Colony: A Hybrid Consistency System for Highly-Available Collaborative Edge Computing

Ilyas Toumlilt

Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

COLONY: A Hybrid Consistency System for Highly-Available Collaborative Edge Computing

---

Ilyas Toumlilt

Soutenue publiquement le : *21 Décembre 2021*

Devant un jury composé de :

Sebastien MONNET, Professeur, Université Savoie Mont Blanc	*Rapporteur*
Etienne RIVIÈRE, Professeur, UCLouvain	*Rapporteur*
Sonia BEN MOKHTAR, Directrice de Recherche, LIRIS, CNRS	*Examinatrice*
Annette BIENIUSA, Maîtresse de Conférences, University of Kaiserslautern	*Examinatrice*
Valerie ISSARNY, Directrice de Recherche, Inria	*Examinatrice*
Pierre SUTRA, Maître de conférences, Télécom SudParis	*Invité*
Marek ZAWIRSKI, Software Engineer, Google	*Invité*
Marc SHAPIRO, Directeur de Recherche, Sorbonne Université, LIP6, Inria	*Directeur de thèse*

À Salma, Latifa et Hassan

Remerciements

"*Il va falloir que tu apportes ta tasse de café*", m'avaient conseillé ***Maxime*** et ***Florian***, pendant que ce dernier libérait son bureau pour me léguer ce qui allait devenir mon espace de travail durant les 5 années qui suivirent, et m'invitèrent ensuite à ma toute première *pause café*. Je compris très vite l'importance de ces moments d'échange entre collègues, mais j'étais loin de me douter du soutien capital que ces discussions allaient m'apporter sur le plan scientifique comme émotionnel, loin de réaliser qu'une pause café en conférence pouvait dessiner ma contribution principale, nouer des collaborations, et à quel point la communication est un pilier fondamental à l'effort collectif qu'est la recherche, loin de penser que j'allais devenir fournisseur officiel de ce breuvage psychotrope dans mon équipe. C'est donc tout naturellement que je souhaite remercier ici toutes les personnes qui ont accompagné ce long périple que fut ma thèse, par leur soutien, leurs encouragements ou leur jus de neurones.

Tout d'abord, je tiens à remercier mes rapporteurs, ***Sébastien Monnet*** et ***Étienne Rivière***, pour le temps qu'ils ont consacré à la lecture attentive et à l'évaluation de ce rapport. Merci également à tous les autres membres du jury, ***Sonia Ben Mokhtar***, ***Annette Bienuisa***, ***Valérie Issarny***, ***Pierre Sutra*** et ***Marek Zawirski***.

Ensuite, cette thèse n'aurait pas été possible sans la supervision de ***Marc Shapiro***. Marc, merci à toi pour ta confiance, de m'avoir accompagné et suivi toutes ces années, d'avoir été suffisamment strict pour orienter mes idées, mais suffisamment souple pour me laisser maîtriser ma thèse, ton expertise en matière de rédaction de papiers de recherche a rendu cette thèse plus facile qu'elle n'aurait pu l'être. Malgré tes occupations de plus en plus nombreuses, tu as toujours été disponible quand j'en avais besoin, merci pour ces innombrables *meetings*, parfois même à des heures improbables. Je ne serai jamais assez reconnaissant pour tout ce que j'ai appris à tes côtés, mais je m'efforcerai de garder la rigueur que tu m'as enseignée pour faire de la recherche en systèmes, et j'espère que je serai à la hauteur de tes attentes.

J'aimerais également adresser un remerciement très particulier à ***Pierre Sutra***, qui n'était pas officiellement mon encadrant, mais qui a suivi mes travaux avec assez de recul pour m'apporter le bon sens nécessaire à leur publication. Son esprit critique et sa capacité à proposer des pistes de recherche prometteuses furent les bienvenues à de nombreuses reprises.

Je souhaite ensuite remercier les membres du ***LIP6***, en particulier les permanents de l'équipe ***Delys***, d'abord en tant qu'enseignants ayant contribué à mon intérêt pour les systèmes, puis en tant que collègues auprès de qui j'ai eu la chance de travailler. Merci ***Pierre Sens***, pour tes nombreux conseils, pour nos échanges toujours source d'encouragement et d'optimisme. Merci ***Jonathan Le Jeune***, d'être le reflet de la convivialité du Nord. Merci ***Julien Sopena***, pour ton dévouement en tant qu'enseignant, ta confiance en mon enseignement, tes précieux conseils, et pour l'étendue de ta curiosité et ton esprit critique qui ont souvent enrichi nos discussions, tant sur le plan scientifique qu'humain.

Viennent ensuite les camarades doctorants des équipes Delys, MoVe et Whisper. En commençant par les plus anciens, ceux qui ont facilité mon intégration. Merci ***Antoine Blin***, mon premier mentor et notre expert du temps-réel; ***Gauthier Voron***, qui apprécie autant les hyperviseurs que la beauté de Perl; ***Damien Carver***, spécialiste des conteneurs, des tableaux et des sports de glisse; ***Hakan Metin***, qui sait tout expliquer par 1-2-3-4; ***Florent Coriat***, notre expert et rescapé réseau; Ainsi qu'à toute la *génération Regal*: ***Maxime Lorrillere, Rudyar Cortés, Lyes Hamidouche, Alejandro Tomsic***. Ensuite je souhaite remercier celles et ceux qui ont partagé mon quotidien au labo, en commençant par mon voisin de bureau, ***Francis Laniel***, le plus sage des Stéphanois; ***Saalik Hatia***, ses 404 cellulaires portatifs et ses Tech Tips; ***Jonathan Sid-Otmane***, sa grande curiosité tech, sauf pour la 5G; ***Vincent Vallade***, champion du monde du SAT; ***Lucas Serrano***, Légion Döner du python; ***Arnaud Favier***, premier doctorant système à promouvoir le web dev; ***Laurent Prosperi***, seul camarade comprenant l'intérêt de la *VO2 Max*; ***Ludovic Le Frioux***, amateur du Clash of Code; Mais également à toute la génération ***Dimitrios Vasilas, Gabriel Le Bouder, Sreeja Nair, Daniel Wilhelm, Benoît Martin, Sara Hamouda, Guillaume Fraysse, Marjorie Bournat, Élise Jeanneau, Mathieu Lehaut, Cédric Courtaud, Darius Mercadier, Yoann Ghigoff*** avec qui j'ai partagé maintes conversations passionnantes. J'aimerais également souhaiter plein de courage à la *génération Adum*, ***Célia Mahamdi, Aymeric Agon-Rambosson*** et ***Étienne Le Louët***.

Je souhaite remercier tout particulièrement, ***Maxime Bittan*** et ***Redha Gouicem***, que je suis heureux de compter parmi mes amis depuis les tout premiers cours à l'université et jusqu'au laboratoire, merci pour les très bons moments (d'informatique, comme de rires) passés ensemble.

Ma vie à l'université n'aurait pas été la même sans les bons moments passés au ***DAPS***, à l'***AS***, à l'***AEIP6/ALIAS*** et toutes les activités proposées par l'écosystème associatif de ***Sorbonne Université***. L'université m'a également permis de coupler mes études à ma pratique sportive qui m'a souvent aidé à tenir mentalement durant cette thèse,

j'en profite pour remercier mon club d'athlétisme, l'***AO Charenton***, et ***adidas***. Le lien fort entre co-équipiers, la discipline à toute épreuve et la persévérance, sont des valeurs fondamentales qu'on retrouve en athlétisme comme en doctorat.

Ces années de thèse auraient été infiniment moins agréables et enrichissantes sans mes amis, de ***Gaillon*** à ***Paris***, du ***Kop Sud*** au ***Central***. Merci les copains.

Et enfin, je tiens à remercier ***ma famille*** pour son soutien indéfectible depuis toujours. Maman, Papa, les mots ne suffiraient pas pour éprouver toute ma reconnaissance pour tout ce que vous avez fait pour moi. Grâce à vous, j'ai pu étudier ce domaine qui me passionne et m'investir dans cette aventure de longue haleine. Grâce à vous, je suis devenu l'homme que je suis aujourd'hui. Merci pour votre confiance et le confort que vous m'avez offert pour mener à bout mon projet d'études. J'espère que vous êtes aujourd'hui fiers des résultats de mon parcours.

Ces remerciements ont été bien plus longs et dénués de blagues que ce que j'aurais espéré, mais croyez-moi, j'ai éprouvé bien plus d'émotions et de nostalgie que je n'ai pu le résumer ici, et si par mégarde j'ai oublié quelqu'un, veuillez m'en excuser.

Merci encore, à toustes, même celleux qui ne consomment pas de caféine.

Abstract

Immediate response, autonomy and availability is brought to edge applications, such as gaming, cooperative engineering, or in-the-field information sharing, by distributing and replicating data at the edge. However, application developers and users demand the highest possible consistency guarantees, and specific support for group collaboration. To address this challenge, COLONY guarantees Transactional Causal Plus Consistency (TCC+) globally, dovetailing with Snapshot Isolation within edge groups. To help with scalability, fault tolerance and security, its logical communication topology is tree-like, with replicated roots in the core cloud, but with the flexibility to migrate a node or a group. Despite this hybrid approach, applications enjoy the same semantics everywhere in the topology. Our experiments show that local caching and peer groups improve throughput and response time significantly, performance is not affected in offline mode, and that migration is seamless.

Keywords: Causal Consistency, Collaborative Computing, Edge Computing, Geo-Replication, Peer-to-Peer Systems

Résumé

La distribution et la réplication des données en périphérie du réseau apportent une réponse immédiate, une autonomie et une disponibilité aux applications de périphérie, telles que les jeux, l'ingénierie coopérative ou le partage d'informations sur le terrain. Cependant, les développeurs d'applications et les utilisateurs exigent les meilleures garanties de cohérence possibles, ainsi qu'un support spécifique pour la collaboration de groupe. Pour relever ce défi, Colony garantit la cohérence Transactional Causal Plus Consistency (TCC+) à échelle planétaire, en complément de l'isolation des instantanés au sein des groupes de périphérie. Pour favoriser le passage à l'échelle, la tolérance aux pannes et la sécurité, sa topologie de communication logique est arborescente, avec des racines répliquées dans le nuage principal, mais avec la possibilité de migrer un nœud ou un groupe. Malgré cette approche hybride, les applications bénéficient de la même sémantique partout dans la topologie. Nos expériences montrent que la mise en cache locale et les groupes collaboratifs améliorent considérablement le débit et le temps de réponse, que les performances ne sont pas affectées en mode hors ligne et que la migration est transparente.

Mots-clés: Cohérence Causale, Systèmes Collaboratifs, Informatique au bord du réseau, Géo-Réplication, Systèmes Pair-à-Pair

Contents

1 Introduction

Internet-scale collaboration is a growing application area, as evidenced by games such as Overwatch or Ingress, shared editors such as Google Docs, Microsoft Office 365 or Apple iWork, or file-sharing systems such as Dropbox or Nextcloud.

Mobile devices with Augmented Reality capabilities support location-aware games such as Pokémon Go and Harry Potter Unite, or collaborative 3D modeling and manufacturing applications [Wan+19; Tak18; SRS21]. Virtual worlds, or the Metaverses [DIG13], constitute a subset of collaborative virtual reality applications. Metaverses are persistent online computer-generated environments where multiple users in remote and close physical locations can interact in real time for the purpose of work or play. Many industries are today building Metaverse applications such as Facebook (Meta) [Fac21], Microsoft's metaverse work spaces [Mic21] or the NVIDIA Omniverse platform [NVI21], and many more industry companies [DIG13]. Those augmented reality worlds generate massive computation for space recognition (e.g., using a smartphone camera to scan relevant objects around the user) on the edge devices, and expect real-time collaboration.

Existing systems are cloud-based, sometimes providing ad hoc application-level caching. Consistency violations are common, baffling users and vexing application developers [TSR15; CNS19; Goo21a; Goo21b; www21]. Support for offline operation is spotty. Users interact through the cloud only, even when direct communication would be possible. These systems lack collaboration features such as group management or versioning.

Cloud computing is an established paradigm and most collaborative applications today, including those running on mobile devices, rely on the cloud to exchange data, download information or offload resource-intensive computations to run on more powerful servers and save battery power. Massively multiplayer mobile games are an example of a large-scale collaborative application requiring low latency between players and generating resource-intensive computation, especially with the advent of augmented reality, which often involves image processing for face recognition or object detection. However, for these applications to be usable, the expected response must be under 5 milliseconds, which is difficult to guarantee when an edge device has to synchronize via remote data centers.

1.1 Overview

In the first part of this thesis, we explore the edge computing paradigms and the challenges that are inherent to its network and architecture. Then, we discuss the safety, scalability, security, hierarchy and programmability requirements for collaborative edge applications. And finally, we present a comprehensive study of the state of the art, and what has been done in the literature to fulfill the edge computing requirements and collaborative applicative requirements.

In the second part of the thesis, we present our main contribution, COLONY, a database and middleware designed to address these issues. A top requirement is an *edge-first* approach [Kle+19] that locates data on the device, to provide availability, fast and seamless response independently of network connectivity and of location, and so that users have ownership of their data. However, this makes it challenging to satisfy consistency and freshness expectations. To answer this challenge, COLONY takes a hybrid approach, and provides the highest consistency guarantees compatible with availability (TCC+), strengthened further (to SI) in well-connected zones. CRDT data types ensure convergence without rollbacks. A related challenge is the overhead of concurrency metadata (fat vector clocks), which we limit thanks to a flexible forest topology and to SI zones.

To address the requirements of group collaboration, COLONY enables edge collaborators to share data without relying on the cloud, and supports collaborative security features, including end-to-end encryption. Our design provides uniform access to data across a spectrum spanning core cloud to the far edge.

Finally, this thesis addresses a number of design and implementation challenges, including disconnected operation, consistency under migration, total-order consensus at the edge, and avoiding single points of failure despite the tree topology.

1.2 Contributions

The main results of this dissertation are as follows:

- A decentralized database architecture, designed for collaborative applications, that provides a continuum spanning from the core cloud to the far edge.
- A hybrid consistency model, based on causal and transactional guarantees globally, strengthened to total-order consistency in edge collaboration groups and in geographical proximity.

- A scalable metadata and topology design that bounds the overhead of the required consistency metadata, and that supports seamless disconnection or migration of a node or of a whole group.
- A novel approach to access control that leverages the consistency model.
- Efficient design and implementation of COLONY and its experimental evaluation demonstrating the benefits of our approach.

Our experimental evaluation shows that: local and group caching improve throughput by 1.4× and 1.6× respectively, and response time by 8× and 20×, compared to a classical cloud configuration; performance in offline mode remains the same as online; both the offline/online transition and migration are seamless.

1.3 Publications

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

- Ilyas Toumlilt, Pierre Sutra, Marc Shapiro. Highly-Available and Consistent Group Collaboration at the Edge with Colony. Middleware 2021 - 22nd International Middleware Conference, Dec 2021, Québec / Virtual, Canada. ⟨10.1145/3464298.3493405⟩. ⟨hal-03353663v3⟩ [TSS21]
- Ilyas Toumlilt, Alejandro Tomsic, Marc Shapiro. Vers une cohérence causale évolutive sans chaînes de ralentissements. Compas 2018: Conférence d'informatique en Parallélisme, Architecture et Système, Jun 2018, Nice Sophia-Antipolis, France. ffhal-01860334f [TTS17]

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

- Ali Shoker, Paulo Sergio Almeida, Carlos Baquero, et al. LightKone Reference Architecture (LiRA). https://www.info.ucl.ac.be/ pvr/LiRAv0.9.pdf. 2020 (cit. on p. 3). [Sho+20]
- LightKone H2020 Project. D6.1: New concepts for heavy edge computing. 2018 (cit. on p. 4) [Pro18]
- LightKone H2020 Project. D6.2: New concepts for heavy edge computing. 2019. [Pro19]

1.4 Organization

This thesis is divided into three parts. The rest of this document is organized as follows:

- Part I introduces the common background of our work, formulate the problem, presents the existing solutions and discusses the use-case requirements, this part is divided into three chapters:
  - Chapter 2 provides a complete and up-to-date review of edge-oriented computing systems by aggregating relevant challenges on their architecture features, management approaches, and design goals.
  - Chapter 3 presents a use-case point of view of the previous review, discussing the safety, scalability, security, hierarchy and programmability requirements that need to be guaranteed for collaborative edge applications.
  - Chapter 4 studies and compares the solutions that have been designed in the state of the art of edge storage systems to answer both challenges and requirements presented before, and discusses the remaining challenges that need to be addressed.
- Part II, in light of what we saw in the existing work, we will here justify some protocol choices used in our approach, which aims to reduce shortcomings observed in some existing systems. And present the design, implementation and the programming model of our main contribution, COLONY, a database and middleware designed to address requirements of group collaboration.
- Part III provides an experimental evaluation demonstrating the benefits of our approach, compared to other solutions from the state of the art.
- the last part IV we summarize our contribution, and present our vision for the future requirements towards more reliable, scalable and secure collaborative edge storage systems.

Part I

Background

2 Edge Computing Paradigms

Although cloud computing has brought paradigm shifts to Internet services, changing the way we live, work and study since its inception around 2005 [Arm+10], it suffers from some problems inherent to its nature such as bandwidth bottlenecks, communication overhead, and location ignorance. The concept of fog/edge computing aims to extend the services from the core in cloud data centers to the edge of the network. In recent years, many systems were proposed to better serve ubiquitous smart devices closer to the user.

Edge Computing [Shi+16] is a model of distributed computing that aims to leverage the processing capacity that exists in devices that operate at the border of the network. Edge Computing reinforces the Cloud computing paradigm, where all processing is done in data centers, to overcome the problems and limitations it suffers from. In the case of edge computing, processing is done cooperatively by edge devices and cloud servers. The computations performed on a particular component depends on a number of factors, including available ressources, node capacity and latency requirements.

This section provides a complete and up-to-date review of edge-oriented computing systems by comparing relevant proposals according to their architecture features, management approaches, and design goals.

2.1 The limits of the Cloud Computing Model

Unfortunately, the cloud computing model is increasingly difficult to collectively guarantee "anywhere, anytime" properties due to the exponential growth in data generation and the processing requirements of many emerging applications, such as applications in the Internet of Things (IoT) or highly collaborative applications in Augmented Reality and Video Games, which produce large amounts of data that must be processed in useful, and sometimes small, time windows. According to IBM [IBM16], 90 percent of the data on Earth was created in the last two years; the IDC analysis [bus15] estimates that IoT workloads will grow by nearly 750 percent between 2014 and 2019, generated by tens of billions of smart devices.

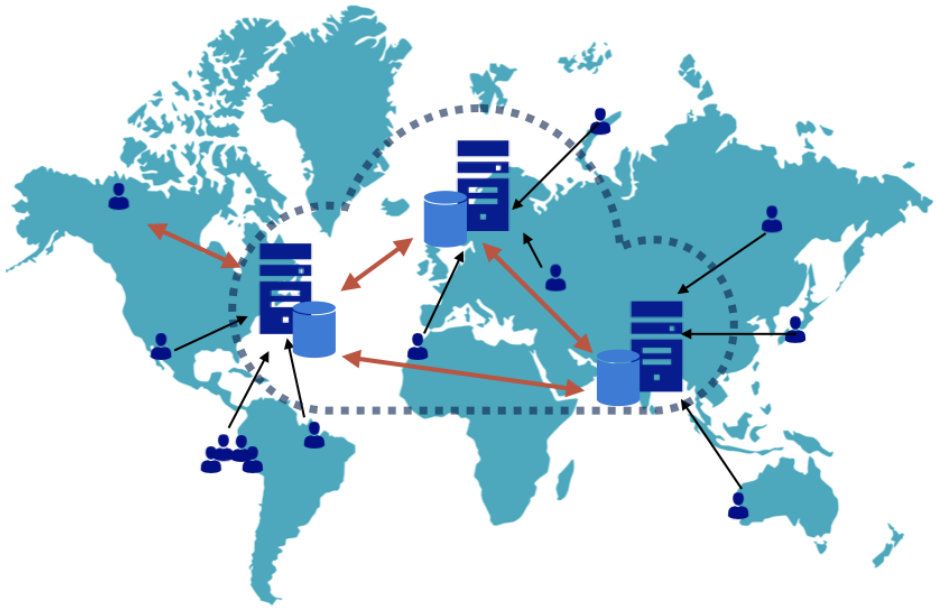

Fig. 2.1.: An example of the cloud computing model architecture, data is geo-replicated over three datacenters, which constitute the Cloud part of the system. The cloud benefits from a wired gygabit communication among them, illustrated by a red arrow. Client applications request data from the nearest datacenter replica, and only collaborate through the Cloud. Client to Cloud communication, illustrated by black arrows are usually slower.

With much of this data often stored and processed in data centers (as described in Figure 2.1), there are many concerns about how these extreme volumes of data must be transported, stored, processed, maintained and made available. Using cloud infrastructures to support such applications is infeasible, not only because of their unprecedented scale and requirements, but also because of the need to move large amounts of data to the cloud, which can cause unbearable delays in data processing. In addition, many components associated with IoT applications often experience poor or intermittent connectivity, which can lead to availability issues if the logic and storage for these applications are delegated entirely to a remote cloud infrastructure.

2.2 Why Edge Computing?

We live in a world where smart devices are a large part of people's daily lives. With the growing intelligence and computational capabilities of machines such as cars, phones, watches, home automation devices, augmented reality and various data sensors, it is becoming increasingly relevant to take advantage of the massive and ubiquitous resources of these smart devices.

Fig. 2.2.: An example of the Edge Computing model. The cloud model presented in figure 2.1 is extended with new data replicas, closer to the users (i.e., the edge or the border of the network), this heterogeneous model mixes partial data replicas in a gigabyte wired school (high quality connections are illustrated with red arrows) or a manufacture company, a disconnected plane that will be reconnected after landing, users also store replicas in their own devices and can collaborate directly among them without the need to synchronize via the cloud server.

Data is increasingly produced at the network edge, therefore, it would be more efficient to process data at the network Edge as well. A cost study by Microsoft [Gre+08] shows that the traditional geo-distributed cloud design is less efficient than layered architectures at processing data when it is produced at the network edge. In this context, the *Fog Computing* model (depicted in Figure 2.2), an emerging

idea, has been designed to provide a broader range of services by extending Cloud Computing to devices that deals with local data [Bon+12]. The essence of the later proposal is that the majority (mote than 90%) of raw data generated by IoT and stored at the cloud data center is arguably useless; in fact, it was observed that aggregates of data are often sufficient for most applications.

We define Edge Computing as a distributed architecture in which client data is processed at the periphery of the network, i.e., computation is done as close at to the originating end users as possible. The concepts of fog and edge computing are expected to pave the way for a variety of possibilities for the future of intelligent embedded systems (IoT) and the Internet.

Today's cloud computing technology has changed the way we use IT services over the past decade. It relies on the massive merging of computing and storage capabilities. However, while cloud computing architecture excels at making internet services easier to use, it also faces many obstacles. Bandwidth limitations, global centralization of data processing and storage have led to many problems such as unwanted latency, reduced mobility, inability to be contextually aware, and data synchronization overheads. This is especially true for collaborative applications such as online gaming, augmented reality and many other applications that require real-time analysis and control.

