NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CLOUD-EDGE HYBRID APPLICATIONS

ALBERT VAN DER LINDE

Master of Computer Science and Engineering

DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon
July, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CLOUD-EDGE HYBRID APPLICATIONS

ALBERT VAN DER LINDE

Master of Computer Science and Engineering

Adviser: Nuno Manuel Ribeiro Preguiça
Associate Professor, NOVA University Lisbon

Co-adviser: João Carlos Antunes Leitão
Assistant Professor, NOVA University Lisbon

Examination Committee

Chair: Pedro Manuel Corrêa Calvente Barahona
Professor Catedrático, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

Rapporteurs: Vivien Quéma
École Nationale Supérieure d'informatique et de Mathématiques Appliquées de Grenoble, França

José Orlando Pereira
Professor Associado com Agregação, Departamento de Informática da Universidade do Minho

Miguel Pupo Correia
Professor Catedrático, Instituto Superior Técnico da Universidade de Lisboa

Hervé Miguel Cordeiro Paulino
Professor Associado, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

Adviser: Nuno Manuel Ribeiro Preguiça
Professor Associado com Agregação, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon
July, 2022

Cloud-edge hybrid applications

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.10).

To my wife, family, and friends.

Acknowledgements

First of all I'd like to thank my advisor Nuno Preguiça and co-advisor João Leitão, for inviting me to the world of research and providing the environment to keep working with them for all these years. I want to thank the Department of Computer Science of the NOVA School of Science and Technology (FCT NOVA) from NOVA University Lisbon, for hosting my research.

I also would like to thank my thesis advisory committee members (Hervé Paulino and Miguel Pupo Correia) for their insightful comments during the earlier phases. The same goes for the anonymous reviewers for their comments on the publications – many of their requests and suggestions led to substantial improvements to the research done. Also, a thank you to the final thesis defense juri for the great discussion at the conclusing of the program.

I'd also like to thank my family, especially my parents, who always supported me throughout the PhD – a very big thank you for helping me during all this time. I'd like to thank my wife Marlene for being part of my life – words are simply not enough to describe the positive impact you have had, without you it would not have been possible.

A special appreciation goes to all my colleagues, in no particular order: Bernardo Ferreira, Tiago Vale, Valter Sousa, Pedro Fouto, Pedro Ákos Costa, Luís Silva, João Silva, and Diogo Serra[†]. Also, thanks to all the students I worked with during these years: Rafael Seara, Francisco Magalhães, Frederico Aleixo, Filipe Luis, André Rijo, Pedro Durães, João Martins, Sara Simões, Francisco Fernandes, and Tiago Costa – the work on Legion in the years that passed, in particular Pedro Fouto for building the first playable game and helping with ExpoFCT each year.

A special thanks to Fundação para a Ciência e Tecnologia (FCT) for my PhD scholarship (SFRH/BD/117446/2016). The research was also supported in part by FCT/MCTES through the project Samoa (PTDC/CCI-INF/32662/2017) and NOVA LINCS (UIDB/04516/2013), and the European Union, through Syncfree (grant agreement n°609551) and LightKone (grant agreement n°732505).

Part of the computing resources used for this work were supported by an AWS in Education Research Grant, AWS Cloud Credits for Research program, and some experiments presented were carried out using the Grid'5000 testbed (www.grid5000.fr).

'[the] common and unfortunate fact of the lack of an adequate presentation of basic ideas and motivations of almost any mathematical theory is, probably, due to the binary nature of mathematical perception: either you have no inkling of an idea or, once you have understood it, this very idea appears so embarrassingly obvious that you feel reluctant to say it aloud; moreover, once your mind switches from the state of darkness to the light, all memory of the dark state is erased and it becomes impossible to conceive the existence of another mind for which the idea appears nonobvious.'

– Mikhael Gromov, 1992, source: M. Berger, Encounter with a geometer II

Abstract

Many modern applications are designed to provide interactions among users, including multi-user games, social networks and collaborative tools. Users expect application response time to be in the order of milliseconds, to foster interaction and interactivity.

