UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO
UNIVERSITÉ CATHOLIQUE DE LOUVAIN

UNDER ERASMUS MUNDUS PROGRAMME

Metadata Management in Causally Consistent Systems

Angel Manuel Bravo Gestoso

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues
Co-Supervisor: Doctor Peter Van Roy

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering
Jury final classification: Pass with Distinction and Honour

2018

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO
UNIVERSITÉ CATHOLIQUE DE LOUVAIN

UNDER ERASMUS MUNDUS PROGRAMME

Metadata Management in Causally Consistent Systems

Angel Manuel Bravo Gestoso

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues
Co-Supervisor: Doctor Peter Van Roy

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering
Jury final classification: Pass with Distinction and Honour

Jury

Doctor Charles Nicolas Lucien Pecheur, Université catholique de Louvain

Doctor Luís Eduardo Teixeira Rodrigues, Instituto Superior Técnico, Universidade de Lisboa

Doctor Gregory Chockler, Royal Holloway, University of London

Doctor Ettienne René Paul Riviere, Université catholique de Louvain

Doctor Helena Isabel de Jesus Galhardas, Instituto Superior Técnico, Universidade de Lisboa

Funding Institutions

European Union
Fundação para a Ciência e Tecnologia

2018

“No será peor de lo que era.
No será peor, seguro que es mejor.”

To my family.

Abstract

Causal consistency has emerged as a key ingredient among the many consistency models and client session guarantees that have been proposed and implemented in the last decade. In fact, it has been proved to be the strongest consistency model that does not compromise availability.

Despite its benefits, causal consistency is not trivial to guarantee: one has to keep track of causal dependencies, and to subsequently ensure that operations are delivered in causal order. Interestingly, the granularity at which causal dependencies are tracked impacts significantly the system's performance. When precisely tracking causal dependencies, the costs associated with the processing and transferring of metadata have a significant impact in throughput. It is possible to mitigate this impact by compressing metadata to reduce the amount of metadata handled. Nevertheless, this comes with the cost of losing precision, which penalizes remote visibility latencies—the delay before an operation's effect is observable at remote replicas, due to the creation of false causal dependencies—two concurrent operations which are ordered as an artifact of the metadata management. This tension between throughput and remote visibility latency is inherent to previous work, and it is typically exacerbated when one wants to support partial replication.

This thesis proposes a set of techniques, which combined, alleviate this tension, allowing designers of causally consistent geo-replicated systems to optimize both throughput and remote visibility latency simultaneously, and attain *genuine* partial replication—a key property to ensure scalability when the number of geo-locations increases. The key technique is a novel metadata dissemination service, which relies on a set of metadata brokers, organized in a tree topology. This thesis experimentally demonstrates that, when the topology is well configured, this mechanism allows to implement genuine partial replication and optimize remote visibility latency while keeping the size of the metadata small and constant, crucial to avoid impairing throughput. Furthermore, this service can be decoupled from the service responsible for managing the data, promoting modular architectures for geo-replicated systems.

The metadata dissemination service assumes that each datacenter is able to serialize, in an order consistent with causality, all updates issued locally. This thesis shows how it is possible to efficiently achieve this by integrating services that operate out of the clients' critical operational path.

We have built a prototype, namely SATURN, that integrates all the aforementioned techniques. SATURN is designed as a metadata service that can be used in combination with several replicated data services. We evaluate SATURN in Amazon EC2 using realistic benchmarks under both full and partial geo-replication. Results show that weakly consistent datastores can lean on SATURN to upgrade their consistency guarantees to causal consistency with a negligible penalty on performance: with only 2% reduction in throughput and 11.7ms of extra remote visibility latency in geo-replicated settings. Also, our extensive evaluation shows that our techniques compare favorably to previous state-of-the-art solutions: SATURN exhibits significant improvements in terms of throughput (38.3%) compared to solutions that favor remote visibility latency; while exhibiting significantly lower remote visibility latency (76.9ms less on average) compared to solutions that favor high throughput.

Résumé

La cohérence causale est devenue un élément clé parmi les nombreux modèles de cohérence et les garanties de session client qui ont été proposés et mis en œuvre au cours de la dernière décennie. Il a été démontré être le modèle de cohérence le plus fort qui ne compromet pas la disponibilité.

Malgré ses avantages, la cohérence causale n'est pas triviale à garantir : il faut garder une trace des dépendances causales, et ensuite s'assurer que les opérations sont livrées dans un ordre causal. Fait intéressant, la granularité à laquelle les dépendances causales sont suivies a un impact significatif sur la performance du système. Lorsque l'on suit les dépendances causales avec précision, les coûts associés au traitement et au transfert des métadonnées ont un impact significatif sur le débit. Il est possible d'atténuer cet impact en compressant les métadonnées. Néanmoins, cela entraîne une perte de précision, ce qui pénalise la latence de visibilité à distance—le délai avant que l'effet d'une opération ne soit observable sur des réplicas distants—en raison de la création de fausses dépendances. Cette tension entre le débit et la latence de visibilité à distance est inhérente au travaux de recherche antérieurs, et elle est typiquement exacerbée lorsque l'on veut supporter une réplication partielle.

Cette thèse propose un ensemble de techniques qui atténuent cette tension, permettant aux concepteurs de systèmes géo-répliqués respectant la causalité d'optimiser le débit et la latence de visibilitè à distance simultanément, et d'obtenir une réplication partielle authentique—une propriété importante lorsque le nombre de géo-localisations augmente. Nous encapsulons ces techniques dans un nouveau service de diffusion de métadonnées, qui repose sur un ensemble de courtiers de métadonnées organisés selon une topologie arborescente. La thése démontre expérimentalement que, lorsque la topologie est bien configurée, ce service permet de réaliser une réplication partielle authentique et d'optimiser la latence de visibilité à distance tout en gardant la taille des métadonnées petite et constante, crucial pour maintenir le débit. De plus, ce service peut être découplé du service de gestion des données, ce qui donne une architecture modulaire pour les systèmes géo-répliqués.

Le service de diffusion des métadonnées suppose que chaque centre de données soit capable de sérialiser, dans un ordre compatible avec la causalité, toutes les opérations émises localement. Cette thèse montre comment il est possible d'y parvenir efficacement en introduisant un nouveau service, Eunomia, qui opère en dehors des chemins critiques des clients.

Nous avons construit un prototype, SATURN, qui intègre toutes les techniques mentionnées ci-dessus. SATURN est conçu comme un service de métadonnées pouvant être utilisé en combinaison avec plusieurs services de données répliqués. Nous évaluons SATURN dans Amazon EC2 en utilisant des tests réalistes sous géo-réplication complète et partielle. Les résultats montrent que les services de données faiblement cohérents peuvent s'appuyer sur SATURN pour obtenir la cohérence causale avec une pénalité négligeable sur les performances : seulement 2% de réduction du débit et 11,7 ms de latence de visibilité à distance supplémentaires dans un cadre géo-répliqué. En outre, notre évaluation approfondie montre que nos techniques se comparent favorablement à l'état de l'art antérieur : SATURN présente des améliorations significatives en termes de débit (38,3%) par rapport aux solutions qui favorisent la latence de visibilité à distance ; tout en présentant une latence de visibilité à distance significativement plus faible (76,9 ms de moins en moyenne) par rapport aux solutions qui favorisent un débit élevé.

Resumo

Apesar das suas vantagens, a coerência causal não é trivial de garantir: concretizar este modelo obriga a manter um registo das dependências entre as operaões e a coordenar a aplicação destas operações em cada centro de dados, de forma a respeitar estas dependências. Manter as dependências causais de forma precisa obriga a manter e transferir uma quantidade significativa de metadados, o que limita o d'ebito do sistema. É possível reduzir o tamanho dos metadados mas, tipicamente, isto obriga a perder precisão, criando falsos positivos, isto é, sugerindo relações causa-efeito potencias que não correspondem a dependências reais, o que amplifica a latência na entrega das mensagens. Esta tensão entre a latência e o débito é comum a todos os trabalhos anteriores, e é tipicamente ampliada quando se pretende suportar replicação parcial.

Esta tese propõe novas estratégias para concretizar coerência causal em sistemas replicados suportando replicação parcial que pretendem superar o compromisso entre o débito e a latência acima referido. A técnica chave para conseguir este objectivo consiste na utilização de um serviço de propagação de informação sobre as dependências causais, organizado na forma de um grafo acíclico de encaminhadores de metadados. A tese mostra experimentalmente que, quando a topologia do grafo é escolhida de forma apropriada, é possível capturar as dependências causais recorrendo a poucos metadados e assegurar que os falsos-positivos que resultam desta compressão não afectam de forma significativa a latência das operações, conciliando desta forma o elevado desempenho com a baixa latência. Este serviço pode ser usado de forma desacoplada dos processos de transferências do conteúdo das operações, promovendo arquitecturas de gestão de dados replicados mais modulares.

O serviço de metadados pressupõe que cada centro de dados é capaz de seriar, de forma coerente com a causalidade, todas as operações que são executadas localmente, antes de as propagar para o serviço de metadados. Esta tese mostra como é possível integrar serviços que executam esta tarefa fora do caminho crítico do cliente, como o Eunomia, de forma eficiente, de forma a criar uma arquitectura coerente para gestão de coerência global em larga-escala.

Desenvolvemos um protótipo deste serviço de gestão de metadados, que designamos por SATURN. Este protótipo foi construído de forma a facilitar a sua integração com diversos serviços de dados replicados. Apresenta-se uma avaliação do SATURN, numa configuração que usa os serviços da Amazon EC2, recorrendo a bancadas de teste realistas, em configurações com replicação total e replicação parcial. Os resultados ilustram que as técnicas propostas, ao contrário dos trabalhos anteriores, conseguem de facto oferecer garantias de coerência causal com uma degradação residual do débito e da latência, quando comparadas com sistemas que não fornecem quaisquer garantias de ordenação.

Keywords

Keywords

Causal consistency
Metadata service
Geo-replication
Partial replication
Key-value storage

Mots clés

Cohérence causale
Service de métadonnées
Géo-réplication
Réplication partielle
Stockage de clé-valeur

Palavras chave

Coereência causal
Serviço de metadados
Replicação geográfica
Replicação parcial
Armazenamento chave-valor

Acknowledgments

I would like to deeply thank those that shared this journey with me and helped me along the way.

First of all, I would like to thank my advisors Luís Rodrigues and Peter Van Roy, without them this thesis would not have been possible. They were always supportive and encouraging. Special thanks to Luís. Without his energy, ideas and guidance, this thesis would only be a poorer version of itself. I am glad to have crossed paths with him once again. I hope we keep in touch for many years.

To the SyncFree and LightKone family. It has been real pleasure to have participated in both projects and have met so many inspiring people. Discussions with all the researchers and practitioners involved definetely pushed my research forward. A special word for Valter Balegas, Christopher Meiklejohn, Tyler Crain, Deepthi Akkoorath, Nuno Preguiça, Marc Shapiro, João Leitão, Annette Bieniusa, Carla Ferreira, Rodrigo Rodrigues, and Carlos Baquero.

To all the wonderful master students I was lucky enough to collaborate with, it is always very rewarding working with enthusiastic, bright young students. A special word to Chathuri Gunawardhana. It was great working with her.

To my EMJD-DC, INESC-ID and UCL colleagues, especially to Shady Issa, Emmanouil Dimogerontakis, João Loff, Vasia Kalavri, Jingna Zeng, Ying Liu, João Neto, Ruma Paul, Cheng Li, Nancy Estrada, Paolo Laffranchini, Amin Mohtasham, Amin Kahn, Sileshi Demesie, Subhajit Sidhanta, Pedro

Joaquim, Diogo Barradas, Mennan Selimi, Daniel Porto, Richard Martínez, and David Gureya for always being available to discuss ideas, and for the good times we spent together.

A special mention to Alejandro Tomsic, Zhongmiao Li and Igor Zavalyshyn. They belong to all the groups of people highlighted here as they are collaborators, colleagues and overall friends. Without the countless conversations, lunches, dinners, and drinks shared, these years would have been much tougher.

To Brussels, and Lisbon for being so entertaining, beatiful cities; and to their welcoming people.

To all the people that took care of all the administrative matters at each of the universities. Special thanks to Vanessa Maons, Sophie Renard and Paula Barrancos.

Finally and most importantly, I would like to thank all the friends and members of my family that made this journey more bearable. My parents Miguel and Mariangeles, my sister Candela, and Virginia overall. A special mention to Pablo who visisted me countless times, and shared many many musical, summer nights with me.

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030; the Fundação para a Ciência e Tecnologia (FCT) via projects PTDC/ EEI-SCR/ 1741/ 2014 (Abyss), UID/ CEC/ 50021/ 2013, and the individual doctoral grant SFRH/ BD/ 115972/ 2016; the FP7 project 609 551 SyncFree; and the Horizon 2020 project 732 505 LightKone.

Madrid, 12 June 2018
Manuel Bravo

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

In this first chapter of the dissertation, we motivate the work by arguing why the study of efficient mechanisms to support causal consistency is of high relevance to both theoreticians and practitioners. Furthermore, we describe two open challenges faced by designers of causally consistent geo-replicated systems. In order to support this argumentation, we present a motivational experiment involving two state-of-the-art, causally consistent geo-replicated systems: GentleRain and Cure. Then, we describe the contributions of this thesis, enumerating a summary of its main contributions and results. Finally, we conclude the chapter with a brief description of the content included in each of the chapters of this document.

1.1 Motivation

Distributed data services are a fundamental building block of modern cloud services. These aim at providing an always-on experience to millions of concurrent users, which expect their requests to always be successfully served in a short period of time. Unfortunately, as proved by the CAP theorem [32, 53], some of these tight availability, latency, and throughput requirements are in conflict with data consistency. Specifically, the CAP theorem proves that it is impossible to design a distributed system that it is always available, tolerant to network partitions and strongly consistent.

As a result, a broad class of services have opted for favoring availability and partition tolerant at the cost of strong consistency [44, 3, 66]. Nevertheless, the observation that delegating consistency management entirely to the programmer makes the application code error prone [15] have spurred the quest for meaningful weaker consistency models, which allow the system to remain always available and can be supported effectively by a data service.

Among the several consistency models proposed, causal consistency seems to be pivotal in the consistency spectrum, given that it has been proved to be the strongest consistency model that does not compromise availability [13, 74]. In fact, ensuring that updates are applied and made visible respecting causality has emerged as a key ingredient among the many consistency criteria and client session guarantees that have been proposed and implemented in the last decade. Mechanisms to preserve causality can be found in systems that offer from weaker [98, 71, 9, 103] to stronger [92, 69, 20] consistency guarantees.

A causally consistent system guarantees that an update does not become visible to users of that system until all its causal dependencies are also visible. Causal dependencies among operations are established as clients interact with the system. Informally, an operation b depends on a second operation a, denoted $a \leadsto b$, either because the client issuing b has previously observed the effects of a or because she has observed the effects of another operation c such that $a \leadsto c$ [67, 5] (§2.1.2 gives a more formal definition of causal consistency). Ensuring this invariant, it is useful for applications such as social networks. Consider, for instance, the interaction of Alice and Bob, two users of a social network. Alice wants to share a photo with Bob. Thus, Alice first uploads the photo and then adds it to an album, such that the album contains a reference to the photo. Under weaker consistency models, such as eventual consistency, Bob could first read the album, getting a list of photo references (in which the recently uploaded photo is included) and then try to read the photo without success. Under causal consistency, updating the album will causally depend on the photo. Therefore, if Bob reads a version of the album that includes the photo, Bob should be able to successfully retrieve the photo.

Unfortunately, the designer of a causally consistent geo-replicated storage system is still faced today with a dilemma: there appears to be a tradeoff between throughput and remote visibility latency—the delay before an operation's effect is observable at remote replicas, derived from the granularity at which causality is tracked [28, 56]. Figure 1.1 reports the results of an experiment that illustrates this tradeoff in current systems. In the two plots starting from the left, we compare the performance of two state-of-the-art solutions,

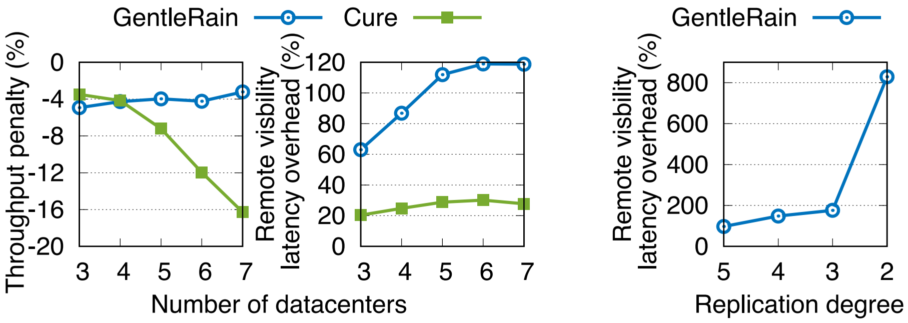


Figure 1.1 – Problems faced by current causally consistent geo-replicated storage systems. Results are normalized against eventual consistency. The latencies among datacentersLeft: Tradeoff between throughput and remote visibility latency. Right: How partial replication affects remote visibility latency.

GentleRain [49] and Cure [7][1]. The former opts for a coarse-grained tracking by compressing metadata into a single scalar. The latter opts for a more fine-grained approach by relying on a vector clock with an entry per datacenter, an approach that most of the causally consistent geo-replicated storage systems in the literature go for. The performance is compared against a store that only guarantees eventual delivery, ensuring no consistency guarantees and therefore requiring no metadata management. In this experiment—deployed in Amazon EC2, we vary the number of datacenters from 3 to 7. The datacenters are added in an order such that each addition increases the maximum latency between any two datacenters. The average latencies among them are listed in Table 4.1. As it can be seen, by keeping little metadata, GentleRain induces a low penalty on throughput but hampers remote visibility latency. This is due to the large number of *false dependencies* inevitably introduced when compressing metadata [39, 40] (a false dependency is created when two concurrent operations are serialized as an artifact of the metadata management). The opposite happens with Cure, that exhibits a low (constant) remote visibility latency penalty but severely penalizes throughput due to the computation and storage overhead

[1] We have chosen these two solutions for two main reasons. First, both solutions share a very similar design with the difference being the amount of metadata used to track causality. This fact serves us to better illustrate the tradeoff. Second, they are—from our perspective—the most scalable and performant solutions of the literature.

associated with the metadata management [16, 49].

Furthermore, current solutions are not designed to fully take advantage of partial replication, a setting of practical relevance [41, 34] in which each datacenter may replicate a different subset of the key-space. The culprit is that causal graphs (a directed graph in which nodes are operations and the edges represent causal dependencies) are not easily partitionable. This fact may force sites to manage not only the metadata associated with the data items stored locally, but also the metadata associated with items stored remotely [71, 16, 103]. Attempts to reduce this effect, by limiting the amount of metadata managed at each site, magnifies the problem of false dependencies, forcing solutions to delay the visibility of remote operations due to operations on data items that are not even replicated locally. To illustrate this problem we run an experiment in which we start from full replication incrementally decreasing the replication degree of each item, until only datacenters close to each other replicate the same data. Figure 1.1 (far right plot) shows the additional visibility latency that is introduced to enforce causal consistency, when using GentleRain. One can observe that GentleRain is incapable of taking advantage of partial replication, imposing longer delays as we reduce the replication factor.

This dissertation studies the fundamental tradeoff, derived from the accuracy in which causality is tracked, between throughput and remote visibility latency, and its relation to both full and partial replication. Is throughput always at odds with remote visibility latency?

1.2 Thesis contributions

In this thesis, we propose a novel modular architecture that integrates a set of techniques, which combined, demonstrate that it is possible to alleviate this tension such that both throughput and remote visibility latency can be optimized simultaneously.

The proposed architecture does a clear separation of the consistency concerns from the responsibilities of the underlying storage system—such as replication and durability—based on the separation between metadata and data management.

The key technique is a novel metadata dissemination service. This service is responsible for notifying datacenters about the order in which these must make remote operations visible to local clients such that causal consistency is guaranteed. The service is devoted exclusively to metadata management. We experimentally demonstrate that when the service is well configured, it enables causally consistent data services to optimize throughput and remote visibility latency simultaneously. In order to add minimal overhead due to metadata handling, the service only requires managing small pieces of metadata, independently of the number of clients, servers, partitions, and locations. This fact is a crucial requirement to avoid impairing throughput. Unfortunately, due to the aggressive metadata compression strategy, a large number of false dependencies is unavoidably generated. In order to diminish its impact on remote visibility latency, the service exploits the fact that causal consistency is a partial order to enforce at each datacenter a different serialization of remote operations, crafted to maximize the performance of that datacenter.

The architecture of the metadata dissemination service is key to achieve these, a priori conflicting, goals. The service is distributed geographically by means of a set of metadata brokers organized in a tree topology. The fact that a tree topology permits ensuring causal consistency trivially enables the service to only handle small pieces of metadata. Nevertheless, in order to optimize remote visibility latency not every tree topology is valid. In this thesis, we propose a configuration service that finds a tree topology that optimizes the average remote visibility latency, given the latencies among the datacenters, a set of possible locations where to place the metadata brokers and the relative importance of paths between pairs of datacenters, reflecting the business goals of the application.

Interestingly, the tree topology is also key to support partial replication efficiently. Thus, the metadata dissemination service only notifies datacenters about the order of those remote operations replicated locally, omitting the order between these and other remote operations of no local interest. This enables genuine partial replication, a desirable scalability property that requires datacenters to manage only the data and metadata concerning items replicated locally.

Finally, although the design of the metadata dissemination service is decoupled from the implementation details of each datacenter, the service—naturally—needs to interact with each datacenter. In this thesis, we also study the requirements imposed by the metadata dissemination service to the datacenter implementation. Specially relevant is the fact that the metadata dissemination service requires each datacenter to generate a causal serialization of the updates issued locally. This thesis addresses this problem and shows how a service operating out of the client's critical operational path is ideal to efficiently solve it. Concretely, we demonstrate how to integrate an existing metadata serialization service, namely Eunomia [59], with the metadata dissemination service and the rest of the intra-datacenter components.

We have built a prototype, namely SATURN, integrating all these techniques that we have deployed on Amazon EC2. Our evaluation using both microbenchmarks and a realistic Facebook-based benchmark shows that eventually consistent systems can use SATURN to upgrade to causal consistency with negligible performance overhead (namely, with only 2% reduction in throughput and 11.7ms of extra remote visibility latency in geo-replicated settings) under both full and partial replication. Furthermore, our solution offers significant improvements in terms of throughput (38.3%) compared to previous solutions that favor remote visibility latency [7]; while exhibiting significantly lower remote visibility latency (76.9ms less on average) compared to previous solutions that favor high throughput [49].

In the following subsections, we first summarize the main contributions and results of this thesis. Then, we present a list of publications that include some of the results presented in this document.

1.2.1 Summary of contributions

In summary, the primary contributions of this dissertation are as follows:

- The design of a modular architecture for ensuring causal consistency in geo-replicated systems. The architecture does a clear separation of the consistency concerns from the responsibilities of the underlying storage system—such as replication and durability—based on the separation between metadata and data management.

- The design of a novel metadata dissemination service responsible for notifying datacenters about the order in which these must make remote operations visible to local clients such that causal consistency is guaranteed. This service leverages a set of metadata brokers, organized in a tree topology, to handle the efficient dissemination of causal metadata among datacenters.

- The design of a configuration service. Pre-configuring the metadata dissemination service is fundamental for its well functioning. We model the problem of configuring the metadata dissemination service as an optimization problem and rely on a heuristic technique that leverages a constraint solver to find a "good" configuration in a reasonable amount of time.

- Specification of the requirements of service provided by the metadata dissemination service and how it should interoperate with the data services' components. Ultimately, this resulted in the design of Eunomia, a metadata serialization service able to efficiently generate a causal serialization of all operations local to a datacenter by operating out of the clients' critical operational path. The actual implementation of Eunomia was carried out in [59].

1.2.2 Summary of results

The main results of this dissertation are as follows:

- The design and implementation of a variant of both GentleRain and Cure with support for partial replication.
- The design and implementation of SATURN, the first distributed metadata service for causal consistency capable of efficiently supporting partial replication, and optimizing both throughput and remote visibility latency simultaneously.
- A tool to configure SATURN such that remote visibility latencies are optimized on average, given a deployment and the application's business goals.
- A sound and complete evaluation of the SATURN's components, as well as a comparison with GentleRain [49] and Cure [7], two of the most performant state-of-the-art solutions.

1.2.3 Publications

Some of the results presented in this thesis have been published as follows:

- M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues. On the use of clocks to enforce consistency in the cloud. *IEEE Data Engineering Bulleting*, 38(1):18-31, 2015 [28].

- M. Bravo, L. Rodrigues, and P. Van Roy. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, Middleware Doct. Symposium '15, pages 5:1-5:4, Vancouver, BC, Canada, 2015 [29].

- M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111-126, Belgrade, Serbia, 2017 [30].

- C. Gunawardhana, M. Bravo, and L. Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 83-95, Santa Clara, CA, USA, 2017 [59].

During my doctoral studies, I have also explored other directions and collaborated in several projects that have helped me to get insights on the challenges of providing consistency in geo-replicated systems. These efforts have led me to contribute to the following publications:

- I. Briquemont, M. Bravo, Z. Li, and P. Van Roy. Conflict-free partially replicated data types. In *Proceedings of the 7th International Conference on Cloud Computing Technology and Science*, CloudCom '15, pages 282-289, Vancouver, BC, Canada, 2015 [33].

- M. Bravo, P. Romano, L. Rodrigues, and P. Van Roy. Reducing the vulnerability window in distributed transactional protocols. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 10:1-10:4, Bordeaux, France, 2015 [31].

- M. Couceiro, G. Chandrasekara, M. Bravo, M. Hiltunen, P. Romano, and L. Rodrigues. Q-opt: Self-tuning quorum system for strongly consistent software defined storage. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 88-99, Vancouver, BC, Canada, 2015 [42].

- C. Bartolomeu, M. Bravo, and L. Rodrigues. Dynamic adaptation of geo-replicated crdts. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 514-521, Pisa, Italy, 2016 [21].

- D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceeding of the IEEE 36th International Conference on Distributed Computing Systems*, ICDCS '16, pages 405-414, Nara, Japan, 2016 [7].

In Chapter 6, we provide further details on how these results relate to this dissertation.

1.3 Outline

The remainder of this disseration is organized as follows.

Chapter 2 provides background on cloud services and causal consistency. Furthermore, we discuss two fundamental aspects to consider when implementing causal consistency: the tradeoff between throughput and remote visibility latency; and the challenges of adopting partial replication.