The emergence of edge-oriented computing paradigms, such as fog/edge computing, is no coincidence. With the explosion in demand and growing dependence on fast information access and efficient data processing, edge-oriented computing paradigms have become an evolving necessity in the IoT (Internet of Things) domain.

Despite its growing popularity, putting edge-oriented computing into practice remains a challenge [Shi+16]. Fog and edge computing systems are still in the early stage of maturity. Over the past few years, researchers have provided new and different perspectives on how these systems should be built. There is no doubt that more effort is needed to realize the vision of fog/edge computing.

2.3 Network Architecture Challenges

Many proposals observe that edge applications offen generate [Con+34; SC+19] large volumes of data at the edge of the network bu push it back to the cloud data centers for computation and storage. This scenario has become increasingly prevalent with the exponential growth of the Internet of Things, where number

of devices and consumed bandwidth are still increasing today [@Pub19]. While the computing power of the cloud infrastructures can be adapted to the increasing pressure generated by IoT and other edge oriented applications, the same cannot be said for the network infrastructures that connect IoT devices to the cloud [Shi+16; TCH16], requiring measures and new technical contributions to, on the one hand, alleviate this load, by delegating some computations to the system's edge, and on the other hand, allow their applications to operate with limited connectivity to the core.

A short-term solution [Bon+12] to this problem is to aggregate IoT sensor data near the location (at the edge) where the data is generated, an idea that has been popularized by fog-based architectures and has been widely adopted by industry [YLL15]. Unfortunately, this approach does not address the fundamental problem of making these applications dependent on heavy computation (and storage) capabilities that are only available at the code in cloud infrastructures. This observation is supported by the exponential growth of the IoT that is expected to continue for at least another decade, with predictions of more than a trillion totals connected and operating IoT devices by that time [Bol19]. The massive amount of data that will be generated by these devices will require massive computing and storage capabilities to be continuously available, which will not be possible by taking advantage of the core cloud infrastructures alone.

Instead, a paradigm shift is needed to take advantage of the multiple layers of compute resources that exist between devices and cloud data centers by decentralizing storage and computation. Moving the components of these applications to the edge will effectively reduce the amount of data that must traverse networks to reach data centers, while allowing data to be processed closer to where it is generated, improving quality of service, namely faster response times, for applications and the end users.

Another key challenge for edge and fog applications that is largely unaddressed by the network and architecture literature and which we are interested int the background of this thesis is data management. Data management is a non-trivial problem, and dealing with it adequately at the application level places a nontrivial burden on application developers, which can lead to incorrect solutions. In this thesis we will focus on this aspect of the state of the art, and in particular on the availability and consistency aspects of data shared and stored on different layers of the cloud-edge spectrum at the architectural level.

2.4 Network Heterogeneity Challenges

In edge computing, the processing load should be distributed cooperatively across edge devices and cloud servers. Computation is distributed on each of these components depending on a number of factors, including network response time and the capacity of each node.

An edge network is typically composed of a large set of heterogeneous and loosely coupled compute nodes. An edge applications will frequently be exposed to several aspects that make their design and execution very challenging, namely high membership dynamics (nodes join, leave, and become unavailable), intermittent connectivity (network partitions), and weak communication order (messages are lost, duplicated or received out of order).

Many approaches have been proposed by the literature to classify heterogeneous devices in the Edge-Cloud networks [APZ18]. The main approaches being: Cloudlets [Sat+09] are computers or cluster of computers with above-average performance and good connectivity to the internet; MEC [Hu+15] is an architecture for deploying a wide variety of applications on top of a host and is critical to the growth of the Internet of Things (IoT); For Computing [Bon+12] requires both the cloud and edge nodes and runs applications to meet a specific use case.

Fog Computing networks can include a mix of powerful and reliable nodes that operate on data centers and small resource-limited and unreliable nodes that operate at the edge. Even among edge nodes, there is considerable heterogeneity, a laptop or a cell phone is much more powerful than other home automation IoT devices. As different devices have different capabilities, it is unrealistic to expect full replication of the data that an application manages.

Network connections are also very heterogeneous and highly dynamic. Latency between edge nodes vary considerably, and some parts of the topology can be very dynamic while others are essentially stable. An algorithm that operates at the edge must account for this diversity and provide network reconfiguration protocols when necessary.

Finally, the edge network suffers from *churn* [SR06], which consists of frequent changes in the operational state of network components. Devices may connect or disconnect very frequently, or even fail permanently. As in the previous challenge, additional solutions are needed to cope with these changes.

In this thesis, we distinguish three areas in the network. The *core* cloud is made of a few tens of resource-rich, well-managed data centers (DCs). At the opposite end of the spectrum, millions of resource-limited and mismanaged *far-edge* devices, such as mobile phones or IoT devices. In between, thousands of *border* nodes, such as CDN Points-of-Presence (PoPs), metropolitan data servers, or 5G MEC servers. We refer as “edge” to the union of the border and far edge, and as “infrastructure” to the union of the core and border.

2.5 Data Management problem

There is an increasing interest in the edge and fog computing paradigms aiming to reduce the dependency on cloud data centers. While these paradigms bring several benefits on security, efficiency and cost, they incur new challenges nicely summarized in the eighth Pilars of the OpenFog Reference Architecture [BS17]: security, scalability, visibility, autonomy, reliability, agility, hierarchy and programmability. Data management is a cross-cutting concern that touches most of these pillars, it gathers all the mecanisms related to storage, data placement, access control and privacy, and most importantly in this thesis data consistency.

At the same network area level (*core, border* or *far-edge*), data management is mainly centralized, most fog architectures does not provide data management directly among lateral nodes. A common pattern to share data in this case is to push it to a "parent" node at a higher level. The parent node plays a centralized data sharing role. The existence of parent nodes reduces the autonomy, robustness, and scalability of fog applications.

Across network area levels, data management is not consistent, which means that applications cannot easily move from one fog level to another without violating correctness. The consequence is that data management solutions have to be defined at each fog level. Sometimes, to improve application responsiveness and reduce latency, loose synchronization patterns are used with loose consistency between fog levels. If not carefully defined, these patterns introduce fundamental data inconsistency between fog levels, which greatly increases the complexity of creating and managing correct applications.

Solving these two problems would simplify collaborative applications development by addressing several requirements and properties, specifically scalability, autonomy, availability, hierarchy and programmability. The following sections will describe how the current systems addressed those problems.

In fact, many distributed applications are currently designed to operate under strong consistency models, using solutions such as replicated state machines. This is due to the complexity of designing applications under weaker consistency models. Unfortunately, the use of these techniques is incompatible with the properties stated above. Therefore, there is a clear need to rethink how to develop distributed applications to work in edge computing environments.

There is currently no solution to easily develop and deploy applications within the edge computing paradigm. There is a lack of support for performing general-purpose computation in such a challenging environment. In particular, the current state of the art for performing computations on edge networks consists of gossip and peer-to-peer algorithms, which are capable of performing aggregate computations and providing immutable data content distribution across the edge network. Unfortunately, these techniques are still limited in scale and are not easily extended to support general computation on the edge, i.e., to perform any form of distributed computation on a shared, mutable state.

2.6 Edge Storage Systems

In the common case, edge storage systems are designed for specific use cases [BS17] and not for general purpose. This is due to the expected high volumes of traffic and the limited bandwidth of the cloud infrastructure. Before data is transferred, there may be some pre-processing to reduce traffic or avoid it at all costs. In addition, these storage systems choose lower consistency guarantees to reduce the storage and latency overhead associated with more powerful models.

One of the first caching systems, Cachier [Dro+17], addresses recognition applications use case. Most sample recognition algorithms operate in a pipelined mode and go through the following steps: feature extraction, feature classification and matching, and best match selection. They are computationally intensive, which does not allow them to run on edge nodes. As a compromise, these edge nodes can store parts of the model, so that they can be used to answer similar requests. This approach relies on the fact that clients in the same time and place request similar things with a high probability. If a client requests something that cannot be satisfied by the cached model, the request is forwarded to the cloud, which responds with the result and the component responsible for obtaining it. The edge node sends the response to the client and caches the new component, following an initial LRU (Least Recently Used) policy that is modified over time. The main goal is to exploit a

strategy that maximizes the number of times the edge node is able to respond to the client on its own.

Another situation is when the data needs to be archived, but not all of it is interesting. In another case [RG18], the system monitors a region through video capture. Since transferring all the images to the cloud requires huge network bandwidth and increases the latency of the analysis, the edge nodes run an algorithm to create feature vectors that allow relatively fast comparison between consecutive images. If two images have a similarity index above a predefined threshold, the differences are not significant and one of them can be discarded. Otherwise, the frame and its feature vector are time-stamped and tagged with a unique node identifier. Then it is placed in a transmission buffer to be sent to the cloud.

2.7 Conclusion

In this chapter, we introduced the edge computing model, its purpose and the problems related to its design. We found that many scalability, efficiency and security challenges are induced to the edge network architecture and the heterogeneous ressources.

3 Requirements for collaborative edge applications

Edge computing enables a new class of distributed cooperative applications.One example is a multiplayer game, where users and bots update a shared game universe. Other examples include users contributing to a cooperative document, sharing files, using version control software, or a group of engineers using augmented reality to design an artifact. Maintenance engineers of a water-distribution company need data in the field to access the pipes. Telecom network engineers set systemwide configuration variables that interact at remote devices. Future applications might include vehicle-to-vehicle collaboration, or informed tactical decision-making in an emergency.

Such applications are based on shared, persistent, mutable state, i.e., a database. Whereas the database is stored centrally in a cloud setting, we enable state to be distributed and replicated across decentralized edge devices. Locating data at the edge enables quick application response (no waiting for a network round-trip) and availability (the application can read and write data, even when disconnected from any remote server). This local-first approach provides users with a sense of ownership and helps with confidentiality [Kle+19]. Nonetheless, we leverage the strengths of the cloud infrastructure, i.e., stronger consistency and well-managed, abundant resources.

Many distributed applications are currently designed to operate under strong consistency models, using solutions such as replicated state machines. This is due to the complexity of designing applications under weaker consistency models. Unfortunately, the use of these techniques is incompatible with the properties stated above. Therefore, there is a clear need to rethink how to develop distributed applications to work in edge computing environments.

It should be possible to execute in the cloud a computation that requires high bandwidth (e.g., analytics) or stronger consistency. The requirement is uniform

semantics, i.e., wherever it executes, the same computation observes the same state and has the same result; only performance should differ.

3.1 Consistency requirements

An edge application must remain available despite unpredictable network connectivity, disconnection, and mobility. Users or groups of users may have to work in isolation. In this context, strong consistency is not possible globally [GL02].

One can argue that users are tolerant, and that a best-effort approach such as Eventual Consistency (EC) is good enough. This was the original approach for large-scale internet applications such as e-commerce or social networks [Vog08]. However, EC loses updates, which is clearly undesirable, e.g., in a collaborative editing context.

Consistency anomalies are frustrating for users and make it difficult to write a correct application code. Thus, Facebook has moved to enforcing the read-your-writes guarantee [Shi+20]. In networked video games, it frequently occurs that the user observes an action, only to see the system roll it back soon afterwards [Pus19]. Applications, AIs, or decision-making software is more likely to have bugs. We take a hybrid approach: globally, provide the strongest model compatible with availability; and locally, strengthen the guarantees where possible.

3.2 Global Consistency Guarantee: TCC+

In the edge context, two different consistency models have been explored. Although they are incomparable, both have been proved to be a strongest possible model compatible with availability under partition.

One is Monotonic Prefix Consistency (MPC), which combines the per-process order into a global total order; however a process is exposed to arbitrary rollbacks [Gir+18]. We argue that a client losing an unpredictable amount of work is an unacceptable user experience.

One is Monotonic Prefix Consistency (MPC) [Gir+18]. In MPC, processes agree on a (monotonically growing) common prefix of their individual sequential history. In other words, MPC provides strong consistency up to the common prefix, but recent history may diverge. After applying some updates, a process might discover

that another update comes earlier in the sequential order, and have to roll back an arbitrary number of times. MPC was the model of Bayou [Ter+95], and lives on in blockchains [PSS17; GKL15] and in some recent file systems [Kle+20]. We argue that arbitrary rollback and loss of work are a poor user experience, which is especially painful when nodes can disconnect for a long duration.

Our preferred alternative is Causal Consistency (CC) [AEM17]. Intuitively, if a client observes some update, it also observes all preceding updates. Only concurrent updates may be observed in different orders.

The following example illustrates ordering anomalies when CC is not observed. Alice makes her photo album private, then adds a new photo. Later, Bob views Alice's area. If the system does not enforce CC, he might see the new photo, unless the website developer went to extra lengths to avoid this. Instead, under CC such anomalies do not occur.

CC can be enforced locally and does not require consensus. On top of CC, atomic transactions and convergence guarantees can be supported without impacting availability [Zaw+15a; Akk+16a]. This improved model is called *Transactional Causal Plus Consistency (TCC+)*. Section 5.2 formalizes the TCC+ guarantees. The drawback is that tracking the partial order in CC (and hence TCC+) can have a heavy metadata cost, as we discuss later.

3.3 Conflict-Free Programming

We briefly introduce the main ideas of conflict-free programming and explain how they adapt to edge computing [Sha+11a; ASB15]. Traditional approaches to distributed system design are not generalizable to edge networks because they depend on strong synchronization between multiple parties, such as that provided by uniform consensus. Unfortunately, such strong synchronization primitives are impossible to realize in a scalable manner on edge networks. An alternative to using such primitives is to rely on conflict-free programming.

Consider a distributed data structure, replicated across n nodes to improve robustness to node failures. Each node has a copy of the current value of the data structure. When an operation is invoked on a node, a new value is calculated and all nodes must be updated with this new value. This operation must be performed consistently in the event of concurrent operations, node failures, and network disruptions ranging from variable latency to message drop or partitions.

Relying on strong synchronization primitives, one would use a solution based on a uniform consensus algorithm such as Multi-Paxos [CGR07] or Raft [OO14]. Many industrial systems use this solution, for example, Google's Chubby lock service uses Multi-Paxos. In this solution, the consensus algorithm is run on all n nodes for each operation, which, in addition to being expensive and not scalable, does not provide availability in the event of a major node failure or network disruption.

The alternative proposed in the context of conflict-free programming is based on the observation that, in most cases, consensus is not strictly necessary to maintain replicated data structures. Replicated data structures can be realized using a conflict-free replicated data type (CRDT) [Sha+11a]. A CRDT satisfies the mathematical property of Strong Eventual Consistency (SEC), which ensures that replicas are consistent as soon as they observe and execute the same set of operations.

This allows programming distributed applications that do not have to explicitly worry about synchronization between components that may be loosely coupled. Instead, it becomes sufficient to ensure that *eventually* the different components of the system are able to communicate with each other and exchange synchronization information, which is much weaker than uniform consensus.

This approach works well and is used in several industry systems, such as applications from SoundCloud, Bet 365, Riot Games, Rovio and Trifork. And its success points to the need to further leverage synchronization-free programming models to develop robust and available applications within the edge-computing model.

3.4 Local Strengthening: Data Center

A set of nodes that are strongly connected to each other can support even stronger guarantees. The system can totally order updates upfront, ensuring for instance Snapshot Isolation (SI). SI is stronger than MPC, as it does not suffer rollbacks, and than CC, as it totally orders updates. The metadata cost for total order is much reduced. We call a set of nodes that enjoy SI among themselves an *SI zone*.

One kind of SI zone is a single data center (DC). A DC has a large number of parallel servers, connected through a high-quality network. COLONY executes transactions across multiple servers in the same DC under SI Snapshot Isolation (SI), an RFTOC model that is strictly stronger than CC [DEZ13; Akk+16a].

In the context of global TCC+, a DC counts for a single sequential process, thus limiting metadata size.

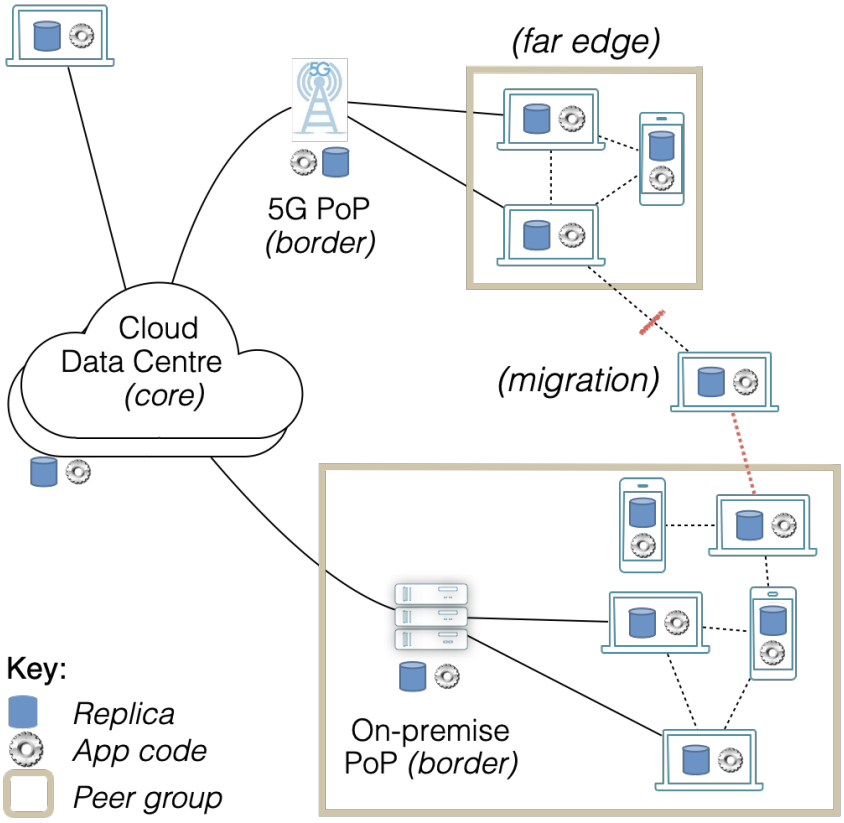

Fig. 3.1.: Example COLONY topology. A small number of DCs forms the core. A far edge device connects either directly to a DC, or via a point-of-presence (PoP) server at the border. A peer group contains devices in geographical proximity. Note the device migrating between subtrees.

Some systems support two classes of transactions, weak and strong; all transactions follow causal order, and in addition strong transactions are globally totally ordered [Li+12; Bra+21a; Bal+15]. COLONY does not currently support strong operations, but does support an intermediate class of "moderate" transactions that are totally ordered within a peer group.

3.5 Local Strengthening: Peer Groups

This section examines another kind of SI zone, the peer group. Inconsistency is especially problematic to users who communicate directly, outside of the database. For instance, in the enhanced-reality game Pokémon Go, two users in close proximity can both become the owner of the same game character [Lei19]; this anomaly confuses the users. Another case is a group of collaborators that wish to follow each other closely, without receiving distracting updates from the outside. This and similar examples argue for groups with stronger consistency.

There are different kinds of groups: A *collaboration group* is a set of nodes collaborating which each other and separately from others. Desirable features include branching versions, and security mechanisms for managing trust. The group may be short- or long-lived, and its nodes may be geographically close or far away.

A lateral group is a set of nodes that communicate directly with one another. This enables them to remain independent, even if the cloud happens to be unavailable.

The group is short-lived and lasts only as long as the lateral connection.

An SI zone is a set of nodes that enforce strong consistency locally, by leveraging stable and low-latency network connection to one another. This kind of group is also short-lived.

For simplicity, in COLONY a lateral group is also an SI zone; we will call this a *peer group*.

In COLONY, edge nodes in network proximity can constitute an SI zone, called a *peer group*. According to SI, transactions become visible in some *a priori* total order with no rollback. This approach improves user experience and helps to manage metadata.

To avoid that these stronger guarantees be at the expense of availability, the system should support disconnected work and mobility within the topology, without losing the TCC+ guarantees.

For instance, users traveling together on a bus might create an SI group to play a multi-user game together; as the bus moves away from its connected DC, it can switch to a closer one.

3.6 Security Requirements

A different concept is the *collaboration group,* a set of nodes that collaborate together, even when far from each other. For this case, To support collaboration, COLONY supports versioning and trust management.

Because it is the edge device that executes and merges updates, data can remain encrypted end-to-end; the untrusted cloud serves merely for transport and persistence [KKB19].

On the other hand, the edge use case poses new security challenges. Information is exposed on compromised edge nodes [LLaP20]; security policy changes and data updates are concurrent [WRT10; WBP16]; and decentralized key management is problematic [KKB19]. We alleviate these difficulties by leveraging the cloud, e.g., for authentication and key management.

Our focus in the security area is on the management of mutual trust to support group collaboration. Every data object comes with an Access Control List (ACL) that describes what updates different users are allowed on the object. The system preventatively enforces ACL in edge devices. Because an edge device may be compromised, every node double-checks the updates it receives, and masks an update that is not allowed by the corresponding ACL, and transitively any update that depends on it. Thus a correct node never depends upon a state that violates the security policy.

3.7 Summary

The ever-increasing amounts of data generated on the Internet pose real challenges to the cloud computing model that often delegates most of its computation and storage to the cloud data center. The Edge/Fog Computing model can be seen as a distributed extension of the cloud computing model by breaking down the core cloud into a network of smaller clouds (i.e., the Fog) often located close to the user, bringing several performance benefits and new classes of applications.

Many distributed applications are currently designed to operate under strong consistency models, using solutions such as replicated state machines. This is due to the complexity of designing applications under weaker consistency models. Unfortunately, the use of these techniques is incompatible with the properties stated above. Therefore, there is a clear need to rethink how to develop distributed applications to work in edge computing environments.

4 Related Work

We have previously seen the challenges of edge computing architectures and the requirements to fulfill collaborative applications needs. We will now study the state of the art of edge storage systems from a set of examples. We will cover the spectrum of fog computing systems from cloud geo-distributed approaches to decentralized edge systems, and assess the strengths and weaknesses of these approaches in terms of consistency, availability, mobility and security. We will also see more distant related work, with fully decentralized web and mobile oriented systems.

4.1 Overview

Much previous work on data in edge computing [Ram+09; Ter+95; Jos+95] focuses on streaming and content delivery.

Examples include sensor systems or propagating database views. We leverage this previous work by propagating shared state as a stream of update events. The distributed sharing of persistent mutable state raises extra challenges, which we address in this state of the art study.

Achieving low response time is an ongoing challenge for many web applications [Aka10; KL07; Lei09]. For example, on amazon.com, a delay of 100 ms costs in average 1% of sales [KL07]. In order to deliver fast response and offline support, a number of web applications cache data at the client side, e.g., in the browser. See for instance Facebook's News Feed [MS15], or the Chrome browser extension that enables Google Docs and Google Maps to be used offline [sup]. Several earlier systems manage data for clients that are connected only intermittently [SS05; Ter13]. Bayou [Ter+95] supports data replicas in mobile computers. Cimbiosys [Ram+09] uses a decentralized model to support Internet Services.

Rover [Jos+95] and Coda [KS92] are general-purpose systems that support disconnected operation, under a weak consistency model.