The design of these applications typically adopts a client-server model, where all interactions are mediated by a centralized component. This approach introduces availability and fault-tolerance issues, which can be mitigated by replicating the server component, and even relying on geo-replicated solutions in cloud computing infrastructures. Even in this case, the client-server communication model leads to unnecessary latency penalties for geographically close clients and high operational costs for the application provider.

This dissertation proposes a cloud-edge hybrid model with secure and efficient propagation and consistency mechanisms. This model combines client-side replication and client-to-client propagation for providing low latency and minimizing the dependency on the server infrastructure, fostering availability and fault tolerance. To realize this model, this works makes the following key contributions.

First, the cloud-edge hybrid model is materialized by a system design where clients maintain replicas of the data and synchronize in a peer-to-peer fashion, and servers are used to assist clients' operation. We study how to bring most of the application logic to the client-side, using the centralized service primarily for durability, access control, discovery, and overcoming internetwork limitations.

Second, we define protocols for weakly consistent data replication, including a novel CRDT model (Δ-CRDTs). We provide a study on partial replication, exploring the challenges and fundamental limitations in providing causal consistency, and the difficulty in supporting client-side replicas due to their ephemeral nature.

Third, we study how client misbehaviour can impact the guarantees of causal consistency. We propose new secure weak consistency models for insecure settings, and algorithms to enforce such consistency models.

The experimental evaluation of our contributions have shown their specific benefits and limitations compared with the state-of-the-art. In general, the cloud-edge hybrid model leads to faster application response times, lower client-to-client latency, higher system scalability as fewer

clients need to connect to servers at the same time, the possibility to work offline or disconnected from the server, and reduced server bandwidth usage.

In summary, we propose a hybrid of cloud-and-edge which provides lower user-to-user latency, availability under server disconnections, and improved server scalability – while being efficient, reliable, and secure.

Resumo

Muitas aplicações modernas são criadas para fornecer interações entre utilizadores, incluindo jogos multiutilizador, redes sociais e ferramentas colaborativas. Os utilizadores esperam que o tempo de resposta nas aplicações seja da ordem de milissegundos, promovendo a interação e interatividade.

A arquitetura dessas aplicações normalmente adota um modelo cliente-servidor, onde todas as interações são mediadas por um componente centralizado. Essa abordagem apresenta problemas de disponibilidade e tolerância a falhas, que podem ser mitigadas com replicação no componente do servidor, até com a utilização de soluções replicadas geograficamente em infraestruturas de computação na nuvem. Mesmo neste caso, o modelo de comunicação cliente-servidor leva a penalidades de latência desnecessárias para clientes geograficamente próximos e altos custos operacionais para o provedor das aplicações.

Esta dissertação propõe um modelo híbrido *cloud-edge* com mecanismos seguros e eficientes de propagação e consistência. Esse modelo combina replicação do lado do cliente e propagação de cliente para cliente para fornecer baixa latência e minimizar a dependência na infraestrutura do servidor, promovendo a disponibilidade e tolerância a falhas. Para realizar este modelo, este trabalho faz as seguintes contribuições principais.

Primeiro, o modelo híbrido *cloud-edge* é materializado por uma arquitetura do sistema em que os clientes mantêm réplicas dos dados e sincronizam de maneira ponto a ponto e onde os servidores são usados para auxiliar na operação dos clientes. Estudamos como trazer a maior parte da lógica das aplicações para o lado do cliente, usando o serviço centralizado principalmente para durabilidade, controlo de acesso, descoberta e superação das limitações inter-rede.

Em segundo lugar, definimos protocolos para replicação de dados fracamente consistentes, incluindo um novo modelo de CRDTs (Δ-CRDTs). Fornecemos um estudo sobre replicação parcial, explorando os desafios e limitações fundamentais em fornecer consistência causal e a dificuldade em suportar réplicas do lado do cliente devido à sua natureza efémera.