Chapter 3 presents the design of SATURN, our prototype that integrates all the proposed techniques. The chapter includes the description of the novel metadata dissemination service; a solution to find the right configuration of the service such that remote visibility latencies are optimized; a list of requirements that the service imposes to geo-replicated dataservices; and its integration with metadata serialization services, such as Eunomia, that operate out of the clients' critical operation path. The chapter also discusses the fault-tolerance and adaptability aspects of our approach.

Chapter 4 presents a complete and sound evaluation of our techniques. We evaluate each of them individually, comparing them to alternative approaches. Also, we compare SATURN, attached to a data service, to state-of-the-art causally consistent data services.

Chapter 5 proposes a taxonomy to classify existing causally consistent systems and discusses the most relevant existing solutions.

Chapter 6 concludes this dissertation with a list of take away messages, a discussion about the limitations of our approach, a list of topics for future work, and final remarks.

Chapter 2

Causal consistency: model & challenges

In this chapter, we first discuss why causal consistency is relevant for today's cloud services. Then, we formally define it. Finally, we discuss two characteristics inherent to causal consistency that should be taken into consideration when designing causally consistent distributed database systems: the throughput vs. remote visibility latency tradeoff; and why partial replication is at odds with metadata compression.

2.1 Causal consistency

2.1.1 Why causal consistency

To better understand why causal consistency is an interesting consistency criterium in practice, we first overview current cloud services architectures and the requirements they imposed over the distributed database systems they rely on to manage the application state.

Cloud services: architecture and requirements

Cloud services handle millons of client requests per second. These services are usually composed of two tiers: the application tier and the storage tier. The former is composed of a set of stateless servers that handle client requests by executing application code that reads and updates the storage tier. The application tier shields clients from the internal complexities of the storage tier; e.g. which machines to contact to read or update a specific piece of data, or how many replicas are maintained. The latter maintains the application state sharded across multiple servers and replicated among multiple datacenters.

These services would ideally require the distributed database system featuring the storage tier to have the following properties:

1. **Always-on.** A cloud service should provides an "always on" user experience to keep users engaged. This implies that the service is always available and that requests are always served in a reasonable amount of time. Therefore, these systems have to be tolerant to *network partitions* and should exhibit *low latency*.

   Network partitions occur within and across datacenters, as some recent studies report [54, 19, 99]. For instance, a study that measures and analyses network failures in several Microsoft datacenters reports that more than 13,300 network failures occurred during one-year period whose effects were observable by end-users [54]. The study also reports that it took five minutes to repair each failure on average, taking up to a week for some of them.

   Low latency is fundamental for cloud services to keep users engaged. In fact, studies have shown that even small increases on latency have a direct negative impact on revenue [46, 88, 70]. Ensuring low latency is challenging under geo-replication as remote communication have substantial cost given the distance among datacenters. For instance, among the São Paolo and Singapore Amazon EC2 regions, ping packets exhibit an average 362.8ms round-trip-time [14]. Besides, the fact that a single user request can be forked in thousand of sub-requests—as reported by Facebook [6]—augments the problem.

2. **Scalability.** The database system should scale-out horizontally. Thus, cloud services operators expect that, when adding new resources to the database system, the aggregate computational and storage power increases accordingly. This is fundamental, as the alternative, namely vertical scaling, could potentially make deployment cost soar, given current load and the necessity to adapt to increases in load.

3. **Strong consistency.** Ideally, systems should ensure *linearizability* [60], the strongest consistency model. This criterium creates the illusion that

the distributed database is a centralized component. This allows developers to reason more simply about what to expect when reading and updating the database, greatly simplifying application development. Under linearizability, operations seem to take effect in the entire system at a single point in time between the moment in which the request is received and the moment in which is completed (the moment the result of the operation is sent to back to the user). Once the system acknowledges the completion of a write operation on a item, all subsequent reads on that item will reflect the written state. Linearizability precludes anomalies observable by end-users; e.g., a user that would not observe its own writes, or a user that would observe a set of events in an order that does not match reality (the order in which an external observer would witness the succession of events). Also, under linearizability, maintaining application invariants becomes trivial; e.g., keeping the balance of a bank account above zero.

Unfortunately, linearizability is expensive to implement in practice, specially under geo-replication. The absence of consistency can be compensated by ad-hoc mechanisms at the application level, but this is error prone [15]. Thus, weaker, but still meaningful, consistency models have been proposed; from stronger models such as RedBlue consistency [69] to weaker models such as causal consistency [71].

Casual consistency, a sweet spot

Unfortunately, some of the desirable properties are in conflict. As stated by the CAP theorem [32, 53], a replicated distributed system cannot offer strong consistency and ensure availability (more concretely be tolerant to network partitions) simultaneously. As a result, one property has to be sacrificed. Interestingly, cloud services have chiefly chosen to favor availability [44, 66], sacrificing consistency. Nevertheless, one does not have to give up consistency completely. Weaker consistency models have been proposed that do not compromise availability [98, 71, 97].

Causal consistency seems to be pivotal in the consistency spectrum, as it has been proved to be the strongest consistency model that an “always-on” system can ensure [13, 74]. Thus, causal consistency is the strongest model that a distributed database system with availability requirements can aim at, making the study of efficient mechanisms to support causal consistency of

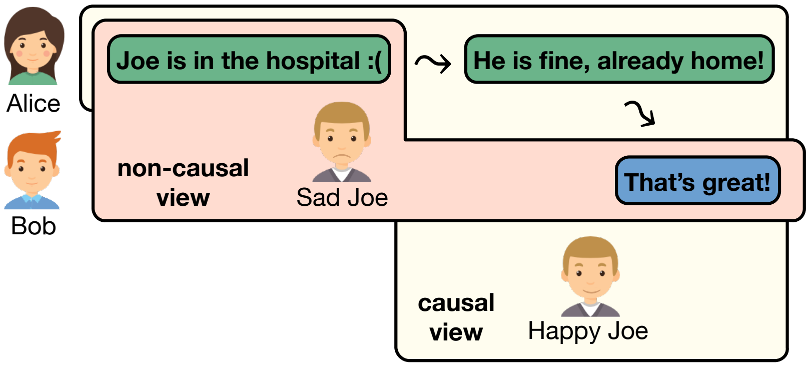


Figure 2.1 – Social network interaction among three users: Alice, Bob and Joe.

absolute practical relevance.

2.1.2 Definition

Causal consistency defines intuitive semantics: it guarantees that, for any operation j, all operations on which j *causally* depends, take effect before j. Causal dependences are determined by the happened-before relation ($\rightsquigarrow$) [67, 5], which is defined by three rules:

1. *Thread of execution*: If a and b are two operations executed by the same thread of execution (e.g., by the same client), then $a \rightsquigarrow b$ if a happens before b.

2. *Reads from*: If a is a write operation and b is a read operation that reads the value written by a, then $a \rightsquigarrow b$.

3. *Transitivity*: If $a \rightsquigarrow c$ and $c \rightsquigarrow b$, then $a \rightsquigarrow b$.

Definition 1 (Causal consistency)**.** *A data service is causally consistent if, when a certain operation is visible to a client, then all of its causal dependencies are also visible.*

Causal consistency precludes some consistency anomalies otherwise observable by end-users under weaker consistency models, e.g., a set of comments in a social network that are displayed in an order that make no sense from the point of view of the observer. Figure 2.1 shows an example. A user Alice posts in a social network that Joe (another user) is in the hospital. Subsequently, Alice discovers that it is nothing serious and comments on her own post: “He is fine, already home”. A third user Bob, which is friend of Joe, reads the second comment and reacts to it replying “That’s great!”. If causal consistency is not enforced, when Joe logs in to the social network and reads the conversation, he could observe Joe’s “That’s great!” comment before Alice’s “He is fine” comment (or not observe the latter at all), suggesting that Bob was actually celebrating the fact that Joe had to go to the hospital. Under causal consistency, Joe could observe only the first comment of Alice, the first and the second, or all of them, but never Bob’s comment without both Alice’s comments.

Figure 2.1 also serves us to illustrate the three rules of the happened-before relation. Both Alice’s comments are causally related due to the first rule (thread of execution). The second comment of Alice causally precedes Bob’s, as Bob reads it before commenting. This is due to the second rule (reads from). Finally, The first comment of Alice also causally precedes Bob’s due to the third rule (transitivity).

Finally, an alternative way of thinking of causal consistency is as the set of all client (or session) guarantees [97]. There are four: read-my-writes, monotonic-reads, monotonic-writes, and writes-follow-reads. The read-my-writes guarantee ensures that writes made by a client are visible to subsequent reads made by that client. The monotonic-reads guarantee ensures that a read made by a client includes at least the effects of the writes already observed by a previous read made by that client. The monotonic-writes guarantee ensures that a write made by a client only takes effect after all previous writes made by that client. Finally, the writes-follows-reads guarantee ensures that writes made by a client take effect only after the writes whose effects were observed by reads made by that client. A database system ensuring the four guarantees, ensures causal consistency [35].

2.1.3 Causality in practice

The great advantage of causal consistency over stronger consistency models is that transparently allows for asynchronous replication. This fact has significant impact on the properties a cloud database can feature. First, operations can be completed in a single datacenter, requiring no inter-datacenter coordination.

The consequences of this fact are twofold: (i) the system becomes tolerant to network failures (at least to the most damaging ones); (ii) long communication round-trip-times are removed from the clients' critical operational path, allowing to achieve lower latencies. Second, casual consistency does trivially allow for sharding, enabling horizontal scaling.

Practical implementations of causal consistency share a common pattern. Read and writes are completed in a single datacenter without requiring synchronous coordination with other datacenters. Writes are tagged with a piece of metadata identifying operation's causal dependencies. Each datacenter asynchronously propagates local writes to the rest of the datacenters replicating the data item being updated. The causal dependencies are either tracked by the clients [71, 47] or by the datacenter servers [103, 9]. In most of the implementations of causal consistency, the effects of a write operation cannot become visible to clients in a datacenter until all operations, on which the write causally depends, have taken effect in that datacenter. Some implementations simply defer applying writes until the dependencies are known to have taken effect [103, 71]; others simply preclude its visibility [49, 7].

2.2 Throughput at odds with remote visibility latency

Interestingly, precisely tracking dependencies have a significant impact on throughput. The cost is associated to the computational and storage overhead derived from the amount of metadata used to represent these dependencies. As a reaction to this problem, the community have proposed solutions to reduce the amount of metadata being handled. One can compress the metadata by serializing sources of potential concurrency. This is for instance, considering that all write operations local to a datacenter happen one after the other, instead of capturing the inherent concurrency of those operations performed concurrently by different clients and on possibly different servers. Therefore, if one depends on two operations local to the same datacenter, it would be enough keeping track of only the operation ordered last, reducing the amount of metadata used considerably.

Unfortunately, when compressing metadata, one penalizes *remote visibility latencies*. We define remote visibility latency as the time interval between the instant in which an update is installed at its originating datacenter and when it becomes visible at remote datacenters. The fundamental problem is that when compressing metadata, one is fusing sources of concurrency and thus creating false dependencies. A false dependency is created when two concurrent operations are serialized as an artifact of the metadata management. Thus, a datacenter receiving an operation that according to its "real" causal dependencies (those captured by the three rules of the happened-before relation) could be applied locally immediately, may have to defer its installation due to another operation, which is concurrent according to the happened-before relation but that appears to be a causal dependency due to the metadata compression.

Let us illustrate this with a simple example. Assume that we opt for serializing all operations local to a datacenter. In this setting, if an operation c causally depends on two other operations a and b, concurrent among them and local to the same datacenter, c only has to carry one of the two as dependency: the operation that comes later in the serialization. Therefore, assuming that a is serialized before b, c will only need to carry b as dependency, as b will carry a as its own dependency, and by transitivity in order to make c visible, one needs to have installed both a and b. This reduces the size of the metadata but adds a false dependency between operations a and b, which are concurrent, forcing datacenters to wait until a is visible before making b visible.

The impact of increasing this latency is twofold. (i) Users observe a staler view of the database, leading to a worse user experience and possibly impacting revenue for some services; e.g., advertising services whose costumers pay based on the pre-agreed number of ad imprints [11]. (ii) Users moving across datacenters experience longer delays. Users may move due to roaming, failures or partial replication.

The research community has extensively explore this tradeoff; proposing solutions that optimize throughput by aggressively compressing the metadata [49], solutions that barely compress metadata [71, 47]—favoring remote visibility latency and solutions that opt for an intermediate approach [7, 103, 9].

2.3 Is partial replication easy to adopt?

Surprisingly, most of previous solutions have been designed for a full replication setting in which all datacenters replicate the full application state. Is partial replication a setting that it is challenging to adopt under causal consistency or it is simply a setting not considered previously? In this section, we take a closer look to this. First, we motivate why partial replication is an interesting setting. Then, we discuss the fundamental challenges when adopting partial replication under causal consistency. Finally, we define *genuine* partial replication, which is ideally the type of partial replication that a solution should aim at implementing.

2.3.1 A case for partial replication

Partial replication is a setting of practical relevance and has the potential of bringing significant savings in deployment costs. We now list a set of anecdotical evidence and recent studies that support our statement:

1. It is obvious that partial replication has the potential of bringing significant savings in deployment costs. One of the fundamental reasons for services to replicate application state in multiple distant locations is to reduce the end-user observable latency. Nevertheless, it seems to be a waste of resources to replicate the full application state, as a single user is usually not interested in accessing all data. Interestingly, in some applications, there seems to be a correlation between the data accessed and the geographical location of the users. For instance, A. Brodersen et al. [34] study the geographic popularity of more than 20 millions of YouTube videos. The study concludes that for about 50% of the videos, more than 70% of their views, correspond to users belonging to a single geographical region. Therefore, minimizing the amount of data shared among datacenters should be something to consider as it could bring significant storage and operational savings, without impacting end-user observable latency substantially.

2. Cloud service providers already consider partial replication when designing their distributed database systems; e.g., Google's Spanner database [41], which is offered as service to users of the Google Cloud Platform [2] and used by multiple Google cloud services such as F1 [91], is specifically designed to allow applications to partition data across different datacenters.

3. Interestingly, given the ever-growing massive amount of data handled by large cloud services, it seems that partial replication will soon become the default setting for them. For instance, Facebook reported in 2014

that "*while the small data stores are replicated globally, Facebook's data warehouse cannot be stored in any single datacenter, so it's partitioned globally*" [1].

4. Research community have acknowledged that the design of causally consistent database systems with support for partial replication [71, 16, 103] is an interesting research challenge. For instance, P. Bailis et al. state that "*While weaker consistency models are often amenable to partial replication (i.e., replicating to a subset of participants), allowing flexibility in the number of datacenters required in causally consistent replication currently remains an interesting aspect of future work*" [16].

5. Edge and fog computing are promising computing paradigms which aim at reducing end-user latency and enhancing scalability by performing data processing at nodes situated at the logical extreme of a network (closer to end-users). An edge network therefore is composed by a set of heterogeneous computing nodes; e.g., points-of-presence, mobile devices, datacenters, and more. Under these paradigms, partial replication is mandatory, as the storage, computation and data transmission capabilities of many of the devices situated on the edge are severely constrained.

2.3.2 Efficient causal consistency in conflict with partial replication

Implementing efficient causally consistent database systems requires minimizing the amount of metadata being handled. As discussed before, when compressing metadata, we merge sources of concurrency; e.g. serializing all operations local to a datacenter, all operations on the same data item, or even all operations happening in the system. This allows solutions to represent multiple causal dependencies as it is was only one, reducing the size of the metadata. Unfortunately, this fact makes difficult for solutions to take full advantage of partial replication.

Let us illustrate this with a simple example (Figure 2.2). Assume a deployment with three datacenters dc_1, dc_2, and dc_3 in which the metadata is compressed into a single scalar, meaning that all operations happening in the system are serialized. Clients in dc_1 issue two operations a and b such that a is serialized before b. a has to be replicated in all datacenters, b only in dc_2. A client in dc_2 reads the effects of b (once this has been installed in dc_2) and

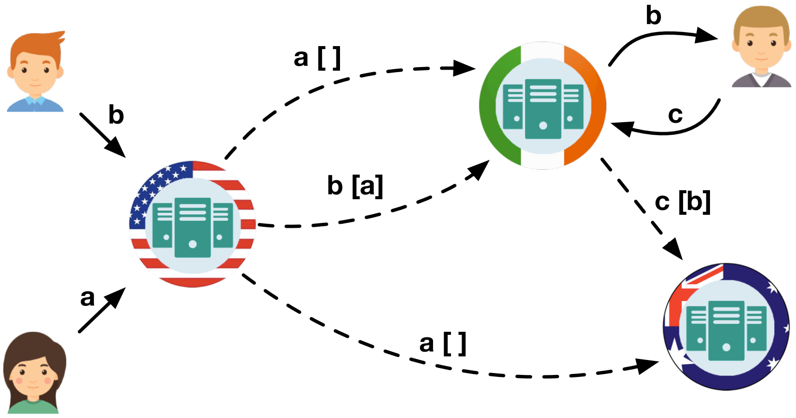


Figure 2.2 – Example of interaction among three datacenters in a partially replicated setting. In the example, all three operations (each issued by a different client) are logically serialized to minimize metadata (abc). Each operation only carries (within brackets) its predecesor in the serialization as dependency.

issues a third operation c such that c is serialized after b. c has to be replicated in dc_3. In order to take advantage of the metadata compression, each operation only carries as dependency, the operation that precedes it in the serialization (benefiting from transitivity). Thus, c can only be installed in a datacenter once b has been installed. Since dc_3 is not interested in b (e.g.; the data item updated by b is not replicated in dc_3), one could say that dc_3 can install c immediately. This is false, as by transitivity, c depends on, not only b, but also a. Therefore, in order to make c visible, dc_3 has to ensure that all dependencies of b are already installed, if replicated locally. Unfortunately, information about b's dependencies is only carried by operation b. This means that even though b does not need to be replicated locally, it has to be received by dc_3 (not necessarily the payload of the operation, but at least the information regarding dependencies). The implication is twofold: (i) there is some computational overhead as datacenters still need to handle at least the metadata of some operations that are not replicated locally; (ii) this fact augments the problem of false dependencies, affecting remote visibility latencies negatively.

The culprit is that causal graphs (a directed graph in which nodes are operations and the edges represent causal dependencies) are not easily partitionable.

Otherwise, one could eliminate this problem by: (i) partitioning the application state in groups of data items such that there will never be causal dependencies among operations mutating data items belonging to different groups; (ii) serializing all operations mutating data items that belong to the same group.

Of course, in our illustrative example, we are aggressively compressing metadata and therefore the impact of this problem is more significant. Nevertheless, the problem is still present in solutions that opt for a fine-grained tracking of causal dependencies such as [71], as we later discuss.

2.3.3 Genuine partial replication

Among the possible implementations of partial replication, the most interesting, due to its scalability properties, is *genuine* partial replication. Roughly speaking, a partially geo-replicated database system is genuine if datacenters are required to manage only the data and the metadata of the data items replicated locally. This enhance scalability. First, it minimizes the computational overheads of non-genuine implementations, as these have to handle metadata and possibly data of operations which are irrelevant locally. Second, it shields the remote visibility latency of the operations being replicated locally from the effects caused by operations on data items that are not replicated locally.

Genuineness was introduce in the context of atomic multicast in asynchronous distributed systems [57] to characterized scalable implementations of atomic multicast. R. Guerraoui et al. [57] state that an atomic multicast implementation is genuine if only the process that sends the message and the processes that have to receive it are involved in the protocol required to deliver the message. Our definition is just a specialized variant in the context of geo-replication, in which processes are datacenters. Genuine partial replication has also been used to characterized distributed transactional protocols [85, 80].

We claim that a causally consistent geo-replicated database system should implement genuine partial replication in order to take full advantage of partial replication.

Chapter 3

The design of Saturn

In this chapter, we describe the key techniques proposed in this thesis. We describe these in detail as we present SATURN, a prototype that integrates them all. In Chapter 4, we use SATURN to evaluate our techniques and compare them to state-of-the-art alternatives.

We first give a general view of the techniques integrated in SATURN and how all the parts blend to achieve our goal: to alleviate the tension between throughput and remote visibility latency inherent to causal consistency, while supporting scalable partial replication (§3.1). Second, we describe the main contribution of this thesis: a novel metadata dissemination service that leverages a set of metadata brokers, namely serializers, organized in a tree topology to propagate causal metadata among datacenters (§3.2). Third, we propose a method to configure the metadata dissemination service. Configuring the service implies finding a tree topology that, given a deployment, permits the optimization of remote visibility latencies. This step is key for the well functioning of SATURN. We model the problem as an optimization problem and rely on a heuristic technique that leverages a constraint solver to find a "good" solution in a reasonable amount of time (§3.3). Then, we describe what the metadata dissemination service requires from data services in order to be attachable (§3.4). Among the requirements, the metadata dissemination service requires datacenters to serialize local updates in an order consistent with causality. In order to maximize the system's throughput, SATURN integrates Eunomia [59], an existing fault-tolerant datacenter service that efficiently undertakes this task at each datacenter. Finally, in the last two sections of the chapter, we discuss the fault-tolerant and adaptability aspects of our techniques.

3.1 Design

SATURN is a metadata service designed to be attached to already existing geo-replicated data services to orchestrate inter-datacenter update visibility. SATURN aims at enforcing causal consistency, with negligible performance penalty under both full and partial geo-replication, such that: *clients always observe a causally consistent state (as defined in [5, 67]) of the storage system independently of the accessed datacenter*. It follows that: i) the metadata handled by SATURN has to be small and fixed in size, independently of the system's scale; ii) the impact of *false dependencies* [39, 40], unavoidably introduced when compressing metadata, has to be mitigated; and, for obvious scalability reasons, (iii) a datacenter should not receive or store any information relative to data that it does not replicate (i.e., it must support *genuine* partial replication [57]).

3.1.1 Overview

Figure 3.1 shows a schematic view of the parties involved. SATURN is attached to a data service that spans multiple datacenters. These are geographically distributed and store the application state. Clients interact with the datacenters by issuing read and update operations. SATURN is responsible for ensuring that causal consistency is always guaranteed by orchestrating the dissemination of update operations among datacenters. To ensure the well functioning of the system—the optimization of the remote visibility latency, SATURN has to be configured by system operators through a configuration service in a prior step. SATURN also integrates mechanisms to allow for online reconfiguration (§3.6).

SATURN is devoted exclusively to metadata management. Thus, it assumes the existence of some bulk-data transfer scheme that fits the application business requirements. The decoupling between data and metadata management is key in the design of SATURN. First, it relieves the datastore from managing consistency across datacenters, a task that may be costly [16, 49]. Second, this separation permits SATURN to handle heavier loads independently of the size of the managed data. To the best of our knowledge, SATURN is the first decentralized implementation of a metadata manager for causal consistency (a centralized metadata service has been previously proposed in [51]).

SATURN only manages small pieces of metadata, called *labels*, in order to add minimal overhead due to metadata handling. Labels uniquely identify op-

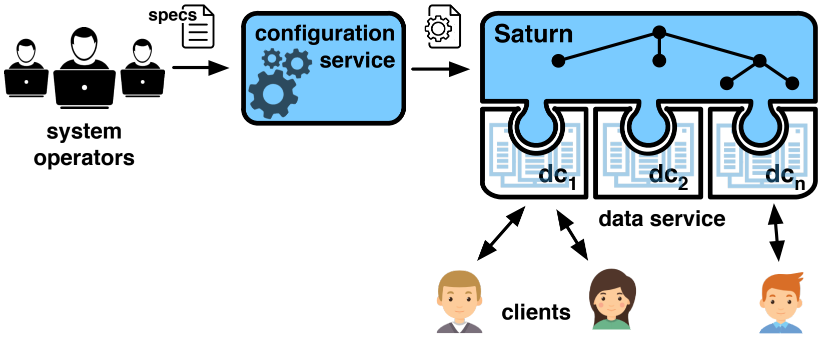


Figure 3.1 – General architecture. SATURN is integrated with a data service, which spans multiple datacenters. System operators configure SATURN through a configuration service. Clients interact with the datacenters of the underlying data service.

erations and have constant size. In SATURN, datacenters are responsible for (i) generating labels (when clients issue update requests), (ii) passing them to the SATURN's metadata dissemination service, in an order that respects causality, and (iii) attaching them to its corresponding update payload before delivering the updates to the bulk-data transfer mechanism. SATURN integrates the necessary mechanisms to support this efficiently. It includes a metadata serialization service (§3.4), namely Eunomia [59], that totally orders all labels local (generated at) to a datacenter in an order consistent with causality and pushes them to the SATURN's metadata dissemination service. Although the Eunomia service is not fundamental for the operation of SATURN (other services, such as a sequencer, could be used instead), it is important to attain high throughput. Finally, label generation and its subsequent attachement to operation payloads is handled by a subcomponent named gear, which is attached to each of the datacenter's servers.

SATURN is then responsible for propagating labels among datacenters and for delivering them to each interested datacenter in causal order. In turn, each datacenter applies remote updates locally when it has received both the update payload (via the bulk-data transfer mechanism), and its corresponding label from the metadata dissemination service. SATURN exploits the fact that causal consistency is a partial order to diminish the impact of false dependencies, otherwise created due to the constraint amount of metadata used. Thus, SATURN delivers to each datacenter a different serialization of labels, which is crafted to maximize the performance of that datacenter. In addition, labels include information w.r.t the data being updated. Based on this information, SATURN can selectively deliver labels to only the set of interested datacenters, enabling genuine partial replication.

The architecture of the metadata dissemination service (§3.2) is key to ensure the well functioning of SATURN. The service leverages a set of metadata brokers, distributed geographically and organized in a tree topology, to propagate labels from the origin datacenter to the possibly multiple destinations. Nevertheless not any tree topology is capable of optimizing the remote visibility latency. SATURN requires thus a prior step, before being operational, to find the appropriate topology. In §3.3, we present a method that, given some characteristics of the deployment and the application—e.g., the number of datacenters and the latencies among them, finds a topology of metadata brokers that effectively optimizes remote visibility latency. Roughly, the topology of metadata brokers must connect nearby datacenters through fast paths and establish slower paths among distant ones.

We assume that clients communicate via the storage system (with no direct communication among them). A client normally connects to a single datacenter (named the preferred datacenter). Clients may switch to other datacenters if they require data that it is not replicated locally, if their preferred datacenter becomes unreachable, or when roaming. Clients maintain a label that captures their causal past (more precisely, this is maintained by library code that runs with the client). This label is updated whenever the client reads or writes an item in the datastore if the new operation is not already included in the client's causal history. The client label is also used to support safe—without violating causality—client migration among datacenters.