Cloud Types [Bur+12] support a programming model for shared cloud data, similar to CRDTs with total order. It supports multiple replicas of data, synchronized periodically. Parse [Bur+12] is similar, under a weaker eventual consistency model.

Strengthening causal consistency with strong convergence was proposed by COPS under the name Causal Plus Consistency (CC+) [Llo+11a]. ChainReaction augments CC+ with transactional reads, implemented with the help of a sequencer per DC. To execute a write, ChainReaction requires that the prior versions read at the client are stable in the DC. TCC+ [Akk+16a] extends the guarantees of CC+ to transactions. This model is closely related to others, such as Parallel Snapshot Isolation (PSI) proposed by the Walter system [Sov+11]. TCC+ is the strongest model compatible with availability under partition. COLONY guarantees TCC+ globally; in zones with good connectivity, it enforces an SI zone to improve metadata overhead and user experience. Whereas TCC+ supports concurrent updates and arbitrary CRDT types, PSI restricts concurrency to a single data type (the cset). Its SI zones, the DCs, are fixed in advance. Walter runs two-phase commit across data centers; this ensures global transactions are strongly consistent, but negatively impacts liveliness and performance.

A hybrid consistency model makes different consistency guarantees for different operations. For instance, in Lazy Replication, operations are causally consistent by default, but can optionally request linearisability [Lad+92a]. References [Fek+99; Bal+15] or [Li+12] are other examples. Unistore [Bra+21b] supports both causal and linearisable transactions over a geo-replicated key-value store. To ensure fault tolerance, Unistore requires that, before a linearisable transaction commits, all of its causal dependencies be stable. Friedman et al. [FRT15] propose a proximity-based hybrid model, called Fisheye Consistency, such that nodes that are close in a "proximity graph" mutually observe strongly-consistent transactions, whereas consistency is weak between far-away nodes.

Depot [Mah+11] and PRACTI [Bel+06a] pioneered highly available caching at the edge, under causal consistency. Their approach uses vector clocks with one entry per replica, which severely limits scalability. Depot targets Byzantine fault tolerance, the most general class of faults, but not transactions. Simba [Per+15] enables the edge application to select among a specific level of consistency (eventual, causal or serialisable). PouchDB [Pou] is a client-side cache that replicates data from a CouchDB server; it supports offline operation and detects conflicts, but does not merge them.

SwiftCloud [Zaw+15a] introduces bounded-size vectors and migration. Legion [Lin+17a] is a web framework that extends applications with peer-to-peer interac-

tion using CRDTs. COLONY is inspired by the above designs; additionally, COLONY supports collaboration groups and peer groups, and ensures that migration is seamless.

Let's study in detail each of these systems.

4.2 Kronos

Kronos [Esc+14] is a centralized service that maintains a dependency graph. Nodes represent events and edges define causal relationships between events.

When a client wants to obtain a reference to an event, he or she must contact Kronos. A reference is an identifier of an event and is managed by the service. After collecting a set of references, clients need to know a possible order to apply the associated events. They must therefore contact Kronos, providing the set as input. The output is a sequence with all elements of the set that does not violate causality. Different clients may get different results, because the concurrent operations may be in a different order.

Adding events to the graph is also done in two phases. In the first one, a client creates the event, receiving its reference. Then, it is able to assign an order, using the new event reference with the other existing references. There are two ways to order events: must and prefer. Must specify a strict order: either all must condition of a call is met, or the operation is aborted. The preferential order is used to assign relationships between concurrent operations. Its validity is checked only after Kronos has analyzed the must conditions. If they introduce inconsistencies, the operation is not aborted, but the order is rejected. Since this is not a geo-replicated solution, there are no remote servers to send new updates. This contributes to a simpler design, but creates a performance bottleneck.

Kronos is an isolated system that deals only with the causality of events. This introduces a new abstraction, as different events can have very different granularities.

The biggest weakness is related to how clients get a correct ordering of events. If a client has not received a reference that depends on another one that has been sent to the server, the ordering will create an inconsistent client state. So either the client must have all references to get a global order, or there must be an application-dependent protocol to prevent this.

Despite an interesting approach to ensuring causal consistency, Kronos cannot be deployed at the edge. It is a centralized service that would quickly become a bottleneck, violating the scalability requirement.

4.3 CloudPath

CloudPath [Mor+17] provides a more general extension of possibly coherent storage systems to the network edge, while adding processing capabilities. It uses path computing to deploy a multi-tiered hierarchy from a data center to the edge node. As nodes get closer to end users, their performance gradually decreases. Overall, CloudPath connects three types of nodes in a tree structure: edge nodes (closest to end users), cloud nodes (most resource-rich and farthest away) and core nodes (any node in between). The root node replicates each object, while each descendant stores a subset of the items available in its parent. Clients read from and write to their local replicas, which propagate changes in the background through the hierarchy. If a client tries to read data that is not available locally, the node redirects the request to its parent and, until the data is found, the request moves up through the nodes. When there is a response, it follows the reverse path, causing each contacted child to store the new data. This creates on-demand data replication, attempting to cache only the latest requests. Updates are recorded in a write log with a label that contains the time of the change and the node ID. As a fault tolerance measure, updates are marked as dirty or clear. Dirty updates are not recognized by the parent node, but clear updates are replicated to both.

4.4 COPS

COPS [Llo+11b] has a timestamp with one part per object. Each part has a logical clock.

Clients are in charge of managing their causal history. In COPS, this history is called a context. Every time a client reads a new object, it adds the version to the context. The data center servers always give the latest consistent version of the object, without looking at the client's context.

When a client wants to write an object, it first uses the context to calculate the closest dependencies. The context can be seen as a dependency graph and the transitivity rule is used to find these dependencies. The closest dependency is obtained when

one version of an object does not occur before another version. Then the client sends the object key, its new value and the closest dependencies to the local server. This server has all the dependencies, because the client maintains a session with it. It can therefore apply the update immediately, marking it with the dependencies and the new timestamp. This timestamp is sent back to the client and replaces all previous context.

The remote update protocol follows similar steps as the local client update. When a server receives an object from another replica, it must check if the dependencies are met. Unlike the previous situation, there may be other updates that belong to the dependencies and are not yet present. Thus, the receiver must delay the update until it can be safely applied.

An interesting feature is that a client can have different contexts for the same connection to the server. Since dependencies are tracked per client context, this allows independent operations to have fewer common dependencies, which reduces the time that remote updates wait before being visible (visibility latency).

The timestamp format introduces some metadata compression, visible by having only one scalar per object and not a vector with one entry per replica. This results in a similar amount of metadata as Kronos, but with higher visibility latency.

COPS was one of the first systems to provide causal consistency over cloud storage. However, the metadata format is not prepared for the size of the network edge. Typically, application workloads tend to include many more reads than writes. In this solution, the processing power required to check for remote update dependencies could overwhelm the nodes. In addition, it does not account for partial replication, as it was not a concern at the time.

4.5 Orbe

This system [Du+13] explores the layout of the data center to choose the number of timestamp partitions. Each data center is divided into N partitions and there are M different replicas. Considering that each partition is assigned to a different server, it is possible to address it through a matrix-like structure with N rows and M columns. Clients track their causal history with a copy of this matrix. It is filled with the logical clock of the corresponding server.

A read-only client sends the key of the object. The server responds with the value of the last available version, the creation clock, and the index of the replica. If the

entry pointed to by the index has a smaller value than the returned clock, then the client overwrites that position with the most recent value.

To write an object, a client sends the key, the new value and its matrix to the local server responsible for the partition. The matrix corresponds to the update dependencies and is used during replication. Before marking the creation time, the server must increment its local clock, thus ensuring that the assigned value is unique. After storing the value and the object's metadata, the server responds with the update clock and its index. As with COPS, the client resets its causal history to default values, saving only the clock returned at the given index.

To replicate updates, each partition communicates with its peers in different data centers. Updates are sent in order of creation with all associated information (key, value, creation clock, dependency matrix and replica index). To track the applied remote updates, there is a vector with one entry per replica and it contains the clocks of the last updates of each replica. The receiving server uses the matrix and the vector to check for dependencies. For its partition, it looks at the corresponding row of the same replicas and compares it to the vector. If the vector is greater than or equal to the row, the server can proceed to the next step. For other partitions, whenever there is a non-default value in the partition column, it means that the operation depends on other elements that the current data center may not store. So, the server contacts the servers that are in this situation, performing an explicit dependency check. If they fulfill the dependencies, the remote update can be made visible and the corresponding vector entry updated.

In an effort to reduce the amount of metadata, the authors store only those matrix entries that are different from the default. This strategy allows clients to store only the clocks of the contacted servers as dependencies for future operations. However, as soon as a client reads a newer version of a server, the matrix must be updated in the corresponding entry. This can lead to a matrix with all entries filled, which brings no gain in the worst case.

Given the coarser grain in which dependencies are tracked, the impact of false dependencies will be greater. For example, updates on one server will depend on all previous updates applied on the same server. So even if there is no direct version dependency between objects, it will be treated as such. The increased wait time before updates can be applied will immediately result in higher visibility latency.

Although Orbe is able to limit the size of metadata, it is not independent of the number of participants. Like COPS, Orbe requires replicas with full system state. Both of these features would not work at the network edge.

4.6 PRACTI

The name of this system [Bel+06b] includes its main features: partial replication (PR), arbitrary consistency (AC), and topological independence (TI). The granularity of partial replication is the finest, as each replica chooses the individual objects it wants to store. Like Orb, timestamping has one part per server and uses logical clocks.

When a node wants to participate in the network, it chooses three sets of other nodes: one to receive updates, one to pick objects for its set of interest, and one to request more recent consistent versions of locally stored objects. The interest set is any group of objects that the node stores, and it can have one or more interest sets.

Each node stores a global timestamp and as many timestamps as interest sets. They are updated at different times, expressing different things. They will be discussed when we talk about replicating updates.

Nodes read their local copy of the object when it is valid. Validity is checked on timestamps and will be explained on update replication.

Like reads, writes are performed on the local replica. Each version is tagged with the node's logical clock value. Then it is sent to the nodes that have subscribed to the object updates.

The replication of updates is done by two types of messages: invalidations and bodies. The first type contains only metadata and can be precise (one message per object) or imprecise (one message for a group of objects and/or multiple updates), but must follow a causal delivery order. Imprecise messages have a set of targets, start and end clocks. Bodies have precise metadata and the actual data, and they do not need a specific delivery order.

Upon receipt of invalidation messages, the node can update the timestamp of the object's interest set as well as that of the local node. The former is only updated with specific invalidations, but the latter joins the two. When the latter has a newer version, the local state is inaccurate. If the body for a specific invalidation has not yet arrived, the community of interest state is invalid. Invalid objects cannot be read, as this prevents inconsistencies for stronger models. However, for causal consistency, old values can be returned, which means that stale reads exist.

Supporting different levels of consistency has an impact on the amount of metadata and the time needed to apply each update. Since all other systems are defined only for causal consistency, this will not be considered a negative feature.

Although nodes can choose to keep only a subset of all existing objects, they must receive all invalidation messages. This is necessary because of the unstructured topology, which could otherwise lead to inconsistent information about the nodes. To reduce the impact of communication overhead, inaccurate invalidations function as a summary of updates whose data is not stored locally. Yet, nodes receive information about objects that they do not replicate, creating inauthentic partial replication. If all messages were about replicated objects, then this would be authentic partial replication.

PRACTI is the first system to reveal some promising strategies that could be used for edge computing. Topology independence is particularly interesting because new nodes do not need to be integrated into a rigid structure using a join protocol. In addition, partial replication is supported, allowing machines with less storage to participate. However, metadata grows with the number of replicas, which could become unsustainable at the edge.

4.7 Lazy Replication

Each node has full data replication, which means it must store all objects in the system. As in PRACTI, there is no predefined network topology and timestamps have a per-node portion.

In lazy replication [Lad+92b], there are two types of timestamps: unique identifiers (UIDs) and tags. The former marks updates and the latter compresses multiple UIDs. Tags create the client's causal history and dependencies for all operations. Servers also keep timestamps to track their current version.

When reading the value of an object, the client sends the key and its label. The server waits for its state to include the entire history of the client. The response contains the last value of the object and the corresponding timestamp. Before delivering the value, the client updates its label by including the partial maximum between its previous label and the received timestamp.

When a client tries to write an object, it constantly contacts the known entities by sending the object key, the new value and its label. Eventually it may send it to different entities and create duplicates. Given the desire to deliver messages at least

once, each client has its client identifier (CID). It is included in the other update metadata, allowing different replicas to recognize duplicate updates. On the server side, if it is a new update, it increments its clock, updating its local label. The UID of the update is obtained by replacing the client label part of the local server with the new clock value. In response, the server sends the update UID, which is able to replace the client label while maintaining the causal history. Finally, the client acknowledges the update, marking the end of the operation.

Updates are replicated by *gossiping*, which ensures that every node sees them. The strategy adopted is log replication. Each replica has its own log with all operations that changed the local state. After some time, the replica sends it to its direct neighbors, who filter out duplicate entries and apply missing operations. The filtering is done by looking at the UID of the update with the CID.

As a garbage collection strategy, messages that take more than a well-defined time interval to arrive are rejected. Acknowledgments of client updates are included in the log, as they serve as a marker to prune old log entries. If future acknowledgements are discarded by the time condition, the log size is reduced.

Lazy replication is not edge-ready, for reasons similar to those of previous systems. However, it does introduce the use of unique identifiers that filter out duplicate updates. This is interesting from a fault tolerance perspective. Since the network edge is much more dynamic, the probability of clients having to send the same request more than once is not negligible, and with the presented strategy, duplicates would not create inconsistencies.

4.8 Cure

Cure [Akk+16b] has successfully allowed causally consistent transactions with object reads and writes. Other solutions allowed transactions with one or the other.

Causal dependencies are expressed by a timestamp with a part per data center and physical clocks. However, the process of creating the timestamp is more complex than in previous systems. It is explained when presenting transactions that include object updates.

Before a client can access objects, it must contact a coordinator to obtain a transaction ID. A coordinator is a server in the local data center that is responsible for committing the client's transaction. The ID functions as a limit for the updates a

client can access. During a transaction, the client can see objects whose version has a timestamp less than that of the identifier.

Clients can read one or more objects at the same time. However, they keep the transaction identifier as immutable metadata. As for writes, a client can also make several at the same time, but they are not final. They will only be stored after a successful commit.

When a client commits a transaction that includes object writes, all these writes will have the same timestamp, so the atomicity property is guaranteed. The timestamp calculation involves the partitions whose objects have been updated. Each partition proposes to the coordinator the current value of its clock. After collecting all the proposals, the coordinator chooses the highest value and notifies the result to the participating partitions. It also replaces the contents of the local data center entry in the dependencies to create the update timestamp.

Update replication is a pairwise process between the same partitions on different data centers. After a specific time interval, the pending committed updates are transmitted to the other data centers. Replicas have two timestamps to control the visibility of such updates. The first tracks all updates (local and remote) and the second is used to display remote updates. The receiving server advances the sender's timestamp entry to the update's timestamp on the structure of all transactions. To know which remote updates can be made visible, the partitions in the same data center periodically exchange their all operations timestamps to calculate the second timestamp. Each part will be the minimum value of timestamps received for that specific entry. Remote updates whose dependencies have a timestamp less than or equal to the global timestamp can be made visible.

The most obvious drawbacks to adopting Cure at the network edge are the need for full replication and the size of the metadata. Looking a little further, the transactional schema is another issue. It requires a fair amount of synchronization between nodes, which at the network edge can lead to a non-negligible number of retries. So, it is preferred to use simpler operations on this setting.

4.9 Chain Reaction

As the name suggests, ChainReaction [ALR13] is built on a chain replication architecture. In the following paragraphs, we explain how this affects the different

operations. This system uses timestamps that have one part per data center. Each part has a logical clock that belongs to a server in the corresponding data center.

Clients track their causal history through a table with one entry per accessed object. The version timestamp and string index vector are the metadata associated with the object key. The chain index vector has as many entries as the number of data centers and stores the last position in the chain at which the object was replicated.

If the chain index entry for the local data center is equal to the predefined chain length, the client can send a read request to any node in the chain. All will be able to respond to the request without waiting for a remote update. Alternatively, the request can be made to any server up to the one specified in the entry. When the client changes data center, the previous version of the object may not be available. Thus, the head of the corresponding chain must wait for it or make an explicit request to a data center that has it. The server returns the value with the timestamp of the version and the index of the chain from the local data center. If it is a new version, both entities are modified. Otherwise, the local data center chain index entry retains the higher value.

Each write is sent to the head of the chain and propagated to subsequent nodes. In the message, the client sends the object key, the new value, and a compression of all the metadata of the objects accessed since the last update. The head of the chain first assigns a new version of the object by incrementing its replica portion of the timestamp. Then the update is sent to the other nodes until it is k-stable. A k-stable update is replicated to at least k nodes in the chain. Only then is the head able to return the object version and index of the last receiving node to the client.

When an update reaches the tail of the chain, it is DC-Write-Stable for that replica. Updates are delayed until every object on which they depend is in this state, preventing clients from reading inconsistent versions. DC-Write-Stable objects do not need to be kept in the client's accessed object table. If all replicas in the chain are in this state, the update is globally stable.

Remote updates are scheduled immediately after a chain head receives an update from the client. For this replication, only the time stamp of the update is required. The data center has a dedicated entity responsible for exchanging these updates, called a remote proxy. It also has a timestamp to track previously applied updates. When a remote update arrives, its timestamp is compared to that of the receiver's proxy. If the latter's timestamp has at least the same input values for the other data centers and one less for the sending data center, the corresponding dependencies are

stable on the current data center. Therefore, this update can be applied. Otherwise, it must wait until these conditions are verified.

It is important to note that a node can have different positions for different objects: for one it can be the head, for another the tail, and for another the middle. Different types of nodes require different fault tolerance strategies. The head is replaced by the second node, the tail is not replaced and the middle nodes are hovered over. The k chosen earlier for a k-stable update functions as the minimum number of nodes for ChainReaction to work properly.

This system relies on a very strict topology for replicating updates, which could cause performance issues at the edge. In addition, the tail of the chain receives remote updates much later than other nodes, as there may be many hops before it is reached. As in previous systems, metadata size remains an issue.

4.10 GentleRain

Unlike previous systems, GentleRain [Du+14] marks updates with a one-piece timestamp for the entire system. This means that it always has the same size, regardless of the number of objects or participants. As far as the clock type is concerned, it uses a physical clock.

The clients store two timestamps: a global stable time and a dependency time. The first one is calculated by the servers and is used as a marker to make remote updates visible. The second is used in some interactions with the local data center. The servers maintain a vector with one entry per data center that allows the calculation of the global stable time. There is also a local stable time that functions as an intermediate result. Their differences become clear in the replication of updates.

When a client wants to read an object, the request includes the key and the global stable time of the client. If the server's value is less than the client's, it replaces its local copy of the global stable time. The response contains the most recent version of the object whose timestamp is less than the global stable time, its value and the server's global stable time. The object's update timestamp is stored in the client's dependency time if it is greater.

For writes, the client attaches its dependency time to the object's key and new value. The server responsible for the object must ensure that the update has a timestamp greater than the client's dependency time. This may cause the operation to stall until the server's internal clock meets the condition. Then the clock replaces the current

data center entry in the server's vector and is assigned as the update timestamp. The server sends its update clock back to the client, which will replace the dependency time. At the same time, the server enables background replication of the update.

Each remote data center entry on the local version clock is updated when an update from that specific data center arrives. It will be changed to the update timestamp value. However, it is possible not to make the update visible, due to the lower value of the global timestamp. There are two intra-datacenter steps in calculating the global time. The first is to broadcast on each time interval the minimum input value of the local version vector of each partition (local stable time). After collecting all values, each partition can obtain the global stable time, which is also the minimum of all exchanged values.

GentleRain is the first monitored system that marks updates with constant and reduced size metadata. However, it still does not offer partial replication, and internally, stabilization protocols rely on information with one entry per data center. The improvement in update metadata size is overshadowed by the structure used for the global stabilization protocol, which prevents its transition to the edge.

4.11 Occult

This system [Meh+17] follows an optimistic approach. Timestamps have one part per partition, instead of per data center. It is the only system to adopt this structure, which leads to a master-slave replication technique. Each entry stores the logical clock of the corresponding partition. Clients have a single timestamp to track their causal history. Servers store only their logical clock. However, updates are stored with their dependency timestamp.

To spread the load of read operations, a client can contact any partition replica by sending the object key. The contacted server immediately responds with the object's last value, its dependency timestamp and the current logical clock. Next, the client must check whether the version is consistent with its causal history by comparing the logical clock with the value of the corresponding partition entry. Inconsistency can occur when the update has been performed on the master and has not yet reached the current replica. Thus, a client can try several times on the same replica and, if still unsuccessful, try the master instead. The last attempt is always successful, because the master has the most up-to-date version of the partition data. Finally, the client updates its own timestamp to reflect the new dependencies.

Writes are always sent to the master server of the partition. The request includes the object key, the new value, and the client timestamp. The master uses the client timestamp to derive the update timestamp, assigning the latest logical clock to the corresponding partition entry. This clock value is also returned to the client, which will use it to update its timestamp.

Updates start being replicated before a master server returns from updating a client object. They are sent to the slaves with the associated timestamp and the master clock. The second will be set as the current value of the slave clock. All updates are replicated by their creation order.

Since the number of partitions tends to be very large, there are three timestamp size optimizations: structural compression, time compression, and data center isolation. On the flip side, metadata compression results in a greater number of false dependencies, which negatively affects update visibility latency.

Structural compression works like a hash function. The timestamp is defined beforehand and it must be much smaller than the number of partitions. Then each partition occupies the position given by its number modulo the size of the timestamp. Despite the fact that different partitions are mapped to the same entry, this entry only stores the maximum value of the clock. This prevents a client from reading an inconsistent version of an object, but may cause the client to hang. If one partition has fewer updates than another that is also mapped to the same entry, reads will always fail, because the client depends on a newer version. The clocks of all partitions must therefore be synchronized.

Time compression also uses a predefined number of entries, but keeps detailed information for the most recently used partitions. If a timestamp has N entries, then the first N - 1 entries will store the partitions with higher clocks and the remaining entry will compress all other partitions by storing their maximum value. This is a dynamic structure that can be updated with each read and update. When reading, the client must have two vectors sorted by decreasing clock value and choose the maximum entry from the two, creating a new vector with a different third order. When writing, this is easier, as there is only the returned partition clock to compare with the vector entries. Starting from the last position, if there is an entry with a larger value than the one received, then it should be on the previous position. The compression of the values of the remaining partitions must also be taken into account.

The final optimization attempts to reduce the visibility latency of updates when the slave and master are on different data centers. Each data center has its own

timestamp, which in the worst case can be considered an approximation of Orb. On read, all client timestamps are updated by looking at the timestamps returned by the server and getting the highest pairwise entries from the timestamps of the same data center (one that the client already had and the other sent by the server). When writing, clients only change the timestamp of the data center that hosts the master shard.

Although the authors consider a fully replicated datastore in their evaluation, Occult offers true partial replication without any changes. The only drawback is that clients may need to visit the master replica more often, in order to get a consistent version of the object being read.