Terceiro, estudamos como o mau comportamento da parte do cliente pode afetar as garantias da consistência causal. Propomos novos modelos seguros de consistência fraca para configurações inseguras e algoritmos para impor tais modelos de consistência.

A avaliação experimental das nossas contribuições mostrou os benefícios e limitações em

comparação com o estado da arte. Em geral, o modelo híbrido *cloud-edge* leva a tempos de resposta nas aplicações mais rápidos, a uma menor latência de cliente para cliente e à possibilidade de trabalhar offline ou desconectado do servidor. Adicionalmente, obtemos uma maior escalabilidade do sistema, visto que menos clientes precisam de estar conectados aos servidores ao mesmo tempo e devido à redução na utilização da largura de banda no servidor.

Em resumo, propomos um modelo híbrido entre a orla (*edge*) e a nuvem (*cloud*) que fornece menor latência entre utilizadores, disponibilidade durante desconexões do servidor e uma melhor escalabilidade do servidor – ao mesmo tempo que é eficiente, confiável e seguro.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Many applications provide an experience where users interact with one another, examples include collaborative editing software, social networks, and multiplayer games. These applications manage a set of shared objects, the application state, and each user reads and writes on a subset of these objects. For example, in a collaborative text editor, users share the document being edited, while in a multi-user game the users access and modify a shared game state. In these applications, the user experience is highly tied with how fast interactions among users can occur.

Despite these applications being user-centric, and often are fully dedicated to enable interaction between users, the architectural model used to create such applications is typically based on a complete separation between the client and server-side of the application – a client-server communication model. In this model client devices communicate only with the server which controls every aspect of running the application and mediates all interactions between users.

Although this model eases reasoning about application logic and how all parts communicate, it promotes contention on the centralized component. This approach leads to several drawbacks, not only for the end user (of the application), but also to application providers (those who develop and/or monetize the application itself).

First, servers are a scalability bottleneck, as all interactions between users have to be managed by them. The work performed by servers has polynomial growth in bandwidth usage with the number of clients, as not only there are more clients producing changes but also as each change must be disseminated to a larger number of clients. Second, when servers become unavailable, clients become unable to interact and, in many cases, they cannot even access the application. Finally, the latency of interaction among nearby users is unnecessarily high since every interaction among users, or communication between their devices, is always routed through a server.

It is not trivial to address these issues, especially if we take into account that users nowadays expect almost minimal latency while the actual amount of users of an application can change by an order of magnitude overnight. Latency is a key property in distributed applications – several studies showing that user engagement drops when latency increases [11–14], even making users believe a website may have compromised security [15] when perceived latency is high.

My thesis is that client-to-client latency can be greatly improved if client-devices are

allowed to interact directly. Additionally, by enabling client-side replicas of the data we are able to offer very fast local writes, and with client-to-client connections we can allow for disconnected operation from the server and increased server capacity as clients can coordinate to reduce server load.

1.1 Context

Creating the next popular application or game might just require an idea that people will like, but creating the underlying system to support that application is not trivial at all. It is difficult to create systems which are not only capable of providing the user with what is expected in terms of availability and response time, but are also able to scale to a growing number of users.

Scaling out is typically not a problem if each user uses a single device to work on its own data and sporadically communicates with a server. There are well studied replication and partitioning techniques for such scenarios [16–18], which in the current world of cloud systems can easily handle very large amounts of users. Providing static contents to a large number of users also poses no problem with the usage of, for example, Content Distribution Networks (CDNs), as the service can easily provide to a virtually unlimited amount of devices the required data [19, 20].

Creating highly available systems gets tricky when users want to share application state among each other while being able to continuously change this state – CDNs are of no use and cloud systems need to be tuned for each application. An application developer has to make sure that all clients are able to operate on data (apply writes) and that clients are updated when data of their interest changes. Interestingly, the supporting system has to deal with a possibly polynomial increase in read and write load per additional client. This increase is polynomial if each new client contributes with its own operations, as each new operation has then to be propagated to all other connected clients. Coping with this increase in load has to be done while delivering in an interactive manner (high availability and low latency) the correct outcome of every users' actions to all users.