Finally, like many other competing systems [71, 72, 47, 49, 7], SATURN assumes that each storage system datacenter is linearizable [60]. This simplifies metadata management without incurring any significant drawback: previous work has shown that linearizability can be scalably implemented in the local area [10], where latencies are low and network partitions are expected to only occur very rarely, especially in modern datacenter networks that incorporate redundant paths between servers [8, 55].

3.1.2 Labels: structure and properties

SATURN implements labels as follows. Each label is a tuple ⟨*type*, *src*, *ts*, *target*⟩ that includes the following fields:

- *type* captures the type of the label. SATURN uses two different label types, namely *update* and *migration*. An *update* label is generated when a client issues a write request. A *migration* label is created when a client needs to migrate to another datacenter. Migration labels are not strictly required to support client movement but may speedup this procedure.
- *src* (source) includes the unique identifier of the entity that generated the label.
- *ts* (timestamp) is a single scalar.
- *target*: indicates either the data item that has been updated (meaningful for update labels), or the destination datacenter (meaningful for migration labels).

Labels have the following properties:

Property 1 (Uniqueness). *The combination of the* ts *and* src *fields makes each label unique.*

Property 2 (Comparability). *Let l_a and l_b be two labels assigned to different updates by* SATURN. *Assuming that source ids are totally ordered, we say that $l_a < l_b$ iff:*

$$l_a.\text{ts} < l_b.\text{ts} \vee (l_a.\text{ts} = l_b.\text{ts} \wedge l_a.\text{src} < l_b.\text{src}) \tag{3.1}$$

Labels can therefore be totally ordered globally. The total order defined by labels respects causality. In particular, given two updates, a and b, if b causally depends on a (denoted $a \rightsquigarrow b$) then $l_a < l_b$. Similarly to Lamport clocks [67], the converse is not necessarily true, i.e. having $l_x < l_y$ does not necessarily indicate that $x \rightsquigarrow y$. In reality, x could be concurrent with y and still $l_x < l_y$. This derives from the fact that causal order is a partial order and, therefore, there are several serializations of the labels that respect causality (the serialization defined by their timestamps is just one of these).

Interestingly, the fact that the timestamp order respects causality, enhances the robustness and availability of the architecture. Therefore, in the unlikely case of a SATURN outage (SATURN has been implemented as a fault-tolerant service), a datacenter may always fallback to make updates visible in timestamp order.

3.2 Label propagation

In this section, we present the design of the SATURN's component in charge of propagating labels among datacenters: the metadata dissemination service. We start with an intuitive example that aims at introducing the tradeoffs involved in the design of this service and at highlighting the potential problems caused by false dependencies. We then define precisely the goals that the metadata dissemination service should meet. Finally, we discuss a concrete architecture for the service.

3.2.1 Rationale

The role of SATURN is to deliver, at each datacenter, in a serial order that is consistent with causality, the labels corresponding to the remote updates that need to be applied locally. Given that there may exist several serial orders matching a given partial causal order of events, the challenge is to select (for each datacenter) the "right" serial order that allows enhancing the system's performance.

In many aspects, SATURN's metadata dissemination service acts as a publish-subscribe system. Datacenters publish labels associated with updates that have been performed locally. Other datacenters, which replicate the item associated, subscribe to those labels. SATURN is in charge of delivering the published events (labels) to the interested subscribers. However, SATURN has a unique requirement that, to the best of our knowledge, has never been addressed by any previously designed publish-subscribe system: SATURN must mitigate the impact of false dependencies that are inevitably introduced when information regarding concurrency is lost in the serialization process. As we have seen, this loss of information is an unavoidable side effect of reducing the size of the metadata managed by the system.

In this section, we use a concrete example to convey the intuition of the tradeoffs involved in the design of SATURN to match the goal above. Consider

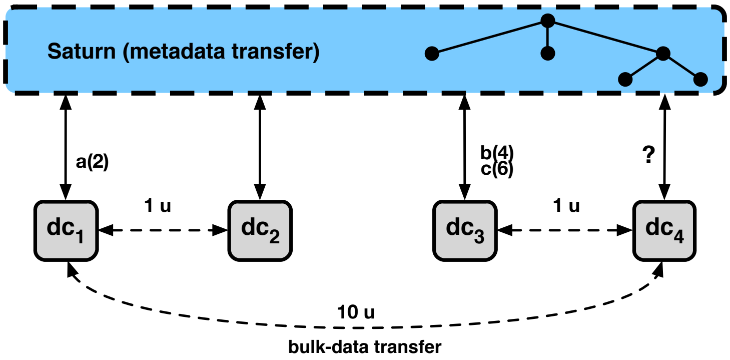


Figure 3.2 – Label propagation scenario.

the scenario depicted in Fig. 3.2. Here, we consider a scenario with four datacenters. Some items are replicated at dc_1 and dc_4 and some other items are replicated at dc_3 and dc_4. Let us assume that the bulk-data transfer from dc_1 to dc_4 has a latency of 10 units while the transfer from dc_3 to dc_4 has a latency of just 1 unit (this may happen if dc_3 and dc_4 are geographically close to each other and far away from dc_1). For clarity of exposition, let us assume that these delays are constant. There are three updates, a, b and c. For simplicity, assume that the timestamp assigned to these updates is derived from an external source of real time, occurring at time $t = 2$, $t = 4$ and $t = 6$ respectively. Let's also assume that $b \rightsquigarrow c$ and that a is concurrent with both b and c. The reader will notice that there are three distinct serializations of these updates that respect causal order: abc, bac, and bca. Which serialization should be provided to dc_4?

In order to answer this question, we first need to discuss how the operation of SATURN's metadata dissemination service can negatively affect the performance of the system. For this, we introduce the following two concepts: *data readiness* and *dependency readiness*. Data readiness captures the ability of the system to provide the most recent updates to clients, as soon as its bulk-data transfer is completed. Dependency readiness captures the ability of the system to serve a request, because all of its causal dependencies (both real dependencies and false dependencies that are created as an artifact of the metadata compression) have been previously applied.

Considering data readiness alone, we would conclude that SATURN should deliver labels to remote datacenters as soon as possible. However, the reader may have noticed that there is no real advantage of delivering label a before instant $t = 12$, as the update can only be applied when the bulk-data transfer is completed. On the contrary, delivering label a very soon may create a false dependency that may affect the dependency readiness of other requests. Assume that SATURN opts to deliver to dc_4 the labels in the serial order abc. This is not only consistent with causality but also consistent with the real time occurrence of the updates. Unfortunately, this serialization creates a false dependency among update a and updates b and c, i.e., updates b and c need to be applied after update a as a result of the serialization procedure. This introduces unnecessary delays in the processing of the later updates: although update b and update c are delivered to dc_4 at times 5 and 7 respectively, they will have to wait until time 12 (while this was not strictly required by causality, given that a is concurrent with b and c). A more subtle consequence is that a correct but "inconvenient" serialization may increase the latency observed by clients. Assume that a client reads from dc_3 update b at time $t = 5$ and then migrates to dc_4 (to read some item that it is not replicated at dc_3). That client should be able to attach to dc_4 immediately, as by time $t = 5$ update b has been delivered to dc_4 and could be made visible. However, the false dependency introduced by the serialization above requires the client to wait until time $t = 12$ for the attachment to complete.

The example above shows that while trying to maximize data freshness by delivering labels not after the data-bulk transfer of its corresponding operations is completed, SATURN should avoid introducing false dependencies prematurely. A prematurely delivered label may unnecessarily delay the application of other remote operations, having a negative impact on the latency experienced by clients and increasing remote updates visibility latencies. Therefore, SATURN has to select a serialization per datacenter which does the best trade-off between these two aspects. In the example above, if a is only delivered at dc_4 after b and c, by selecting the serialization bca, clients migrating from dc_3 to dc_4 would not be affected by the long latency of the bulk-data transfer link from dc_1 to dc_4; and a, b and c will become visible at dc_4 as soon as the bulk-data transfer is completed.

3.2.2 Selecting the best serializations

In order to precisely define which is the best serialization that should be provided to a given datacenter, we first need to introduce some terminology.

Let u_i be a given update i performed at some origin datacenter dc_o, and l_i the label for that update. Let t_i be the real time at which the update was created at the origin datacenter dc_o. Let dc_r be some other datacenter that replicates the data item that has been updated. Let $\Delta(dc_o, dc_r)$ be the expected delay of the bulk-data transfer from the origin datacenter to the replica datacenter. For simplicity of notation we assume that $\Delta(dc_o, dc_s) = 0$ if datacenter s does not replicate the item, and therefore, it is not interested in receiving the update. The expected availability time for the update at dc_r would be $t_i + \Delta(dc_o, dc_r)$. Finally let $\mathcal{H}(u_i) = \{u_j, u_k, ...\}$ the set of past updates that are in the causal past of u_i.

Definition 2 (Optimal visibility time)**.** *We then define the* optimal visibility time*, denoted vt_i^r of an update i at some replica dc_r, as the earliest expected time at which that update can be applied to dc_r. The optimal visibility time of an update at a target datacenter dc_r is given by:*

$$vt_i^r = max(t_i + \Delta(dc_o, dc_r), \max_{u_x \in \mathcal{H}(u_i)} vt_x^r) \tag{3.2}$$

From the example above it is clear that if the label l_i is delivered at dc_r after vt_i^r data freshness may be compromised. If l_i is delivered at dc_r before vt_i^r, delays in other requests may be induced due to false dependencies and lack of dependency readiness. Thus, SATURN should—ideally—provide to each datacenter dc_r a serialization that allows each label l_i to be delivered exactly at vt_i^r on that datacenter.

3.2.3 Architecture of the metadata dissemination service

SATURN's metadata dissemination service is implemented by a set of metadata brokers, namely *serializers*, in charge of aggregating and propagating the streams of labels collected at each datacenter. We recall that our main goal is to provide to each datacenter a serialization of labels that is consistent with causality. This can be obtained by ensuring that serializers and datacenters are organized in a tree topology (with datacenters acting as leaves), connected with FIFO channels, and that serializers forward labels in the same order it receives them.

Let us illustrate its principles with the simplest example. Consider for instance a scenario with 3 datacenters dc_1, dc_2 and dc_3 connected to a single serializer S_1 in a star network. Consider a data item that is replicated in all datacenters and two causally dependent updates to that item, a and b ($a \leadsto b$), where a is performed at dc_1 and b is performed at dc_2 (both updates need to be applied at dc_3). In this scenario, the following sequence of events would be generated: a is applied at dc_1 and its label is propagated to S_1. In turn, S_1 propagates the label to dc_2 which, after receiving its payload (via the bulk-data transfer), applies update a locally. Following, at dc_2, some local client reads a and issues update b. The label associated with b is sent to S_1, that in turn will forward it to the other datacenters. Since serializers propagate labels preserving arrival order, datacenter dc_3 will necessarily receive a before b, as S_1 observes, independently of arbitrary network delays, a before b[1].

Although a star network, with a single server, will trivially satisfy causality, such a network may offer sub-optimal performance. In the previous section, we have seen that the metadata dissemination service must deliver a label to a datacenter approximately at the same time the associated bulk data is delivered; for this requirement to be met, the metadata path cannot be substantially longer than the bulk data path. In fact, even if labels are expected to be significantly smaller than data items (and therefore, can be propagated faster), their propagation is still impaired by the latency among the SATURN serializers and the datacenters. Consider again the example of Fig. 3.2: we want labels from dc_3 to reach dc_4 within 1 time unit, thus any servers on that metadata path must be located close to those datacenters. Similarly, we want labels from $dc_1$to reach dc_2 fast. These two requirements cannot be satisfied if a single server is used, as in most practical cases, a single server cannot be close to both geo-locations simultaneously.

To address the efficiency problem above we use multiple serializers distributed geographically. Note that the tree formed is shared by all datacenters, and labels are propagated along the shared tree using the source datacenter as the root (i.e., there is no central root for all datacenters). This ensures that we can establish fast metadata paths between datacenters that are close to each other and that replicate the same data.

Resorting to a network of cooperative serializers has another advantage: labels regarding a given item do not need to be propagated to branches of the tree that contain serializers connected to datacenters that do not replicate

[1] One can easily derive a correctness proof for any tree topology based on the idea that for any two causally related updates a and b ($a \leadsto b$) such that b was generated at dc_i (this implies that a was visible at dc_i before b was generated), the lowest common ancestor serializer between dc_i and any other datacenter interested in both updates, observes a label before b label.

that item. This fact enables genuine partial replication at the data service: datacenters will only receive labels that correspond to the data items replicated locally. Nevertheless, the metadata dissemination service itself is not genuine as defined by R. Guerraoui et al. [57] (a multicast implementation is genuine if only the process that sends the message and the processes that have to receive it are involved in the protocol required to deliver the message) as in order to propagate a label from the origin datacenter to a destination datacenter, this has to traverse a set of serializers, which may include serializers located at a third datacenter location. Nevertheless, the fact that the service is distributed prevents all serializers from processing all labels, contributing to the scalability of the system.

Finally, since we expect labels to be disseminated faster that their correspondent bulkier payloads, it may happen that labels become available for delivery before their optimal visibility time. In fact, in current systems, and for efficiency reasons, bulk data is not necessarily sent through the shortest path [62]. Thus, for optimal performance, SATURN may introduce artificial delays in the metadata propagation, as discussed in §3.3.

3.3 The configuration problem: finding the topology

As discussed in the previous section (§3.2.3), SATURN relies on a network of metadata brokers, organized in a tree topology, to disseminate the metadata among datacenters. Nevertheless, in order to optimize remote visibility latencies not every tree topology is valid. Thus, the quality of the serialization served by SATURN to each datacenter depends on how the metadata dissemination service is configured.

In this section, we present a configuration service that can be used to find a SATURN's configuration that allows to optimize remote visibility latencies given a deployment and certain application characteristics. This is a fundamental step that has to be done before deploying SATURN in order to ensure the well functioning of the service.

3.3.1 The configuration service

The configuration service is composed of two main components: the topology generator and the solver. The former iterates through multiple tree topologies

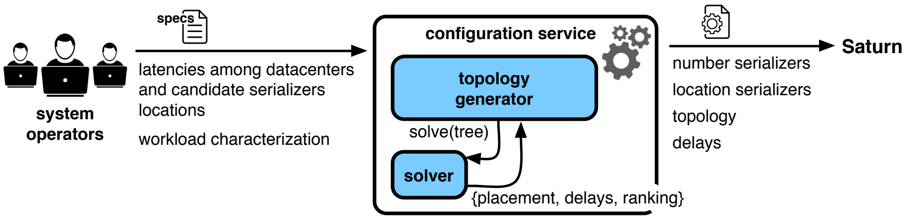


Figure 3.3 – The configuration service. It is composed of two subcomponents: the solver and the topology generator.

and relies on the solver component to determine the one that optimizes remote visibility latencies, given a deployment.

Figure 3.3 shows the interaction between these two components, the input that the service expects, and the output that it generates.

Inputs

The configuration service expects the following inputs (the first three items of the list correspond to the first input in Figure 3.3, the last item corresponds to the last input in the figure):

- The set V of datacenters that need to be connected (we denote $N = |V|$ the total number of datacenters).

- The latencies of the bulk data transfer service among these datacenters; lat_{ij} denotes the latency between datacenters i and j. Note that lat_{ij} and lat_{ji} are not necessarily equal.

- The set W of potential locations for placing serializers ($M = |W|$). Note that, in practice, when deploying SATURN, one has not complete freedom to select the geo-location of serializers. Instead, the list of potential locations for serializers is limited by the availability of suitable points-of-presence that results from business constraints. Since each datacenter is a natural potential serializer location, W is a superset of V and $M \geq N$. Let d_{ij} denote the latency between two serializer locations i and j.

- If available, a characterization of the workload generated by the applications using the data service to which SATURN is attached, e.g.; the distribution of clients requests among items and datacenters.

Outputs

The configuration service outputs a SATURN's configuration. This is defined by:

- The number of serializers to use and where to place them.
- A topology: how these serializers are connected, among each other and with datacenters.
- What delays (if any) should a serializer artificially add when propagating labels (in order to match the optimal visibility time).

3.3.2 Modelling the problem

Given a limited set of potential locations to place serializers and the constraint of having to organize them in a tree topology, it is unlikely (impossible in most cases) to match the optimal label propagation latency for every pair of datacenters. Therefore, the best we can aim when setting-up SATURN is to minimize the mismatch between the achievable label propagation latency and the optimal label propagation latency.

The optimal label propagation latency is determined by the expected arrival time of the data. It approximates the optimal visibility time (Equation 3.2) as follows. The latter defines the earliest expected time at which an update can be applied at a remote (target) datacenter. This time is determined by the time that it takes to propagate the update and its causal dependencies to the target datacenter. Given that our architecture (the tree topology of serializers) ensures that the label of an update is always serialized after its causal dependencies, the best we can do is to approximate that the arrival time of an update's label matches the arrival time of that update's payload.

Let us precisely model the optimization problem. Consider that the path for a given topology between two datacenters, i and j, denoted $P^M_{i,j}$ is composed by a set of serializers $P^M_{i,j} = \{S_k, ..., S_o\}$, where S_k connects to datacenter i and S_o connects to datacenter j. The latency of this path $\Delta^M(i, j)$ is defined by the latencies (d) between adjacent nodes in the path, plus any artificial delays that may be added at each step—δ_{ij} denotes the artificial delay added by serializer i when propagating metadata to serializer j., i.e.:

$$\Delta^M(i,j) = \sum_{S_k \in P^M_{i,j} \setminus \{S_o\}} (d_{k,k+1} + \delta_{k,k+1}) \tag{3.3}$$

and the mismatch between the resulting latency and the optimal label propagation latency is given by:

$$\textit{mismatch}_{i,j} = |\Delta^M(i,j) - \Delta(i,j)| \tag{3.4}$$

Finally, one can observe that in general, the distribution of client requests, among items and datacenters may not be uniform, i.e., some items and some datacenters may be more accessed than others. As a result, a mismatch that affects the data visibility of a highly accessed item may have a more negative effect on the user experience than a mismatch on a seldom accessed item. Therefore, in the scenario where it is possible to collect statistics regarding which items and datacenters are more used, it is possible to assign a *weight* $c_{i,j}$ to each metadata path $P^M_{i,j}$, that reflects the relative importance of that path for the business goals of the application. Using these weights, we can now define precisely an optimization criteria that should be followed when setting up the serializers topology:

Definition 3 (Weighted Minimal Mismatch)**.** *The configuration that better approximates the optimal visibility time (Equation 3.2) for data updates, considering the relative relevance of each type of update, is the one that minimizes the weighted global mismatch, defined as:*

$$min \sum_{\forall i,j \in V} c_{i,j} \cdot \textit{mismatch}_{i,j} \tag{3.5}$$

Note that we are summing the mismatches as we want to optimize the average remote visibility latency. If one wants to optimize for the worst-case, the weighted minimal mismatch should rather multiply mismatches.

Algorithm 3.1 Find the best configuration.

```
 1: function FIND_CONFIGURATION($V$, $Threshold$)
 2:     $\langle First, Second \rangle \leftarrow$ PICK_TWO($V$)
 3:     InitTree = rooted tree with $First$ and $Second$ as leaves
 4:     $Trees \leftarrow \{InitTree\}$
 5:     $V \leftarrow V \setminus \{First, Second\}$
 6:     while $V \neq \emptyset$ do
 7:         $NextDC \leftarrow$ HEAD($V$)
 8:         $NewTrees \leftarrow \emptyset$                              ▷ ordered set
 9:         for all $Tree \in Trees$ do
10:             $NTree \leftarrow$ NEW_ROOTED($NextDC$, $Tree$)
11:             $NTree.ranking \leftarrow$ SOLVE($NTree$)
12:             $NewTrees \leftarrow NewTrees \cup \{NTree\}$
13:             for all $Edge \in Tree$ do
14:                 $NTree \leftarrow$ NEW_TREE($NextDC$, $Tree$, $Edge$)
15:                 $NTree.ranking \leftarrow$ SOLVE($NTree$)
16:                 $NewTrees \leftarrow NewTrees \cup \{NTree\}$
17:         $V \leftarrow V \setminus \{NextDC\}$
18:         $Trees \leftarrow$ FILTER($Threshold$, $NewTrees$)
19:     return HEAD($Trees$)
```

3.3.3 Configuration generator

The problem of finding a configuration that minimizes the Weighted Minimal Mismatch criteria, among all possible configurations that satisfy the constraints of the problem, is NP-hard.[2] Therefore, we have designed a heuristic that approximates the optimal solution using a constraint solver as a building block. We have modeled the minimization problem captured by Definition 3.5 as a constraint problem that for a given tree, the solver finds the optimal location of serializers (for a given set of possible location candidates) and the optimal (if any) propagation delays.

The proposed algorithm, depicted in Alg. 3.1, works as follows. Iteratively, starting with a full binary tree with only two leaves (Alg. 3.1, line 3), the algorithm generates all possible isomorphic classes of full binary trees with N labeled leaves (i.e., datacenters). The algorithm adds one labeled leaf (data-

[2] A reduction from the Steiner tree problem [63] can be used to prove this.

center) at each iteration until the number of leaves is equal to the total number of datacenters. For a given full binary tree T of f leaves, there exist $2 * f - 1$ isomorphic classes of full binary trees with $f + 1$ leaves. One can obtain a new isomorphic class by either inserting a new internal node within an edge of T from which the new leaf hangs (Alg. 3.1, line 14), or by creating a new root from which the new leaf and T hang (Alg. 3.1, line 10). We could iterate until generating all possible trees of N leaves. Nevertheless, in order to avoid a combinatorial explosion (for nine datacenters there would already be 2,027,025 possible trees), the algorithm selects at each iteration the most promising trees and discards the rest. In order to rank the trees at each iteration, we use the constraint solver. Therefore, given a totally ordered list of ranked trees, if the difference between the rankings of two consecutive trees T_1 and T_2 is greater than a given threshold, T_2 and all following trees are discarded (Alg. 3.1, line 18). At the last iteration, among all trees with N leaves, we pick the one that produces the smallest global mismatch from the optimal visibility times by relying on the constraint solver.

Note that Algorithm 3.1 always returns a binary tree. Nevertheless, SATURN does not require the tree to be binary. One can easily fuse two serializers into one if both are directly connected, placed in the same location, and the artificial propagation delays among them are zero. Any of these fusions would cause the tree to change its shape without reducing its effectiveness.

3.4 Datacenter operation: unobtrusive ordering

The design of SATURN is decoupled from the implementation details of each datacenter. In this way, SATURN can be cast to operate with different geo-replicated data services. Naturally, SATURN needs to interact with each datacenter, and the datacenter implementation must allow attaching the hooks that provide the functionality required by SATURN: the generation of labels associated with each update, the propagation of remote labels to the local storage, and the capability of exporting a single serial stream—consistent with causality—of labels, as if it were a logically centralized store, even when each datacenter's storage service spans multiple servers.

SATURN could employ well-known solutions to meet these goals. For instance, systems with similar requirements [9, 103] resort to a logically centralized service (one at each datacenter) to address the problem of generating a

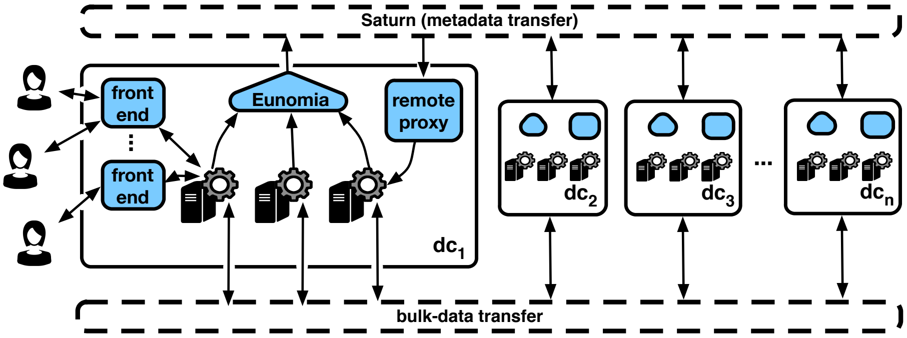


Figure 3.4 – Datacenter operation.

single causal serialization of local updates. This service, namely *sequencer*, totally orders local updates by assigning increasing sequence numbers. In order to trivially ensure that the total order derived from the sequence is consistent with causality, the coordination between storage servers, which handle clients requests, and the sequencer is done synchronously before returning to clients.

Unfortunately, sequencers are known to limit datacenter's concurrency and increase operation latencies. In order to use SATURN to the best advantage, we integrate Eunomia, a recent metadata serialization service [59]. Eunomia is conceived to replace sequencers as building blocks in weakly consistent geo-replicated storage systems. Unlike traditional sequencers, Eunomia lets local client operations execute without synchronous coordination. Then, in the background, Eunomia establishes a serialization of all updates occurring in the local datacenter in an order consistent with causality, based on timestamps generated locally by the individual servers that compose the datacenter. We refer to this process as *site stabilization procedure*. Thus, Eunomia is capable of abstracting the internal complexity of a multi-server datacenter without limiting the concurrency.

In order to give full support to the functionality required by SATURN, together with the Eunomia service (the fundamental piece), we have designed three more subcomponents, as illustrated by Figure 3.4. Note that in the exposition, we assume that the key-space is divided into N non-overlapping *partitions* distributed among the storage servers composing the datacenter. Table 3.1 provides a summary of the notation used in the algorithms. These subcomponents have the following responsibilities:

M	Number of datacenters
N	Number of partitions
Label_c	Client c label
p_n^m	Partition n at datacenter m
g_n^m	Gear attached to p_n^m
Clock_n^m	Current physical time at p_n^m
MaxTs_n^m	Greatest timestamp ever assigned by g_n^m
l_a	Label assigned to operation a
Labels_m	Set of unstable labels at the Eunomia service of datacenter m
$\mathit{VectorTime}_m$	Vector with an entry per gear at the Eunomia service of datacenter m

Table 3.1 – Notation used in the protocol description.

- Stateless *frontends* shield clients from the details of the internal operation of the datacenter (how the key-space is partitioned, how many replicas of each item are kept, etc). Frontends intercept client requests before they are processed by partitions— therefore storage servers. They have two roles: (i) to ensure that clients observe a causally consistent snapshot of the datastore; and (ii) to forward updates to responsible partitions and later return the labels assigned to the operations to clients.

- *Gears* are responsible for generating labels, and propagating the data and metadata associated to each update. A gear is associated to each partition; it intercepts update requests (coming from a frontend) and, once it is made persistent, ships the update to remote datacenters via the bulk-data transfer service. Furthermore, it forwards locally generated labels to the *Eunomia* service.