The compression of causal timestamps and the genuine partial replication make Occult a valid option for a causally consistent edge store. The only drawback is that clients do not have a true local replica, as writes must always be made to the master replica.

4.12 Saturn

This is the second system [BRVR17a] to use a timestamp with a single part for the entire system. In this case, the metadata is a tag, which functions as a tag for updates. It is created in a gear, which exists in each server. The tag sink is responsible for sending the updates to the other data centers and they are received by a remote proxy. The clocks used in the labels are physical clocks.

Clients track their causal history by keeping an updated label that is used in operations. The new Saturn components handle all metadata, but the labels are stored with the object data.

If a client reconnects to a data center, it must first attach to it. It sends the label seen earlier and the process ensures that the client's causal history is consistent with the state of the data center. Sometimes the client has to wait for the remote updates to arrive at the new data center.

For a read, the client only needs to send the object key. The engine intercepts the request to retrieve the tag associated with the stored value. The client receives both and replaces its label if the returned one is newer. The client writes contain the object key, the new value and the client label. The server engine creates a new label and stores it with the new value on the server's persistent storage. Then, the update

data and metadata are sent to the label sink for replication. Finally, the new label is sent back to the client, replacing the old value.

Object replication separates the propagation of the data and the label from the object. Label transmission follows a well-defined tree topology through serializers, but data can be delivered in any order. If a serializer does not have lower nodes that replicate the object associated with the incoming label, then it does not propagate it to them. This little detail gives Saturn true partial replication.

Remote updates are only visible when the label arrives. False dependencies are tightly coupled to the arrival of individual labels, which means that concurrent operations are executed in that order. If a label arrives before the corresponding data and the transfer time of all these data is high, the other remote updates will be blocked. Thus, the tag dissemination tree is constructed to minimize the difference between the arrival times of metadata and data. Sometimes it is necessary to introduce some delays in the transmission of labels, as they tend to be smaller and arrive earlier.

When a serializer fails, data centers eventually realize this and switch to a different tree topology. The transition occurs without shutting down the system, but updates received from the new configuration are applied only after ensuring that all other data centers share the new tree. This is achieved by using a special type of label, which signals the tree change for each data center.

To facilitate the migration of clients to other data centers, there is another special type of label that is replicated in the new client's data center. It ensures that all of the client's causal history is available at the destination.

Finally, this system meets the majority of edge computing requirements. It supports authentic partial replication, the size of the metadata does not depend on the number of participants or the number of objects, and it improves the amount of false dependencies. However, the complex topology does not allow for fast tree reconfiguration, and the migration must use the original data center to create the special label, resulting in increased latency for this operation.

4.13 POCC

This system [SDZ17] has an optimistic approach to causal consistency. Whenever servers receive an update (client or remote), they include it directly in the version chain. To ensure causal consistency, clients are responsible for managing the value

returned to the application in their upper layer. The authors argue that it is not necessary for servers to maintain heavy dependency control mechanisms, because clients typically request updates that are already replicated.

Clients have two timestamps with one part per data center. One tracks the dependencies of all operations and the other is updated on reads. Servers have a timestamp that controls the stored objects. Each entry stores a physical clock value.

To join the system, a client creates a session to a particular server in the local data center. Either the server owns the objects the client wants to access or it passes the operations to a responsible server.

When a client tries to read an object, it sends the object's key and its read timestamp. If the server's timestamp is greater than or equal to the client's read timestamp, it can respond to the request with a consistent version of the object. Otherwise, it must wait to receive a new version from another replica. The server responds with the value of the object and all associated metadata. The most important ones are the object dependencies (it updates the read and all operation timestamps of the client) and the creation clock (it only updates the timestamp of all operations).

It is possible to identify a situation in which a read never completes. For example, if a client depends on an update from a remote data center and a network partition separates the two data centers, the client will hang for an indefinite time. To solve this problem, the client is given a time interval to complete the operation. Otherwise, it starts a new session to the current data center, but in a pessimistic approach (similar to Cure).

To write to an object, the client sends the object's key, its new value and the timestamp of all operations. The server expects its current clock to be greater than the maximum value of the received timestamp, thus ensuring a higher clock than any of its dependencies. The clock value replaces the local data center entry in the server's timestamp and marks the object's creation clock. The object's dependencies are expressed by the client's timestamp for all operations. The server responds to the client with the update creation clock, which replaces the local data center entry in the timestamp for all operations.

Updates are replicated asynchronously in the order of their creation clock. The receiving server adds the object version to the version chain and updates the local timestamp of the sender's input in the update creation clock.

Although it does not meet the metadata size and partial replication requirements, POCC presents a change in the strategy for enforcing causal consistency. The

optimistic approach may be reasonable for a causally consistent edge store, as clients can retry in other replicas and sometimes continue to operate in the presence of failures in their local replica. However, the transition to the pessimistic solution raises some drawbacks discussed earlier

4.14 Eunomia

At the time of the proposal of this system [GBR17], the two most common types of mechanisms for enforcing causal consistency were: sequencers and global stabilization procedures. The first had the problem of being in the critical path of the client, introducing a delay for completion of operations. The second used a more complex dependency check, which required more communication between partitions in different data centers.

To combine the best of both approaches, Eunomia uses a site stabilization procedure. There is a new component in the data center, which is responsible for exchanging remote updates with other data centers. When the current data center has the primary replica, the update is passed to this new component in the background. The process is detailed in the propagation of remote updates.

Each client maintains a timestamp with one part per datacenter, storing the highest hybrid clock for each data center. The Eunomia service has a timestamp to monitor the latest remote updates applied and a vector with one entry per partition.

When a client wants to read an object, it sends the key to the partition server. The server returns the value of the object and the creation timestamp. The client keeps the partial maximum between the received timestamp and its own local timestamp.

For a write, the client sends the object's key, its new value and its local timestamp. This timestamp is used as the basis for the update timestamp. It is only changed in the local data center entry by giving it a higher value. The server sends the data and metadata to the Eunomia service and the update timestamp to the client. The client can override its local timestamp, as the new timestamp is guaranteed to be larger.

Replication of updates takes place only when Eunomia is sure not to receive an update with a lower timestamp from a local partition. It keeps track of the data center clocks of previously received updates on the partition vector. Each update whose value on the local data center entry is less than the minimum value of all entries on the vector is sure to be replicated to other data centers. For a data center

that receives a replicated object, it places it in the sender's pending update queue. After some time, a dependency check is performed to make these updates visible. If, for each entry that does not belong to either the sender or the receiver, the local timestamp has a value greater than or equal to the update timestamp, then the update is applied. The local timestamp is changed accordingly to allow for future updates. It is important to note that the queues are sorted with the oldest updates at the top and the newest at the bottom.

Although Eunomia does not support partial replication and its metadata is not constant, the propagation of updates uses a lightweight protocol. Nodes have all the information to autonomously choose when to make updates visible. This autonomy is valuable for edge devices, primarily because of the increased resiliency to foreign node failure.

4.15 Okapi

In Okapi [DSZ17], clients track their causal history using timestamps with one part per data center. Servers have three timestamps of the same size: one for all updates received, one for globally stable updates (similar to the Cure) and one for universally stable updates. A universally stable update is one that is globally stable across all data centers and its importance is explained when discussing update replication. Okapi uses hybrid clocks as input values.

In total, clients have two time stamps and one time dependency. One is a consistent version of the universal stable timestamp of a local server and the other is a merge of the former with the dependency time in the local data center input. A dependency time is returned for objects that are not yet universally stable, but were created locally. This is the clock value of the server in which the update was created.

When a client wants to read an object, it sends the key and the local copy of the universal stable timestamp. If the server is late, it takes the received timestamp for its local copy. At most, the returned version has a timestamp as large as the client's timestamp. Along with the object value, the server also returns its universal stable timestamp and, as mentioned in the previous paragraph, possibly the dependency time, which are stored on the client side.

Writes are completed with the new object value and the merged client timestamp. The update creation timestamp is assigned to the server, using the last clock value in its corresponding entry of the merged client timestamp. The server's timestamp for

all updates is also updated in its entry. As a confirmation, the client receives the new dependency time associated with the update.

The replication of updates is similar to that of Cure. However, the primary replica only sends its portion of the timestamp. The receiver knows which entry to update, because there is pairwise communication. The global timestamp is calculated as in Cure. The universal timestamp uses the global timestamps, which are now exchanged between the same partitions in different data centers. As before, each entry is the minimum value of all exchanged timestamps. The universal timestamp ensures that clients observe the same updates when they change data centers, unlike in Cure.

When considering the transition to the edge, Okapi struggles with full replication and metadata size. In addition, the update propagation protocol is too cumbersome, requiring a lot of stabilization between nodes. Edge nodes provide no guarantee of availability, which could lead to their demise and block the application of updates.

4.16 SwiftCloud

In SwiftCloud [Zaw+15b] The timestamp has one part per data center and one part for the client. The client part is necessary because each client maintains a local object cache and can send updates for different data centers, before they are recognized. This is a similar strategy to the CID of lazy replication. Each timestamp part stores the logical clock of the corresponding entity.

When a client wants to access SwiftCloud, it must maintain a session with the node contacted first. Clients request the objects they want to cache and include them in their interest set. All operations are performed in the cached objects and can update the timestamp of the client's causal history.

A read gets the locally available value and merges the update timestamp with the client's history. Updates are tagged with dependencies (a copy of the client's timestamp) and with an initial timestamp from the client's clock. In the background, new updates are sent to the data center node in their chronological order. After receipt, the server reassigns a timestamp with its logical clock and makes the update visible to other clients.

To deliver the updates remotely, the node has the client's current version vector and sends it only the changes that the client has not seen. If a client wants to change the number of items in its package, it must inform the associated node. When

there are new objects to replicate, the node delivers the most recent consistent versions as remote updates. In both situations, the client's causal history timestamp is updated.

For inter-data center replication, the receiving data center must wait until the local version is consistent with the update dependencies. When these updates become k-stable, they can be sent to clients that want to replicate them. To detect which updates are k-stable, each data center stores a version vector that is causally updated when it receives k - 1 acknowledgements from other data centers. To reduce the size of the timestamp, the dependencies may not take into account all existing data centers. They may only mention those that were read/written before the update.

SwiftCloud k-stability can be a promising fault tolerance measure for an edge deployment. It has a negative impact on visibility of remote updates, but avoids causal consistency violations when replicas fail. Delivering remote updates to the client cache is not the best option for the edge, as replicas must maintain a specific state for each client. This approach hinders scalability. In addition, clients are the only partial replicas, which means that servers must store all objects.

4.17 Legion

The main motivation for this system [Lin+17b] is to change the interactions in collaborative applications. In the past, there were data centers with replicas of the objects and clients had to use them as an intermediary. This leads to long delays in receiving updates and blocks the distribution of changes when the data center is unavailable. Therefore, Legion allows direct communication between clients.

A client device does not have the available storage to be a complete replica of a data center. To get around this problem, each client replicates a subset of all objects and organizes them into one or more containers. For example, one container might contain all the objects that belong to a collaborative job. Although clients can use a subset of the container, they still need to replicate the entire container. This can be considered a non-authentic partial replication method, like PRACTI. Each object in the container has its own timestamp, with one part per replicating node. Unlike the systems presented earlier, Legion has entries with physical clocks.

When a client wants to join a container, it must connect to a number of nearby and distant nodes. They are chosen based on their latency to known servers, which is calculated by the RTT of ping messages. The difference between these two types of

nodes allows for lower latency visibility for updates as well as greater tolerance to network partitions.

Reads and writes are applied to locally available versions of objects. A write creates a new version of the object, whose timestamp is changed on the local node's input to the current clock value. At the same time, the update is added to the container version chain. This structure allows only the differences between the old and new container state to be transmitted, encoded with CRDTs and transferred in the background. Between each pair of clients, there is a FIFO channel, which preserves the causal order on the replication of updates.

In order to cope with legacy applications, a special client is responsible for communication with the data center. Legacy applications may not have the changes necessary for direct communication with the client, which leads to the client sending local updates only to the data center. The special node maintains global consistency by sending changes from inter-client communication to the data center and disseminating new versions from the data center among its peers.

The ability for clients to propagate updates among themselves is an interesting prospect for the network edge. It could reduce the load on servers and speed up the delivery of updates. However, synchronization between nodes may not allow the current version of the protocol to be used at the edge. In addition, the multipart timestamp has one entry per replication node. This means that more popular objects have more metadata, which may not be possible to process efficiently.

4.18 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.

4.18.1 Summary of existing systems

Table 4.1 summarizes the described systems to simplify their comparison. We characterized them based on our taxonomy: the key technique behind their implementation of causal consistency, the amount of metadata used to repre- sent 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 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.

Other approaches avoid sequencers while tracking dependencies more precisely. Unfortunately, these systems may generate a very large amount of metadata, incurring a significant overhead due to its management costs.

As a reaction to the limitations of previous approaches, researchers have envisioned solutions based on background stabilization mechanisms. 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.

Finally, as mentioned before, Kronos and Eunomia 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. Eunomia falls between the sequencer-based and the background stabilization techniques. It relies on a per-datacenter service, called Eunomia, 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, Cure and Okapi. This is an interesting design that permits Eunomia 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, Eunomia is a less scalable solution than approaches that rely on background stabilization mechanisms. The latter do not exhibit any potential performance bottleneck, while Eunomia employs a single receiver per datacenter.

Tab. 4.1.: 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.

System	Key Technique	Metadata Size	False Dependencies	Partial Replication	Dissem. Scheme
Bayou	Sequencer	Scalar O(1)	I+P+DC+G	no	pair-wise
TACT	Sequencer	Scalar O(1)	I+P+DC+G	no	pair-wise
PRACTI	Sequencer	Scalar O(1)	I+P+DC+G	non-gen	pair-wise
Lazy Repl.	Sequencer	vector[dcs] O(M)	P+DC	no	all-to-all
ChainReact.	Sequencer	vector[dcs] O(M)	P+DC	no	all-to-all
SwiftCloud	Sequencer	vector[dcs] O(M)	P+DC	at the Edge side	all-to-all
Legion	Sequencer	matrix [dcs][part.] O(MxN)	P+DC	at the Edge side	all-to-all
COPS	Explicit check	vector[keys] O(K)	I	no	all-to-all
Eiger	Explicit check	vector[keys] O(K)	I	no	all-to-all
Bolt-on	Explicit check	vector[keys] O(K)	I	no	all-to-all
Karma	Explicit check	vector[keys] O(K)	I	non-gen	all-to-all
Orbe	Explicit check	matrix [dcs][part.] O(MxN)	P	no	all-to-all
GentleRain	stabilization	scalar O(1)	I+P+DC+G	no	all-to-all
Okapi	stabilization	scalar O(1)	I+P+DC+G	no	all-to-all
Cure	stabilization	vector[dcs] O(M)	P+DC	no	all-to-all
POCC	lazy resolution	vector[dcs] O(M)	P+DC	no	all-to-all
Occult	lazy resolution	vector[part.] O(N)	I+P	no	master-slave
Kronos	-	all O(graph)	none	non-gen	all-to-all
Eunomia	-	vector[dcs] O(M)	P+DC	no	all-to-all

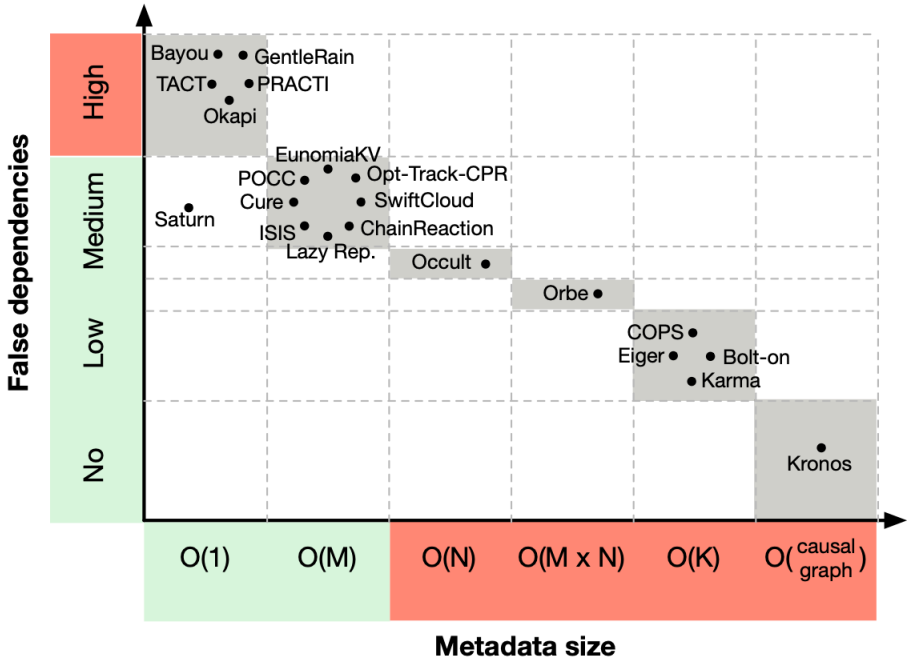

Fig. 4.1.: Graphic distribution of existing causally consistent systems based on the metadata size used to capture causal dependencies and the amount of false dependencies that each solution generates. Colored cells represent the diagonal. M, N, and K refers to the number of datacenters, partitions and keys respectively. Colored in green (lightly colored) and red (darkly colored) the different sizes of metadata and types of false dependencies.

4.18.2 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 4.1 shows the graphic distribution of existing systems 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 exclusively from the metadata. Thus, on the one hand, when metadata is aggregated, false dependencies induce poor remote visibilities compared to systems tracking causality

more precisely. On the other hand, when metadata is not aggregated, the associated computation and storage overhead has an impact in throughput.

Part II

Contributions

In this chapter, we present the design, algorithms and architecture of COLONY, a group collaborative database that addresses the challenges of edge replication. It efficiently ensures causally consistent, available, and convergent access to high number of client replicas, tolerating failures. COLONY allows the development of collaborative applications with seamless support for replication at the client machine leveraging peer-to-peer interactions to propagate operations among clients.

We first describe an abstract design that addresses the challenges of client-side replication, starting with the no-failure case, and then, how we support DC failure. Our design demonstrates how to apply the principles of causally consistent algorithms for full-replication systems to build cloud-based support for partial client replication. To avoid the complexity of consistent client-side, device-wide partial replication on the client side, we leverage full DC replicas to provide a consistent view of the database to the client. The client merges this view with its own updates. To tolerate DC failures, we explore a trade-off between data liveliness and freshness, and serve clients a slightly delayed, but consistent and sufficiently replicated version.

Then, we describe how COLONY clients can synchronize directly among each other, using a peer-to-peer interaction model. To support these interactions, geographically close clients form overlay networks to propagate objects and updates among them. Which induces low latency for propagating updates and objects between the nearby clients. However, while leveraging direct client interactions brings significant benefits, it also creates security challenges. We address these challenges by making it impossible for an unauthorized client to access objects or interfere with operations issued by authorized clients. Our design uses lightweight cryptography and benefits from access control mechanism of the cloud to securely distribute keys among clients.

We then discuss and justify our design and specify our consistency choices, where we take a hybrid approach, providing the highest consistency guarantees compatible with availability and convergence (TCC+), strengthened further (to SI) in proximity connected zones. A related challenge is the overhead of concurrency metadata (fat vector clocks), which we limit thanks to a flexible forest topology and to SI zones.

Finally, this section addresses a number of design challenges, including disconnected operations, consistency under client and groups migration, total-order consensus at the edge, transaction migration, and avoiding single points of failure despite the tree topology.

Our approach has been implemented as a software named COLONY [Tou21] and published at the ACM Middleware Conference 2021 [TSS21].

5 Protocol design

5.1 Design Objectives

Our requirements are to provide a highly-available and fast access to shared data, with uniform semantics across the core-edge spectrum, while ensuring the strongest possible consistency guarantees.

COLONY provides distributed access to shared data. A client application process can execute transactions at an arbitrary node in the cloud-edge spectrum, with uniform guarantees, enabling nodes or code to migrate seamlessly.

In light of what we saw in the existing work, we will here justify some protocol choices used in our approach, which aims to reduce shortcomings observed in some existing systems.

We turn now to a system design for satisfying the above requirements efficiently. Our design is an extension of the SwiftCloud approach [Zaw+15a].

COLONY uses caching and replication to ensure that a client can execute locally. The system must remain safe at all times; specifically, the data observed by a client always satisfies the TCC+ and security invariants defined below. It should also remain available.

The trade-off is that, during some failures, liveness cannot be ensured. A client cannot make progress in two cases: if it requires data that cannot be retrieved; or if it runs out of storage. Furthermore, there are corner cases (described later) where a client commits updates, but they cannot become visible. The above situations are temporary, and last only until the problem is repaired.

Our system ensures convergence by using operation-based CRDTs, which merge concurrent conflicting operations deterministically [Sha+11b]. As underlined in the Introduction, supporting causal consistency (CC) can have high metadata overhead; our design bounds metadata to a small size. Similarly to recent CC designs [BRVR17b; Akk+16a], COLONY separates (internal) state management from (external) visibility: the backend layer transmits and stores states efficiently, without

regard for correctness, whereas the visibility layer manages metadata and ensures that an application observes only those states that satisfy the TCC+ guarantees.

5.2 The TCC+ Guarantees

We now tersely specify the TCC+ guarantees. We use the following notations and definitions. Nodes (at any level of the topology) are noted p, p'. A node behaves sequentially, executing one transaction at a time. A node might fail, in which case it ceases executing (fail-stop); a node that does not fail is said correct. x, y designate data objects. Transactions are noted T, T'. A transaction consists of a sequence of reads and updates. A transaction is interactive, i.e., the objects it accesses are not known in advance. A read has no side effect; an update does not return a response value. We write $a \in T$ when operation a (a read or an update) belongs to transaction T. A transaction executes at a single replica; if it commits, its updates are broadcasted to be replayed by the other replicas. An operation is noted $a, b, \ldots$; in more detail, updating x is written $u(x)$, and a read $r(x)$. The response value of $r(x)$ is $\mathsf{res}(r(x))$.

Following Viotti and Vukolić [VV16], an abstract execution $A = (H, \overset{vis}{\rightarrow}, \overset{ar}{\rightarrow})$ consists of an interleaving of operations executed by the nodes, or history (H), a visibility relation ($\overset{vis}{\rightarrow}$), a partial order that accounts for the propagation of updates in the system, and the arbitration relation ($\overset{ar}{\rightarrow}$), a total order over H that helps to resolve concurrency conflicts. The order in which nodes execute operations is called the program order. The happened-before relation ($\prec$) is the transitive closure of the union of visibility and program order [VV16].

Hereafter, we consider only transactions that commit; we can safely ignore the operations of a transaction that aborts, since it has no effect.

Note that, with TCC+, even a transaction that aborts reads a consistent snapshot.

The phrase "visible in node p" refers to an operation that is visible by some operation executed at node p.

Each object starts in some known initial state. The return value of a read is computed according to the semantics of prior updates to the object (including updates in the same transaction). That is, for each read operation $r(x)$, $\mathsf{res}(r(x))$ results from some linearization $l_{r(x)}$ of the updates visible to $r(x)$ consistent with $\prec$.