Besides having to deal with data-access and data-consistency, networking itself is an issue as server instances are limited in how many connections they can keep open, and how much data can be transferred on those connections. It is difficult to design a per-application specific load balancing system which has to keep hundreds of thousands of simultaneous connections to a server farm [21].

In particular, it is challenging to devise general purpose techniques that allow an application to balance the load imposed on servers by an increasing number of clients, and to enforce adequate semantics over the data accessed and manipulated by clients.

In this work, we aim to support user-centric applications – where users use their devices to interact directly – without sacrificing the application's scalability, reliability, and security.

Latency-sensitive applications A particularly interesting example is that of location-based and augmented reality games, such as Ingress [22] and Pokémon Go [23], where a player interacts with nearby players and low latency is crucial for interactivity. Pokémon Go, one of the most

desired games at the time (2016), received much criticism for technical issues. While the game was originally launched on Google Cloud, with regional launches to keep up with the increasingly large user base, the issues were mainly related to the sheer amount of users that played the game.

Pokémon Go is a good example as the application is very data-intensive: client-devices near-constantly apply writes (from catching Pokémon to continously updating the user device's location) and must be kept constantly updated with the events that happen around the user (other user's actions and locations). Each client device interacts continuously with a server, both sending data back and forth nearly constantly as the application is used. There is no offline mode, meaning the user is unable to do anything unless a server can be reached.

The application requires that all clients must, at all times, be able to read and write data to and from the server, and that all clients must constantly be updated on any changes that must be shown to the user as these can impact the users' actions.

Another important aspect to consider is users misbehaving – in multiplayer games this is especially important as a single cheating user can impact the experience of all other users. Users should not be able to apply operations nor be able to observe data in a way that was not originally intended by the application developer.

In summary, these are the main aspects to reason about when building a large scale application with the focus on users interacting with each other:

- first, managing the supporting system, by keeping all clients connected (to the servers) and coping with the aggregated required bandwidth and server load (networking and computing capacity);
- second, keeping data consistent and fresh at all time, led by the continuous global write load by all clients (data consistency and update latency);
- third, ensuring clients apply correct updates in what is possible in the context of the application (application security).

1.2 A cloud-edge hybrid

1.2.1 Thesis

My thesis is that a hybrid interaction model, where client devices can interact with each other directly while leveraging the server (cloud infrastructure) mostly for durability and assisting in some key aspects for correct system operation, is the better approach for creating applications.

Especially for applications that focus on interactions among users, user experience can be improved by introducing client-side data replicas and allowing for peer-to-peer interactions among client devices.

With client-side data replicas it is possible to provide reduced application response time, as well as the ability to continue operating even when disconnected from the server. A side effect is the possibility for a reduction in server load as multiple operations can be compressed semantically (for example, multiple counter increments can be merged into a single unified operation).

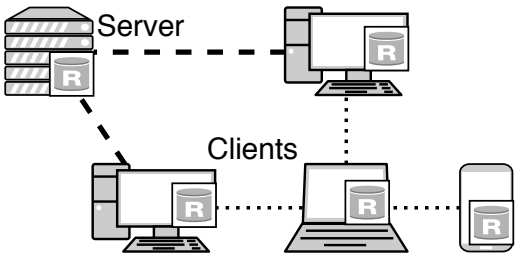


Figure 1.1: Architecture of a cloud-edge hybrid application.

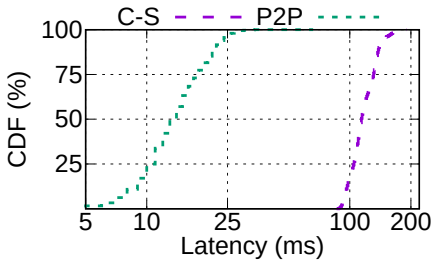


Figure 1.2: Latency eCDF of update propagation for peer-to-peer (P2P) and Client-server (C-S) deployments, with a server on AWS Ireland and clients in Grid5000 in France.