- The Eunomia service is a logically centralized component that collects all the labels associated with the updates performed in the local datacenter and forwards them to the metadata transfer service of Saturn, in a serial order that is compliant with causality.

- The *remote proxy* applies remote operations in causal order. For this, it relies on the order proposed by Saturn and on the label timestamp order as explained in §3.4.3.

Algorithm 3.2 Operations at frontend q of datacenter m

```
▷ Handles the attachement of a client to a datacenter
function ATTACH($Label_c$)
    $g_n^k \leftarrow Label_c.src$                    ▷ gear $n$ of dc $k$
    if $k == m$ then                                 ▷ $Label_c$ was locally generated
        return ok
    else                                             ▷ $Label_c$ was remotely generated
        WAIT_FOR_STABILIZATION($Label_c$)
        return ok
▷ Forwards an update request to the responsible storage server. Note the
operation is intercepted by the gear attached to it.
function UPDATE(Key, Value, $Label_c$)
    server ← RESPONSIBLE(Key)
    send UPDATE(Key, Value, $Label_c$) to server
    receive Label from server
    return Label
▷ Forwards a read request to the responsible storage server. Note the operation is intercepted by the gear attached to it.
function READ(Key)
    server ← RESPONSIBLE(Key)
    send READ(Key) to server
    receive ⟨Value, Label⟩ from server
    return ⟨Value, Label⟩
▷ Forwards a migration request to any gear
function MIGRATE(TargetDC, $Label_c$)
    $g_n^m \leftarrow$ GEAR(random_key)              ▷ gear $n$ of local dc $m$
    send MIGRATION(TargetDC, $Label_c$) to $g_n^m$
    receive Label from $g_n^m$
    return Label
```

3.4.1 Client interaction

Clients interact directly with the frontends through the client library. A frontend exports four operations: *attach*, *read*, *write*, and *migrate*. The latter is described in §3.4.4. Algorithms 3.2 and 3.3 describe how events are handled by frontends and gears, the two key subcomponents to understand the generation of labels and the interaction with clients.

Attach. Before issuing an update, read, or migration request, a client c is required to *attach* to a datacenter. Being attached to a datacenter m signifies that client c causal past is visible in m and, therefore, c can safely interact with m without violating causality. A client attaches to a datacenter by providing the latest label it has observed (stored in the client's library). The frontend waits until that label is *causally stable*, i.e., until it is sure that all updates that are in the causal past of the client have been locally applied. When this condition is met, it replies back to the client. From this point on, the client may issue requests. The condition that indicates the stability of the presented label depends on the type and source of the label. If the label was created on the same datacenter, the fronted may return immediately (Alg. 3.2, line 4). If the label was created on a remote datacenter (Alg. 3.2, line 5), and it is of type *migration* (§3.4.4 discusses the generation of this type of labels), it waits until SATURN delivers that label and all previous labels have been applied (in the order provided by SATURN). Finally, if the label was created on a remote datacenter, and it is of type *update*, the frontend waits until an update with an equal or greater timestamp has been applied from every remote datacenter.

Update. A client c's update request is first intercepted by the client library, then tagged with the label that captures the client's causal past (*Label*$_c$), and forwarded to any local frontend. The frontend forwards the update operation to the local responsible partition (Alg. 3.2, line 9). This operation is intercepted by the gear attached to that partition g_n^m. The gear first generates a new label for that update (Alg. 3.3, line 2). Then, the value and its associated label are persistently written to the store. Subsequently, the update's payload —tagged with its corresponding label— is sent to the remote replicas (Alg. 3.3, lines 5–6), and the label is handed to the local Eunomia service (Alg. 3.3, line 7). The new label is then returned to the frontend that forwards it to the client library. Finally, the new label replaces the client's old label, capturing the update operation in the client's causal past.

Read. A read request on a data item *Key* is handled by a frontend by forwarding the request to the local responsible partition (Alg. 3.2, line 14). The request is intercepted by the gear g_n^m attached to the partition that returns the associated value and label (Alg. 3.3, line 11). The label associated is the one assigned by g_n^m to the update operation that generated the current version. If the label associated with the value is greater than the label stored in the client's library (*Label*$_c$), the library will replace the old label by the new one, including

Algorithm 3.3 Operations at gear n of datacenter m (g_n^m)

```
    ▷ Updates the local store and propagates to remote datacenters
function UPDATE($Key$, $Value$, $Label_c$)
    $Ts \leftarrow$ GENERATE_TSTAMP($Label_c.ts$)
    $Label \leftarrow \langle update, Ts, g_n^m, Key\rangle$
    $ok \leftarrow$ KV_PUT($Key$, $\langle Value, Label\rangle$)
    for all $k \in$ REPLICAS($Key$) $\setminus\{m\}$ do
        send NEW_PAYLOAD($Label$, $Value$) to $k$
    send NEW_LABEL($Label$) to $Eunomia$
    return $Label$ 5
    ▷ Reads the most recent version of $Key$ from the local store
function READ($Key$)
    $\langle Value, Label\rangle \leftarrow$ KV_GET($Key$)
    return $\langle Value, Label\rangle$
    ▷ Generates a migration label
function MIGRATION($TargetDC$, $Label_c$)
    $Ts \leftarrow$ GENERATE_TSTAMP($Label_c.ts$)
    $Label \leftarrow \langle migration, Ts, g_n^m, TargetDC\rangle$
    send NEW_LABEL($Label$) to $Eunomia$
    return $Label$
    ▷ Sends a heartbeat to Eunomia
function HEARTBEAT                                  ▷ Every $\delta$ time
    if no update for $\delta$ time then
        $Ts \leftarrow$ GENERATE_TSTAMP(0)
        send HEARTBEAT($g_n^m$, $Ts$) to $Eunomia$
    ▷ Generates a timestamp larger than $Min$
function GENERATE_TSTAMP($Min$)
    $MaxTs_n^m \leftarrow$ MAX($Clock_n^m$,$MaxTs_n^m + 1$,$Min$+1)
    return $MaxTs_n^m$
```

thus the retrieved update into the client's causal past.

3.4.2 Integration of the Eunomia service

Eunomia requires that the labels, assigned by gears to client operations, satisfy the following two properties.

Algorithm 3.4 Operations at Eunomia of datacenter m

```
▷ Queues a new label
function NEW_LABEL($l_a$)
    $Labels_m \leftarrow Labels_m \cup l_a$
    $VectorTime[g_n^m] \leftarrow l_a.ts$
▷ Handles the reception of a heartbeat coming from a local gear
function HEARTBEAT($g_n^m$, $Ts$)
    $VectorTime[g_n^m] \leftarrow Ts$
▷ Computes and processes stable labels
function PROCESS_STABLE                                    ▷ Every $\theta$ time
    $STime \leftarrow$ MIN($VectorTime_m$)
    $SLabels \leftarrow$ FIND_STABLE($Labels_m$, $STime$)
    PROCESS($SLabels$)
    $Labels_m \leftarrow Labels_m \setminus SLabels$
```

Property 3 (Site-causality). *Given two operations a and b, both local to the same datacenter, if a causally depends on b, then the timestamp assigned to l_b ($l_b.ts$) is strictly greater than $l_a.ts$.*

Property 4 (Gear-monotonicity). *For two labels l_a and l_b received by Eunomia coming from the same gear g_n^m, if l_a is received before l_b then $l_b.ts$ is strictly greater than $l_a.ts$.*

These two properties imply that labels are causally ordered across all gears and that once Eunomia receives a label coming from a gear g_n^m, no label with a smaller timestamp will ever be received from g_n^m.

Label generation. In addition to the restrictions imposed by Eunomia to the generation of labels, SATURN requires labels to be unique and their timestamp order to respect causality, not only within a datacenter (as the Property 3 of Eunomia) but across datacenters. Our design ensures that labels guarantee these four properties

First, the gear-monotonicity property is guaranteed by ensuring that each gear generates monotonically increasing timestamps. As the combination of a label's fields timestamp and source make it unique, this also guarantees the uniqueness property. Second, ensuring a causal order among labels requires that, when generating a label for an update issued by some client c, the timestamp assigned to that label is strictly greater than all the labels that c has previously observed. In SATURN, each client's causal past is represented by the greatest label the client has observed when interacting with the system (*Label$_c$*). Since clients are not tied to a specific frontend, this label has to be stored in the client's library and be piggybacked with client requests. Therefore, upon an update request, gears only need to guarantee that the timestamp of the label being generated (Alg. 3.3, lines 2 and 13) is greater than the client's label timestamp. Note that to ensure correctness, client libraries have to update client's labels (as described in §3.4.1) when they interact with SATURN frontends to ensure that all operations observed by the client are included in the client's causal past. Third, given that the Eunomia's site-causality property is strictly weaker than ensuring the global causal order of labels required by SATURN, the above method also ensures the site-causality property of Eunomia.

Gears rely on hybrid clocks [64] to generate labels' timestamps, which combine logical and physical time (Alg. 3.3, lines 21–23). Although we could simply use logical clocks and still be correct, the rate at which clocks from different partitions progress would depend on the rate in which partitions receive update requests. This may cause Eunomia services to process local updates in a slower pace and thus increase remote visibility latencies, as the stable time is set to the smallest timestamp received among all partitions. Differently, physical clocks naturally progress at similar rates independently of the workload characterization. This fact—previously exploited by [49, 7]—makes stabilization procedures resilient to skewed load distribution. Unfortunately, physical clocks do not progress exactly at the same rate, forcing protocols to wait for clocks to catch up in some situations in order to ensure correctness [48, 49, 7, 50]. The logical part of the hybrid clock makes the protocol resilient to clock skew by avoiding artificial delays due to clock synchronization uncertainties [64]. Briefly, if a gear g_n^m receives an update request with *Label$_c$.ts*$>$ *Clock$_n^m$*, instead of waiting until *Clock$_n^m$* $>$ *Label$_c$.ts* to ensure correctness (monotonicity and causal order among labels), the logical part of the hybrid clock (*MaxTs$_n^m$*) is moved forward. Then, when g_n^m receives an update from any client, if the physical part *Clock$_n^m$* is still behind the logical (*MaxTs$_n^m$*), the update is tagged with *MaxTs$_n^m$* $+ 1$.

Stabilization Procedure. When Eunomia receives a label from a given gear g_n^m, it adds it to the set of non-stable labels *Labels$_m$* and updates the g_n^m entry in the *VectorTime$_m$* vector with the label's timestamp (Alg. 3.4, lines 2–3). A label l_a is considered *stable* when one is sure that no label with a smaller timestamp will be received from any gear (i.e., when Eunomia is aware of all labels with timestamp $l_a.ts$ or smaller). Periodically, Eunomia computes the value of the maximum stable timestamp (*STime*), which is computed as the minimum of the *VectorTime*$_m$ vector (Alg. 3.4, line 7). Property 4 implies that no partition will ever timestamp an update with an equal or smaller timestamp than *STime*. Thus, Eunomia can confidently serialize all operations tagged with a timestamp smaller than or equal to *STime* (Alg. 3.4, line 8). Eunomia serializes them in timestamp order, which is consistent with causality (Property 3), and then hands them to SATURN's metadata dissemination service (Alg. 3.4, line 9). Note that non-causally related updates coming from different partitions may have been timestamped with the same value. In this case, operations are concurrent. Eunomia processes concurrent labels according to Property 2.

Heartbeats. If a gear g_n^m does not receive a client update request for a fixed period of time (δ), it will send a heartbeat including its current time to Eunomia (Alg. 3.3, lines 17–20). Thus, even if a gear receives updates at a slower pace than others, it will not slow down the processing of other gears updates at Eunomia. When Eunomia receives a heartbeat from g_n^m, it simply updates its entry in the *VectorTime*$_m$ vector (Alg. 3.4, line 5).

3.4.3 Handling remote operations

The remote proxy collects updates generated at remote datacenters and applies them locally, in causal order. A remote proxy has at its disposal two sources of information to derive an order that does not violate causality: the timestamp order of the labels associated with the updates (that defines one valid serialization order), and the label serialization provided by SATURN, which also respects causal order (although it may differ from the timestamp order). As we will show in the evaluation section, SATURN can establish a valid remote update serialization order significantly faster than what is feasible when just relying on timestamp values Therefore, unless there is an outage on the metadata service, the serialization provided by SATURN is used to apply remote updates, and timestamp order is used as a fallback. Moreover, these two causal serialization can also be leveraged by the remote proxy to infer that two remote operations a and b are concurrent. Specifically, this can be inferred if SATURN delivers their corresponding labels (l_a and l_b) in an order that does not match timestamp order. Since both serializations are consistent with causality, the fact that they order two operations differently means that both operations are concurrent, otherwise at least one of the serializations would not be consistent with causality. This can be exploited by remote proxies to increase the parallelism when handling remote operations. By using this optimization, remote proxies can issue multiple remote operations in parallel to the local datacenter.

3.4.4 Client migration support

Applications may require clients to switch between datacenters, especially under partial replication, in order to read data that is not replicated at the client's preferred datacenter. In order to speedup the attachment at remote datacenters, SATURN (frontends specifically) expose a migration operation. When a client c –attached to a datacenter m– wants to switch to a remote datacenter, a migration request is sent to any local frontend f_q^m, specifying the target datacenter (*TargetDC*) and the client's causal past $Label_c$. f_q^m forwards the request to any local gear. The receiving gear g_n^m generates a new label and hands it to the local Eunomia service (Alg. 3.3, lines 13–15). g_n^m guarantees that the generated label is greater than $Label_c$ to ensure that Eunomia hands it to SATURN after any update operation that c has potentially observed. In turn, SATURN will deliver the label in causal order to the target datacenter, which will immediately allow client c to attach to it, as c's causal past is ensured to be visible locally.

The procedure above, in particular the creation of a migration label, is not strictly required to support client migration, but aims at optimizing this process. In fact, when attaching to a new datacenter, the client could just present the update label that captures its causal past. However, the stabilization procedure could force the client to wait until an update from each remote datacenter with a timestamp equal or greater than the timestamp of the client's label has been applied locally. The creation of an explicit migration label prevents the client from waiting for a potentially large number of false dependencies.

3.5 Fault-tolerance

In this section, we discuss the fault-tolerant mechanisms of the components in charge of the propagation of labels among datacenters: the metadata dissemination service and the intra-datacenter metadata serialization service. We disregard others failures in datacenters—such as partitions, as the problem of making data services fault-tolerant has been widely studied and is orthogonal to the contributions of this thesis.

3.5.1 Replicating Eunomia

In §3.4, we have described how the Eunomia metadata serialization service integrates with the metadata dissemination service and the rest of intra-datacenter components. Naturally, as any other service in a datacenter, Eunomia must be made fault-tolerant. In fact, if Eunomia fails, the site stabilization procedure stops, and thus, local updates can no longer be propagated to other datacenters. Although these mechanisms are described in detail in [59], for self-containment, we include here a brief description of the techniques used to avoid such limitation. Our description of a fault-tolerant Eunomia service follows closely the one presented in [59].

In the fault-tolerant version, Eunomia is composed by a set of *Replicas*. Algorithm 3.5 shows the behaviour of a replica e_f of the fault-tolerant Eunomia service. We assume the initial set of Eunomia replicas is common knowledge: every replica knows every other replica and every gear knows the full set of replicas. Gears send labels and heartbeats (Alg. 3.3, lines 7, 15 and 20) to the whole set of Eunomia replicas. The correctness of the algorithm requires the communication between gears and Eunomia replicas to satisfy the *prefix-property* [98]: an Eunomia replica r_f that holds a label l_b originating at g_n^m also holds any other label l_a originating at g_n^m such that $l_a.ts < l_b.ts$. This property can be ensured with inexpensive protocols that offer only *at-least-once delivery*. Stronger properties, such as inter-gear order or exactly-once delivery are not required to enforce the prefix-property. Our implementation achieves the prefix-property by having each gear keeping track of the latest timestamp acknowledged by each of the Eunomia replicas in a vector denoted as Ack_n. Thus, to each Eunomia replica e_f, a gear g_n^m sends not only the lastest label but the set of labels including all labels l_a such that $l_a.ts > \mathit{Ack}_n^m[f]$. Upon receiving a new batch of labels *Batch* (Alg. 3.5, lines 1–5), e_f process it—in timestamp order—filtering out those labels already seen, and updating both Labels_f and $\mathit{VectorTime}_f$ accordingly with the timestamps of the unseen labels. After processing *Batch*, e_f acknowledges g_n^m including the greatest timestamp observed from labels originating at g_n^m ($\mathit{VectorTime}_f[g_n^m]$). Although this algorithm adds redundancy as some labels are sent multiple times, it is resilient to message loss and unordered delivery.

Algorithm 3.5 Operations at Eunomia replica e_f

```
     ▷ Processes a new batch by queueing new labels
 1: function NEW_BATCH($Batch$, $g_n^m$)
 2:     for all $l_a \in Batch, VectorTime_f[g_n^m] < l_a.ts$ do
 3:         $VectorTime_f[g_n^m] \leftarrow l_a.ts$
 4:         $Labels_f \leftarrow Labels_f \cup l_a$
 5:     send ACK($VectorTime_f[g_n^m]$) to $g_n^m$
     ▷ Computes and processes stable labels
 6: function PROCESS_STABLE                                ▷ Every $\theta$ time
 7:     if $Leader_f == e_f$ then
 8:         $STime \leftarrow$ MIN($VectorTime_f$)
 9:         $SLabels \leftarrow$ FIND_STABLE($Labels_f$, $STime$)
10:         PROCESS($SLabels$)
11:         $Labels_f \leftarrow Labels_f \setminus SLabels$
12:         send STABLE($STime$) to $Replicas_f \setminus \{e_f\}$
     ▷ Discards already stable labels
13: function STABLE($STime$)
14:     $SLabels \leftarrow$ FIND_STABLE($Labels_f$, $STime$)
15:     $Labels_f \leftarrow Labels_f \setminus SLabels$
16:     for all $g_n^m \in VectorTime_f$ do
17:         $VectorTime_f[g_n^m] \leftarrow$ MAX($VectorTime_f[g_n^m]$, $STime$)
     ▷ Sets the new leader
18: function NEW_LEADER($e_g$)
19:     $Leader_f \leftarrow e_g$
```

In addition, to avoid unnecessary redundancy when exchanging metadata among datacenters, a leader replica is elected to propagate this information. The existence of a unique leader is not required for the correctness of the algorithm; it is simply a mechanism to save network resources. Thus, any leader election protocol designed for asynchronous systems (such as Ω [38]) can be plugged into our implementation. A change in the leadership is notified to a replica e_f through the NEW_LEADER function (Alg. 3.5, line 19). The notion of a leader is used to optimize the service's operation as follows. When the PROCESS_STABLE event is triggered, only the leader replica computes the new stable time and processes stable labels (Alg. 3.5, lines 7–10). Then, once the labels have been processed, the leader sends the recently computed *STime* to the remaining replicas (Alg. 3.5, line 12). When a replica e_f receives the new stable time, it removes the labels already known to be stable from its pending set of labels, since it is certain that those operations have been already processed and sent to metadata dissemination service of SATURN (Alg. 3.5, lines 14–15).

3.5.2 Failures in label propagation

Although SATURN's metadata service is instrumental to improve the global system performance (in particular, to speedup update visibility and client migration), it is never an impairment to preserve data availability. The fact that the global total order of labels defined by timestamps respects causality makes SATURN robust to failures. Thus, even if the metadata service suffers a transient outage, and stops delivering labels, updates can be still applied based on the timestamp order (we recall that labels are also piggybacked in the updates delivered by the bulk-data service).

A transient outage may be caused by a serializer failure or a network partition among serializers. Both situations lead to a disconnection in the serializers tree topology, possibly preventing the metadata service from delivering each label to all interested datacenters. Failures in serializers can be tolerated using standard replication techniques. Our current implementation assumes a fail-stop fault model [86], as serializers are made resilient to failures by replicating them using chain replication [100]. Nevertheless, SATURN's design does not preclude the use of other techniques [87, 37] in order to weaken the fault assumptions that we have made when building the current prototype. Connectivity problems in the tree may be solved by switching to a different tree, using the online reconfiguration procedure described next.

3.6 Adaptability

Configuring SATURN is an offline procedure performed before the system starts operating. Substantial changes in the workload characterization may require changes in the serializers tree topology. A change to a new tree may be also required if connectivity issues affect the current tree (backup trees may be pre-computed to speedup the reconfiguration).

In this last section of the chapter, we discuss two on-line reconfiguration protocols that enable SATURN to switch among configurations: a fast reconfiguration that relies on the old configuration to accelerate the transition, and a slower reconfiguration that must be used when the metadata dissemination service in its current configuration is unusable.

3.6.1 Assisted (fast) reconfiguration

We have implemented a simple mechanism to switch among configurations without interrupting SATURN's operation. Let C_1 denote the configuration currently being used. Let C_2 denote the tree configuration to which we have decided to switch. SATURN switches configurations as follows:

- All datacenters input a special label, namely *epoch* change, in the system through the C_1 tree.
- At each datacenter, labels produced after the epoch change label are sent via the C_2 tree.
- A datacenter can start applying labels arriving from the C_2 tree as soon as it has received the *epoch* change label for every datacenter and all previously received labels delivered by the C_1 tree have been applied locally.
- During the transition phase, labels delivered by the C_2 tree are buffered until the *epoch* change is completed.

This mechanism provides fast reconfigurations, namely, in the order of the largest latency among the metadata paths in C_1 (in our experiments, always less than *200ms*).

3.6.2 Unassisted (slower) reconfiguration

When reconfiguring because C_1 has failed, or if C_1 breaks during the reconfiguration, the following (slower) switching protocol is used:

- During the transition phase, updates are delivered in timestamp order and labels delivered by the C_2 tree are buffered.

- A datacenter can start applying labels arriving from the C_2 tree as soon as the update associated with the first label delivered by C_2 is stable in timestamp order.

In this case, the reconfiguration time is bounded by the time it takes to stabilize updates by timestamp order.

Chapter 4

Evaluation

In this chapter, we present the evaluation of SATURN, our prototype. We evaluate SATURN in Amazon EC2. In order to compare SATURN, a metadata service, to state-of-the-art solutions, which are data services, we attach SATURN to a data service that only guarantees eventual delivery: eventually, every update operation is received by each of the datacenters that replicate the data item updated by the operation. This data service exhibits no performance penalty due to consistency management. We also use it, detached from SATURN, as baseline to quantify the performance overhead produced by our techniques.

The main result of the evaluation is the experimental demonstration that the techniques integrated in SATURN are valid to build causally consistent geo-replicated systems that optimize both throughput and remote visibility latencies simultaneously. Our experiments show that upgrading our baseline to causal consistency only produces a 2% of throughput penalty and 11.7ms of extra remote visibility latency on average in both full and partial geo-replicated settings. Also, we show that our techniques compare favorably to previous state-of-the-art solutions: SATURN exhibits significant improvements in throughput (38.3%) compared to solutions that favor remote visibility latency such as Cure [7]; while exhibiting significantly lower remote visibility latency (76.9ms less on average) compared to solutions that favor high throughput such as GentleRain [49].

The chapter also presents a set of experiments that investigate few internal aspects of the features included in SATURN. In §4.4.1, we experimentally demonstrate that the choice of relying on a set of metadata brokers organized in a tree topology is key to optimize remote visibility latencies. In §4.4.2, we show the benefits of genuine partial replication by comparing SATURN to non-genuine solutions. Finally, in §4.4.3, we study the impact of latency variability by artificially injecting extra delays between datacenters.

4.1 Goals

Our primary goal is to determine if, unlike previous work (see §4.5.1), a data service attached to SATURN can simultaneously optimize throughput and remote update visibility latency under both full and partial geo-replication. For this, we run SATURN and other relevant competing solutions under:

(i) Synthetic workloads that allow us to explore how the different parameters that characterize a workload impact the performance (§4.5); and

(ii) A benchmark based on the Facebook's dataset and access patterns, to obtain an assessment of SATURN under complex realistic workloads (§4.5.4).

Secondarily, we evaluate a set of other characteristics of SATURN. First, we experiment with alternative architectures of the metadata dissemination service to better understand its impact on remote update visibility (§4.4.1) and the importance of genuine partial replication (§4.4.2). We then study the impact of latency variability in SATURN (§4.4.3).

In order to compare SATURN with other solutions from the state-of-the-art (data services), we attached SATURN to an eventually consistent geo-replicated data service we have built. This service ensures eventual delivery: eventually, every update operation is received by each of the datacenters that replicate the data item updated by the operation. Replication between datacenters is done asynchronously. Throughout the evaluation, we use this data service as the *baseline*, as it adds no overheads due to consistency management (remote operations are not delivered in any specific order at each datacenter), to better understand the overheads introduced by SATURN. Note that this baseline represents a throughput upper-bound and a latency lower-bound. Thus, when we refer to the optimal visibility latency throughout the experiments, we are referring to the latencies provided by the eventually consistent system.

	N. California	**Oregon**	**Ireland**	**Frankfurt**	**Tokyo**	**Sydney**
N. Virginia	*37 ms*	*49 ms*	*41 ms*	*45 ms*	*73 ms*	*115 ms*
N. California	-	*10 ms*	*74 ms*	*84 ms*	*52 ms*	*79 ms*
Oregon	-	-	*69 ms*	*79 ms*	*45 ms*	*81 ms*
Ireland	-	-	-	*10 ms*	*107 ms*	*154 ms*
Frankfurt	-	-	-	-	*118 ms*	*161 ms*
Tokyo	-	-	-	-	-	*52 ms*

Table 4.1 – Average latencies (half round-trip-time) among Amazon EC2 regions

4.2 Implementation

Our SATURN prototype implements all functionality described in §3. It has been built using the Erlang/OTP programming language. To balance the load among frontends at each datacenter, we use Riak Core [23], an open source distribution platform.

Eunomia internally uses a *red-black tree* [58], a self-balancing binary search tree optimized for insertions and deletions, which guarantees logarithmic search, insert and delete cost, and linear in-order traversal cost. Note that for Eunomia to work, we need to store a potentially large number of labels, coming from all the gears composing a datacenter, and periodically traverse them in timestamp order when a new stable time is computed. In our case, the *red-black tree* turned out to be more efficient than other self-balancing binary search trees such as AVL trees [4]: red-black trees are more efficient than AVL trees in add/delete intensive tasks, the type of task demanded by our instantiation of Eunomia.

The SATURN's configuration service is implemented in Scala. The solver, which models the optimization problem defined by Definition 3.5 and it is used by Algorithm 3.1, is implemented using OscaR [96], a Scala toolkit for solving Operations Research problems. The solver uses a depth-first search algorithm to find the optimal solution for a given tree.