TCC+ is defined by the following invariants.

Causal Consistency (CC) Causal consistency requires that every update that happened-before an operation is visible to that operation, and that arbitration is consistent with happened-before. Formally, $(\prec \subseteq \overset{vis}{\rightarrow}) \wedge (\prec \subseteq \overset{ar}{\rightarrow})$.

Rollback-freedom Once a node has read a value, it does not roll it back: If $r(x) \prec_p r'(x)$ then $l_{r(x)}$ prefixes $l_{r'(x)}$.

Strong Convergence Any two nodes that observe the same set of updates read the same value. Formally, $\forall r(x), r'(x) : (\forall u(x); u(x) \overset{vis}{\rightarrow} r(x) \iff u(x) \overset{vis}{\rightarrow} r'(x)) \implies \mathsf{res}(r(x)) = \mathsf{res}(r(x))$.

The above invariants constrain the behavior of individual operations. Below, we formalize the fact that a transaction is atomic (i.e., all-or-nothing). We define the following equivalence relation, written $\equiv$: if operations a and b are in the same transaction T, then $a \equiv b$. For some relation R over the set of operations, we say that R is left-compatible with $\equiv$ when for any three operations a, b and c, if $a \equiv b$ and $(a, c) \in R$ then $(b, c) \in R$. Right-compatibility is defined symmetrically, that is $b \equiv c \wedge (a, b) \in R \implies (a, c) \in R$. Relation R is compatible with $\equiv$ when it is both left- and right-compatible with it.

Atomicity If two updates occur in the same transaction, then they are visible atomically, and arbitrated in the same way. Formally, visibility and arbitration are compatible with transactional $\equiv$.

Snapshot A transaction takes all its reads (independently of their order) from a same snapshot, which is sound both causally and for the atomicity relation.

Additionally, the following liveness property should hold:

Eventual Visibility If two correct nodes p and p' are not permanently disconnected from one another, and $u(x)$ is visible in p, then eventually $u(x)$ is visible in p'.

TCC+ extends Transactional Causal Consistency, as defined by Zawirski et al. [Zaw+15a], with the Strong Convergence and Rollback-Freedom properties. This ensures that progress is monotonic at each node.

To illustrate the concepts in this section, consider the history in Figure 5.1, which depicts the evolution of a CRDT counter (x) when nodes execute increment operations (inc), and propagate such updates (depicted by arrows).[1]

The history in the figure is causally consistent. Indeed, every new increment updates the counter to a state also containing the preceding operations (e.g., after the event ⑥, the counter value is 2). Similarly, there is no rollback at any node. Nodes that received the same increments (e.g., events ⑦ and ⑧) are in the same state; therefore this history satisfies strong convergence. Moreover, since every transaction contains a single operation, the history trivially ensures the atomicity and snapshot requirements.

5.3 Strengthening to SI

COLONY strengthens the above TCC+ guarantees to strong consistency in an SI zone. Each DC forms a distinct SI zone. Furthermore, a group of edge nodes in network proximity may elect to form an SI zone (which an individual node can join or leave using standard membership mechanisms).

In a SI zone, COLONY ensures Snapshot Isolation (SI). This means that $\stackrel{ar}{\rightarrow}$ is gapless [SSAP16], i.e., for any operation b visible to a, every operation $c \stackrel{ar}{\rightarrow} b$ is also visible to a.

5.4 Bounding metadata

This and the following sections detail the logic to achieve the above consistency guarantees.

Supporting CC requires metadata, which can represent a substantial overhead; this section explains how COLONY bounds metadata to a small size.

The CC invariant dictates that an update may become visible only if its *dependencies* (i.e., the updates that happened-before it) are themselves visible. To check this, when transmitting an update, COLONY piggy-backs some associated *visibility metadata*, a vector timestamp (or version vector) that summarizes its dependencies [Mat89; Fid88]. Vector timestamps support efficiently computing the set of

[1] For now, ignore the version, commit and snapshot information, which will be detailed later.

missing dependencies [Pet+97]. (in contrast with storing explicit dependencies [Llo+11a])

A precise representation of the happened-before order among N concurrent writers requires a vector of size $\geq N$ [CB91]. As N grows, the overhead on every message quickly becomes unacceptable.[2] The following sections describe some techniques that we use to keep the size small, at the cost of spuriously ordering some concurrent events. (i) Since under SI writes are totally ordered, the set of nodes in an SI zone all count for one, and require only a single component. (ii) The metadata for concurrent nodes connected to a common server node can be reduced to a scalar (called a *dot*) on top of the vector [Alm+14]. (iii) As a safe approximation is sufficient to enforce the CC invariant, size N can be made as small as desired [TRA99]. (iv) Connecting nodes in a tree of FIFO links also provides a safe approximation of CC, without added metadata [BRVR17b].

5.5 Topology and metadata design

We first turn to the topology design (illustrated in Figure 3.1) and the metadata design.

Each DC forms an SI zone; therefore, the updates of a given DC are totally ordered; externally, it behaves as a single sequential node. On the other hand, DCs are connected in a full peer-to-peer mesh; their updates are partially ordered, which requires a vector. Since each DC appears sequential, a timestamp vector V of size N suffices to a point in the CC partial order between DCs. Component $V[i]$ numbers the (sequentially ordered) transactions committed at DC i.

The *least upper bound* (LUB) of two vectors is defined as their component-wise maximum. Each node maintains its *state vector*, which is the LUB of the commit timestamps (defined next) that it has observed.

A transaction has a unique identifier called its *dot* [Alm+14].

Successive transactions submitted at the same node have monotonically-increasing dots.

Edge nodes (border or far-edge) are partitioned into distinct trees. A tree is rooted at a specific DC, which we call its *connected DC*.[3] This is illustrated in Figure 3.1.

[2] In COLONY each component of the vector is 8 bytes, in order to store a monotonic clock that does not wrap around.

[3] A peer group counts for a single node in the tree.

A subtree may detach itself from its parent and *migrate* to a different tree, e.g., to accommodate mobility or a failure.

5.6 Transaction metadata

We now describe the metadata that COLONY associates with a transaction T: its snapshot and commit timestamp vectors, and its dot.

Transaction T's *snapshot vector* $T.S$ describes the (previous) transactions it depends upon. $T.S$ forms a snapshot closed under causal consistency and atomicity. The meaning of $T.S[j] = n$ is the following: T reads from all the transactions T' committed at DCj up to time n, and no later, i.e., such that $T'.C[j] \leq n$.

A read-only or aborted transaction terminates without side effects. The *commit vector* $T.C$ of an update transaction represents the point where it commits.[4] It is greater than its snapshot vector; if the transaction commits at DCi, they differ only at index i, i.e., $T.S[i] < T.C[i] \wedge \forall j \neq i : T.S[j] = T.C[j]$, where $T.C[i]$ is a timestamp assigned by DCi.

Transaction T is before T' if $T.C \leq T'.S$. If neither of T or T' is before the other, they are said concurrent.

Finally, a transaction has a unique timestamp called a *dot* $T.D$, which both serve as a unique identifier and provides the (total) arbitration order between concurrent transactions (as defined in Section 5.2).

Figure 5.1 description: This system has three data centres DC0, 1, and 2, and edge nodes A and B. Vector components refer to DC0, 1 and 2 respectively. Dots are omitted from the figure. The k-stability objective is 2.

❶ Initially, the DCs observe $x = 0$ with version vector $[0, 0, 0]$.

❷, ❸ Transaction T0 (resp. T1) increments x at DC0 (resp. DC1), committing $x = 1$ with version $[1, 0, 0]$ (resp. $[0, 1, 0]$). Both have $k = 1$.

❹ DC0 transmits T0 to DC2. Being replicated at two DCs, it is 2-stable ($k = 2$).

[4] In fact, to tolerate faults, it may have multiple equivalent commit vectors, as described later, in Section 5.9.

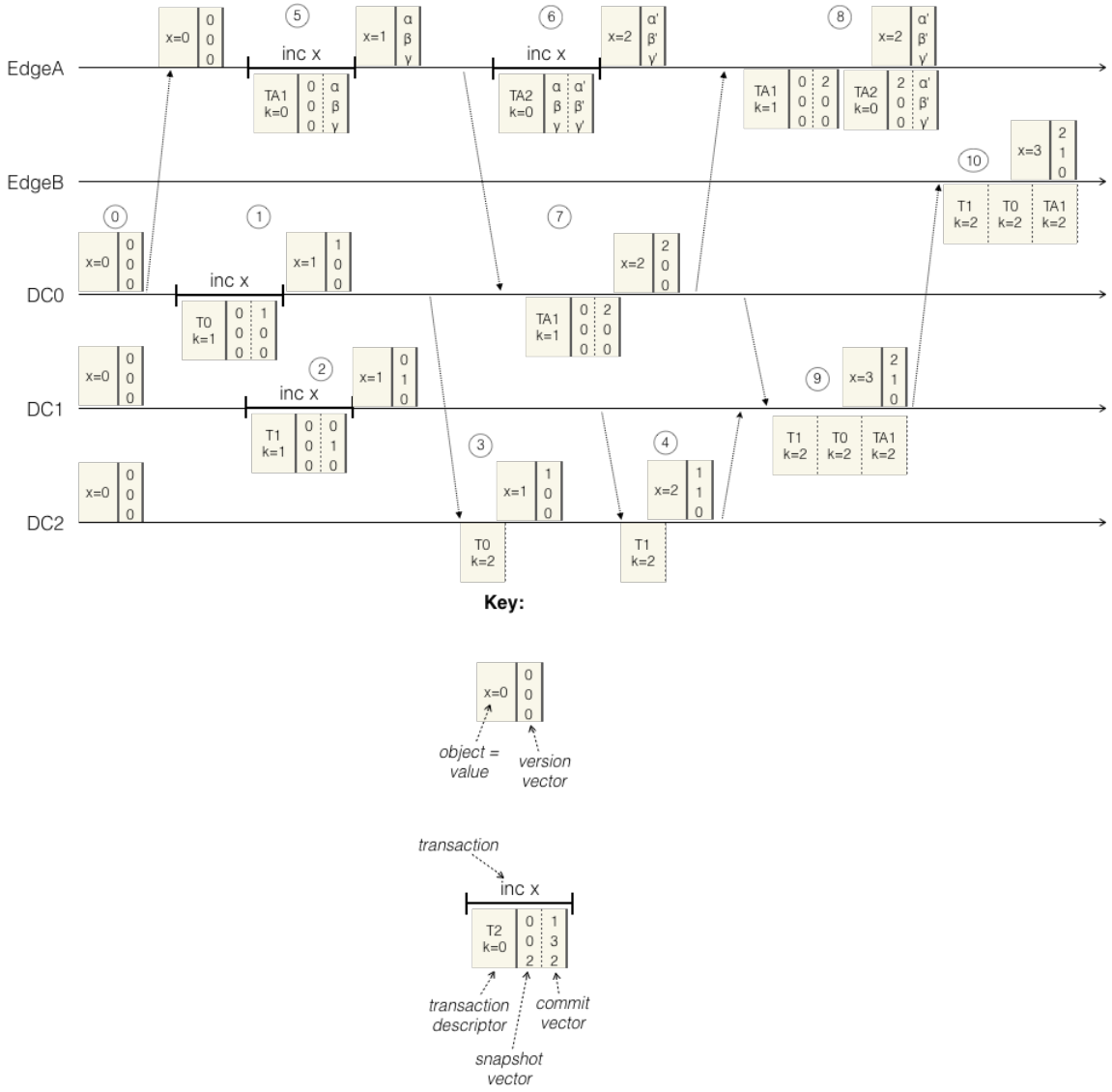

Fig. 5.1.: DC and edge transaction protocols

4. DC1 transmits T1 to DC2. T1 is now 2-stable. DC2 observes two increments, T0 and T1: now $x = 2$ with version $[1, 1, 0]$, the least-upper-bound of the commit vectors of T0 and T1.

5. EdgeA caches x. Transaction TA1 increments x and commits locally. Its commit vector remains the symbolic $[\alpha, \beta, \gamma]$. As TA1 has not been transmitted to a DC, it has $k = 0$.

6. Transaction TA2 at EdgeA again increments x. Its snapshot vector is symbolic $[\alpha, \beta, \gamma]$; its commit vector is symbolic $[\alpha', \beta', \gamma']$. The value of x at EdgeA is now 2.

7. Concurrently, EdgeA transmits TA1 to DC0. DC0 assigns commit vector $[\alpha, \beta, \gamma] := [2, 0, 0]$. TA1 being known in one DC, it has $k = 1$. DC0 observes two increments, T0 and TA1: now $x = 2$, with version $[2, 0, 0]$.

8. DC0 transmits back the concrete descriptor of TA1, informing EdgeA that $[\alpha, \beta, \gamma] = [2, 0, 0]$. EdgeA observes local increments TA1 and TA2, hence $x = 2$. T0 is not visible to EdgeA because $T0.k = 1$.

9. DC0 transmits T0 and TA1 to DC1, making them both 2-stable.

10. T0, T1 and TA1 being 2-stable at DC1, are made visible to EdgeB, where $x = 3$.

Eventually (not depicted): TA2 is delivered to a DC, filling in the values for $[\alpha', \beta', \gamma']$; TA2 becomes 2-stable; all four transactions reach all replicas; all replicas observe $x = 4$.

5.7 In-DC transaction protocol

Let us describe how the system computes metadata in the simple case of a transaction that executes within some DCi. By default, $T.S$ is assigned the current state vector of DCi. The system checks that $T.S$ represents a consistent cut [Akk+16a; SASS13] such that $T.S[i] \leq current\_time$. Its unique dot is $T.D := (current\_time, i)$.

When the transaction reads an object x, the system returns its latest version included in the snapshot; choosing a snapshot vector less or equal to the state of DCi ensures that every read can be satisfied. When the transaction updates y, the update is buffered, and further reads of y in the same transaction are satisfied from the buffer.

The commit protocol is a standard two-phase commit among the servers of DCi (we use ClockSI [DEZ13]). The commit vector is equal to the snapshot vector, except that $T.C[i] := current\_time$. Object versions created by the transaction are marked with version timestamp $T.C$.

As COLONY objects are operation-based CRDTs, materializing a version may require to apply multiple updates [Sha+11b; Bur+14]. Conversely, concurrent transactions that update the same CRDT object can be merged and by default do not abort, although it can abort for semantic reasons, e.g., if it would violate some invariant; we assume a higher level of concurrency control to detect such violations [Got+16; HL19; Alv+11], which is out of the scope of this paper.

When a DC receives a new remote transaction, it applies its updates to the local store. Concurrent updates are merged according to CRDT semantics; the merged version's timestamp is the least upper bound of the corresponding commit timestamps.

We illustrate the in-DC transaction lifecycle in Figure 5.1, events ⓪ through ④. Focus on the three DCs, numbered 0, 1 and 2, and on the CRDT counter x.[5] ⓪ All DCs have a copy of x with value 0 and version timestamp $[0,0,0]$. The three DCs are in state $[0,0,0]$. ① DC0 executes transaction T0. Its snapshot vector is set from its current state, at $[0,0,0]$. T0 increments x. Its commit vector is $[1,0,0]$, i.e., its snapshot vector incremented by 1 at the component for DC0. It commits version $[1,0,0]$ of x, with value 1. ② Concurrently, DC1 executes T1, with the same snapshot. T1 also increments x. As x is a CRDT, T1 can also commit; its commit vector is $[0,1,0]$ and it updates x to version $[0,1,0]$ with value 1. At this point, T0 is visible only to DC0, and T1 only to DC1. ③ DC0 replicates T0 to DC2, where x has version $[1,0,0]$. ④ DC1 replicates T1 to DC2. All three versions of x are visible at DC2, as well as a merged version with least-upper-bound timestamp $[1,1,0]$. As this version includes the increments from both T0 and T1, its value is 2.

5.8 Basic Edge Transaction Protocol

A transaction may execute and commit in an edge node. In this case, commit is asynchronous, i.e., for availability, the edge node continues to execute further transactions without waiting for the DC to assign its commit vector.

Starting a transaction T is similar to the in-DC case: the edge node assigns its snapshot, and a dot using the edge node's unique identifier. The transaction commits

[5] Ignore for now the two edge nodes, and references to k and to stability.

locally at the edge node, which can immediately start another dependent transaction. Until it receives the DC's acknowledgement, the commit timestamp remains *symbolic*, i.e., indeterminate, subject only to the invariant $T.S < T.C$.

Eventually, the edge node sends the transaction to its connected DCi, which acknowledges with a concrete commit vector. Similarly to the in-DC case, the commit vector differs from the snapshot only in the component corresponding to the connected DC: $T.C[i] := current\_time$. However, because of migration, index i cannot be predicted, as described in the next section.

Let us return to Figure 5.1 to illustrate the lifecycle of edge transactions. ⑤ Edge node A pulls x into its interest set, then starts transaction TA1 with snapshot vector $TA1.S = [0, 0, 0]$. TA1 reads version $[0, 0, 0]$ of x. TA1 increments x. TA1 commits; its commit vector is still uncertain, noted with the symbolic $TA1.C = [\alpha, \beta, \gamma] > [0, 0, 0]$. ⑥ EdgeA starts a second transaction TA2. To be able to read the writes of TA1 from the local cache, it assigns snapshot vector $TA2.S = [\alpha, \beta, \gamma]$. TA2 increments x and commits with symbolic $TA2.C = [\alpha', \beta', \gamma'] > TA2.S = [\alpha, \beta, \gamma]$. ⑦ Concurrently, EdgeA sends TA1 to DC0. Similarly to an in-DC transaction, DC1 assigns the commit vector $TA1.C = [\alpha, \beta, \gamma] := [1, 0, 0]$. ⑧ EdgeA receives the updated descriptor for TA1 and fills in the concrete values $TA1.C = TA2.S = [1, 0, 0]$.

5.9 Node Migration and K-Stability

The topology should be arranged to minimize response time and maximize availability. From this perspective, a fixed tree is inflexible, as a single faulty link disconnects a whole subtree, and as latency changes when a mobile node travels around.

A fixed tree is inflexible, and a single fault may have a disproportionate impact.

Therefore, COLONY supports migrating a node and the subtree attached to it. Ideally, node migration should be seamless and transparent to applications, but unfortunately this is not completely possible.

Migration creates some extra complications to the edge transaction protocol, which we consider next. For simplicity, we focus on the case of a single migrating edge node. Hereafter, we focus on the migration mechanism, and ignore the policy decision of why or when to migrate, e.g., in response to a network failure.

Avoiding Duplicates Migration can change the connected DC of the node. Consider the edge transaction protocol described above. Suppose that some edge node sends its transaction T to its connected DCi, loses the connection to DCi, then migrates to DC$j \neq i$. As the edge node does not know whether DCi received T, it sends T again to DCj. Although T might now be received twice, via both DCs, a replica should replay it only once; the transaction's dot $T.D$ serves to filter out such duplicates.

According to causal ordering, dots must be received in monotonically non-decreasing order.

To this effect, every node keeps track of the highest dot assigned by another node, and ignores a transaction whose dot is less or equal this value.

K-stability to avoid causal incompatibility The migrating edge node and its new connected DCj each transmits the updates it has visible and the other not.

Recall that snapshot $T.S$ was chosen less or equal to the state of DCi, ensuring that its dependencies will be satisfied.

Consider an edge node that migrates from DCi to a new connected DCj. If the state of DCj includes that of DCi, the edge node's dependencies remain satisfied, and migration is seamless. We say that the states are causally compatible.

However, it might happen (for instance, because of a communication failure) that an edge transaction T' depends on a transaction T that was visible at DCi but not yet at DCj. The snapshot of T' does not satisfy the CC invariant at DCj, which cannot apply it and cannot assign its commit vector. The edge node remains effectively disconnected, and its transactions are non-visible to the rest of the system. We say the edge node state is incompatible with DCj.

If T was not visible to T', the above dependency could not exist, and the nodes would remain compatible. Thus, one possible approach would be to let transaction T become visible at the edge only once it is known at all DCs. However, a single slow DC would delay edge visibility of all transactions.

Our solution, taken from SwiftCloud [Zaw+15a], is twofold. First, to ensure the Read-My-Writes session guarantee [Ter+94], an edge node's transactions are always visible to itself. Second, to decrease the probability of incompatibility, transaction T becomes visible to edge nodes only after it is visible by $\geq K$ DCs, where $1 \leq K \leq N$ [Zaw+15a]. The higher K, the higher the probability that the new DC is compatible with the old one, i.e., that its state includes the dependencies of T'. The exact value of K is a trade-off between two extremes. If $K = 1$, the probability of incompatibility is

high. If $K = N$, a single slow DC could prevent all edge transactions from becoming visible.

To illustrate K-stability, refer again to Figure 5.1. $T.k$ counts the number of DCs where T is stable. The visibility limit is set to $k \geq K = 2$. At ⑧,

even though DC0 transmitted its state to EdgeA,

Transaction T0 is not visible to EdgeA, because $TO.k = 1$. At ④, DC1 sends T1 to DC2; now $T1.k = 2$. At ⑨, DC0 transmits T0 and TA1 to DC1; now all three transactions T0, T1 and TA1 have $k = 2$. As DC1 has observed three increments, $x = 3$ with version $[2, 1, 0]$, the least-upper-bound $TC0.C, TC1.C$ and $TA1.C$. Therefore, in ⑩, DC1 can make them visible to EdgeB, where later transactions may depend on $x = 3$ with version $[2, 1, 0]$.

Transaction Ordering Finally, both DCi and DCj may assign different commit vectors, $T.C_i \neq T.C_j$. This could cause an ordering anomaly: if some transaction T' could depend on T with $T.C_i \leq T'.S$, where $T.C_j \not\leq T'.S$; a node that knows only of $T.C_j$ is not aware that T happens before T'. Observe, however, that $T.C_i$ and $T.C_j$ conceptually denote the same point in the TCC+ partial order. To ensure this concretely, COLONY considers the two timestamps as equivalent; they already have the same causal past, and the equivalence ensures they have the same causal descendants.

Thus, a same transaction may carry up to N equivalent commit timestamps.

We optimize their memory size as follows. Recall that a commit vector differs from the snapshot vector in a single component, that of the DC that accepted it; the others are not significant. Therefore, COLONY stores multiple commit vectors into a single vector of size N, containing a significant value only for a DC that accepted the transaction. For simplicity, Figure 5.1 does not depict this optimization.

5.10 Transaction Migration

Running at the edge provides fast response and high availability. However, an edge device has limited resources, such as CPU capacity, memory, power and bandwidth resources. In contrast, the core cloud is rich in resources and can ensure better consistency. There is a trade-off and some computations are better placed in the core. Examples include analytics or large queries. Although we expect edge execution to

be the common case, COLONY also supports moving a transaction from the edge, to execute in a trusted environment at the connected DC.

If the state of the DC differs from the edge node, the result might be inconsistent. Thanks to TCC+, this is not an issue. The result of a transaction depends only on its reads, i.e., on its snapshot. To ensure consistency, it suffices to run the transaction under the same snapshot vector as if it was running on the client device.

Resource-hungry transactions should run in the core cloud rather than the edge. Examples include analytics or large queries. COLONY supports migrating them to a trusted node in the core cloud for execution.