4.3 Setup

We use Amazon EC2 `m4.large` instances running Ubuntu 12.04 in our experiments. Each instance has two virtual CPU cores, and 8 GB of memory. We use seven different regions in our experiments. Table 4.1 lists the average latencies we measured among regions. Our experiments simulate one datacenter per region. Clients are co-located with their preferred datacenter in separate machines. Each client machine runs its own instance of a custom version of Basho Bench [22], a load-generator and benchmarking tool. Each client eagerly sends requests to its preferred datacenter with zero thinking time: a client issues her next request as soon as she gets the response on her previous request. We deploy as many clients as necessary in order to reach the system's maximum capacity, without overloading it. Each experiment runs for more than 5 minutes. In our results, the first and the last minute of each experiment are ignored to avoid experimental artifacts. We measure the visibility latencies of remote update operations by storing the physical time at the origin datacenter when the update is applied locally, and subtracting it from the physical time at the destination datacenter when the update becomes visible. To reduce the errors due to clock skew, physical clocks are synchronized using the NTP protocol [77] before each experiment, making the remaining clock skew negligible in comparison to inter-datacenter travel time.

4.4 Evaluating the internals of Saturn

In this section, we experiment with a few aspects of SATURN.

In §4.4.1, we compare the architecture of the SATURN's metadata dissemination service (a multi-serializer architecture) to two alternative architectures: a centralized version in which the architecture is composed of a single serializer; and a variant that completely dispenses with serializers in which the metadata is not necessarily delivered in causal order at each datacenter. Our experiment shows that our chosen architecture compares favorably to both alternatives.

In §4.4.2, we show the benefits of genuine partial replication by comparing SATURN to a non-genuine solution. The experiment shows how genuineness have a positive impact on remote visibility latencies under partial replication.

Finally, in §4.4.3, we study the impact of latency variability by artificially injecting extra delays among datacenters. We show that, unless a large latency variability is experienced, SATURN does not need to be reconfigured. In our experiment, only when the latency between two datacenters is 5× the original, reconfiguring SATURN is worthy.

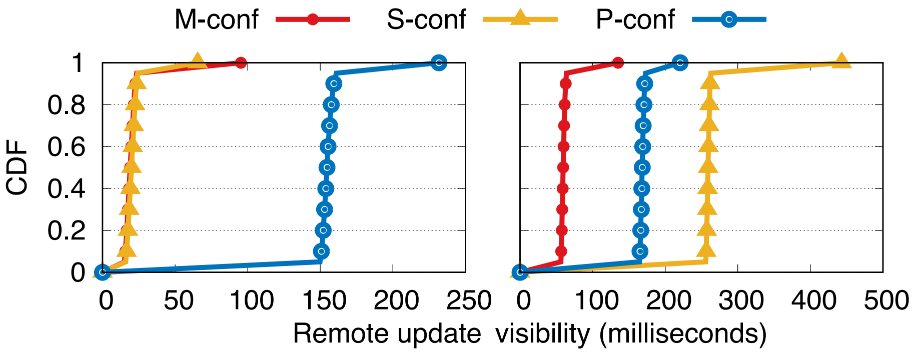


Figure 4.1 – Left: Ireland to Frankfurt (*10ms*); Right: Tokyo to Sydney (*52ms*)

4.4.1 The architecture of Saturn matters

We compare different architectures of the metadata dissemination service to better understand their impact on the system performance. We compare three alternative architectures: (i) a centralized, single-serializer architecture (S), (ii) a multi-serializer architecture (M), and (iii) a peer-to-peer version of SATURN that relies on the conservative label's timestamp order to apply remote operations (P). The P-architecture does not require the usage of serializers as an operation's metadata is sent directly from the origin datacenter (the datacenter local to the client that issued the operation) to the destination datacenters. We focus on the visibility latencies provided by the different implementations.

For the S-architecture, we placed the serializer in Ireland. For the M-architecture, we build the serializers tree by relying on Algorithm 3.1. We run an experiment with a read dominant workload (90% reads). Figure 4.1 shows the cumulative distribution of the latency before updates originating in Ireland become visible in Frankfurt (left plot) and before updates originating in Tokyo become visible in Sydney (right plot). Results show that both the S and M architectures provide comparable results for updates being replicated in Frankfurt. This is because we placed the serializer of the S-architecture in Ireland, and therefore, the propagation of labels is done efficiently among these two regions. Unsurprisingly, when measuring visibility latencies before updates originating in Tokyo become visible in Sydney, the S-architecture performs poorly because labels have to travel from Tokyo to Ireland and then from

Ireland to Sydney. Plus, results show that the P-architecture, that relies on the label's timestamp order, is not able to provide low visibility latencies in these settings. This is expected as, when tracking causality with a single scalar, latencies tend to match the longest network travel time (161ms in this case) due to false dependencies. In turn, the M-architecture is able to provide significantly lower visibility latencies to all locations (deviating only 8.2ms from the optimal on average).

4.4.2 The importance of genuine partial replication

In this section, we study the importance of genuine partial replication, a feature implemented by SATURN. Apart from the obvious scalability gains that brings—each datacenter only stores the data it replicates, genuine partial replication fully shields datacenters from updates on data that it is not replicated locally.

The problem is that, when compressing metadata to make causal consistency affordable, without genuine partial replication, updates on data items that are not replicated locally, may negatively impact the visibility of other updates unnecessarily. Let us illustrate this phenomenon with a simple example. Assume a system deployed in three datacenters dc_1, dc_2 and dc_3 that compresses metadata into a vector clock with an entry per datacenter (VC). dc_1 generates update a with vector $[1, 0, 0]$, which has to be replicated in dc_2 but not in dc_3; while dc_2 generates update b with vector $[1, 1, 0]$, which is replicated in dc_3. a does not depend on any other update. b is ordered after a. In this case, dc_3 should be able to make b visible as soon as it is received, as a is not replicated locally. Nevertheless, dc_3 can only make b visible when it receives an update from dc_1 such that can infer that a is not replicated locally; i.e., by receiving an update from dc_1 such that $VC[dc_1] > 1$, assuming FIFO links between datacenters.

In order to experimentally measure this phenomenon, we run a simple experiment in which we compare SATURN with a system, namely *Vector*, that compresses metadata into a vector with an entry per datacenter, as many solutions in the state-of-the-art do [7, 103, 59, 9]. The implementation of the latter is done on the codebase of SATURN by enriching the metadata, removing the propagation tree and replacing it by a peer-to-peer propagation schema in which datacenters multicast local updates to only the datacenters that replicate the data items being updated. Furthermore, a datacenter sends a heartbeat

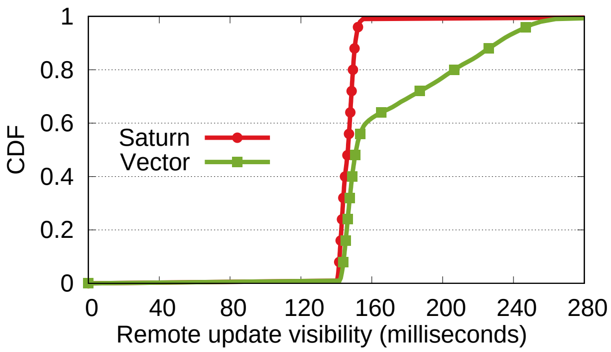


Figure 4.2 – Benefits of genuine partial replication in remote update visibility.

to another datacenter if this did not propagate any update to it during the last 5ms. Heartbeats are fundamental to ensure liveness and to reduce the negative impact of not having genuine partial replication. We chose to compare to a solution that compresses the metadata into a vector because it allows us to better assess the benefits of genuine partial replication. If we compared to a system that compresses causal dependencies in a single scalar without diminishing the impact of false dependencies [49], the potential gains of SATURN would come from two sources: the mitigation of false dependencies and the implementation of genuine partial replication. When comparing to *Vector*, the number of false dependencies observed in SATURN will, in the best case, match the ones observed in *Vector*. Therefore, the potential benefits would exclusively come from implementing genuine partial replication. We expect these benefits to be more substantial when compared to non-genuine systems that compress metadata into a single scalar.

We deploy both systems in 7 datacenters. Clients at each datacenter randomly read and update data items (each with potentially different replication groups) except for the clients in N. Virginia that only update keys replicated locally and in Ireland; and the clients in Ireland that only read the updates from N. Virginia and update data replicated locally and in Sydney. The idea is to create situations as the described above to measure its impact in both SATURN and *Vector*. Figure 4.2 shows a cumulative distribution of the latency before updates originating in Ireland become visible in Sydney. The results confirm that, by implementing genuine partial replication, SATURN is shielded from this phenomenon, as the latencies observed are considerably close to the aver-

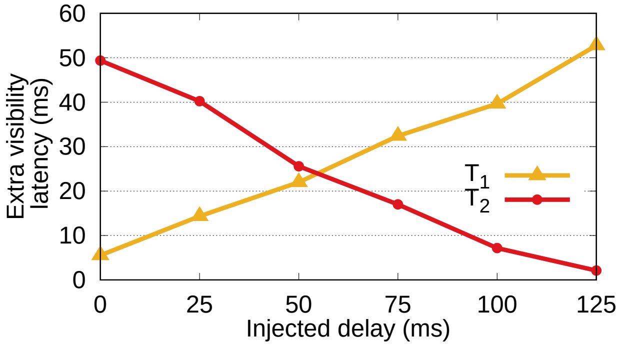


Figure 4.3 – Impact of latency variability on remote update visibility in SATURN.

aged latency among the datacenters. *Vector*, unlike SATURN, exhibits latencies that significant deviate from the average. Starting from the 60^{th} percentile, latencies start increasing significantly. For instance in the 90^{th} percentile, *Vector* adds already 80ms of latency when compared to the ones observed in SATURN. This is a strong result. It shows that using less metadata than other solutions, SATURN is able to provide better visibility latencies under partial replication. Next sections further confirm this result and also demonstrate that the size of the metadata have a significant negative impact in throughput.

4.4.3 Impact of latency variability on Saturn

The goal of this section is to better understand how changes in the link latency affect SATURN's performance. We have just seen that the correct configuration of the serializers' tree has an impact on performance. Therefore, if changes in the link latencies are large enough to make the current configuration no longer suitable, and these changes are permanent, a reconfiguration of SATURN should be triggered. In practice, transient changes in link latencies are unlikely to justify a reconfiguration; therefore we expect their effect on performance to be small.

To validate this assumption, we set up a simple experiment with three datacenters, each located in a different EC2 region: N. Carolina, Oregon and Ireland. For the experiment, we artificially inject extra latency between N. Carolina and Oregon datacenters (average measured latency is 10ms). From our experience, we expect the latency among EC2 regions to deviate from its average only slightly and transiently. Nevertheless, to fully understand the consequences of latency variability, we also experimented with unrealistically large deviations (up to 125ms).

Figure 4.3 shows the extra remote visibility latency that two different configurations of SATURN add on average when compared to an eventually consistent storage system which makes no attempt to enforce causality. Both configurations, T_1 and T_2, use a single serializer: configuration T_1 places the serializer in Oregon, representing the optimal configuration under normal conditions and configuration T_2, instead, places the serializer in Ireland.

As expected, under normal conditions, T_1 performs significantly better than T_2, confirming the importance of choosing the right configuration. As we add extra latency, T_1 degrades its performance, but only slightly. One can observe that, in fact, slight deviations in the average latency have no significant impact in SATURN: even with an extra delay of 25ms (more than twice the average delay), T_1 only adds 14ms of extra visibility latency on average. Interestingly, it is only with more than 55ms of injected latency that T_2 becomes the optimal configuration, exhibiting lower remote visibility latency than T_1. Observing a long and sustained increase of 55ms of delay on a link that averages 10ms is highly unlikely. Indeed, this scenario has the same effect of migrating the datacenter from N. Carolina to São Paulo. Plus, if such large deviation becomes the norm, system operators can always rely on SATURN's reconfiguration mechanism to change SATURN configuration.

4.5 Saturn vs. the state-of-the-art

We compare the performance of SATURN against eventual consistency and against the most performant causally consistent storage systems in the state-of-the-art.

4.5.1 GentleRain and Cure

We consider GentleRain [49] and Cure [7] the current state-of-the-art. These solutions are, from our perspective, the most scalable and performant solutions of the literature. We do not compare to solutions that rely on sequencers to compress metadata because (i) the Eunomia paper [59] already experimentally demonstrates that relying on Eunomia (the metadata serialization service that our prototype integrates) to compress metadata is more efficient than relying on sequencers, and (ii) GentleRain and Cure are representative of most of the sequencer-based solutions in terms of the amount of metadata used, as most of these use, as GentleRain, a single scalar [98, 81, 102, 43], or, as Cure, a vector with an entry per datacenter [26, 65, 9, 103]. Note that all these solutions, unlike SATURN, do not rely on clever metadata dissemination services. Therefore, the remote visibility latency is mostly determined by the precision in which causality is tracked (the size of the metadata). We have also experimented with solutions based on explicit dependency checking such as COPS [71] and Eiger [72]. Nevertheless, we concluded that approaches based on explicit dependency checking are not practical under partial geo-replication. Their practicability depends on the capability of pruning client's list of dependencies after update operations due to the transitivity rule of causality [71]. Under partial geo-replication, this is not possible, causing client's list of dependencies to potentially grow up to the entire database.

At their core, both GentleRain and Cure implement causal consistency very similarly: they rely on a background *stabilization mechanism* that requires all partitions in the system to periodically exchange metadata. This equips each partition with sufficient information to locally decide when remote updates can be safely–with no violation of causality—made visible to local clients. In our experiments, GentleRain and Cure's stabilization mechanisms run every 5ms following the authors' specifications. The interested reader can find more details in the original papers [49, 7]. We recall that SATURN does not require such a mechanism, as the order in which labels are delivered to each datacenter already determines the order in which remote updates have to be applied.

The main difference between GentleRain and Cure resides in the way causal consistency is tracked. While GentleRain summarizes causal dependencies in a single scalar, Cure uses a vector clock with an entry per datacenter. This enables Cure to track causality more precisely—lowering remote visibility latency—but the metadata management increases the computation and storage overhead—harming throughput. Concretely, by relying on a vector, Cure remote update visibility latency lower-bound is determined by the latency between the originator of the update and the remote datacenter. Differently, in GentleRain, the lower-bound is determined by the latency to the furthest datacenter regardless of the originator of the update [49, 7, 59].

In order to guarantee a fair comparison between SATURN, GentleRain and Cure, we have implemented our own version of the last two in the codebase of SATURN. The original systems where implemented in different programming languages (GentleRain is implemented in `C++`, and Cure and SATURN are implemented in `Erlang/OTP`). Also, GentleRain and Cure include extra features such as causally consistent read-only transactions, or the integration of high level data types with rich confluent semantics (CRDTs) [36, 90] that may be costly to support and add extra overhead unrelated to the cost of maintaining causal consistency.

4.5.2 Throughput experiments

In the following set of experiments, we aim at understanding how different parameters of the workload characterisation may impact SATURN's throughput in comparison to state-of-the-art solutions. We explore the workload space varying a single parameter, setting the others to a fixed value. We play with multiple aspects (default values within the parenthesis): values size (2B), read/write ratio (9:1), the correlation among datacenters (exponential), and the percentage of remote reads (0%).

Value size. We vary the size of values (operation's payload) from 8B up to 2048B. Sizes have been chosen based on the measurement study discussed in the work of Armstrong et al. [12]. Results (Figure 4.4) show that all solutions remain unaffected up to medium size values (128B). Nevertheless, as we increase the value size up to 2048B, solutions exhibit, as expected, a similar behavior handling almost the same amount of operations per second. This shows that, with large value sizes, the performance overhead introduced by GentleRain and, above all, Cure, is masked by the overhead introduced due to the extra amount of data being handled.

R/W ratio. We vary the read/write ratio from a read dominant workload (99% reads) to a balanced workload (50% reads). Results (Figure 4.5) show that solutions are similarly penalized as the number of write operations increases.

Correlation. We define the correlation between two datacenters, as the amount of data shared among them. The correlation determines the amount of traffic generated in SATURN due to the replication of update operations. We define four patterns of correlation: *exponential*, *proportional*, *uniform*, and *full*. The exponential and proportional patterns fix the correlation between the datacenters based on their distance. Thus, two datacententers closely located (e.g., Ireland and Frankfurt) have more common interests than distant datacenters

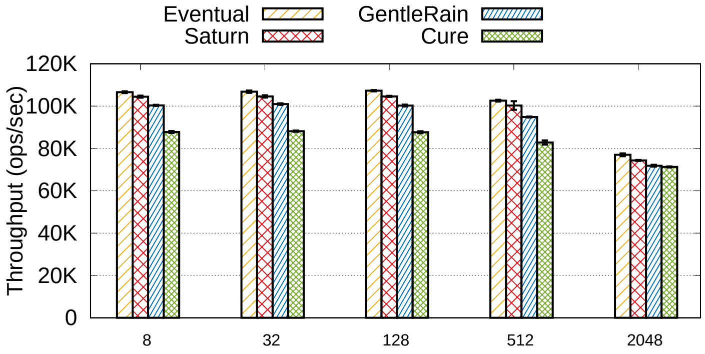


Figure 4.4 – Dynamic workload throughput experiments: varying the operation's payload size (bytes)

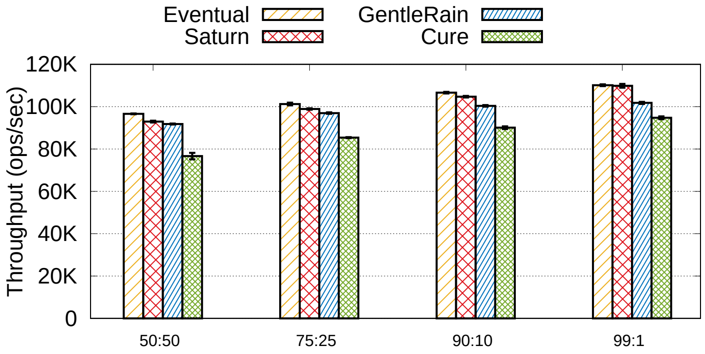


Figure 4.5 – Dynamic workload throughput experiments: varying the read:write ratio

(e.g., Ireland and Sydney). The exponential pattern represents a more prominent partial geo-replicated scenario by defining very low correlation among distant datacenters. The proportional pattern captures a smoother distribution. The uniform pattern defines an equal correlation among all datacenters. Lastly, the full pattern captures a fully geo-replicated setting. Results show that the more prominent the partial geo-replication scenario is, the better results SATURN presents when compared to GentleRain and Cure, that are required to send heartbeats constantly, adding an overhead when compared to SATURN. Interestingly, even in the full geo-replicated scenario, the best case scenario

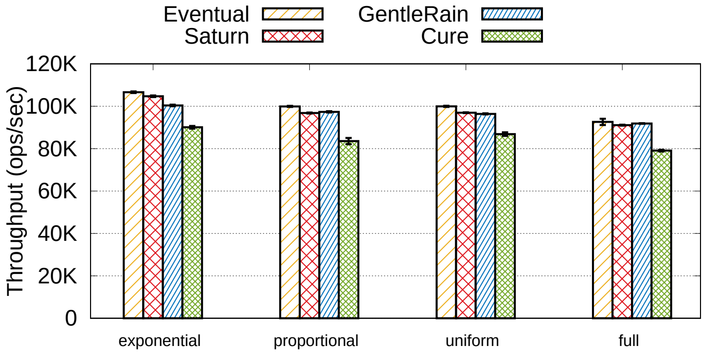


Figure 4.6 – Dynamic workload throughput experiments: varying the correlation distribution

for GentleRain and Cure, SATURN still provides a throughput comparable to GentleRain and significantly outperforms Cure (15.2% increase).

Remote reads. We vary the percentage of remote reads from 0% up to 40% of the total number of reads. Interestingly, results show (Figure 4.7) that GentleRain and, above all, Cure are significantly more disrupted than SATURN by remote reads. To better understand the cause of this behavior, we need to explain how remote reads are managed in our experiments by GentleRain and Cure. As in SATURN, a client requiring to read from a remote datacenter first needs to attach to it. An attach request is performed by providing the latest timestamp observed by that client (a scalar in GentleRain and a vector in Cure). The receiving datacenter only returns to the client when the stable time—computed by the stabilization mechanism—is equal or larger than client's timestamp. This significantly slows down clients. Results show that with 40% of remote reads, SATURN outperforms GentleRain by 15.7% and Cure by 60.5%.

We can conclude that SATURN exhibits a performance comparable to an eventually consistent system (2.2% of overhead on average) while showing a slightly better throughput than GentleRain (4.8% average) and significantly better than Cure (24.7% on average). Cure overhead is dominated by the managing of a vector for tracking causality instead of a scalar as in GentleRain and SATURN. Regarding GentleRain, SATURN exhibits slightly higher throughput due to the overhead caused by GentleRain's stabilization mechanism.

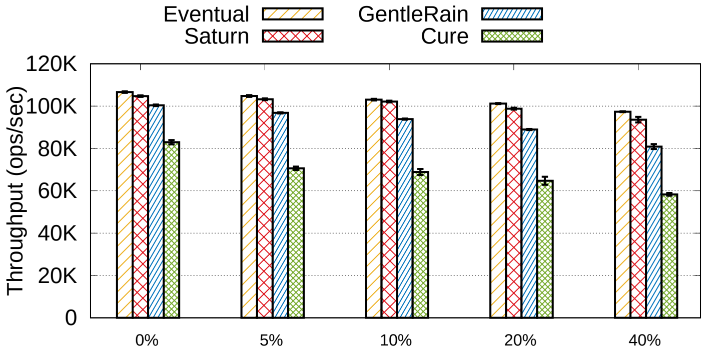


Figure 4.7 – Dynamic workload throughput experiments: varying the percentage of remote reads

4.5.3 Visibility latency experiments

In the following experiment, we measure the visibility latency provided by each of the systems. We expect SATURN to exhibit lower visibility latencies on average than both Cure and GentleRain. We expect to slightly outperform Cure as we avoid the costs incurred by Cure's stabilization mechanism. We expect to significantly outperform GentleRain since it does not mitigate false dependencies and the exhibited remote visibility latencies should theoretically tend to match the longest travel time among datacenters.

In addition to measuring the average visibility latencies provided by each solution, we analyze both the best and the worst case for SATURN. Given that serializers possible locations are limited, labels traversing the whole tree of serializers are likely to be delivered with some extra undesired delay. Concretely, in the following experiment, the major deviation from the optimal latencies is produced by the path connecting Ireland and Sydney (extra 20ms).

For these experiments, we have used the default values defined in §4.5. Results show that SATURN only increases visibility latencies by 7.3ms on average when compared to the optimal, outperforming GentleRain and Cure that add 97.9ms and 21.3ms on average respectively. Figure 4.8 shows the cumulative distribution of the latency before updates originating in Ireland become visible in Frankfurt (left plot) and Sydney (right plot). The former represents the best case scenario with no extra delay imposed by the tree. The latter represents the worst case scenario. Figure 4.8 shows that SATURN almost matches

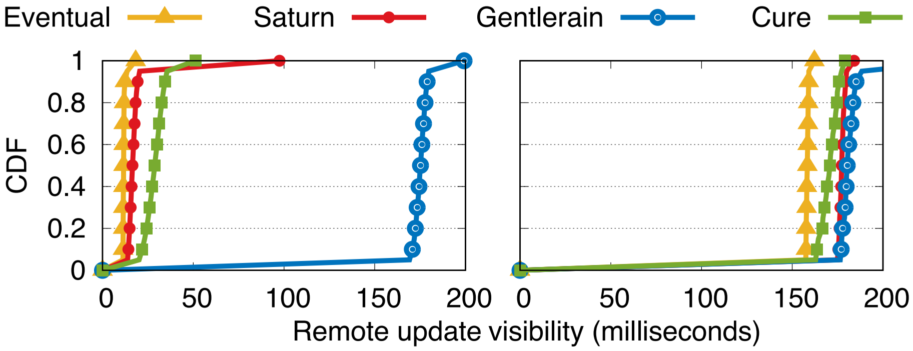
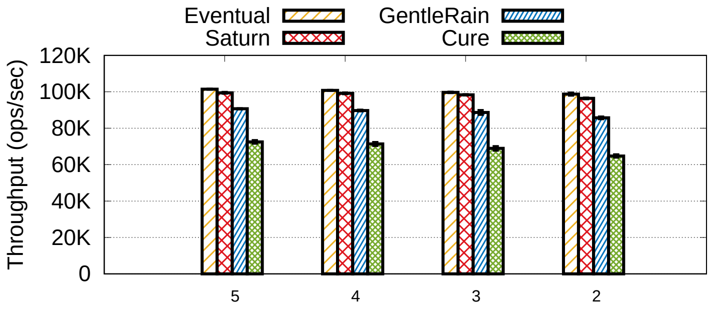


Figure 4.8 – Left (best-case scenario): Ireland to Frankfurt (*10ms*); Right (worst-case scenario): Ireland to Sydney (*154ms*)

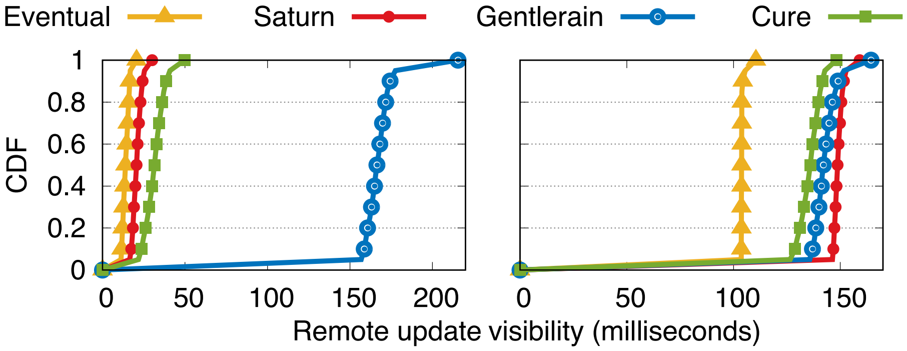
the optimal visibility latencies in the best case scenario (only 7ms of extra delay in the 90^{th} percentile) and, as expected, adds an extra of 20.4ms (90^{th} percentile) in the worst case. Results also show that SATURN is able to provide better visibility latencies than both GentleRain and Cure in the best case and to GentleRain in the worst case. As expected, GentleRain tends to provide visibility latencies equal to the longest network travel time, which in this case is between Frankfurt and Sydney. Interestingly, in SATURN's worst case, Cure only serves slightly lower latencies (3.6ms less in the 90^{th} percentile). Although the metadata used by Cure to track causality theoretically allows it to make visible remote updates in optimal time, in practice, the stabilization mechanism results in a significant extra delay.

4.5.4 Facebook benchmark

To obtain an assessment of SATURN's performance under complex realistic workloads, we experiment with a social networking workload we integrated into Basho Bench, our benchmarking tool. Our workload generator is based on the study of Benevenuto et al. [24]. The study defines a set of operations (e.g., browsing photo albums, sending a message, editing user settings among many others) with its corresponding percentage of occurrence. This serves us not only to characterize the workload in terms of the read/write ratio, but it also tells us whether an operation concerns user data (e.g., editing settings), a friend's data (e.g., browse friend updates), or even random user data (e.g.,



(a) Throughput. The horizontal axis defines the maximum number of replicas per data item.



(b) Left (best-case scenario): Ireland to Frankfurt (*10ms*); Right (worst-case scenario): Ireland to Tokyo (*107ms*).

Figure 4.9 – Facebook-based benchmark results.

universal search), which relates to the number of remote operations.

We use a public Facebook dataset [101] of the New Orleans Facebook network, collected between December 2008 and January 2009. The dataset contains a total of 61096 nodes and 905565 edges. Each node represents a unique user, which acts as a client in our experiments. Edges define friendship relationships among users. In order to distribute and replicate the data among datacenters (seven in total), we have implemented the partitioning algorithm described in [82], augmented to limit the maximum number of replicas each partition may have, to avoid partitionings that rely extensively on full replication. As in [82], partitions are made to maximize the locality of data regarding a user and her friends, thus, minimize remote reads.

We examine the throughput of SATURN in comparison to an eventually consistent system, GentleRain, and Cure. Figure 4.9a shows the results of a set of experiments in which we fix the minimum of replicas to 2, and vary the maximum from 2 to 5, which indirectly varies the number of remote read operations. Results show that SATURN exhibits a throughput comparable to an eventually consistent system (only 1.8% of averaged overhead) and significantly better than GentleRain and Cure, handling 10.9% and 41.9% more operations per second on average respectively.

In a second experiment, we measure the remote update visibility latency exhibited by the solutions. SATURN increases visibility latencies by 16.1ms on average when compared to the optimal, outperforming GentleRain and Cure that add 79.2ms and 23.7ms on average respectively. In addition, we analyze the best and the worst case scenario for SATURN. Figure 4.9b shows the visibility latency of updates replicated from Ireland to Frankfurt (on the left) and Tokyo (on the right). The former represents the best case scenario for SATURN; the latter represents the worst. In the worst case scenario, SATURN introduces significant overheads when compared to the optimal (47.2ms in the 90^{th} percentile). This is expected, as it has to traverse the whole tree. Nevertheless, it still exhibits a performance comparable to both GentleRain and Cure, only adding 0.9ms and 9.9ms respectively in the 90^{th} percentile. Moreover, in the best case scenario, SATURN exhibits visibility latencies very close to the optimal (represented by the eventually consistent line), with only a difference of 8.7ms in the 90^{th} percentile.

Chapter 5

Related work

This chapter describes, classifies and compares the most relevant techniques and solutions proposed in the past. We focus on causally consistent systems. We do not include strongly consistent systems such as Spanner [41] in our discussion, as in our opinion these are designed to satisfy the needs of a different set of application designers. Causally consistent solutions are proposed to help developers to program applications with strict availability and performance requirements. In constrast, strongly consistent systems prioritize consistency at the cost of penalizing performance: these require using coordination-intensive protocols, such as distributed agreement [68, 78], to guarantee consistency.

We start by introducing a taxonomy that classifies previous systems based on the characteristics of their solution and the features that integrate. Secondly, using our taxonomy, we describe, in detail, each of the most relevant previous solutions. Then, we compare them, discussing the advantages and disadvantages of each solution for different settings. Finally, we briefly compare them to our work.

5.1 A taxonomy for causally consistent systems

We group solutions in four categories based on the key technique behind their implementation of causal consistency:

- **Sequencer-based.** This group of solutions rely on centralized components, commonly called *sequencers*, to compress causal metadata. Sequencers allow designers to trivially serialize multiple sources of concurrency. Typically, solutions use a sequencer per datacenter, being able to compress causal metadata into a vector with an entry per datacenter.

- **Explicit check.** This group of solutions tag remote updates with a list of explicit dependencies. Upon arrival to a remote datacenter, dependencies are locally checked by issuing a set of dependency check messages to the corresponding datacenter servers.
- **Background stabilization.** This group clusters solutions that rely on some sort of background stabilization. These mechanisms are typically used to ensure that remote operations are only made visible in a datacenter to local clients when its causal dependencies are already visible locally.
- **Lazy resolution:** Finally, this last group clusters solutions that allow datacenters to install remote operations as these are received without any consistency check. Then, causality is enforced when clients read by ensuring that the version returned is not in conflict with clients' causal history (what a client has already observed).

Within each of the above groups, we characterize each solution based on four categories:

- **Metadata size:** This category characterizes the amount of metadata a solution uses to capture causal dependencies. Under causal consistency, datacenters replicate local operations asynchronously in other datacenters. In order to make remote updates visible in an order that does not violate causal consistency, operations are tagged with a piece of metadata, capturing operations' causal dependencies. The size of the metadata is usually proportional to some characteristics of the system. For instance, it can be proportional to the number of datacenters, or to the number of data items. When metadata is not compressed, the size its variable and proportional to the number of causal dependencies each operation has. In solutions in which the metadata is aggressively compressed, its size is constant independently of the system characterization. We introduce the following notation: M denotes the number of datacenters; N the number of partitions per datacenter (we assume that each datacenter is equally partitioned to simplify the notation); and K the total number of data items.

- **False dependencies:** The metadata structure determines, in most of the cases, the amount of false dependencies artificially created by the solutions. This category quantifies the number of false dependencies introduced as an artifact of the metadata management. We identify four types of false dependencies: *data-item (I)* that represents dependencies among concurrent update operations over the same data item local to different replicas, *partition (P)* that represents dependencies among concurrent update operations on data items belonging to the same logical partition or shard, *intra-datacenter (DC)* that represents dependencies among concurrent update operations local to the same datacenter, *inter-datacenter (G)* that represents dependencies among concurrent update operations local to different datacenters.

  Note that not all types of false dependencies are equally damaging. Out of the four, the inter-datacenter (G) is the most harmful one. This is because a G-false-dependency will force a datacenter to prevent the installation of a remote update coming from a nearby datacenter until the false dependency, originating at a farther datacenter, is received. Given that latencies among datacenters can be of more than 160ms (see Table 4.1), this type of false dependencies will add significant extra delays in remote visibility latencies. Contrary, other types of false dependencies, such as the intra-datacenter type, will impact the remote visibility latency less significantly, as the latency among parties is of few milliseconds at the most.

- **Partial replication support:** This category captures the ability of the systems to support partial replication or not (whether this setting was considered in the design or not). A system can be characterized as one of the following: *no support* if they assume full replication, *non-genuine* if it supports partial replication but it is no genuine, or *genuine* if the solution implements genuine partial replication.

- **Type of dissemination:** Solutions propose different ways of disseminating updates among replicas. In previous work, we have identified three schemes:

  - The *all-to-all* scheme in which only the update's origin datacenter (the one local to the client that issued the update operation) propagates to the rest of interested datacenters.

- The *pair-wise* scheme in which datacenters periodically synchronize pair-wise: one datacenter propagates to a second datacenter all update operations (originated in any datacenter) that have not been seen by the latter.
- The *master-slave* scheme in which each data item gets assigned a master replica. Updates are all first executed in the master replica and then propagated to the slave replicas. Reads can generally be executed at any replica (master or slave).

5.2 Causally consistent replicated systems

In this section, we describe the most relevant solutions. For each of the groups introduced before, we first describe in detail how each technique works and discuss its advantages and disadvantages. Then, we describe each solution characterizing it based on our four categories: metadata structure, false dependencies, partial replication support and type of dissemination; and briefly discussing additionally features that may integrate.

5.2.1 Sequencer-based solutions

Sequencers are centralized components that trivially enable metadata compression, key to efficiently implement causal consistency. Imagine a database system, in which the application state is sharded among multiple servers, and therefore, update requests may be handled in parallel by different servers. Having a sequencer allows these servers to establish a total order—compliant with causality—among clients' update requests without requiring explicit coordination among them, as this coordination is done through the sequencer. In such a setting, before an update is considered completed, and thus made visible to other clients, the sequencer is contacted. This assigns a timestamp to the request. The sequencer proposes increasing timestamps (usually monotonically increasing), establishing thus a total order among update requests that is a linear extension of the causal order defined by them. It is simple to see that the total order is compliant with causality. An update request a can only be a causal dependency of another update b, if a was completed before b: either because both were executed one after the other by the same client, the client that executed b observed a before, or by transitivity (§2.1.2). Therefore, if $a \leadsto b$, the sequencer would assign a timestamp to a before than to b. Since the sequencer assigns increasing timestamps, a will be ordered before b in the total order, consistently with causality.

Having a central serialization point, instead of having to devise a distributed algorithm among the entities that one wants to serialize, greatly simplifies the problem. Unfortunately, to ensure that the total order is consistent with causality, sequencers have to operate in the critical operational path of clients, limiting concurrency. Thus, sequencers represent, not only single points of failure, but also potential performance bottlenecks. The system's throughput upper-bound is then limited by the amount of requests a sequencer can order per unit of time. Furthermore, delays on the sequencer are directly observable by end-users.

Typically, sequencer-based solutions rely on a sequencer per datacenter [98, 43, 26, 65, 103, 9]. This way, the metadata is compressed into a *Vector* of scalars with an entry per datacenter. The vector indicates on which update from each datacenter, the update b tagged with *Vector*, depends. Thus, *Vector[*dc_1*]* indicates that update b depends on all updates generated in dc_1 with timestamp *ts* (assigned by the sequencer local to dc_1) such that *ts*$\leq$*Vector[*dc_1*]*[1]. Thus, a datacenter can only make a remote update visible once all its dependencies are already visible locally.

Note that some of the solutions [26, 65, 43, 98] we are including in the group of sequencer-based solutions do not explicitly use sequencers, as they assume single-machine replicas. Nevertheless, we argue that in order to adapt their design to the multi-server architecture of current datacenters, using a sequencer is the most straightforward way of doing it.

Another relevant, common characteristic of this group of solutions is that they require a logically-centralized, unique receiver per datacenter. This fact adds another single point of failure and makes these solutions not only bounded performance-wise by the sequencer capacity, but also by the receivers capacity. Receivers queue remote updates until its dependencies are known to be installed locally. Thus, when the pace in which remote updates are received is faster than the pace in which remote updates are installed in the local datacenter, the queue grows indefinitely, increasing the remote visibility latency of updates until the receiver starts dropping incoming remote updates or simply

[1] Except for the entry in the vector that corresponds to the update's local datacenter dc$_{local}$, as, in such a case, the update depends on any other update with timestamp ts originating at the same datacenter such that *ts*$<$*Vector[*dc_{local}*]*

crashes due to a memory issue.

We now describe each of the systems individually, highlighting the particularities of each of them.

The ISIS toolkit [26] is a distributed programming environment that integrates three multicast primitives: a casual multicast primitive called *cbcast*, an atomic multicast primitive called *abcast*, and a group multicast primitive called *gbcast*. It assumes single-machine replicas. One could implement a causally consistent geo-replicated database system by using the cbcast primitive. In order to ensure causal consistency in a fully replicated setting, cbcast uses a vector clock with an entry per datacenter (replica) to track causal dependencies. Authors also propose a solution to provide causal consistency under partial replication. In the latter case, the metadata is enriched to a set of vectors, one per replication group. The size (number of entries) of each vector is equal to the number of replicas composing the replication group it represents. In the worst case, each update would need to carry a total of 2^{dcs}-1 (total number of possible replication groups given a set a datacenters) vectors, significantly increasing the metadata size and penalizing throughput. We name this variant of cbcast that supports partial replication *cbcast**. Both cbcast and cbcast* adds intra-datacenter false dependencies, as each datacenter generates a single stream of totally ordered update requests.

Lazy Replication [65] proposes a way of building causally consistent replicated systems. In this solution, update operations local to a replica are tagged with their causal dependencies, and sent asynchronously to the rest of the replicas. A remote replica only installs an operation coming from another replica when its dependencies are satisfied locally. In their proposed implementation, they argue that in order to make their solution efficient, causal dependencies are compressed into a vector with an entry per replica (each datacenter in the setting being considered). It assumes single-machine replicas. Thus, it adds intra-datacenter false dependencies as their metadata does not reflect the concurrency exhibited internally at each datacenter. Their solution is envisioned for a fully replicated setting. R. Ladin et al. propose two additional types of operations to cope with applications that require stronger-than-causal consistency: *forced*, and *immediate*. Forced operations are totally ordered among themselves and installed in all replicas respecting such order. This is useful, for instance, to ensure uniqueness; e.g., to ensure that two users do not successfully sing up with the same user-name. Immediate operations are the stronger type of operations. They are installed in every replica in the same order relative to every other operation (immediate, forced, and causal). These operations are ordered consistently with external events [52], making immediate operations a very powerful primitive.

Bayou [98, 81] is a causally consistent, replicated storage system designed for mobile computing. It assumes single-machine replicas and that each replica stores the whole database (full replication). It allows replicas to modify the database state while being disconnected and synchronize with any other replica that happens to find. Bayou ensures that each replica eventually reaches the same final state. To achieve this, it integrates conflict detection and resolution mechanisms that may cause operations to be reordered. Bayou relies on a log-based (updates are stored in a causally ordered log) pair-wise replica synchronization. When a replica receives an update coming from a local user, it timestamps it and adds it to the log. Replicas generate monotonically increasing scalar timestamps. When a replica (the receiving replica) wants to synchronize with other replica (the sending replica), it first notifies the latter which updates are already in its local log. This information is maintained in a vector clock with an entry per replica that stores the timestamp of the latest update installed from each of the replicas. The sender replica, by comparing the timestamps of the operations in its log and the corresponding entries in the receiving replica vector clock, can then determine which operations have to be sent to the receiving replica. These operations have to be installed at the receiving replica in the sender's log order. Since each log is causally consistent and when synchronizing, the sender replica do not only send its local updates but any other update that may precede it in the causal order, this dissemination scheme trivially ensures causal consistency. Regarding the amount of false dependencies, it will depend on how the replicas synchronize; e.g., if all replicas synchronize with all, this solution would probably only introduce intra-datacenter false dependencies. Bayou does not support partial replication.

TACT [102] is a middleware layer that enables replicated systems to tune the level of inconsistency allowed among replicas based on three metrics: *numerical error*, *order error*, and *staleness*. Numerical error limits the number of writes a replica can install before propagating to other replicas; order error limits the number of operations whose final global order is still unknown, and therefore are subject to reordering, a replica can have in their local log; and staleness bounds with real-time the delay of update propagation among replicas. Independently of the values of each of the three boundary metrics, the system always remains causally consistent. The underlying mechanism that ensures causality is similar to Bayou's. Therefore, the TACT solution has the same characteristics than Bayou's solution in terms of metadata structure, false dependencies, and dissemination scheme. As Bayou, TACT does not includes support for partial replication and assumes single-machine replicas.

The **PRACTI [43]** approach defines a set of three properties that, according to the authors, an ideal replication framework should provide: partial replication; arbitrary consistency, meaning that the system can provide both strong and weak consistency; and topology independence, meaning that any replica can exchange updates with any other replica. They propose an architecture that supports the PRACTI properties. As other solutions [65, 102] supporting several consistency levels, their proposed architecture remains causally consistent at all time. The PRACTI solution is based on Bayou, and therefore, has the same characteristics: single-machine replicas, log-based pair-wise replica synchronization, updates tagged with a scalar, vector clocks maintained at each replica in order to reduce the amount of data exchange when synchronizing. The PRACTI architecture supports partial replication but not genuine partial replication, as a replica may still need to observe update operations on data items that are not replicated locally. They propose two optimizations to reduce the impact of this problem: separation of data and metadata, and imprecise invalidations. By separating data and metadata, replicas do not have to receive the payload of operations that are not replicated locally, considerably reducing the amount of data handled by the system when partially replicated. The data is sent through a channel that requires no ordering guarantees. The metadata, namely *invalidations*, is sent through the causally consistent channel. An update is visible to clients when both the data and the metadata have been received. On the other hand, imprecise invalidations are metadata messages that compress the metadata information regarding multiple operations, further reducing the amount of communication among replicas.

ChainReaction [9] is a causally consistent geo-replicated key-value store. It supports intra- and inter-datacenter replication. Intra-datacenter replication is supported by a variant of chain replication [100] that ensures causal consistency. This variant relaxes the consistency guarantees provided by the original chain replication scheme to enhance performance. To ensure causal consistency across datacenters, ChainReaction relies on a sequencer per datacenter. It compresses causal dependencies in a vector with and entry per datacenter. Updates are tagged with this vector. A receiver datacenter only makes remote updates visible to local clients when its causal dependencies (encoded in the vector) are already visible locally. ChainReaction assumes full replication. It assumes an all-to-all dissemination schema. It introduces intra-datacenter false dependencies.

SwiftCloud [103] is a causally consistent geo-replicated key-value store. It offers single and multi-object operations. A multi-object operation contains both read and write operations, it is executed in a causally-consistent snapshot of the database, and ensures atomicity (either all the updates belonging to the same multi-object operation are visible or none). We refer to this set of guarantees as *transactional causal consistency (TCC)*. It assumes full replication among datacenters. Additionally, it allows clients to partially replicate the application state. The idea behind SwiftCloud is to significantly reduce latency by allowing clients to read and write from their local replicas (or caches) and only propagate to the local datacenter asynchronously. This is achieved efficiently by relying on high level data types with rich confluent semantics [36, 90]. The metadata is compressed into a vector clock with an entry per datacenter. Each datacenter integrates a sequencer in order to serialize local updates. As with other solutions that rely on a vector clock with an entry per datacenter, SwiftCloud introduces intra-datacenter false dependencies. It assumes an all-to-all dissemination scheme.

M. Shen et al. [61] propose algorithms to achieve causal consistency in both partial and full replication settings. They detail three algorithms: Full-Track, Opt-Track, and Opt-Track-CPR. All algorithms assume an all-to-all dissemination scheme.

The two former algorithms ensure causal consistency under partial replication by relying on a matrix clock of size $n \times n$, where n is the number of sites (equivalent to datacenters in our nomenclature). They assume single-machine sites. In such a setting, their algorithm will not add any false dependency. Nevertheless, adapting their algorithms to multi-server datacenters would require serializing all updates happening at each datacenter, adding thus false dependencies among updates local to the same datacenter. Opt-Track further optimizes the Full-Track algorithm by reducing the amortized complexity of both message size and space by exploiting the transitivity rule of causal consistency. Nevertheless, the message size upper bound complexity remains $\mathcal{O}(n^2)$ in both, which may substantially impact the algorithms performance.

Finally, the Opt-Track-CPR algorithm ensures causal consistency under full replication. It relies on a vector with an entry per replica. As with previous algorithms, when assuming single-machine replicas, the algorithm does not add any false dependency. Nevertheless, adapting it to multi-server datacenters would require serializing all updates happening at each datacenter, adding thus false dependencies among updates local to the same datacenter.

5.2.2 Solutions based on explicit check messages

This type of solutions ensure causal consistency by having datacenters explicitly checking the dependencies of each remote update before applying it locally. It works as follows. Clients keep track of causal dependencies, in a data structure commonly called causal context. When a client issues an update request, the list of dependencies is attached to the request. In turn, the local datacenter propagates the update, together with its dependencies, to remote datacenters. On arrival, the receiver datacenter issues a set of messages to explicitly check whether the update's dependencies are already installed locally. How many of these check messages are sent per request depends on how causal dependencies are represented and how the datacenters are sharded. For instance, if one tracks causal dependencies at the granularity of data items, and the application state is sharded at each datacenter in multiple partitions, there will be a maximum of one check message per partition if the request depends on operations that updated at least one data item on each partition.

These type of solutions do not require a centralized component to order events, such a sequencer, eliminating a potential bottleneck. Furthermore, unlike sequencer-based solutions, in which there is a single stateful receiver per datacenter, solutions based on explicit check messages can deploy a set of stateless receivers at each datacenter. This fact allows them to eliminate another potential performance bottleneck, as the receiver in sequencer-based solutions solely coordinates the local application of remote updates checking dependencies and delaying the visibility of unordered updates.

We now describe each of the systems individually, highlighting the particularities of each of them.

COPS [71] is a causally consistent geo-replicated key-value store. Its design assumes that each datacenter replicates the full application state. This fact is fundamental for the practicability of their solution. COPS opts for a fine-grained dependency tracking approach. It tracks dependencies at the granularity of data items. Thus, in the worst case, if a client's update request depends on every other data item, the metadata attached to the request is a vector with an entry per data item in the database. Nevertheless, in practice, an update request rarely piggybacks the full vector. A client only includes as an update's dependencies its previous update, and all updates observed (by reading from the database) in between its previous update and current one. Due to the transitivity rule of the happened-before relation, a client can clean their dependency context after issuing an update. COPS' design assumes an all-to-all dissemination schema. Despite using a large amount of metadata, COPS still adds false dependencies among concurrent updates over the same key (local to different replicas), as these are all serialized. The authors propose a protocol to enforce causally consistent read-only transactions: a multi-key read operation that reads from a causally consistent snapshot of the database.

Eiger [72] is a geo-replicated store built on top of Cassandra [66]. As in COPS, causal dependencies are tracked at the granularity of data items and clients' causal context is cleaned after update operations. Eiger offers stronger semantics than COPS by supporting atomic write-only transactions: multi-key write operations that create the illusion that all writes belonging to the same transaction are applied atomically. Furthermore, it improves COPS read-only transactional protocol efficiency. Instead of requiring the explicit dependencies of each version being read to compute the snapshot from which a read-only transaction reads, Eiger's transactional protocol solely relies on lightweight logical clocks. This comes with the cost of sometimes requiring a third read round. COPS, in constrast, only requires one round in the normal case and a maximum of two read rounds per read transaction.

The **bolt-on [18]** architecture separates consistency concerns from liveness, replication and durability concerns. The architecture includes a *shim* layer, placed between an eventually consistent storage system and clients. This layer upgrades an eventually consistent storage system to convergent causal consistency. Clients interact directly with the shim layer, which contains a local store. Reads are served from the shim layer's local store, which is guaranteed to always be causally consistent with clients' causal history. Updates are stored in the layer's local store and eventually propagated to the eventually consistent storage system. The shim layer periodically pulls new versions from the underlying storage system in order to update its local store and thus make other clients' updates visible to its local clients.

Authors opt for a fine-grained dependency tracking approach, similar to the one of COPS and Eiger. Thus, the size of the metadata associated to each update is proportional to a client's causal history; being proportional, in the worst case, to the total number of data items in the store. It only adds false dependencies among concurrent updates on the same data items. Its design assumes that each datacenter replicates the full application state.

Karma [75] is a distributed key-value store designed for partial replication. The goal of Karma is to enforce causal consistency while minimizing the amount of remote accesses (unavoidable under partial replication) by leveraging cache techniques. Its design assumes an all-to-all dissemination scheme. Karma assumes that each datacenter can replicate only part of the application state. Nevertheless, it assumes that given a deployment of a set of datacenters, one can cluster them into a set of groups, namely rings, each replicating the full application state. This limits the generality of their model.

Karma ops for a fine-grained dependency tracking approach, similar to the one of COPS. Thus, it tracks dependencies at the granularity of data items, adding false dependencies among concurrent updates on the same data item originating at different replicas. Unlike COPS, Karma keeps track of updates that have been stored in all replicas (globally stable) and avoid adding them as causal dependencies of other updates, as it is guaranteed that the latter update will be installed at any datacenter after any update that was globally stable when this was issued.

We consider that Karma does not implement genuine partial replication, as the partial replication model supported is not flexible enough. Karma is designed assuming that one can cluster datacenters in a way that each group replicates the full application state, in which case Karma ensures genuineness. Nevertheless, Karma will not work as it is if one cannot ensure this invariant. The modifications required to make Karma work under this—more general—setting would cause Karma to become non-genuine or require a larger amount of causal metadata per update (carrying more dependencies).

Orbe [47] is a distributed key-value store. Its design assumes that each datacenter replicates the full application state. This fact, as in COPS and Eiger, is fundamental for the practicability of their solution. Orbe represents causal dependencies in a matrix, with an entry per partition per datacenter. Thus, all operations, local to a partition, are serialized, which has a negative impact on remote update visibility when compared to systems such as COPS that opt for a more fine-grained representation of dependencies. Nevertheless, if an update depends on two other updates over different keys, stored by the same partition, it will carry only one dependency (the one ordered after in the serialization) rather than two, as in COPS. This has a positive impact on throughput. Orbe assumes and all-to-all dissemination scheme. Due to its metadata compression, it adds false dependencies among operations local to the same partition. Orbe also offers read-only transactions. Their implementation is based on loosely-synchronized physical clocks which make their protocol resilient to skewed workloads. Nevertheless, the efficiency, but not the correctness, of their protocol is affected by clock drifts.

5.2.3 Solutions that rely on background stabilization

These solutions ensure causal consistency by relying on a background stabilization mechanism. This mechanism permits full decentralization. It works as follows. Partitions composing each datacenter accept requests from local clients and handle them without any coordination. Each update operation is tagged with a piece of metadata, generated locally, that captures the update's causal dependencies. This is asynchronously replicated to sibling partitions: equivalent partitions belonging to remote datacenters. Thus, these solutions assume that datacenters are logically, equally partitioned. Upon receiving a remote update, a partition stores it in its local multiversion storage, but it does not necessarily make it visible immediately.

Periodically, a stabilization mechanism that coordinates all partitions runs in the background. The goal is to ensure that if a partition makes a remote update visible (local updates are always immediately visible to other local clients), it is certain that its causal dependencies are also visible in the local datacenter. Thus, a client can safely—without violating causality—read from multiple partitions of the same datacenter with no consistency check. The procedure comprises the following steps. Each partition keeps track of the updates run locally and at sibling partitions. Periodically, each partition gathers this information from every other partition of the same datacenter and computes the current *stable* consistent snapshot. A snapshot is stable if it is known that any operation belonging to it has already been received and it is visible in the local datacenter. For this, the metadata attached to each update must be comparable to how snapshots are identified. Normally, this is simply a scalar or a vector with an entry per datacenter.