The migrated transaction must have the same effect as if it ran on the edge node; only performance should differ. Thanks to TCC+, it suffices to assign the same snapshot vector.

Thus, the client primes the snapshot with its own state vector and sends the transaction code. Before the transaction starts, the DC must have received the client's local transactions, which that the new one depends upon (Section 7.1.3 explains how we accelerate this).

Since the edge node's state is less or equal the connected DC's, plus its local transactions, this ensures that every read can be satisfied.

This ensures that every read can be satisfied.

The migrated transaction executes in the DC just like a standard local client, and its results are sent back to the requesting edge node.

6 Data Management

COLONY ensures convergence by using operation-based CRDTs, which merge concurrent conflicting operations deterministically [Sha+11b]. Similarly to recent CC designs [BRVR17b; Akk+16a], COLONY separates (internal) state management from (external) visibility: the backend layer transmits and stores the state efficiently, without regard for correctness, whereas the visibility layer manages metadata and ensures that an application observes only those states that satisfy the TCC+ guarantees.

6.1 Versioning

COLONY stores an object persistently as a base version and a journal of updates since the base version. To materialize an arbitrary object version, the cache first reads the base version from the store, and applies the missing updates from the journal. Occasionally, the system advances the base version.

A transaction reads from its snapshot, logs its updates to the journal, and materializes new versions in a private buffer. When the transaction commits, it updates the cache from the buffer. Both the updates recorded in the journal, and object versions that result from committed transaction T, are labeled with vector $T.C$ and dot $T.D$.

6.2 Edge caching

An edge node cannot replicate the whole database, but can only cache some small fraction of it. An edge client may declare *interest* in some object to add it to its node's cache. The connected DC regularly informs the client of updates to its interest set.

At any point in time, the state vector of an edge node is the LUB of the state received from its connected DC (itself $\leq$ the DC's current state vector) and the commit vectors of local transactions. Choosing a snapshot vector $\leq$ the node's state vector ensures that every read could be satisfied either from the local cache or from the

connected DC. It may happen that the client requires an object version that cannot be retrieved (in the cache, from the DC or from another node), in which case the transaction cannot proceed. This limitation of availability is inherent to the edge environment.

7 Groups

COLONY supports two distinct group mechanisms: the peer group, an SI zone at the edge, and the collaboration group, nodes that update the same data. Peer groups are disjoint, whereas collaboration groups may overlap. All nodes in a peer group are in the same collaboration group.

7.1 Peer groups

A peer group is a set of nodes with high-availability, low-latency connection to one another.

It makes sense to provide SI within the group. This enhances the user experience, and simplifies metadata management. A peer group creates opportunities to improve performance, by pooling resources into a collaborative cache, and to decrease network load to the cloud by collecting the updates from many clients.

It may also isolate itself from the cloud, keeping out distracting remote updates and improving security.

In summary, a read by a member of the group is satisfied either from its local cache, or from that of a neighbor, or from the connected DC. Reads and writes satisfy the TCC+ and SI guarantees, as well as security guarantees described elsewhere.

Conceptually, a peer group consists of four related components, with distinct roles: managing group membership, sharing content within the group, communicating with the outside, and enforcing the SI order. They are described in further detail below.

7.1.1 Membership

Membership of a peer group is seeded and managed by a single node, called the group's *parent*. The parent maintains a connection to each of the group member, stores the list of the members, and informs them of any membership change. The

parent is fixed but arbitrary, possibly located in the DC or on a point-of-presence (PoP) server. A node may serve as a member and a parent at the same time.

To join or leave a group, a node contacts the group's parent. The parent responds with the membership list, as well as the session security key (described shortly). When a node migrates between groups, it uses the migration protocol previously described (Section 5.9); the new group must be causally compatible with the node's state.

7.1.2 Content Sharing

Using the membership list, the group members and their parent maintain point-to-point connections. Above these connections, they construct a collaborative cache using a simple peer-to-peer protocol. Each member publishes its current interest set to all its neighbors (other members and parent). This subscribes the member to receive all updates to its interest set. When a member updates an object in a neighbor's interest set, it pushes that update in a best-effort manner. Conversely, if a member observes that it is missing an update to its interest set (by examining the visibility log described below), it pulls the transaction from some neighbor. Objects evicted from a cache are unsubscribed to save resources.

When a node first joins a group, or when it extends its interest set, some neighbor will push the objects it requires, as described above.

The parent maintains an interest set that is the union of those of the group members. It subscribes for updates outside the peer group on behalf of its members, as detailed next.

7.1.3 Communicating Outside the Group

As illustrated in Figure 3.1, a subtree communicates with another one via some common ancestor. For simplicity, the following description assumes this ancestor is its connected DC.

1

[1] So far, we glossed over the fact that a DC contains a number of concurrent shard servers. The group might communicate with any of these servers.

Let us call *synchronization point* (sync point) a node within a group that communicates with the DC. In the common case, this is the parent, but any member may also unilaterally become a sync point (for instance before migrating a transaction to the DC, Section 5.10), thus avoiding any single point of failure.

A sync point sends all the updates that the DC is missing, and symmetrically subscribes to updates to its interest set. Importantly, the sync point makes updates visible to the DC in the visibility order described in the next section. This ensures that different sync points send identical information.

7.1.4 Transaction Protocol for Peer Groups

A peer group as a whole should behave like a single, sequential edge node, from the perspective of the rest of the system. To ensure sequential ordering, causality and progress within the group, COLONY relies on EPaxos [MAK13] within the peer group. Compared to other consensus protocols, EPaxos improves availability and performance, by allowing any group member to become the leader for some particular transaction, and by minimizing synchronization between non-conflicting transactions.

In addition to improving the user experience, consensus is essential to correct metadata management.

Recall from Section 7.1.3 that possibly multiple sync points send transactions to the DC. Without consensus, conflicting transactions would be sent in different orders, breaking causality and causing unsafe commit vectors.

When a peer node commits a transaction, it submits it to EPaxos. EPaxos ensures consensus on the order in which versions become visible sequentially according to SI, which we call *visibility order*. Every peer maintains the list of visible transactions in a *visibility log*.

The transaction then executes in isolation against the local cache. Its dependencies are the union of the state vector, the node's previous transactions, and the transactions in the node's visibility log.

The node assigns it a unique dot, executes and persists its updates in a local journal, and updates its local cache.

Within a peer group, two different variants of commit exist. In the first, the node submits the transaction to EPaxos in the critical path of commit. This has the effect of

ordering the commitment of conflicting transactions within the peer group, possibly leading to aborts; non-conflicting transactions commit in parallel. This variant maintains Parallel Snapshot Isolation (PSI) within a group [Sov+11], ensuring that the group behaves as an SI zone.

The second variant follows a similar approach to Section 5.8. It assumes that transactions never conflict. The transaction commits locally as soon as it reaches the commit statement, and a new transaction can follow immediately. The transaction is then submitted to EPaxos in the background.

Committed transactions become visible in the order assigned by EPaxos. A sync point sends visible transactions to the connected DC according to the visibility order, where they get assigned a commit timestamp.

7.2 Migration Between Peer Groups

Just as a node can migrate between DCs (Section 5.9), it may migrate between peer groups. Similar consistency issues occur here. In this case, the base version of cached objects on the migrating node must be compatible with that of the new group. If the client is not missing any dependencies, or can retrieve them, then migration is seamless.

However, if the client is missing dependencies and the new peer group is offline, migration cannot succeed. If the client waits, its pending commits remain logged until the communication problem is fixed and they can be merged into the DC. In the meantime, the client might start a session with the new group, but its pending updates in the old session become invisible. Alternatively, the client might attempt to migrate again.

7.3 Collaboration groups

The mechanisms related to collaboration groups are trust management and versioning.

Messages are protected using symmetric cryptography. The authentication service provides a client with a session key per shared object, which she uses to decrypt data and sign her updates. This ensures that only legitimate clients can read an object. The key remains valid through disconnection and reconnection.

To keep out untrusted or unwanted updates, we leverage the separation between state and visibility, previously discussed in Section 5. Recall that an update is visible only if it satisfies the TCC+ invariants. In addition, it is visible only if it satisfies collaboration constraints.

To manage trust, the security administrator sets ACL. Furthermore, a collaboration group can, for instance, restrict visibility to include only versions produced within the group. An update that does not satisfy the corresponding ACL or group constraints remains invisible, and transitively the updates that depend upon it. Thus, security policies and groups can evolve dynamically. Technically, this violates the monotonicity invariant, but in a very restricted manner. The store remains TCC+, but security and group constraints expose only a variable-size window thereof.

The data model of COLONY together with TCC+is able to express (and maintain) several classes of applicative invariants. We illustrate this below using two examples, a control access system and a collaborative social application.

We consider a fail-stop distributed model of computation. Clients execute wait-free operations in COLONY whatever failures occur in the system.

8 System API and Implementation

The COLONY middleware is designed to provide a simple API for developing and deploying collaborative applications. This section presents its implementation and programming interface. The code is open-source and available on GitHub.

8.1 Modular design

COLONY enjoys a modular design, where each component (DC, PoP, Client, API) can be deployed and run independently.

Server components are implemented in Erlang/OTP, a functional language built for distribution and concurrency. We provide Docker containers for each server module.

The client component has two versions: a lightweight Typescript library for web applications, and an Erlang module that can run on the edge device, serving multiple applications through a language-agnostic Protocol Buffer interface.

8.2 API and programming model

An application node connects to a session manager (currently implemented in the core cloud), which authenticates the node. With the session opened, the node may join a collaboration or peer group, and run transactions accessing database objects. The node is notified of group change events (e.g., a new peer joins).

The database stores CRDT objects, such as counters, registers, sets, maps, or sequence datatypes.

COLONY can also support custom data types, but details are omitted.

An object is stored in a namespace called a bucket. Opening a bucket caches it in the node; optional parameters can specify cache policies (e.g., LRU, writeback, etc.). The application can subscribe to an object's update events, in order to implement reactive programming patterns. A transaction is atomic (all-or-nothing) against multiple updates, and reads a TCC+-consistent snapshot of its opened buckets. COLONY supports both interactive and batch transactions.

```
1  let dc_connection = colony_dc.connect(CONFIG.dbURI, CONFIG.credentials);
2  let cnt = dc_connection.counter("myCounter");
3  dc_connection.update(
4    cnt.increment(3)
5  )
6
7  let peer_connection = colony_pop.connect(CONFIG.signalingServers, CONFIG.
       credentials)
8  let tx = await peer_connection.startTransaction()
9  let map = tx.gmap("myMap");
10 tx.update([
11   map.register("a").assign(42),
12   map.set("e").addAll([1, 2, 3, 4])
13 ])
14 tx.commit().then(
15   console.log(
16     await peer_connection.gmap("myMap").set("e").read()
17   )
18 )
```

Fig. 8.1.: An example of COLONY program.

The TypeScript example in Figure 8.1 illustrates the API. This application opens a session (Line 1). Then, it creates and increments a CRDT counter object (Lines 2–5). Then it connects to a peer group (line 7), and updates the grow-only map (gmap) `"myMap"` in a transaction (lines 8–12). This map contains references to a register object (key `"a"`) and a set object (key `"e"`). The counter update and the commit are both asynchronous (Lines 9 and 13), returning a promise. At line 13, the client waits for the promise, and displays the content of the set.

8.3 Communication protocol

Edge nodes communicate over WebRTC. Opening a client session occurs in the signaling phase of WebRTC and currently relies on a server in the core cloud, to simplify authentication and trust management. The session provides the networking information required to communicate with the system, i.e., the IP addresses and ports of nearby peers, and the keys required to establish secure point-to-point

connections with them. [1] To migrate to a different peer group, the node relies again on the session server.

8.4 Storage

Cloud nodes (DCs and PoPs) have secondary storage and persist their data to it. They also cache data in memory for performance. Data in a DC is sharded by consistent hashing across multiple server machines, leveraging `riak_core` [Klo10].

We do not assume that a far-edge node has disk, and store their data in browser memory. When a disconnected client reconnects again, it repopulates its cache, either from its peer group's content-sharing network, or from its connected DC.

8.5 Security

The authentication keys received from the session server serve to encrypt communication between nodes, using symmetric encryption. This ensures that only authenticated clients are able to observe and update objects. Decentralised authentification [KKB19] is left for future work.

A system administrator can set a security policy with the help of access-control lists (ACL*s*). An ACL is a tuple from the set objects $\times$ users $\times$ permissions. It defines that a given user is granted access to some object and the operation she is allowed to execute on that object. Right inheritance (RI) is modelled using two forests, atop objects and users. If user u inherits from user v, then u holds the same ACL as v. Similarly, if an object x inherits from some object y, then any ACL granted on y also holds for x. Checking an ACL evaluates a first-order logic predicate over the RI and ACL relations following the above logic. For instance, (C1) $(\mathsf{book}, \mathsf{Alice}, \mathsf{own}) \in ACL$, or the more complex
(C2) $(\mathsf{book}, \mathsf{shelf}) \in RI \wedge (\mathsf{shelf}, \mathsf{Bob}, \mathsf{read}) \in ACL$ specify respectively that Alice owns a book and that this book is on a shelf readable by Bob.

An ACL check must respect the order in which clients modify both data and the security policy, to avoid unexpected behavior. More precisely, the system must ensure [Pan+19] that: (i) ACLs are applied in the order they were issued, and (ii) ACL

[1] The first connection is established via STUN. If this fails (due to a firewall or NAT), COLONY falls back to using TURN [Win+08].

checks are evaluated on a fresh copy of data and metadata. If data and security metadata are mutually consistent according to TCC+, the first constraint is trivially satisfied.

Formalizing the second one, if x and y are related by some ACL check, and if ux occurs in real time before vy, then every transaction that reads vy must access at least version ux. This is a form of external consistency, and is therefore not available under partition [GL02].

Let us use an example to illustrate the problem with the second constraint. Consider predicate C2, and assume that Alice, Bob and Carl share the bookshelf. Suppose Alice removes a book from the shelf on her node, while Bob makes the shelf readable by everyone. The two are concurrent from the causality perspective, and thus Carl may observe them in any order. However, by the second constraint, if Bob's update occurs later in real time, then Carl must never see Alice's book on the shelf. If Bob's node is disconnected or slow to transmit, this requirement is violated.

COLONY alleviates this problem as follows. First, object versions are visible according to the local copy of the ACL and RI relations. Second, it defers ACL checks to after commit. A committed transaction that fails an ACL check is not visible. In the above example, Alice's book may appear briefly on Carl's node; but as soon as Bob's update is delivered, it will disappear.

Part III

Experimental Evaluation

This last part of the thesis presents an empirical evaluation of COLONY. We first demonstrate the implementation of a realistic collaborative application atop the middleware. With this as our main benchmark, we then evaluate the platform experimentally, comparing it to a classical client-server approach in the cloud, and to a simple caching approach. We consider both the online and the offline case. In the former, we evaluate transaction throughput vs. response time, and behavior under load. In the offline case, we measure reconnection time, i.e., the time it takes for disconnected clients to be synchronized again. We also evaluate the performance benefit of peer groups. Finally, we study migration in mobile setups, measuring the time to return to normal performance.

COLONY, the examples used in the evaluation, the scripts to launch the evaluation and the scripts to parse the results are all available at the following url: `https://gitlab.inria.fr/itoumlilt/colony`

9 COLONYChat benchmark application

Overview COLONYChat emulates a team collaboration application modeled after the Slack and Mattermost communication platforms [Sla13; Mat15]. It consists of approximately 1500 lines of Typescript. COLONYChat represents its three main entities, users, workspaces and bots with the help of CRDT objects. In detail, a *user* has a profile, a list of events, a set of friends, and a set of workspaces she is a member of. A *workspace* contains the users that collaborate through the application and a set of channels. It also maintains the status of the users within the workspace (e.g., owner, ordinary, invited, or deleted). A *channel* holds a brief description, and the list of messages posted to it. A *bot* is a special kind of user. It automatically triggers an action when it observes some event, or a specific message on a channel. For instance, a bot might monitor activities within a file-system tree, or display weather information. Bots play an important role in the benchmark, as they generate a large number of update transactions.

The TCC+ guarantees of COLONY ensure that there are no ordering anomalies in the application. For instance, an answer is guaranteed to be visible in a chat after the corresponding question. Moreover, transactions are atomic, allowing maintaining invariants such as "a user is in a workspace if and only if the workspace is in the user's profile." Furthermore, within an SI zone such as a peer group, users observe updates in the same order, greatly simplifying collaboration.

Workload The workload consists of a modified trace from a popular Mattermost server. The trace contains the actions of around 2,000 users spreads over 3 workspaces; each workspace holds 20 channels on average. A user can be in more than one workspace, and one of the workspaces contains 1,000 users. Around 10% of the users are bots that act randomly upon receiving a message on the channel, they have subscribed to. An action of a user follows a 90/10 read/write ratio. A user refreshes its local copy of a channel every 5 transactions. The trace follows a Pareto distribution for the action, where 20% of the users execute 80% of operations. It contains 40 days of activity in total on the Mattermost server and exhibits a diurnal cycle. In the experiments, the trace is accelerated to execute in a few minutes only.

For each experiment, we indicate when users are scattered in peer groups, or directly connected to a remote DC. The experiments use the second variant of the peer group commit protocol, i.e., EPaxos is off the critical path of commitment (Section 7.1.4). The current version of our benchmark does not exercise transaction migration (Section 5.10); this will be added in future work. Each experiment is executed 10 times, and we report the average.

10 Performance Evaluation

The performance evaluation will focus on situations where all clients are continuously online, as well as on situations where the network is interrupted. For online situations, we are especially interested in the time it takes to distribute and apply an update to all other clients that are editing the same data. For the offline situation, we are especially interested in the time it takes for all clients to get back in sync with each other after the network disruption, and in the time it takes to restore normal interactive performance.

10.1 Setup

We deploy each COLONY component (edge client, cloud server, peer group, etc.) as a Docker container, on a set of dedicated servers in a cluster. Each server has two Intel Xeon Gold CPUs, each with 16 cores per CPU, 128 GB of memory, and 2 TB of (spinning) hard disk. Nodes are all connected through 10 Gb/s network switches. A monitoring server, deployed on a separate container, captures the performance metrics.

We measure an average 0.15 ms network latency within the cluster.

We use the Linux traffic shapping tool (`tc`) to simulate larger network latency, with a mean of 50 ms for mobile cellular data and 10 ms for carrier Ethernet. DCs are connected in a mesh using RabbitMQ sockets above TCP; peer groups are connected using WebRTC.

The performance evaluation in this chapter is performed using the COLONYChat Benchmark application presented in Chapter 9, as this scenario has the largest set of collaborative transactions on shared data and objects between users.

To compare performance, we first evaluated the performance of COLONY compared to classical cloud and single client caching (without peer-to-peer groups), to demonstrate the benefits of our approach; we then ported the COLONYChat application on top of representative platforms for web-based data synchronization: Yjs [yjs19] (a widely used open source technology, with 5.5k Github stars) and Automerge [KB18],

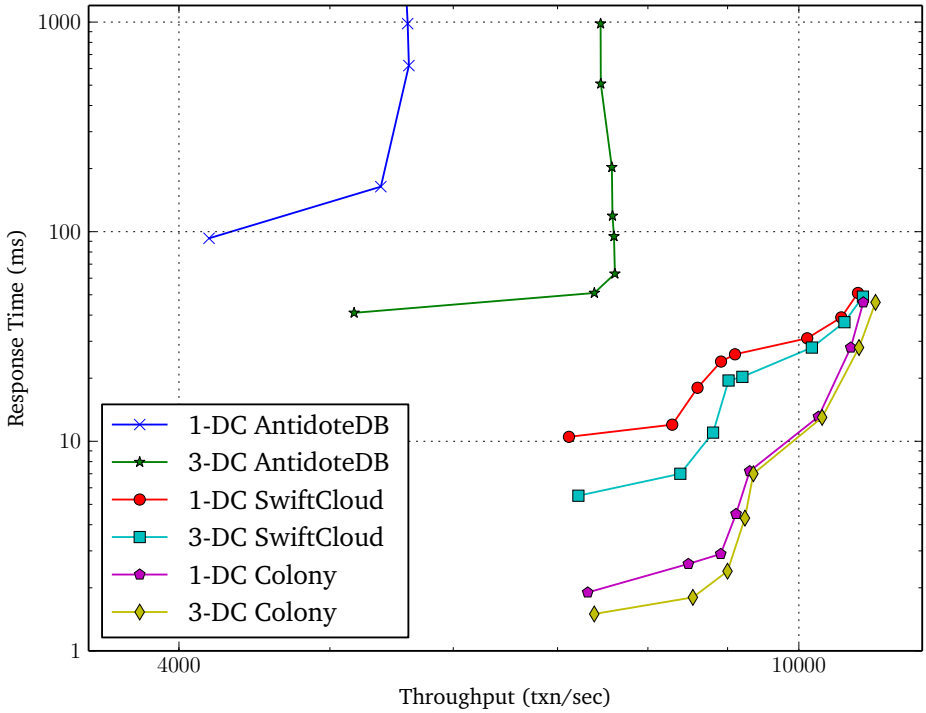

Fig. 10.1.: Performance of COLONY.

both are using state-based CRDTs, Automerge being the JSON based implementation.

10.2 Response time and throughput

In this first experiment, we evaluate system performance when scaling up, increasing the number of clients until performance saturates. We compare three approaches. One emulates AntidoteDB [The21], a classical geo-replicated approach, where a client does not have a local cache, and must contact the DC for each operation. Another emulates SwiftCloud [Zaw+15a], where clients have a local cache but do not form peer groups. Finally, the COLONY label indicates a system with peer groups enabled. In each case, we evaluate a deployment with a single DC and one with three DCs.

Figure 10.1 reports throughput vs. response time. It uses a log-log scale; down and to the right is better. Load doubles from one point to the next, from 4 to 1024 clients.

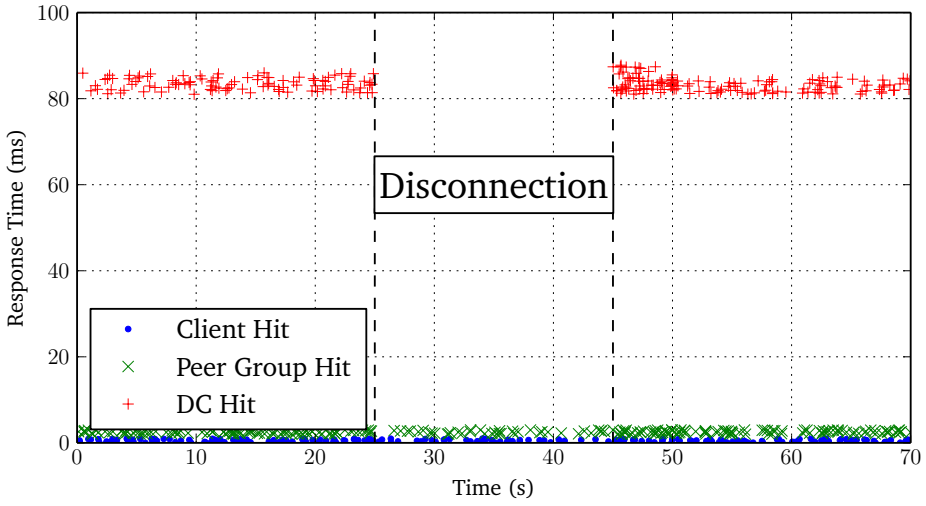

Fig. 10.2.: Impact of a DC disconnection – *Interval between the dashed lines indicates the disconnection period.*

As expected, at the beginning of the curve, throughput improves and response time remains stable. At some point, throughput levels out and response time degrades, indicating saturation.