Interestingly, solutions based on stabilization mechanisms have the advantage of enabling fully decentralized implementations: partitions coordinate with sibling partitions without the need of having a centralized ordering service or a local receiver at each datacenter. Nevertheless, background stabilization mechanisms augment the consequences of the tradeoff between throughput and remote visibility latency. First, in order to make remote updates visible, datacenters have to run the stabilization mechanism periodically. Thus, the time interval between runs is an extra delay added to the remote visibility latency, which is determined by the metadata used to represent causal dependencies. Second, the overheads associated to the amount of metadata used have a more significant impact than in other type of solutions. This is caused by the fact that these solutions require keeping multiple versions of each data item, and therefore they require consistency checks when reading from the store in order to find the "safe" version (consistent with causality) to be returned. Furthermore, every run of the background stabilization mechanism requires each partition not only to receive messages from every other partition of the same datacenter, but also to compute the stable time. This computation has a penalization on throughput determined by the periodicity in which the stabilization runs and the size of the metadata.

To illustrate this, Figure 5.1 plots the remote visibility latency (left plot) and the throughput penalty (right plot) of GentleRain [49] and Cure [7] when varying the time interval between stabilization runs from 1ms to 100ms. The throughput penalty is normalized against a geo-replicated system that does not add any overhead due to consistency management. In this simple experiment, we have deployed both solutions spanning three datacenters. The round-trip-times across datacenters are 80ms between datacenter 1 (dc_1) and both dc_2 and dc_3; and 160ms between dc_2 and dc_3. Latency values refer to the (90th percentile) delays incurred by each system at dc_2 for updates originating at dc_1. For each experiment, we deploy as many clients as possible without saturating the system.

As expected Cure, which uses a vector clock with an entry per datacenter to represent causal dependencies, exhibits worse throughput and better remote visibility latency than GentleRain, which relies on a single scalar. More inter-

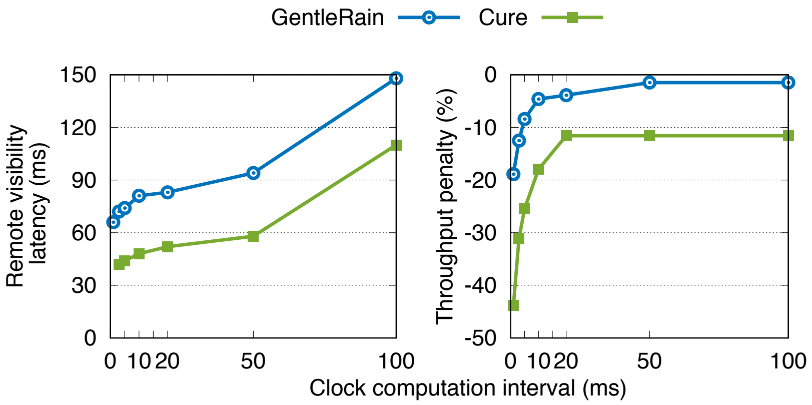


Figure 5.1 – Remote update visibility (left) and throughput penalty (right) exhibited by GentleRain and Cure when varying the time interval between stabilization runs.

estingly, the experiment demonstrates the impact that the time interval between stabilization computations have in the system. To avoid impairing throughput, solutions are forced to pick time intervals large enough such that the impact in throughput is diminished. This is paramount for solutions relying on a vector, such as Cure, as when using very short time intervals the penalization is very significant: of more than 31% when using a time interval of 3ms. Results suggest that the throughput overhead is amortized when using time intervals of about 20ms. Unfortunately, this adds a significant delay to remote visibility latency.

We now describe each of the systems individually, highlighting the particularities of each of them.

GentleRain [49] is a distributed key-value store. Its design assumes that each datacenter replicates the full application state and that this state its equally partitioned at each of them. It assumes an all-to-all communication scheme. GentleRain design goal is to optimize throughput. Thus, it compresses metadata to a single scalar. Unfortunately, this fact has a negative impact on the remote update visibility latency. When using a singe scalar to represent causal dependencies, one is serializing all updates happening in the system (at all datacenters). Thus, GentleRain adds false dependencies not only among updates local to the same datacenter but also across datacenters. Indeed, because of the aggressive metadata compression, the lower-bound remote visibility latency is the latency between the destination datacenter and the datacenter that is the farthest from it, despite the originating datacenter. GentleRain also integrates two read-only transaction primitives: snapshot reads and causally consistent snapshot reads. The former allows clients to read from a consistent snapshot, but it does not guarantee that this is consistent with the client's causal history. Thus, versions that have been previously seen by the client may be excluded from the snapshot, violating causality. The latter guarantees that the snapshot is consistent with the client's causal history. Snapshot reads' protocol is wait-free and single round. Nevertheless, the casually consistent snapshot reads' protocol may block. If the expecting blocking period is longer than a threshold, GentleRain leverages Eiger's read protocol, which completes in a maximum of three rounds. GentleRain relies on loosely synchronized physical clocks.

Okapi [45] is a distributed key-value store. Its design assumes that each datacenter replicates the full application state and that this state its equally partitioned at each of them. It assumes an all-to-all communication scheme. It offers read-only transactions that read from a causally consistent snapshot.

Okapi's stabilization mechanism is slightly different to GentleRain's. At its core, it works similarly. Nevertheless, the condition to make a remote update visible at a datacenter to local clients is different. Unlike previous solutions that make updates visible as soon as their causal dependencies are known to be visible in the local datacenter, Okapi only makes a remote update visible when it is known to have been replicated in the whole system (all datacenters). The goal is to enhance the availability of the system. Upon a datacenter failure, progress is compromised, as the stable time does not advanced due to the failing datacenter. However, unlike previous solution, in Okapi, healthy datacenters may implement a recovery protocol such that the failing datacenter is removed from the system. This is possible because they could have only established dependencies on remote updates originating at failing datacenter that all have received. To orchestrate such a recovery protocol in other solutions would not be trivial, requiring datacenters to propagate remote updates (originated in the failing datacenter) to other healthy datacenters. Unfortunately, enhancing availability comes with the cost of serving a higher remote update visibility latency, as remote updates have to be acknowledged by every datacenter to become visible.

Okapi uses a vector clock with an entry per datacenter to represent causal dependencies. This, in combination with the usage of hybrid clocks [64], permits Okapi to serve read-only transactions efficiently (never blocking) at the cost of delaying the visibility of updates at remote datacenters. Nevertheless, they reduce the computational and storage overhead associated to the metadata size by tagging remote updates with a single scalar. As a consequence, similar to GentleRain, the lower-bound update visibility latency is the latency between the destination datacenter and the datacenter that is the farthest from it, despite the originating datacenter.

Cure [7] is a distributed key-value store. Its design assumes that each datacenter replicates the full application state and that this state its equally partitioned at each of them. It assumes an all-to-all communication scheme. Cure guarantees transactional causal consistency (TCC), the strongest semantics an always-available distributed system can guarantee. It offers interactive transactions that read from a causally consistent snapshot and that respect atomicity: all updates belonging to a transaction are made visible simultaneously, or none does. Cure integrates CRDTs: high-level datatypes with rich confluent semantics, which guarantee convergence. Their protocols rely on loosely synchronized physical clocks. Cure uses a vector with an entry per datacenter to represent causal dependencies. Thus, it capable of exhibiting a lower remote visibility latency at the cost of damaging throughput due to the extra storage and computational overhead. Unlike GentleRain, the lower-bound update visibility latency is the latency between the destination and originating datacenters. Nevertheless, Cure still adds false dependencies among updates originating in the same datacenter, as these are serialized.

5.2.4 Solutions based on lazy resolution

Solutions belonging to the above categories ensure that datacenters only make remote operations visible to local clients when its causal dependencies are known to be already installed locally. Differently, solutions based on lazy resolution allows datacenters to make remote updates visible to local clients even before its causal dependencies are visible locally. Causality is then enforced when a client reads by only returning versions that are causally consistent with client's causal history (what the client has already observed). For this, updates still have to be tagged with some piece of metadata representing update's causal dependencies and clients have to keep track of the updates already observed (causal history). Differently to other solutions [71, 72, 49], clients tag read requests with their causal history. Thus, a server receiving a read request can determine, by comparing the metadata of the locally stored versions with client's causal history, which version should be returned. A server always returns the most up-to-date, or freshest, version that is consistent with client's causal history.

These solutions have three main advantages when compared to previous approaches: (i) do not require expensive mechanisms, such as background stabilization mechanisms or explicit dependency checking messages, to ensure causal consistency, (ii) are resilient to slowdown cascades [6]: when a straggling server affects other healthy servers, and (iii) can potentially reduce remote visibility latency, as operations can be observed by clients even when its dependencies are not installed. Nevertheless, this is a double-edged sword. First, clients may block indefinitely after reading an update whose dependencies have not arrived yet, compromising availability. Imagine the following scenario in which a client reads operation b that depends on another operation a. When the client observed b, a had still not been installed, but it can be read as it does not conflict with client's causal history. Then, the client reads the data item that a updates. In this case, since the client has already read b, the system should return a or a newer update on that data item. The client could be indefinitely blocked waiting for a to arrive. These solutions therefore trade availability (probably the most important requirement of cloud services and the main reason to adopt weaker consistency models) to improve other performance metrics. Second, it adds computation overhead on read operations that otherwise are fairly light (as consistency is enforced when installing remote updates). On each read operation, consistency needs to be checked to ensure that causality is not being violated. Given that cloud services are characterized by read dominant workloads, this overhead may be quite significant.

We now describe each of the systems individually, highlighting the particularities of each of them.

Occult [76] is a distributed key-value store that implements a weaker variant of PSI [92], a relaxation of snapshot isolation [25] specifically crafted for geo-replicated system that allows for asynchronous replication and different ordering of updates across datacenters. Occult leverages a master-slave scheme to make the enforcement of consistency more efficient. Its design assumes that each datacenter replicates the full application state and that this state its equally, logically partitioned at each of them.

Their basic implementation uses a vector clock with an entry per partition to represent causal dependencies. Since they assume a master-slave scheme, only the master accept writes. Therefore, one entry in the vector per partition, despite the number of replicas per partition is enough to capture causality. We refer to this variant as simply *Occult*. This variant still introduces false dependencies among updates local to the same partition.

In order to reduce the metadata size—note that the number of partitions can be very large—they propose three variants. *Occult*$_{opt1}$ bounds to n the total size of the vector by merging entries that are congruent modulo n: one entry in the vector correlates to multiple partitions. Thus, when a client's update depends on two updates, local to two different partitions that share an entry in the vector, this has to use the largest number as dependency, adding a significant amount of false dependencies. *Occult*$_{opt2}$ addresses this problem by, instead of mapping each entry in the bounded vector to roughly the same amount of partitions, letting clients to assign $n - 1$ entries in the dependency vector to the most recent dependencies (largest scalars) and compress the rest in a single entry. The intuition is that larger timestamps are more likely to create false dependencies. They propose a final variant *Occult*$_{opt3}$. This variant aims at further reducing the amount of false dependencies by increasing the metadata size. Authors noticed that the skew between datacenters' clocks and the time that it takes for updates to be replicated exacerbates the impact false dependencies. Assume a client is attached to a datacenter that is the master of a partition p_a and slave of a second partition p_b. The client first updates p_a and immediately after tries to read from p_b. If both partitions are represented by the same entry in the dependency vector, the client will be block until an update, coming from the datacenter that is master of p_b, is replicated in client's datacenter with an associated timestamp at least equal to the one returned when p_a was updated. Authors solve this problem by keeping distinct timestamps for each datacenter.

POCC [93, 94] is a distributed key-value store. Its design assumes that each datacenter replicates the full application state and that this state its equally partitioned at each of them. It assumes an all-to-all communication scheme. It offers read-only transactions that read from a causally consistent snapshot. POCC relies on a vector with an entry per datacenter to represent causal dependencies. Therefore, it introduces false dependencies among updates local to the same datacenter.

5.2.5 Other solutions

Finally, we describe two systems that do not perfectly fit in any of the previous categories, but are related to our work.

Kronos [51] is a generic service that allows to precisely track any partial order, avoiding false dependencies. Kronos flexibility comes at the cost of a centralized implementation that, in geo-replicated settings, forces clients to pay the cost of a potentially large roundtrip to use the service.

EunomiaKV [59] is a causally consistent distributed key-value store. Its design assumes that each datacenter replicates the full application state. It assumes an all-to-all communication scheme.

EunomiaKV introduces a novel service, namely Eunomia. Each datacenter integrates an instance of this fault-tolerant service. Similarly to solutions based on stabilization, the Eunomia service lets local client operations to execute without synchronous coordination, an essential characteristic to avoid limiting concurrency and increasing the latency of operations. Then, in the background, Eunomia establishes a serialization of all updates occurring in the local datacenter in an order consistent with causality, based on timestamps generated locally by the individual servers that compose the datacenter. As this serialization is being generated, Eunomia notifies other datacenters of this order. Based on this information and in the metadata attached to each update, remote datacenters can safely make remote updates visible locally in causal order. EunomiaKV relies on a vector with an entry per datacenter to represent causal dependencies. Therefore, it introduces false dependencies among concurrent updates local to the same datacenter.

5.3 Summary and comparison

In this final section of the chapter, we first summarize the characteristics of previous solutions and briefly compare the existing techniques. Then, we take a closer look to the correlation between metadata size and false dependencies. Finally, we compare them to SATURN.

5.3.1 Summary of existing systems

Table 5.1 summarizes the described systems to simplify its comparison. We characterized them based on our taxonomy: the key technique behind their implementation of causal consistency, the amount of metadata used to represent causal dependencies, the amount of false dependencies introduced due to metadata compression, and whether partial replication is supported or not.

The fact that sequencer-based approaches [98, 81, 102, 43, 26, 65, 9, 103, 61] rely on a sequencer per datacenter, simplifies the implementation of causal consistency. Sequencers allow to aggregate metadata effortlessly, which is key to avoid metadata explosion. Unfortunately, this comes with the cost of limiting intra-datacenter parallelism. The sequencer has to be contacted before requests are processed (on the client's critical operational path), limiting datacenter capacity to the number of requests the sequencer is capable of processing per unit of time. Note that in Table 5.1, both ISIS [26] and SwiftCloud [103] are classified as if they do not support partial replication. ISIS can actually support it by extending its metadata to multiple *vector[dcs]* (one per communication group; a maximum of $2^{dcs} - 1$). SwiftCloud in contrast supports partial replication at the client-side, a challenge we have not addressed.

Other approaches avoid sequencers while tracking dependencies more precisely [71, 72, 18, 75, 47]. Unfortunately, these systems may generate a very large amount of metadata, incurring a significant overhead due to its management costs [49, 16].

As a reaction to the limitations of previous approaches, community has envisioned solutions based on background stabilization mechanisms [49, 45, 7]. This stabilization mechanism runs periodically, coordinating all partitions belonging to the same datacenter, in order to orchestrate when remote updates can safely become visible. From our perspective, this type of solutions are the most scalable and performant solutions of the literature. Unfortunately, as demonstrated in this thesis, minimizing the amount of metadata used to represent causal dependencies is more critical than in other type of solutions. This is due to the associated costs of the background stabilization mechanism.

On the other hand, solutions based on lazy resolution [76, 93, 94] have the potential of improving performance by (i) letting remote updates to be installed at remote datacenters without any consistency check, and making these visible to clients before it is safe (before the causal dependencies of updates are known to be visible locally). Nevertheless, these benefits come with the cost of sacrificing availability, a high cost in our opinion given the cloud services' requirements.

Finally, as mentioned before, Kronos [51] and EunomiaKV [59] do not fit into any of the categories. Kronos is an interesting solution that can cope with composed services—services composed by multiple distributed systems—but it incurs a high cost due to fact of being centralized. EunomiaKV falls between the sequencer-based and the background stabilization techniques. It relies on a per-datacenter service, called Eunomia (also integrated in our prototype), whose goal is to totally order local updates in an order consistent with causality (same goal than sequencers). Nevertheless, Eunomia operates out of the client's operational critical path. This is achieved by relying on a local stabilization mechanism that shares some similarities with the stabilization mechanisms of GentleRain [49], Cure [7] and Okapi [45]. This is an interesting design that permits EunomiaKV to incur a small throughput penalty, similar to GentleRain's, while tracking causal dependencies more precisely (using more metadata), and thus reduce the amount of false dependencies (lowering remote visibility latency). Unfortunately, a priori, EunomiaKV is a less scalable solution than approaches that rely on background stabilization mechanisms. The latter do not exhibit any potential performance bottleneck, while EunomiaKV employs a single receiver per datacenter.

5.3.2 Correlation between metadata size and false dependencies

To better understand the relation between the metadata size and the amount of false dependencies introduced by each of the systems, Figure 5.2 shows the graphic distribution of existing systems (including SATURN) based on these two characteristics. Note that the solutions placed in the same box use the same amount of metadata and introduce the same amount of false dependencies. This does not imply that in practice these solutions will exhibit the same throughput and remote visibility latencies, as performance is also determined by many other characteristics of their design such as the key technique used to ensure causality.

As expected, there is a direct correlation between the metadata size and the amount of false dependencies, which is why most of the solutions stand in the diagonal (colored cells). This is due to the fact that, in all existing solutions, the order in which each datacenter applies remote updates locally must be inferred

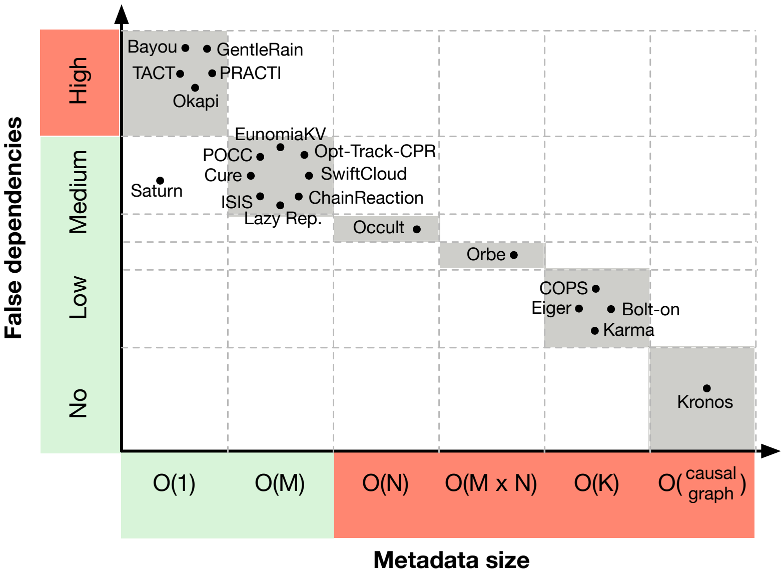


Figure 5.2 – Graphic distribution of existing causally consistent systems based on the metadata size used to capture causal dependencies and the amount of false dependencies that each solution generates. Colored cells represent the diagonal. M, N, and K refers to the number of datacenters, partitions and keys respectively

exclusively from the metadata (unlike in SATURN, where metadata is served in the correct order). Thus, on the one hand, when metadata is aggregated, such as in [49, 98, 43], false dependencies induce poor remote visibilities compared to systems tracking causality more precisely [7, 71, 72, 47, 26, 65]. On the other hand, when metadata is not aggregated, the associated computation and storage overhead has an impact in throughput.

In Figure 5.2, we have colored in green (lightly colored) and red (darkly colored) the different sizes of metadata and types of false dependencies. We consider that metadata sizes of a single scalar or proportional to the number of datacenters are practical. The former is obviously practical as it is the minimum possible size and has the advantage of being constant (independently of the system's scale). The latter is still reasonably practical as one can except between 10 and 20 datacenters at most. Nevertheless, we consider solutions that require more metadata unpractical, as in the best case the metadata size will be proportional to the number of partitions, which is expected to be quite large: from hundreds to thousands of partitions. Regarding the types of false dependencies, we have only colored in red the highest level. This is the level including solutions that exhibit inter-datacenter false dependencies, the most damaging ones (§5.1).

Note that we have left the Full-Track and Opt-Track solutions [61] out of the graph. This is because these are designed to support genuine partial replication, which forced them to use a large amount of metadata. Concretely, they use M times (the total number of datacenters) more metadata than its counterpart Opt-Track-CPR, which is crafted for full replication and generates the same amount of false dependencies. This fact would placed these solutions above the diagonal in the graph.

5.3.3 A comparison with Saturn

As Chapter §4 demonstrates, SATURN operates on a sweet-spot among these approaches. Note that SATURN is the only solution in Figure 5.2 that is below the diagonal.

1. SATURN keeps the size of metadata small and constant, independently of the system's scale (number of datacenters, partitions, etc). Specifically, causal dependencies are tracked with a single scalar. This fact is key to optimize throughput by incurring a negligible computational and storage overhead.

2. Furthermore, SATURN mitigates the impact of false dependencies by using a novel metadata dissemination technique: a set of metadata brokers organized in a tree topology. Concretely, SATURN mitigates the impact of the most damaging type of false dependencies, namely the inter-datacenter, as explained in §5.1. This allows SATURN to achieve significantly lower visibility latencies than other solutions that, like it, rely on a single scalar [49].

3. Finally, SATURN is optimized for partial replication. It enables genuine partial replication, requiring datacenters to only manage the data and metadata of the items replicated locally. Previous solutions either do not ensure genuineness [43, 75, 51], negatively impacting remote visibility latency in partially replicated settings (see §4.4.2); or are genuine [61]

at the cost of increasing significantly the amount of metadata used and consequently penalizing throughput. SATURN is able to achieve genuineness using only one scalar to represent causal dependencies. This is enabled by the tree-based dissemination technique.

	Key Technique	Metadata Size	False Dependencies	Partial Replication	Dissem. Scheme
Bayou [98, 81]	*sequencer*	*scalar* $\mathcal{O}(1)$	*I+P+DC+G*	*no*	*pair-wise*
TACT [102]	*sequencer*	*scalar* $\mathcal{O}(1)$	*I+P+DC+G*	*no*	*pair-wise*
PRACTI [43]	*sequencer*	*scalar* $\mathcal{O}(1)$	*I+P+DC+G*	*non-gen.*	*pair-wise*
ISIS [26]	*sequencer*	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
Lazy Repl. [65]	*sequencer*	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
ChainReaction [9]	*sequencer*	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
SwiftCloud [103]	*sequencer*	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
Opt-Track-CPR [61]	*sequencer*	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
Full-Track [61] **Opt-Track** [61]	*sequencer*	*matrix[dcs][dcs]* $\mathcal{O}(M \times M)$	*P+DC*	*genuine*	*all-to-all*
COPS [71]	*explicit check*	*vector[keys]* $\mathcal{O}(K)$	*I*	*no*	*all-to-all*
Eiger [72]	*explicit check*	*vector[keys]* $\mathcal{O}(K)$	*I*	*no*	*all-to-all*
Bolt-on [18]	*explicit check*	*vector[keys]* $\mathcal{O}(K)$	*I*	*no*	*all-to-all*
Karma [75]	*explicit check*	*vector[keys]* $\mathcal{O}(K)$	*I*	*non-gen.*	*all-to-all*
Orbe [47]	*explicit check*	*matrix[dcs][part.]* $\mathcal{O}(M \times N)$	*P*	*no*	*all-to-all*
GentleRain [49]	*stabilization*	*scalar* $\mathcal{O}(1)$	*I+P+DC+G*	*no*	*all-to-all*
Okapi [45]	*stabilization*	*scalar* $\mathcal{O}(1)$	*I+P+DC+G*	*no*	*all-to-all*
Cure [7]	*stabilization*	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
POCC [93, 94]	*lazy resolution*	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
Occult [76]	*lazy resolution*	*vector[part.]* $\mathcal{O}(N)$	*I+P*	*no*	*master-slave*
Kronos [51]	-	*all* $\mathcal{O}(graph)$	*none*	*non-gen.*	*all-to-all*
EunomiaKV [59]	-	*vector[dcs]* $\mathcal{O}(M)$	*P+DC*	*no*	*all-to-all*
Saturn	-	*scalar* $\mathcal{O}(1)$	*P+DC*	*genuine*	*tree-based*

Table 5.1 – Summary of causally consistent systems. The metadata sizes are computed based on the worst case scenario. M, N, and K refers to the number of datacenters, partitions and keys respectively. I, P, DC and G refers to data-item, partition, intra-datacenter and inter-datacenter false dependencies respectively. These types of false dependencies are described in detail in §5.1.

Chapter 6

Conclusion

In this chapter, we conclude this dissertation by first highlighting the key lessons learnt while designing and implementing SATURN. Second, we list the limitations of our approach. Third, we discuss few directions we explored during the development of this dissertation, in addition to the work described in this document, that had a positive impact on this thesis. Then, we discuss opportunities for future work. Finally, we conclude with a summary of the results obtained during the development of this thesis.

6.1 Aspects to consider when building causally consistent geo-replicated storage systems

During the development of SATURN, we realized a set of aspects that we consider key for someone designing causally consistent geo-replicated storage systems. In this section, we outline four:

Topology-based dissemination. Having a tree to propagate the metadata is key in SATURN's design. As described in this dissertation, a dissemination tree can be used to trivially enforce causal consistency. This enables SATURN to keep the size of the metadata small and constant, despite the number of clients, servers, partitions and replicas; while optimizing the remote visibility latency and supporting genuine partial replication. Interestingly, most of previous solutions rely on pure peer-to-peer metadata dissemination scheme (all-to-all scheme in our nomenclature) that forces them to exclusively rely on the metadata attached to each update to determine the order in which each datacenter must install remote updates. As widely discussed in this dissertation, this forces solutions to choose between compressing metadata, such that the remote visibility latency increases and less scalable partial replication models are allowed; or to keep the size of metadata large such that throughput is significantly penalized. We strongly believe that relying on topology-based dissemination schemes (not only in a tree) has a lot of potential and it is a scheme to consider by future system designers.

Separation of concerns. Our approach opts for decoupling the dissemination of data and metadata, which fundamentally decouples consistency concerns from most of the responsibilities of the underlying storage system, such as replication and durability. We advocate this decoupling in future designs, which brings few advantages when compared to a fully coupled approach. First, it enhances composability. Thus, one could with hopefully little effort use the consistency component in different datastores and viceversa [18]. Second, decoupling can potentially help devising a solution to ensure causal consistency in composed services. Most of the solutions are designed to work under the assumption that there is a unique distributed system. The reality is that many of the modern cloud services are compose by multiple distributed systems. Third, under partial replication (specially if genuineness is not supported), it is fundamental to avoid sending the payload of updates to datacenters that do not replicate the data item being updated. We are not the first to advocate this separation of concerns [51, 18, 43].