Observe that the COLONY's response time is approximately 5 times better than Swiftcloud's, which itself performs one order of magnitude better than AntidoteDB (both for throughput and response time). This difference is explained by the caching policy. AntidoteDB does not have a client-side cache. In the SwiftCloud configuration, 90% of transactions hit the local cache. The hit rate reaches 95% in the shared cache of the COLONY peer group configuration.

Adding more DCs spreads the load in the AntidoteDB configuration, improving the maximum throughput of the system, by 40% from a single to three DCs. However, adding more data centers does not improve response time, since clients need still to contact them for each operation. This is 8x slower than the SwiftCloud configuration. In contrast, the number of DCs has a minor impact in the SwiftCloud and COLONY configurations.

10.3 Response time of offline collaboration

single group of users in a channel (has already fulfilled its cache) there are disconnected after some time workspace w. 36 users/ 36 clients the peer group contains

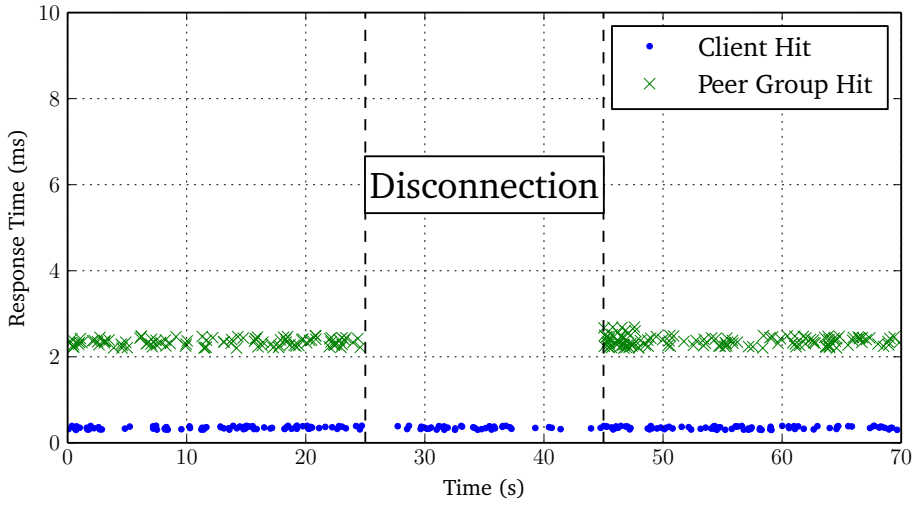

Fig. 10.3.: Impact of a peer group disconnection – *Interval between the dashed lines indicates the disconnection period.*

12 users; the rest is directly connected to the DC. in red, we measure the response time for these guys in blue, hit locally within the peer group in green, hit within the group but not local.

We now evaluate how the response time varies under offline collaboration, or when the sync point of a group fails to connect to a DC. To this end, we use a single COLONYChat workspace that contains 36 users. We pack 12 of these users in a peer group, whereas the others remain independent. All users start with a warmed-up cache. Figure 10.2 shows the response time perceived at each user during the experiment. Each dot in this figure corresponds to a transaction.

In Figure 10.2, we observe that the response time for local cache hits is near zero (in blue). Users that belong to the same peer group benefit from an average 2.3 ms response time when data is fetched from the collaborative cache (in green). This raises to around 82 ms when the user needs to perform a remote read from the DC (in red).

Approximately 25 s after the start of the experiment, the peer group goes offline, and only collaborates on its shared interest set of objects. After this event, we observe that both the local and peer response time is unchanged: users in the group will not observe remote transactions due to the disconnection, but they continue their collaboration seamlessly. Around 45 s after the beginning of the experiment, the group is then reconnected to the DC. In Figure 10.2, we can observe a slight

increase of the response time at the reconnection, yet it has minimal impact on performance.

In Figure 10.3, we consider the same workload but this time disconnect a user from its peer group. The disconnection occurs after 25 s and the user reconnects 20 s later. In this figure, we may observe that the response time of the ColonyChat application is slightly impacted by the reconnection to the peer group. Upon reconnecting to the peer group, the user notices a slight increase (below the millisecond) in its transactions. This variation comes from the fact that the channels were updated with the new content published by the users in the peer group.

10.4 Migration effect on response time

Mobile clients, especially in location-based collaborative applications, like games, frequently switch from one peer group to another. Our last experiment studies the synchronization time for a client to connect to a group when its cache is invalid. This experiment exercises both the cache refreshing mechanism of Colony and the collaborative cache in a peer group. The results are presented in Figure 10.4.

In this figure, 45s after the start of the experiment, a mobile client migrates and joins the peer group. The client has a completely invalid chat history. She thus needs to synchronize her cache before interacting with the peer group. Figure 10.4 plots the (average) response time observed by the connecting user (in blue), and the rest of the group (in green). As previously, each dot in the plot represents the response time of a transaction. In this figure, we can observe that the first transactions of the connecting user have a higher response time (below 12 ms). This performance degradation is way lower than the cost of reconnecting to a DC, and fetching data from it (as in Figure 10.2). Moreover, after only a few seconds, it returns to the normal and matches the perceived response time of the group users (in green).

10.5 Performance compared to related work

In this experiment, we compare system performance of three approaches Yjs [yjs19], Automerge [KB18] and our system Colony when scaling up, the setup for this experiment is exactly the same as in Section 10.2.

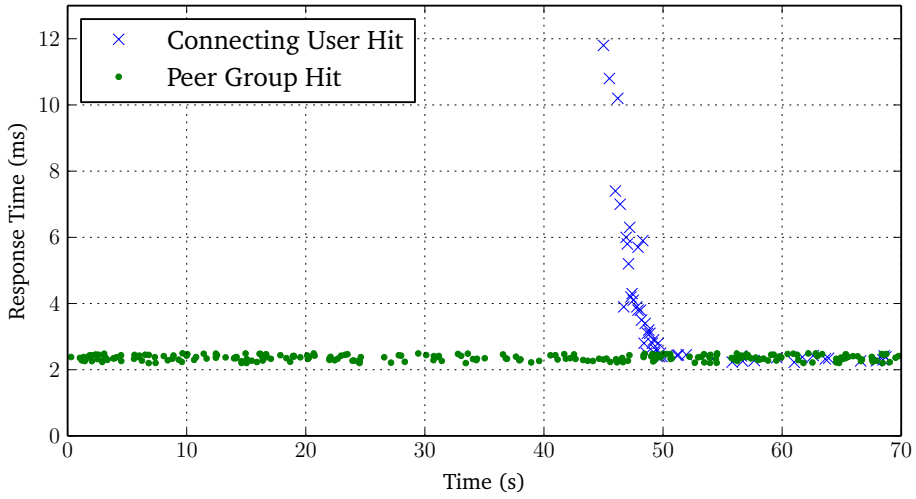

Fig. 10.4.: Synchronizing with a peer group.

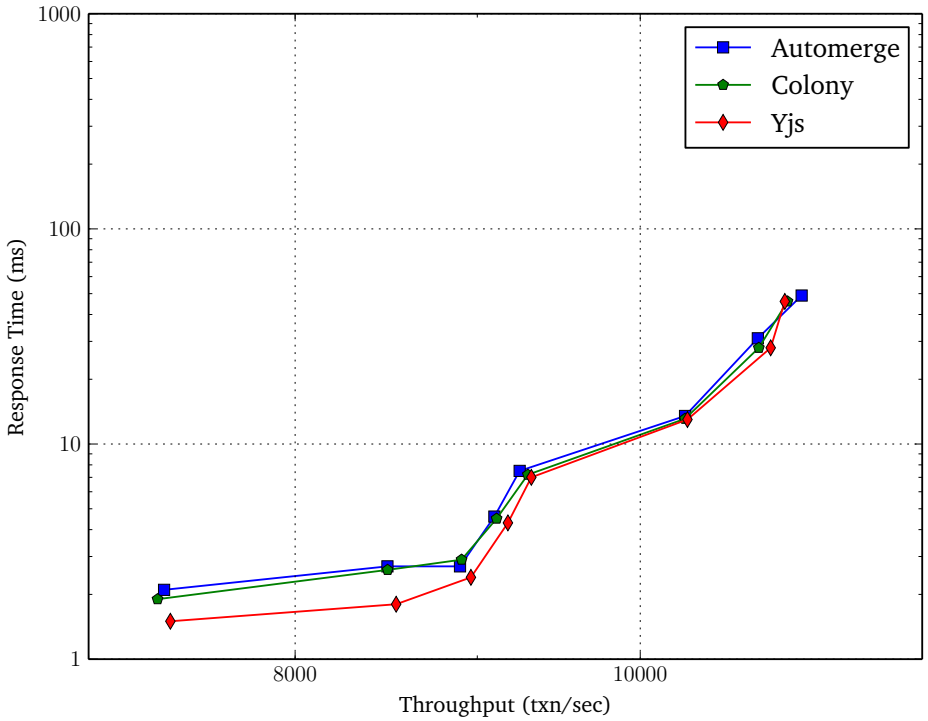

Fig. 10.5.: Performance evaluation comparing throughput and latency of SwiftCloud, Automerge, Colony and Yjs. Using ColonyChat benchmark applications

Figure 10.5 reports throughput vs. response time. It uses a log-log scale; down and to the right is better. Load doubles from one point to the next, from 4 to 2024 clients.

Automerge requires more bandwidth because it stores the whole history and uses fat metadata to identify transactions and clients. All three approaches uses state-based CRDTs, COLONY performance is lower but remains very close to Yjs, this is expected from the consensus at SI Zones, which we see as a trade-off to improve the average reconnection time and metadata size as we will see in the next section.

10.5.1 Reconnection time

Tab. 10.1.: Average resynchronization time (in seconds).

System	4 clients	8 clients	16 clients	32 clients
Automerge	1.1s	1.3s	9.2s	102s
Yjs	4s	8.3s	14s	73s
Colony	0.9s	1.1s	4.2s	8.3s

We now present the performance analysis when the network between the clients and the server is disrupted, running the same workload presented in Section 10.2, with an increased disconnection rate (10%). And as in Section 10.4 we measured the average synchronization time when a client is connected back to its failure point, or (in the Colony case) when it switches to another sync point.

Table 10.1 shows the results of this experiment, thanks to its ability to synchronize from multiple points of failure, the forest topology and its migration feature, COLONY achieves an under 10s synchronization time while increasing the number of group clients. While in Automerge and Yjs, that needs a single failure point to synchronize a peer-to-peer group after failure, the synchronization time increases exponentially.

10.5.2 Metadata and implementation features comparison

Tab. 10.2.: Features and metadata comparison of similar systems.

System	Edge CRDTs + Offline	Edge Metadata	P2P Groups	Client Migration	Txn Movement	Brower Support	IoT Support	Web Server
WebCure	Yes	O(N)	No	No	No	No	No	Yes
SwiftCloud	Yes	O(1)	No	Yes	No	No	No	Yes
Legion	Yes	O(N)	Yes	No	No	Yes	No	Yes
Automerge	Yes	O(N)	Yes	No	No	Yes	Yes	Yes
Yjs	Yes	O(N)	Yes	No	No	Yes	Yes	Yes
Colony	Yes	O(1)	Yes	Yes	Yes	No	No	Yes

This last section illustrates in Table 10.2 the set of features our approach offers while keeping low metadata overhead.

As presented in Section 7.1.4, our approach relies on EPaxos [MAK13] within the peer group. Committed transactions become visible in the order assigned by EPaxos. In addition to improving the user experience, consensus is essential to correct metadata management. Thanks to this choice, COLONY has a low metadata size compared to other approaches, while maintaining a similar performance 10.5, and many edge features.

However, the current implementation of COLONY can only be used as a local web server, which restricts the type of devices it can be executed on (computers and laptops mostly), In-Browser (Service Workers) or IoT devices are future work development.

11 Summary

This evaluation part of the thesis presented an experimental evaluation demonstrating the benefits of our approach.

Our experimental evaluation shows that: local and group caching improve throughput by 1.4× and 1.6× respectively, and response time by 8× and 20×, compared to a classical cloud configuration; performance in offline mode remains the same as online; both the offline/online transition and migration are seamless.

COLONY achieves a similar performance to other collaborative edge systems, even with consensus at the peer groups, which helps reducing the metadata cost. The forest topology and the ability to synchronise from multiple point of failure makes the resynchronization after failure in less than 2secs even with large group of clients.

All the approaches used in our evaluation uses state-based CRDTs, an improvement idea can be to make use of Delta-based CRDTs [ASB15] to improve the resynchronisation time.

Part IV

Conclusion

Many web applications are built around direct interactions among users, from collaborative applications and social networks to multiuser games. Despite being user-centric, these applications are usually supported by services running on servers that mediate all interactions among clients. When users are in close vicinity of each other, relying on a centralized infrastructure for mediating user interactions leads to unnecessarily high latency while hampering fault-tolerance and scalability.

This thesis first explored the edge computing paradigms and the challenges that are inherent to its network and architecture. Then, we presented the safety, scalability, security, hierarchy and programmability requirements for todays collaborative edge applications. We also explored the solutions from the state of the art.

In the second part of the thesis, we presented the design of COLONY, a system that brings geo-replication guarantees to the Edge. COLONY allows applications to run transactions in the client machine, for common operations that access a limited set of objects, with immediate, consistent and offline response, or in the DC, for transactions that require accessing a large number of objects. COLONY also proposes a client-assisted failover mechanism that trades response time by a small increase in staleness.

11.1 Limitations and perspectives

Several aspects remain open for improvements and investigation.

Better caching heuristics the current design caches at the client side the least recently used versions of clients objects. We would like to take into account the interest set of the application, and have more level of LRU caches, to avoid evicting objects that are actively used by the application while loading temporary objects.

Support for transaction migration Client computation resources can be poor and limited, although being resource-friendly and metadata lightweight, in COLONY, some heavy operation can be done faster using the Datacenter ressources. We have explored a hybrid model where we can move computation from the client to the server in the heavy jobs case. This raises some interesting challenges like preserving the causal state of the client, handling updates and scheduling operations.

Data and Point-of-Presence placement Placing clients at different levels of the hierarchy, in particular in Content Delivery Network points of presence, might improve perceived response time even more. In addition to prefetching heuristics and better caching at the Point-of-Presence level.

Extending the peer-to-peer communications across edge services would also make the system less dependent on the cloud for updates dissemination.

Bibliography

[APZ18] Yuan Ai, Mugen Peng, and Kecheng Zhang. “Edge computing technologies for Internet of Things: a primer”. In: *Digital Communications and Networks* 4.2 (2018), pp. 77–86 (cit. on p. 26).

[Aka10] Akamai. *New Study Reveals the Impact of Travel Site Performance on Consumers*. `https://www.akamai.com/us/en/about/news/press/2010-press/new-study-reveals-the-impact-of-travel-site-performance-on-consumers.jsp`. June 2010 (cit. on p. 39).

[Akk+16a] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, et al. “Cure: Strong semantics meets high availability and low latency”. In: *icdcs*. Nara, Japan, June 2016, pp. 405–414 (cit. on pp. 33, 34, 40, 69, 76, 83).

[Akk+16b] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, et al. “Cure: Strong semantics meets high availability and low latency”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 405–414 (cit. on p. 47).

[Alm+14] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. “Scalable and Accurate Causality Tracking for Eventually Consistent Stores”. In: *dais*. Ed. by Kostas Magoutis and Peter Pietzuch. Vol. 8460. lncs. ifip. Berlin, Germany: springer, June 2014, pp. 67–81 (cit. on p. 73).

[ASB15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Efficient state-based crdts by delta-mutation”. In: *International Conference on Networked Systems*. Springer. 2015, pp. 62–76 (cit. on pp. 33, 109).

[ALR13] Sérgio Almeida, João Leitão, and Luís Rodrigues. “ChainReaction: a causal+ consistent datastore based on chain replication”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 85–98 (cit. on p. 48).

[Alv+11] Peter Alvaro, Neil Conway, Joe Hellerstein, and William Marczak. “Consistency Analysis in Bloom: a CALM and Collected Approach”. In: *cidr*. Asilomar, CA, USA, Jan. 2011 (cit. on p. 77).

[Arm+10] Michael Armbrust, Armando Fox, Rean Griffith, et al. “A view of cloud computing”. In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on p. 21).

[AEM17] Hagit Attiya, Faith Ellen, and Adam Morrison. “Limitations of Highly-Available Eventually-Consistent Data Stores”. In: *tpds* 28.1 (Jan. 2017), pp. 141–155 (cit. on p. 33).

[Bal+15] Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, et al. “Putting Consistency back into Eventual Consistency”. In: *eurosys*. Indigo. Bordeaux, France, Apr. 2015, 6:1–6:16 (cit. on pp. 36, 40).

[Bel+06a] Nalini Belaramani, Mike Dahlin, Lei Gao, et al. “PRACTI Replication”. In: *nsdi*. usenix. San Jose, CA, USA: usenix, May 2006, pp. 59–72 (cit. on p. 40).

[Bel+06b] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, et al. “PRACTI Replication.” In: *NSDI*. Vol. 6. 2006, pp. 5–5 (cit. on p. 45).

[Bol19] Prof. David Bol. *Ecological transition in ICT: A role for open hardware ?* `https://open-src-soc.org/2019-10/media/slides/2nd-RISC-V-Meeting-2019-10-01-14h30-David-Bol.pdf`. Accessed: 2020–11-06. 2019 (cit. on p. 25).

[Bon+12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh 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*. 2012, pp. 13–16 (cit. on pp. 24–26).

[Bra+21a] Manuel Bravo, Alexey Gotsman, Borja de Régil, and Hengfeng Wei. “UniStore: A fault-tolerant marriage of causal and strong consistency”. In: *usenix-atc*. usenix. Virtual event, July 2021 (cit. on p. 36).

[Bra+21b] Manuel Bravo, Alexey Gotsman, Borja de Régil, and Hengfeng Wei. “UniStore: A fault-tolerant marriage of causal and strong consistency”. In: *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. Ed. by Irina Calciu and Geoff Kuenning. Virtual: USENIX Association, 2021, pp. 923–937 (cit. on p. 40).

[BRVR17a] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. “Saturn: A distributed metadata service for causal consistency”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, pp. 111–126 (cit. on p. 53).

[BRVR17b] Manuel Bravo, Luìs Rodrigues, and Peter Van Roy. “Saturn: A Distributed Metadata Service for Causal Consistency”. In: *eurosys*. acm. Belgrade, Serbia: acm, 2017, pp. 111–126 (cit. on pp. 69, 73, 83).

[Bur+12] Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Benjamin P. Wood. “Cloud Types for Eventual Consistency”. In: *ecoop*. Beijing, China: springer, June 2012, pp. 283–307 (cit. on p. 40).

[Bur+14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. “Replicated Data Types: Specification, Verification, Optimality”. In: *popl*. San Diego, CA, USA, Jan. 2014, pp. 271–284 (cit. on p. 77).

[bus15] businesswire. *Datacenter Investments Critical to Internet of Things Expansion, IDC Says | Business Wire*. `https://www.businesswire.com/news/home/20150427005027/en/Datacenter-Investments-Critical-to-Internet-of-Things-Expansion-IDC-Says`. (Accessed on 11/07/2021). 2015 (cit. on p. 21).

[BS17] C Byers and R Swanson. “OpenFog consortium OpenFog reference architecture for fog computing”. In: *OpenFog Consortium Archit. Working Group, Fremont, CA, USA, Tech. Rep. OPFRA001* 20817 (2017) (cit. on pp. 27, 28).

[CNS19] W. Cai, A. Ng, and C. Sun. “Design and Implementation of a Concurrency Benchmark Tool for Cloud Storage Systems”. In: *cscw*. ieeecs. Porto, Portugal: ieeecs, May 2019, pp. 386–391 (cit. on p. 15).

[CGR07] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. “Paxos made live: an engineering perspective”. In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 2007, pp. 398–407 (cit. on p. 34).

[CB91] Bernadette Charron-Bost. “Concerning the size of logical clocks in distributed systems”. In: *Information Processing Letters* 39.1 (July 1991), pp. 11–16 (cit. on p. 73).

[Con+34] OpenFog Consortium et al. “IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing”. In: *IEEE Std* 2018 (1934), pp. 1–176 (cit. on p. 24).

[DSZ17] Diego Didona, Kristina Spirovska, and Willy Zwaenepoel. “Okapi: Causally consistent geo-replication made faster, cheaper and more available”. In: *arXiv preprint arXiv:1702.04263* (2017) (cit. on p. 57).

[DIG13] John David N Dionisio, William G Burns III, and Richard Gilbert. “3D virtual worlds and the metaverse: Current status and future possibilities”. In: *ACM Computing Surveys (CSUR)* 45.3 (2013), pp. 1–38 (cit. on p. 15).

[Dro+17] Utsav Drolia, Katherine Guo, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. “Cachier: Edge-caching for recognition applications”. In: *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*. IEEE. 2017, pp. 276–286 (cit. on p. 28).

[Du+13] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. “Orbe: Scalable causal consistency using dependency matrices and physical clocks”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–14 (cit. on p. 43).

[DEZ13] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. “Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks”. In: *srds*. Braga, Portugal: ieeecs, Oct. 2013, pp. 173–184 (cit. on pp. 34, 77).

[Du+14] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. “Gentlerain: Cheap and scalable causal consistency with physical clocks”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–13 (cit. on p. 50).

[Esc+14] Robert Escriva, Ayush Dubey, Bernard Wong, and Emin Gün Sirer. “Kronos: The design and implementation of an event ordering service”. In: *Proceedings of the Ninth European Conference on Computer Systems*. 2014, pp. 1–14 (cit. on p. 41).

[Fac21] Meta Facebook. *Connect 2021: Our vision for the metaverse*. `https://tech.fb.com/connect-2021-our-vision-for-the-metaverse/`. (Accessed on 11/04/2021). 2021 (cit. on p. 15).

[Fek+99] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. “Eventually-Serializable Data Services”. In: *Theoretical Computer Science* 220 (1999). Special issue on Distributed Algorithms, pp. 113–156 (cit. on p. 40).

[Fid88] C. J. Fidge. “Timestamps in message-passing systems that preserve the partial ordering”. In: *11th Australian Computer Science Conference*. University of Queensland, Australia, 1988, pp. 55–66 (cit. on p. 72).

[FRT15] R Friedman, M Raynal, and François Taïani. “Fisheye Consistency: Keeping Data in Synch in a Georeplicated World”. In: *International Conference on NETworked sYStems (NETYS'2015)*. Networked Systems : Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers 9466. Agadir, Morocco: Springer International Publishing, May 2015, pp. 246–262 (cit. on p. 40).