Minimum metadata. According to Facebook, the reason causal consistency has not been adopted in production yet, despite having acknowledged its benefits, is because designers are concerned to damage the performance of their system [6]. When experimenting with SATURN and several alternatives in its design, we have noticed that one of the major factors that negatively impact the system's throughput is the size of the metadata used to track causal dependencies. We noticed that as soon as the number of datacenters increases slightly, solutions whose metadata is proportional to the number of datacenters significantly damages throughput; e.g., in a deployment with 7 datacenters, Cure [7], which uses a vector with an entry per datacenter, penalizes the system's throughput up to 12.5% when compared to its counterpart GentleRain that uses a single scalar. Thus, among other design choices, we advocate a solution should prioritize the use of a minimum amount of metadata to capture causal dependencies.

Coordination out of clients' critical path. SATURN integrates Eunomia, a service inside each datacenter responsible for serializing all local updates in an order consistent with causality and push them to SATURN. The design of Eunomia is motivated by the observation that taking the coordination between the datacenter servers and the Eunomia service out of client's critical operational path can potentially bring significant performance gains. The Eunomia's original paper [59] empirically demonstrates the benefits of it by comparing it to a traditional sequencer: Eunomia can handle up to 7.7× more operations than a sequencer. We believe that this is a key design pattern that designers should embrace when possible. When removing coordination from the clients' critical path one is not only taking the contention of the service—or the damage a straggling server may cause—out of client's observable latency, but it is also enabling techniques, such as batching, which are key to enhance performance in real systems.

6.2 Limitations of our approach

We strongly believe that the techniques and solutions described in this dissertation are of relevance. Nevertheless, we identify several limitations, derived from positioning our selves into one extreme of a given tradeoff. In this section, we outline three:

Large number of datacenters, or replicas. SATURN is designed to work very efficiently under the assumption that the system is deployed over a relatively small set of datacenters. Nevertheless, without farther experimentation, we are not sure how SATURN will behave when deployed over a larger amount; e.g., tens of datacenters. Following, we identify the strong and weak points of our techniques regarding to scaling up in the number of datacenters.

On the one hand, SATURN's metadata is small and constant, independently of the system's scale. This means that throughput is likely to remain high, despite an increase in the number of datacenters, as the overhead associated with the processing of metadata is the most significant factor affecting throughput. Other solutions, whose metadata is proportional to the number of servers or datacenters, will have more problems scaling up.

On other hand, we suspect that SATURN will deviate more from the optimal remote visibility latency (the one exhibited by an eventually consistent system) as the number of datacenters increases. Two aspects motivate this observation. First, our configuration generator will have to employ a more restrictive threshold to find a good-enough configuration in a reasonable amount of time. This will lead to a suboptimal configuration that it is possibly less optimal than a configuration generated using a less restrictive threshold or no threshold at all. Second, given the constraint of having a tree topology, even when using the optimal configuration, it is not always possible to optimize all metadata paths (the path between two datacenters when traversing the tree). Specially problematic are those connecting the two extremes of the tree, as a label generated in one extreme has to traverse the whole tree to reach the other extreme (see Figures 4.8 and 4.9b). Intuitively, the more datacenters, the more suboptimal metadata paths with a major deviation from the optimal, which will have an impact on the average remote visibility latency. I would be interesting exploring alternative topologies, e.g.; graphs with cycles or overlapping trees, such that faster paths between pairs of datacenters are established. This nonetheless would require maintaining extra metadata in the serializers (nodes of the topology), causing an extra storage and processing overhead that has to be measured to consider the validity of such approaches.

Remote metadata receivers. Each datacenter receives labels, coming from SATURN, through a single remote metadata receiver. This component is responsible for taking these labels and propagating them to the responsible partition in the local datacenter, ensuring that these are delivered at the partitions in the order that were received at the receiver. This is a quite lightweight procedure, as, unlike most of previous solutions [103, 71, 72], the receiver does not need to do any consistency check. Nevertheless, this receiver is still a potential bottleneck of our design that other solutions, such as those based on background stabilization [49, 7, 45], do not exhibit. It would be interesting to explore techniques to efficiently distribute remote metadata receivers.

Robustness of the serializers tree. In SATURN, the tree is made resilient by replicating (leveraging chain-replication [100]) each tree node. If failures still cause a disconnection in the tree, SATURN falls back into making remote updates visible in label (timestamp) order, which requires a background stabilization mechanism that adds coordination among partitions. While the tree is being repaired, SATURN exhibits significantly higher visibility latencies and lower throughput (due to the costs associated with the stabilization mechanism). It would be interesting to find ways to strengthen the service to reduce the likelihood of being forced to fall back to label order. Similarly to the problem of having long paths when using SATURN in a deployment with a large number of datacenters, the usage of alternative (and more resilient) topologies such as graphs with cycles or overlapping trees could also help to overcome this issue.

Local-area networks. We have experimentally demonstrated that relying on a tree topology to disseminate metadata between datacenters (in a wide-area network) is an efficient technique to mitigate the impact of inter-datacenter false dependencies, the most damaging ones (§5.1). Nevertheless, it is not clear whether this technique would be efficient in other settings, such as local-area networks. Local-area networks e.g., a datacenter network, exhibit latencies of a few millisecond, and a uniform distribution of these among parties. In contrast, in geo-replicated settings, latencies can be of hundreds of milliseconds, having significant differences in the latency exhibited by pairs of datacenters, e.g., two datacenters co-located in the same country and two datacenters, each located in a different continent. The fact that latencies are of at least an order of magnitude lower in local-area networks would make the overheads introduced by our techniques more significant. Moreover, the fact that there are not very significant differences in latencies, makes the tree topology inefficient: for the distant parties (those connected by longer tree-paths), the label propagation time could be several times the data propagation time.

6.3 Other explored directions and collaborations

During the development of this dissertation, we explored other directions, mostly through collaborations, that helped us to gain insights on the design of distributed systems in general and on the challenges of enforcing consistency in geo-replicated systems in particular.

Of special relevance for this thesis is Cure [7]. As greatly discussed in this document, since it is one of state-of-the-art solutions we compared to, Cure is a causally consistent geo-replicated storage system that tracks causal dependencies by means of a vector clock with an entry per datacenter. Designing it, we realized of the practical relevance of the tradeoff between throughput and remote visibility latency. We noticed that when relying on background stabilization mechanisms as the key technique to implement causal consistency, the impact of this tradeoff is quite significant on throughput. This fact motivated us to investigate strategies to reduce this impact without hampering the remote visibility latency.

Other projects [33, 31, 42, 21] gave us a broader understanding of consistency, not only causal consistency, that helped us to better understand the tradeoffs involved in ensuring consistency in replicated systems:

- In [33], we propose a set of high-level datatypes with rich confluent semantics that can be partially replicated (partitionable CRDTs [36, 90]). We built a prototype integrating this into SwiftCloud [103], a causally consistent geo-replicated storage system. This work helped us to better understand the challenges of adopting partial replication under causal consistency.

- In [31], we propose a new type of hybrid clock [64] designed to be exploited by distributed transactional protocols. This work helped us to better understand the tradeoffs of using different types of clocks (physical, logical or hybrid) to ensure consistency in distributed systems.

- In [42], we explore techniques to automatically and dynamically reconfigure quorum-based replication systems. Our approach uses a combination of complementary techniques, including top-k analysis to prioritise quorum adaptation, machine learning to determine the best quorum configuration, and a non-blocking quorum reconfiguration protocol that preserves consistency during reconfiguration. This work helped us to devise the on-line reconfiguration protocols integrated in SATURN and presented in this dissertation.

- In [21], we study how different state dissemination techniques, such as state transfer or operation transfer, have an impact in throughput, network traffic and remote visibility latency in causally consistent systems. We built a system, namely Bendy, that automatically adapts between these techniques based on the application needs and the observed system configuration, such that the best approach is used at each point in time. Specifically, this work helped us to better understand how increases in remote visibility latency have an impact in other performance aspects of the system.

6.4 Future work

We see several interesting directions for future work. All this promising directions have as starting point the lessons learnt when designing and developing SATURN.

6.4.1 Supporting stronger semantics

Our plan is to investigate how to add stronger semantics to SATURN. Specifically, we are interesting in designing protocols to support transactional causal consistency [7] (TCC), the strongest possible semantics that an always-available distributed system can provide.

A TCC system offers a transactional interface, in which developers can modelled their applications by defining a set of multi-key operations. Each of these multi-key operations, or transactions, combines read and write operations. A TCC system guarantees that:

1. Transactions read from a causally consistent snapshot. A snapshot S is causally consistent iff for any two object versions $x_i, y_j \in S$ where x, y are the object identifiers and i, j are the versions, $\nexists x_k$ such that x_k causally precedes y_j and causally follows x_i (denoted as $x_i \rightsquigarrow x_k \rightsquigarrow y_j$). In other words, transactions read from a snapshot of the database system that includes the effects of all transactions that causally precede it. This guarantee simplifies the development of applications. For instance, in the context of social networks, imagine that Alice—a user—has a photo album which is public to everyone. Let us assume that the permissions are stored under a different key that the album and its photos. She decides to change the visibility of the album to *friends-only*, and only then to add new (private) photos. A system that does not support causally consistent snapshot reads may return (after two sequential single-key reads) the new (private) photos to a user Bob that it is no friend of Alice by allowing reading the old permissions and the new state of the album. Under TCC, this is not possible if both keys are read in the same transaction. Thus, Bob would either read the old permissions and the public photos, or the new permission, being unable to have access to Alice's album.

2. All updates composing a transaction occur and are made visible simulta-

neously, ensuring read atomicity [17]. This property is instrumental for ensuring state transitions consistently with respect to certain invariants such as foreign key constrains to represent relationships between data records, secondary indexing to optimize location of partitioned data by attributes, and the maintenance of materialized views.

We plan to integrate these guarantees to SATURN. The goal is to design transactional protocols that add negligible performance overhead. A specially important requirement is that read-only transactions are latency-optimal [73], given the tight latency requirements of cloud services—in which a single read request may fork into thousand of sub-requests [6]; and the negative impact in user engagement and revenue that slight increases in latency carries [46, 89].

6.4.2 Moving towards edge computing

Our plan is to investigate the design of replication protocols to ensure causal consistency on edge networks. The goal is to help developers to program the edge.

Edge computing [83, 79, 27] is a promising computing paradigm which aims at (i) reducing end-user latency, (ii) enhancing scalability, and (iii) enable applications that are latency-sensitive and resource-eiger, such as augmented reality applications [84, 95], by performing data processing at nodes situated at the logical extreme of a network (closer to end-users). An edge network therefore is composed by a set of heterogeneous computing nodes; e.g., points-of-presence, mobile devices, datacenters, and more.

Unlike cloud networks where nodes are resourceful, almost never-failing datacenters, ensuring consistency on edge networks is more challenging: one can expect a large number of nodes, some of which may be severely resource-constrained; nodes may join and leave constant and unexpectedly (high churn); and privacy, security and data integrity issues are a major concern.

To address this challenge we plan to leverage our experience based on building SATURN, which has the following properties that are useful in this context:

- Saturn keeps the metadata size small a constant independently of the system's scale to optimize throughput while simultaneously optimizing remote visibility latencies by using a tree-based dissemination technique.

This is paramount on edge networks as the number of nodes is expected to be large and therefore the metadata size cannot be proportional to the number of nodes.

- Saturn implements genuine partial replication, the most scalable form of partial replication. Adopting partial replication is inevitable in edge computing, given that some edge nodes are typically resource-constrained.

Nevertheless, SATURN's design choices aimed at a smaller set of stable datacenters, and need to be revised to operate in edge networks. We outline a few problems:

- First, the tree-based dissemination is key in SATURN to ensure genuineness and to optimize remote visibility latency while keeping the metadata small and constant. Nevertheless, SATURN's mechanism for building the tree, as discussed in the previous section, has been designed for a handful set of datacenters and will not scale well. We need to envision new ways of building the tree, even if this means finding a suboptimal tree.

- Second, the on-line reconfiguration in SATURN implies rebuilding the whole tree, disrupting end-users, under the assumption that reconfiguration happens very rarely. We need mechanisms to add and remove nodes disrupting the minimum possible the rest of the system; e.g., by applying local changes to the tree, involving a minimum number of nodes. This is paramount when placing replicas in client devices (mobile phones), as one can expect a high churn.

- Third, we need to strengthen SATURN's fault tolerant mechanisms. In SATURN, the tree is made resilient by replicating (leveraging chain-replication [100]) each tree node. We plan to explore alternative (and more resilient) topologies such a graph with cycles or overlapping trees. This nonetheless would require maintaining extra metadata in the nodes of the topology, causing an extra storage and processing overhead.

- Fourth, in a deployment in which user devices—e.g., mobile phones—are part of the edge network, we need to find solutions to guarantee security, privacy and integrity.

- Fifth, nodes may leave unexpectedly (specially when considering user devices as edge components), and with them, updates that have not yet been propagated to other nodes. Thus, we will need to ensure that an update has been replicated in multiple nodes (or at least in some "stable" node) before letting other clients depend on it. Otherwise, clients may be blocked forever or have to restart their session.

We plan to first address a simpler problem in which we assume that edge devices are points-of-presence controlled by the services' providers with similar guarantees than datacenters but less resources. This simplifies the problem by making the third, fourth and fifth points less critical and less challenging. The plan is to eventually devise a solution that also considers end-user devices; e.g., smartphones or tablets, as edge devices. This is a more challenging setting, as the total number of nodes will increase significantly, as well as the churn rate (the rate in which nodes join and leave) and the security, privacy and integrity concerns.

6.4.3 Coping with composed services

Modern cloud services are increasingly built on top of multiple subsystems: storage systems, monitoring systems, processing systems; in an effort to make cloud services modular, and therefore easier to maintain and optimize. Interestingly, even if each of the subsystems ensures causal consistency, or stronger consistency criteria such as linearizability, consistency violations may still occur [104]. Thus, current solutions, which are designed assuming a single replicated system (usually a geo-replicated storage system)[1], must be revisited.

Our plan is to investigate how the ideas behind SATURN can be useful to address the problem of ensuring causal consistency for composed services. We believe SATURN is a good starting point: (i) it decouples the management of consistency from other concerns (by decoupling metadata from data dissemination), a key characteristic that a solution for composed services should have; (ii) it efficiently supports partial replication, key in composed services as not all subsystems will necessarily interact with all other subsystems. Nevertheless, we identify new challenges unaddressed in this thesis.

[1] Except Kronos [51] that, as SATURN, decouples the metadata from the data management. Nevertheless, Kronos, unless SATURN, is centralized, imposing high latencies in geo-replicated settings.

First, in our work, we assume that the replication groups of each key are known by the metadata service in order to ensure genuineness. Even when this is not ensured, our solution ensures progress (but not genuineness) under the assumption that a receiver datacenter can discard labels corresponding to data items that are not replicated locally. These assumptions are too strong for composed services. In these settings, one may not know a priori which subsystems an event may reach. Note that if a subsystem replica receives a label, whose corresponding payload is not meant to be received, this is blocked forever (being unable to install remote operations), unless the label is ignored.

Second, events originating at a source subsystem replica may fork and mutate throughout the way before reaching one of its final destinations. For instance, a user of a social network Alice that writes a private message to a second user Bob. This event needs to (i) be stored in the local replica of the geo-replicated storage system; (ii) be replicated to possibly a subset of the storage system replicas; and (iii) mutate into a notification event that is handled by the notification service replica in charge of notifying Bob. We believe this type of scenarios bring challenges in the consistency management—e.g., identification and timestamping of events—unsolved by our work.

6.5 Final remarks

In this thesis, we have investigated the tradeoff inherent to causally consistent replicated data services between throughput and remote visibility latency derived from the granularity in which causal dependencies are tracked. We have studied its impact not only under full replication, which assumes that the application state is fully replicated at all datacenters, but also under partial replication, a more challenging setting that it is gaining prominence.

We have proposed a set of techniques and mechanisms that combined enable data services to upgrade their consistency guarantees to causal consistency. The key proposed technique of this thesis is a novel metadata dissemination service. This service leverages a set of metadata brokers, geographically distributed and organized in a tree topology to disseminate causal metadata among datacenters. This permits solutions to ensure genuine partial replication; and to optimize remote visibility latency while using small and constant pieces of metadata—independently of the system's scale—imperative to avoid impairing throughput. Furthermore, we advocate decoupling metadata dissemination from the data dissemination, which is key to enhance composability and to make metadata brokers as light as possible. Finally, the metadata dissemination service requires to be notified by each datacenter of a total order, consistent with causality, of the updates issued locally. In this thesis, we have studied the integration into our architecture of metadata serialization services able to achieve this goal very efficiently by operating out of clients' critical operational path. Concretely, we have demonstrated how to integrate an existing metadata serialization service, namely Eunomia, with the metadata dissemination service and the rest of the intra-datacenter components.

We have presented our prototype, namely SATURN, a distributed metadata service that integrates all the mentioned techniques. Among many experiments evaluating each of the techniques individually, we have shown that SATURN, when attached to a data service, exhibits a throughput comparable (only 2% overhead on average) to systems providing almost no consistency guarantees. At the same time, SATURN mitigates the impact of false dependencies—unavoidably introduced when compressing metadata—by relying on a metadata dissemination service that can be configured to match optimal remote visibility latencies. These results confirm that the techniques proposed are effective.

Bibliography

[1] Facebook's top open data problems. https://research.fb.com/facebook-s-top-open-data-problems/. Accessed: 2018-03-07.

[2] Google Cloud Platform. `https://cloud.google.com/`.

[3] Riak KV. `https://github.com/basho/riak_kv`.

[4] M. Adelson-Velskii and E. Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.

[5] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[6] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HotOS'15, pages 13–19, Kartause Ittingen, Switzerland, 2015.

[7] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceeding of the IEEE 36th International Conference on Distributed Computing Systems*, ICDCS'16, pages 405–414, Nara, Japan, 2016.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM*

Conference on Data Communication, SIGCOMM ’08, pages 63–74, Seattle, WA, USA, 2008.

[9] S. Almeida, J. a. Leitão, and L. Rodrigues. ChainReaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys ’13, pages 85–98, Prague, Czech Republic, 2013.

[10] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP ’09, pages 1–14, Big Sky, Montana, USA, 2009.

[11] Application and environment requirements. Deliverable: Natural language requirements. SyncFree Project, April 2015.

[12] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD ’13, pages 1185–1196, New York, New York, USA, 2013.

[13] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC ’15, pages 385–394, Donostia-San Sebastián, Spain, 2015.

[14] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, Nov. 2013.

[15] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15, pages 1327–1342, Melbourne, Victoria, Australia, 2015.

[16] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC ’12, pages 22:1–22:7, San Jose, California, 2012.

[17] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with ramp transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 27–38, Snowbird, Utah, USA, 2014.

[18] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, New York, USA, 2013.

[19] P. Bailis and K. Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, Sept. 2014.

[20] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, Bordeaux, France, 2015.

[21] C. Bartolomeu, M. Bravo, and L. Rodrigues. Dynamic adaptation of geo-replicated crdts. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 514–521, Pisa, Italy, 2016.

[22] Basho. Basho Bench.
`http://github.com/basho/basho_bench`.

[23] Basho. Riak core.
`http://github.com/basho/riak_core`.

[24] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 49–62, Chicago, Illinois, USA, 2009.

[25] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, San Jose, California, USA, 1995.

[26] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3), Aug. 1991.

[27] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, Helsinki, Finland, 2012.

[28] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues. On the use of clocks to enforce consistency in the cloud. *IEEE Data Engineering Bulleting*, 38(1):18–31, 2015.

[29] M. Bravo, L. Rodrigues, and P. Van Roy. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, Middleware Doct Symposium '15, pages 5:1–5:4, Vancouver, BC, Canada, 2015.

[30] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, Belgrade, Serbia, 2017.

[31] M. Bravo, P. Romano, L. Rodrigues, and P. Van Roy. Reducing the vulnerability window in distributed transactional protocols. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 10:1–10:4, Bordeaux, France, 2015.

[32] E. A. Brewer. Towards robust distributed systems. In *Keynote at the ACM Symposium on Principles of Distributed Computing*, PODC, 2000.

[33] I. Briquemont, M. Bravo, Z. Li, and P. Van Roy. Conflict-free partially replicated data types. In *Proceedings of the 7th International Conference on Cloud Computing Technology and Science*, CloudCom '15, pages 282–289, Vancouver, BC, Canada, 2015.

[34] A. Brodersen, S. Scellato, and M. Wattenhofer. Youtube around the world: Geographic popularity of videos. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 241–250, Lyon, France, 2012.

[35] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *12th Euromicro Conference on Parallel,*

Distributed and Network-Based Processing, 2004. Proceedings., pages 152–158, 2004.

[36] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, San Diego, California, USA, 2014.

[37] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, New Orleans, Louisiana, USA, 1999.

[38] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[39] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[40] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP '93, pages 44–57, Asheville, North Carolina, USA, 1993.

[41] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Hollywood, CA, USA, 2012.

[42] M. Couceiro, G. Chandrasekara, M. Bravo, M. Hiltunen, P. Romano, and L. Rodrigues. Q-opt: Self-tuning quorum system for strongly consistent software defined storage. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 88–99, Vancouver, BC, Canada, 2015.

[43] M. Dahlin, L. Gao, A. Nayate, A. Venkataramana, P. Yalagandula, and J. Zheng. Practi replication. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '06, 2006.

[44] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, Washington, USA, 2007.

[45] D. Didona, K. Spirovska, and W. Zwaenepoel. Okapi: Causally consistent geo-replication made faster, cheaper and more available. *Arxiv preprint arXiv:1702.04263*, Feb. 2017.

[46] P. Dixon. Shopzilla site redesign: We get what we measure. In *Velocity Conference Talk*, 2009.

[47] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, Santa Clara, California, 2013.

[48] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems*, Braga, Portugal, 2013.

[49] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the 5th ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, Seattle, WA, USA, 2014.

[50] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone. Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Atlanta, Georgia USA, 2014.

[51] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer. Kronos: The design and implementation of an event ordering service. In *Proceedings of the*

9th European Conference on Computer Systems, EuroSys '14, pages 3:1–3:14, Amsterdam, The Netherlands, 2014.

[52] D. K. Gifford. *Information Storage in a Decentralized Computer System.* PhD thesis, Stanford, CA, USA, 1981. AAI8124072.

[53] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[54] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, Toronto, Ontario, Canada, 2011.

[55] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, SIGCOMM '09, pages 51–62, Barcelona, Spain, 2009.

[56] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Trade-offs in replicated systems. *IEEE Data Engineering Bulletin*, 39:14–26, 2016.

[57] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1):297–316, 2001.

[58] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, Michigan, USA, 1978.

[59] C. Gunawardhana, M. Bravo, and L. Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 83–95, Santa Clara, CA, USA, 2017.

[60] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[61] T.-Y. Hsu, A. Kshemkalyani, and M. Shen. Causal consistency algorithms for partially replicated and fully replicated systems. *Future Generation Computer Systems*, 2017.

[62] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '13, pages 3–14, Hong Kong, China, 2013.

[63] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[64] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Proceedings of the 18th Internaltional Conference on Principles of Distributed Systems*, OPODIS '14, 2014.

[65] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[66] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[67] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978.

[68] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[69] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 265–278, 2012.

[70] G. Linden. Make data useful.
`http://www.gduchamp.com/media/`
`StanfordDataMining.2006-11-28.pdf`.

[71] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage

with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, 2011.

[72] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 313–328, 2013.

[73] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 135–150, Savannah, GA, USA, 2016.

[74] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. *Technical Report TR-11-21, University of Texas at Austin, Austin, Texas*, 2011.

[75] T. Mahmood, S. Puzhavakath Narayanan, S. Rao, T. Vijaykumar, and M. Thottethodi. Achieving causal consistency under partial replication for geo-distributed cloud storage. Technical report, Department of Electrical and Computer Engineering Technical Reports, 2016.

[76] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, 2017.

[77] NTP. The network time protocol.
`http://www.ntp.org`.

[78] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, Toronto, Ontario, Canada, 1988.

[79] M. Patel, Y. Hu, P. Heédeé, J. Joubert, C. Thornton, B. Naughton, J. Ramos, C. Chan, V. Young, S. Tan, D. Lynch, N. Sprecher, T. Musiol, C. Manzanares, U. Rauschenbach, S. Abeta, L. Chen, K. Shimizu,

A. Neal, P. Cosimini, A. Pollard, and G. Klas. Mobile-edge computing - introductory technical white paper. Technical report, European Telecommunications Standards Institute (ETSI), Sept. 2014.

[80] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 455–465, Atlanta, GA, USA, 2012.

[81] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, Saint Malo, France, 1997.

[82] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '10, pages 375–386, New Delhi, India, 2010.

[83] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct. 2009.

[84] M. Satyanarayanan, P. B. Gibbons, L. Mummert, P. Pillai, P. Simoens, and R. Sukthankar. Cloudlet-based just-in-time indexing of IoT video. In *Proceedings of the 2nd Global Internet of Things Summit (GIoTS)*, GIoTS '17, pages 1–8, Geneva, Switzerland, June 2017.

[85] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 2010 IEEE 29th International Symposium on Reliable Distributed Systems*, SRDS '10, pages 214–224, New Delhi, Punjab, India, 2010.

[86] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984.

[87] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[88] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[89] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Velocity Web Performance and Operations Conference*, San Jose, CA, USA, 2009.

[90] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Grenoble, France, 2011.

[91] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[92] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, Cascais, Portugal, 2011.

[93] K. Spirovska, D. Didona, and W. Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. In *Proceeding of the IEEE 37th International Conference on Distributed Computing Systems*, ICDCS'17, pages 2626–2629, Atlanta, GA, USA, 2017.

[94] K. Spirovska, D. Didona, and W. Zwaenepoel. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. Technical report, EPFL, oai:infoscience.epfl.ch:225991, 2017.

[95] C. Streiffer, A. Srivastava, V. Orlikowski, Y. Velasco, V. Martin, N. Raval, A. Machanavajjhala, and L. P. Cox. ePrivateeye: To the edge and beyond! In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 18:1–18:13, San Jose, California, 2017.

[96] O. Team. Oscar: Scala in or.
`https://bitbucket.org/oscarlib/oscar`.

[97] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Sep 1994.

[98] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, Copper Mountain, Colorado, USA, 1995.

[99] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 315–326, New Delhi, India, 2010.

[100] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '04, 2004.

[101] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks*, WOSN '09, pages 37–42, Barcelona, Spain, 2009.

[102] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, San Diego, California, 2000.

[103] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, Vancouver, BC, Canada, 2015.

[104] I. Zhang, N. Lebeck, P. Fonseca, B. Holt, R. Cheng, A. Norberg, A. Krishnamurthy, and H. M. Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *Proceedings of the*

12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pages 723–738, Savannah, GA, USA, 2016.