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. “The bitcoin backbone protocol: Analysis and applications”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 281–310 (cit. on p. 33).

[GL02] Seth Gilbert and Nancy Lynch. “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *sigact-news* 33.2 (2002), pp. 51–59 (cit. on pp. 32, 94).

[Gir+18] Alain Girault, Gregor Gössler, Rachid Guerraoui, Jad Hamza, and Dragos-Adrian Seredinschi. “Monotonic Prefix Consistency in Distributed Systems”. In: *forte*. Ed. by C. Baier and Caires L. Vol. 10854. lncs. springer, June 2018 (cit. on p. 32).

[Goo21a] Google. *Fix common issues in Google Drive*. `https://support.google.com/drive/answer/2456903`. Accessed: 2021-05-05. 2021 (cit. on p. 15).

[Goo21b] Google Drive Doc Editors Help, Community. *Lost doc going from offline to online. I looked in trash, recents, any ideas?* `https://support.google.com/docs/thread/10026411/lost-doc-going-from-offline-to-online-i-looked-in-trash-recents-any-ideas?hl=en`. Accessed: 2021-05-05. 2021 (cit. on p. 15).

[Got+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. “‘Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems”. In: *popl*. St. Petersburg, FL, USA: acm, 2016, pp. 371–384 (cit. on p. 77).

[Gre+08] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. *The cost of a cloud: research problems in data center networks*. 2008 (cit. on p. 23).

[GBR17] Chathuri Gunawardhana, Manuel Bravo, and Luis Rodrigues. “Unobtrusive deferred update stabilization for efficient geo-replication”. In: *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 2017, pp. 83–95 (cit. on p. 56).

[HL19] Farzin Houshmand and Mohsen Lesani. “Hamsaz: Replication Coordination Analysis and Synthesis”. In: *popl*. Vol. 3. procacmpl. Cascais, Portugal: acm, Jan. 2019, 74:1–74:32 (cit. on p. 77).

[Hu+15] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. "Mobile edge computing—A key technology towards 5G". In: *ETSI white paper* 11.11 (2015), pp. 1–16 (cit. on p. 26).

[IBM16] IBM. *Bringing big data to the enterprise*. `http://www-01.ibm.com/software/data/bigdata/`. (Accessed on 19/05/2018). Jan. 2016 (cit. on p. 21).

[Jos+95] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. "Rover: A Toolkit for Mobile Information Access". In: *osr* 29.5 (Dec. 1995), pp. 156–171 (cit. on p. 39).

[KS92] James J. Kistler and M. Satyanarayanan. "Disconnected operation in the Coda file system". In: *ACM Trans. on Comp. Sys. (TOCS)* 10.5 (Feb. 1992), pp. 3–25 (cit. on p. 39).

[KB18] Martin Kleppmann and Alastair R Beresford. "Automerge: Real-time data sync between edge devices". In: *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018). https://mobiuk. org/abstract/S4-P5-Kleppmann-Automerge. pdf*. 2018 (cit. on pp. 101, 105).

[Kle+20] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. *A highly-available move operation for replicated trees and distributed filesystems*. Work-in-progress draft. University of Cambridge, June 2020 (cit. on p. 33).

[Kle+19] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. "Local-First Software: You Own Your Data, in spite of the Cloud". In: *onward*. sigplan. Athens, Greece: acm, Oct. 2019, pp. 154–178 (cit. on pp. 16, 31).

[Klo10] Rusty Klophaus. "Riak Core: Building Distributed Applications without Shared State". In: *Commercial Users of Functional Programming (CUFP)*. ICFP'10. Baltimore, Maryland, USA: acm, 2010 (cit. on p. 93).

[KL07] Ron Kohavi and Roger Longbotham. "Online experiments: Lessons learned". In: *Computer* 40.9 (2007) (cit. on p. 39).

[KKB19] Stephan A. Kollmann, Martin Kleppmann, and Alastair R. Beresford. "Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing". In: *Privacy Enhancing Technologies (PoPETS)*. Vol. 2019. Stockholm, Sweden, July 2019 (cit. on pp. 37, 93).

[Lad+92a] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. "Providing High Availability Using Lazy Replication". In: *tocs* 10.4 (Nov. 1992), pp. 360–391 (cit. on p. 40).

[Lad+92b] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. "Providing high availability using lazy replication". In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 360–391 (cit. on p. 46).

[Lei09] Tom Leighton. "Improving performance on the internet". In: *Communications of the ACM* 52.2 (2009), pp. 44–51 (cit. on p. 39).

[Lei19] João Leitão. “Pokémon Go in-the-field anomaly”. Private communication. Sept. 2019 (cit. on p. 36).

[Li+12] Cheng Li, Daniel Porto, Allen Clement, et al. “Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary”. In: *osdi*. Hollywood, CA, USA, Oct. 2012, pp. 265–278 (cit. on pp. 36, 40).

[Lin+17a] Albert van der Linde, Pedro Fouto, João Leitão, et al. “Legion: Enriching Internet Services with Peer-to-Peer Interactions”. In: *www*. WWW ’17. Perth, Australia: International World Wide Web Conferences Steering Committee, 2017, pp. 283–292 (cit. on p. 40).

[Lin+17b] Albert van der Linde, Pedro Fouto, João Leitão, et al. “Legion: Enriching internet services with peer-to-peer interactions”. In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 283–292 (cit. on p. 59).

[LLaP20] Albert van der Linde, João Leitão, and Nuno Preguiça. “Practical Client-side Replication: Weak Consistency Semantics for Insecure Settings”. In: *pvldb* 13.11 (July 2020), pp. 2590–2605 (cit. on p. 37).

[Llo+11a] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. “Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS”. In: *sosp*. Cascais, Portugal: acm, Oct. 2011, pp. 401–416 (cit. on pp. 40, 73).

[Llo+11b] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. “Don’t settle for eventual: Scalable causal consistency for wide-area storage with COPS”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 401–416 (cit. on p. 42).

[Mah+11] Prince Mahajan, Srinath Setty, Sangmin Lee, et al. “Depot: Cloud Storage with Minimal Trust”. In: *tocs* 29.4 (Dec. 2011), 12:1–12:38 (cit. on p. 40).

[MS15] Chris Marra and Alex Sourov. *Continuing to build News Feed for all types of connections*. `https://engineering.fb.com/2015/12/09/android/continuing-to-build-news-feed-for-all-types-of-connections/`. Dec. 2015 (cit. on p. 39).

[Mat15] Mattermost Inc. *Mattermost*. `https://mattermost.com`. 2015 (cit. on p. 99).

[Mat89] Friedmann Mattern. “Virtual Time and Global States of Distributed Systems”. In: *Int. W. on Parallel and Distributed Algorithms*. Elsevier Science Publishers B.V. (North-Holland), 1989, pp. 215–226 (cit. on p. 72).

[Meh+17] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, et al. “I can’t believe it’s not causal! scalable causal consistency with no slowdown cascades”. In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 453–468 (cit. on p. 51).

[Mic21] Microsoft. *Microsoft Cloud at Ignite 2021: Metaverse, AI and hyperconnectivity in a hybrid world - The Official Microsoft Blog*. https://blogs.microsoft.com/blog/2021/11/02/microsoft-cloud-at-ignite-2021-metaverse-ai-and-hyperconnectivity-in-a-hybrid-world/. (Accessed on 11/04/2021). 2021 (cit. on p. 15).

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. "There is More Consensus in Egalitarian Parliaments". In: *sosp*. SOSP '13. Farminton, PA, USA: acm, Nov. 2013, pp. 358–372 (cit. on pp. 87, 108).

[Mor+17] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal De Lara. "Cloudpath: A multi-tier cloud computing framework". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 2017, pp. 1–13 (cit. on p. 42).

[NVI21] NVIDIA. *What Is the Metaverse? | NVIDIA Blog*. https://blogs.nvidia.com/blog/2021/08/10/what-is-the-metaverse/. (Accessed on 11/04/2021). 2021 (cit. on p. 15).

[OO14] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319 (cit. on p. 34).

[Pan+19] Ruoming Pang, Ramon Caceres, Mike Burrows, et al. "Zanzibar: Google's Consistent, Global Authorization System". In: *usenix-atc*. usenix. Renton, WA, USA, July 2019 (cit. on p. 93).

[PSS17] Rafael Pass, Lior Seeman, and Abhi Shelat. "Analysis of the blockchain protocol in asynchronous networks". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017, pp. 643–673 (cit. on p. 33).

[Per+15] Dorian Perkins, Nitin Agrawal, Akshat Aranya, et al. "Simba: Tunable End-to-End Data Consistency for Mobile Apps". In: *eurosys*. Bordeaux, France: acm, Apr. 2015, 7:1–7:16 (cit. on p. 40).

[Pet+97] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. "Flexible Update Propagation for Weakly Consistent Replication". In: *sosp*. ACM SIGOPS. Saint Malo, Oct. 1997, pp. 288–301 (cit. on p. 73).

[Pou] PouchDB. *PouchDB website*. https://pouchdb.com/ (cit. on p. 40).

[Pro18] LightKone H2020 Project. *D6.1: New concepts for heavy edge computing*. https://wordix.inesctec.pt/wp-lightkone/wp-content/uploads/2019/04/D6.1-New-concepts-for-heavy-edge-computing.pdf. (Accessed on 11/07/2021). Jan. 2018 (cit. on p. 17).

[Pro19] LightKone H2020 Project. *D6.2: New concepts for heavy edge computing*. https://wordix.inesctec.pt/wp-lightkone/wp-content/uploads/2019/04/D6.2_Newconceptsforheavyedgecomputing.pdf. (Accessed on 11/07/2021). Jan. 2019 (cit. on p. 17).

[Pus19] Ricky Pusch. “Explaining how fighting games use delay-based and rollback netcode”. In: *Ars Technica* (Oct. 2019) (cit. on p. 32).

[Ram+09] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, et al. “Cimbiosys: A platform for content-based partial replication”. In: *nsdi*. usenix. Boston, MA, USA: usenix, Apr. 2009, pp. 261–276 (cit. on p. 39).

[RG18] Arun Ravindran and Anjus George. “An edge datastore architecture for latency-critical distributed machine vision applications”. In: {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*. 2018 (cit. on p. 29).

[SASS13] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. “Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems”. In: *srds*. ieeecs. Braga, Portugal, Oct. 2013, pp. 163–172 (cit. on p. 76).

[SS05] Yasushi Saito and Marc Shapiro. “Optimistic Replication”. In: *acmcs* 37.1 (Mar. 2005), pp. 42–81 (cit. on p. 39).

[Sat+09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. “The case for vm-based cloudlets in mobile computing”. In: *IEEE pervasive Computing* 8.4 (2009), pp. 14–23 (cit. on p. 26).

[SRS21] Alexander Schäfer, Gerd Reis, and Didier Stricker. “A Survey on Synchronous Augmented, Virtual and Mixed Reality Remote Collaboration Systems”. In: *arXiv preprint arXiv:2102.05998* (2021) (cit. on p. 15).

[Sha+11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-free replicated data types”. In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400 (cit. on pp. 33, 34).

[Sha+11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-free Replicated Data Types”. In: *sss*. Ed. by Xavier Défago, Franck Petit, and V. Villain. Vol. 6976. lncs. Grenoble, France: springer, Oct. 2011, pp. 386–400 (cit. on pp. 69, 77, 83).

[SSAP16] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. “Consistency in 3D”. In: *concur*. Ed. by Josée Desharnais and Radha Jagadeesan. Vol. 59. lipics. Québec, Québec, Canada: dagstuhl-pub, Aug. 2016, 3:1–3:14 (cit. on p. 72).

[Shi+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. “Edge computing: Vision and challenges”. In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on pp. 21, 24, 25).

[Shi+20] Xiao Shi, Scott Pruett, Kevin Doherty, et al. “FlightTracker: Consistency across Read-Optimized Online Stores at Facebook”. In: *osdi*. Banff, Alberta, Canada: usenix, Nov. 2020, pp. 407–423 (cit. on p. 32).

[Sho+20] Ali Shoker, Paulo Sérgio Almeida, Carlos Baquero, et al. *LightKone Reference Architecture (LiRA)*. `https://www.info.ucl.ac.be/~pvr/LiRAv0.9.pdf`. (Accessed on 11/07/2021). Jan. 2020 (cit. on p. 17).

[SC+19] Inés Sittón-Candanedo, Ricardo S Alonso, Juan M Corchado, Sara Rodríguez-González, and Roberto Casado-Vara. "A review of edge computing reference architectures and a new global edge proposal". In: *Future Generation Computer Systems* 99 (2019), pp. 278–294 (cit. on p. 24).

[Sla13] Slack Technologies. *Slack*. `https://slack.com`. 2013 (cit. on p. 99).

[Sov+11] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. "Transactional storage for geo-replicated systems". In: *sosp*. Cascais, Portugal: acm, Oct. 2011, pp. 385–400 (cit. on pp. 40, 88).

[SDZ17] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. "Optimistic causal consistency for geo-replicated key-value stores". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 2626–2629 (cit. on p. 54).

[SR06] Daniel Stutzbach and Reza Rejaie. "Understanding churn in peer-to-peer networks". In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. 2006, pp. 189–202 (cit. on p. 26).

[sup] support.google.com. *Work on Google Docs, Sheets, & Slides offline*. `https://support.google.com/docs/answer/6388102` (cit. on p. 39).

[Tak18] Dean Takahashi. *Onshape lets engineers collaborate on 3D designs with Magic Leap's AR glasses*. `https://venturebeat.com/2018/10/13/onshape-lets-engineers-collaborate-on-3d-designs-with-magic-leaps-ar-glasses/`. Oct. 2018 (cit. on p. 15).

[TSR15] Vinh Tao, Marc Shapiro, and Vianney Rancurel. "Merging Semantics for Conflict Updates in Geo-Distributed File Systems". In: *systor*. Haifa, Israel: acm, May 2015, pp. 10.1–10.12 (cit. on p. 15).

[TCH16] William Tärneberg, Vishal Chandrasekaran, and Marty Humphrey. "Experiences creating a framework for smart traffic control using aws iot". In: *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2016, pp. 63–69 (cit. on p. 25).

[Ter13] Doug Terry. "Replicated Data Consistency Explained Through Baseball". In: *cacm* 56.12 (Dec. 2013), pp. 82–89 (cit. on p. 39).

[Ter+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, et al. "Session Guarantees for Weakly Consistent Replicated Data". In: *pdis*. Austin, Texas, USA, Sept. 1994, pp. 140–149 (cit. on p. 79).

[Ter+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, et al. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *sosp*. ACM SIGOPS. Copper Mountain, CO, USA: ACM Press, Dec. 1995, pp. 172–182 (cit. on pp. 33, 39).

[The21] The SyncFree Consortium. *AntidoteDB: A planet scale, highly available, transactional database*. Website `http://antidoteDB.eu/`. 2021 (cit. on p. 102).

[TRA99] Francisco J. Torres-Rojas and Mustaque Ahamad. “Plausible clocks: constant size logical clocks for distributed systems”. In: *Distrib. Comput.* 12.4 (Sept. 1999), pp. 179–195 (cit. on p. 73).

[Tou21] Ilyas Toumlilt. *Ilyas Toumlilt / antidote-edgeant · GitLab*. `https://gitlab.lip6.fr/itoumlilt/antidote-edgeant`. (Accessed on 11/07/2021). 2021 (cit. on p. 67).

[TSS21] Ilyas Toumlilt, Pierre Sutra, and Marc Shapiro. “Highly-Available and Consistent Group Collaboration at the Edge with Colony”. In: *Middleware 2021 - 22nd International Middleware Conference*. Québec / Virtual, Canada: ACM, Dec. 2021 (cit. on pp. 17, 67).

[TTS17] Ilyas Toumlilt, Alejandro Tomsic, and Marc Shapiro. “Vers une cohérence causale évolutive sans chaînes de ralentissements”. In: *Compas 2017: Conférence d’informatique en Parallélisme, Architecture et Système*. Nice Sophia-Antipolis, France, June 2017 (cit. on p. 17).

[VV16] Paolo Viotti and Marko Vukolić. “Consistency in Non-Transactional Distributed Storage Systems”. In: *acmcs* 49.1 (July 2016), 19:1–19:34 (cit. on p. 70).

[Vog08] Werner Vogels. “Eventually Consistent”. In: *acmqueue* 6.6 (Oct. 2008), pp. 14–19 (cit. on p. 32).

[Wan+19] Peng Wang, Shusheng Zhang, Mark Billinghurst, et al. “A comprehensive survey of AR/MR-based co-design in manufacturing”. In: *Engineering with Computers* (2019), pp. 1–24 (cit. on p. 15).

[WBP16] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. “Access Control for Weakly Consistent Replicated Information Systems”. In: *intwkonSecurity and Trust Management*. Ed. by Gilles Barthe, Evangelos P. Markatos, and Pierangela Samarati. Vol. 9871. lncs. ACGreGate. Heraklion, Crete, Greece: springer, Sept. 2016, pp. 82–97 (cit. on p. 37).

[Win+08] Dan Wing, Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. “Session traversal utilities for NAT (STUN)”. In: *RFC5389, October* (2008) (cit. on p. 93).

[WRT10] Ted Wobber, Thomas L. Rodeheffer, and Douglas B. Terry. “Policy-based access control for weakly consistent replication”. In: *eurosys*. Ed. by Christine Morin and Gilles Muller. Paris, France: acm, Apr. 2010, pp. 293–306 (cit. on p. 37).

[www21] www.reddit.com/r/pokemongo. *Incubator duplication glitch??* `https://www.reddit.com/r/pokemongo/comments/lxzsf1/incubator_duplication_glitch/`. Accessed: 2021-05-05. 2021 (cit. on p. 15).

[YLL15] Shanhe Yi, Cheng Li, and Qun Li. “A survey of fog computing: concepts, applications and issues”. In: *Proceedings of the 2015 workshop on mobile big data*. 2015, pp. 37–42 (cit. on p. 25).

[yjs19] yjs. *yjs/yjs: Shared data types for building collaborative software*. `https://github.com/yjs/yjs`. (Accessed on 11/12/2021). 2019 (cit. on pp. 101, 105).

[Zaw+15a] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, et al. “Write Fast, Read in the Past: Causal Consistency for Client-side Applications”. In: *middleware*. ACM/IFIP/Usenix. Vancouver, BC, Canada, Dec. 2015, pp. 75–87 (cit. on pp. 33, 40, 69, 71, 79, 102).

[Zaw+15b] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, et al. “Write fast, read in the past: Causal consistency for client-side applications”. In: *Proceedings of the 16th Annual Middleware Conference*. 2015, pp. 75–87 (cit. on p. 58).

Webpages

[@Pub19] Cisco Public. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017–2022*. 2019. URL: `https://s3.amazonaws.com/media.mediapost.com/uploads/CiscoForecast.pdf` (visited on Nov. 3, 2021) (cit. on p. 25).

List of Figures

List of Tables

List of Listings

Résumé

La collaboration à large échelle sur Internet est un domaine d'application en pleine expansion, comme peuvent en témoigner les éditeurs de texte partagés tels que Google Docs ou Microsoft Office 365; les systèmes de partage de fichiers comme Dropbox ou NextCloud. Certains appareils dotés de capacités de réalité augmentée prennent en charge des jeux géolocalisés tels que Pokémon Go ou Harry Potter Unite, ou des applications collaboratives de modélisation et de conception d'objets en 3D.

Les systèmes existants sont basés sur des plateformes de stockage dans le nuage (*Cloud*), rajoutant parfois une mise en cache ad hoc au niveau de l'application. Les violations de cohérence sont courantes dans ce type de configurations, ce qui perturbe les utilisateurs et rend la conception des applications plus contraignante pour les développeurs. De plus, la prise en charge du fonctionnement en déconnecté est limitée, voire inexistante. En effet, les utilisateurs interagissent uniquement par le biais du *Cloud*, même lorsqu'une communication directe serait possible entre eux, puisque ces systèmes sont dépourvus de fonctions de collaboration telles que la gestion des groupes et des versions.

Cette thèse explore ces problèmes en profondeur, en étudiant l'état de l'art lié aux systèmes de stockage et services de synchronisation des données aux bords du réseau, et présente la base de données et l'intergiciel *Colony*, conçu pour répondre aux problématiques exposées. L'une des principales exigences est une approche *Edge-First* qui stocke les données au plus près de l'utilisateur, directement sur son dispositif, afin de fournir une disponibilité, une réponse rapide et transparente indépendamment de la connectivité du réseau et de l'emplacement géographique, et afin que les utilisateurs soient propriétaires de leurs propres données. Cependant, cela rend difficile la satisfaction des attentes en matière de cohérence et de fraîcheur des données.

Pour répondre à ces défis, nous avons fait le choix d'adopter une approche hybride, en fournissant les plus fortes garanties de cohérence compatibles avec la disponibilité (modèle *TCC+* formalisé dans la contribution), tout en renforçant davantage (avec un modèle *SI* présenté dans l'état de l'art) ces garanties dans les zones les plus proches et connectées (que nous appelons *zones à cohérence forte*). De plus, nous utilisons le format de données *CRDT* afin d'assurer les propriétés de convergence

sans avoir recours à un retour en arrière (*Rollback*). Un défi connexe est le surcout des métadonnées liées à la gestion de la concurrence (gros vecteurs d'horloges), que nous avons limité grâce à une topologie en arbre flexible, couplée aux zones à cohérence forte.

Pour répondre aux exigences de la collaboration de groupe, notre solution versionne les données, permet à un groupe périphérique (terminaux d'utilisateurs géographiquement proches et connectés) de synchroniser leurs versions sans dépendre du serveur central, tout en fournissant des garanties de sécurité collaborative. Notre conception fournit un accès uniforme aux données sur un spectre allant du nuage central (*Cloud*) à la périphérie du réseau (*Edge*).

Enfin, cette thèse aborde un certain nombre de défis de conception et de mise en oeuvre du système, notamment le fonctionnement déconnecté, le consensus à ordre total à la périphérie du réseau et l'absence de points de défaillance uniques malgré la topologie en arbre.

Les contributions de cette thèse peuvent être résumées comme suit:

- Une architecture de base de données décentralisée, conçue pour les applications collaboratives, qui fournit un continuum allant du nuage central à la périphérie du réseau;
- Un modèle de cohérence hybride, basé sur des garanties causales et transactionnelles à large échelle, renforcé à la cohérence d'ordre total dans les groupes de collaboration à proximité géographique au bord du réseau;
- Une conception évolutive des métadonnées et de la topologie qui limite l'encombrement des métadonnées de cohérence requises, tout en prenant en charge la déconnexion ou la migration transparente d'un noeud ou d'un groupe entier;
- Une nouvelle approche du contrôle d'accès qui s'appuie sur le modèle de cohérence présenté;
- La conception et l'implémentation efficaces de *Colony*, et son évaluation expérimentale démontrant les avantages de notre approche.

Notre évaluation expérimentale montre que : la mise en cache locale et de groupe améliore le débit de 1,4× et 1,6× respectivement, et le temps de réponse de 8× et 20×, par rapport à une configuration classique de nuage ; la performance en mode déconnecté reste la même qu'en mode connecté ; la transition et la migration sont transparentes dans les deux modes de connexion.