Peer-to-peer interactions among client devices, allowing for synchronization of state among their local replicas, primarily benefits user-to-user interaction latency. If operations are allowed to flow directly among client-devices instead of always following the client-to-server-to-client route, client-to-client latency can be reduced, especially noticeable for nearby users.

Additionally, clients are able to continue interaction even if temporarily disconnected from the server (as long as client devices were connected before server disconnection). Server load can also further be reduced as application instances on client devices that are connected to each other no longer need to be all individually connected to the server. This possibly leads to a further reduction in bandwidth usage if the connected clients efficiently aggregate operations from multiple clients into unified (and smaller) updates (not only due to data compression, but due to summarizing groups of operations).

1.2.2 Motivation

Many applications can benefit from the direct interaction among nearby clients' devices, from collaborative applications such as document editors [24], games [22, 23] and audience engagement applications [25], to location-dependent information sharing, such as geo-social [26, 27] or event networks [28], traffic information [29], and contact tracing applications [30]. Note that any direct client-to-client interaction is especially relevant if client devices are located close to each other.

By leveraging client-local replicas and direct interactions among client devices many drawbacks of the client-server model can be overcome, making the system less dependent on the centralized infrastructure. Figure 1.1 depicts how such a cloud-edge model would look like: a connected network including all actors, clients and server(s), where client devices connect directly and only subsets of clients connect to the server. A direct outcome of such a model is a reduction in client-to-client latency, especially noticeable for nearby clients – in Figure 1.2 we show client-to-client latency results comparing the proposed cloud-edge hybrid model to a typical client-server deployment (these results are detailed in Section 3.3 and Section 5.5).

1.2.3 Goals and challenges

We propose a gradual shift of responsibilities from the server-side to the client-side of the application:

- replication of application data at the client, allowing for local operation generation and execution and also offline work;
- peer-to-peer connections among client devices to synchronize application state.

The overall goals of allowing client-side replicas with client-to-client synchronization are threefold.

First, considering communication among the client-side of the application, peer-to-peer connections must be used among client devices, while leveraging efficient network usage and elimination of redundancy (of network paths).

Second, client-side replicas and client-to-client synchronization must be enabled and be efficient. Additionally, a replica on the client is unable to store large amounts of data, requiring partial replicas for large settings.

Finally, dealing with malicious behaviour of replicas, as the replicas themselves are no longer running solely on the centralized servers controlled by the application developer, but now run directly on client devices.

Each of these goals has its challenges that need to be overcome. We summarized the challenges the hybrid approach brings, which are explored throughout this thesis, as follows:

- a connected graph must exist among the server and all client devices which share interest in the same data as to ensure all clients observe all operations (overlay networks);
- data on client-devices and the server must be kept consistent while allowing for high-availability and network partitions (weak consistency models);
- data on client-devices must be partitionable to allow applications to only store at each client the data that client is interested in (partial replicas);
- clients must not be able to tamper with the application's underlying mechanisms for their own benefit, as malicious behaviour from users can impact all connected users (security).

1.2.4 Results

The main results can be semantically divided into three parts:

- a modular system design which allows for client-side replicas and client-to-client propagation of data;
- modules which allows for weak consistency guarantees and also partial replication on client devices;
- security modules which allow for weak consistency semantics in insecure settings.

1.2.4.1 Client-side replicas and client-to-client propagation

We designed a modular framework for creating web applications which we named Legion. Legion enables the part of the application that is running on the client's device to replicate data from servers and continuously apply operations locally, while using the server mainly for durability. More importantly, Legion allows for synchronization to happen directly among client devices, moving from the client-server communication model to one where clients interact directly when possible.

This approach, detailed in Chapter 3, enables many benefits to the application provider and its users – the main outcomes are reduced user-to-user latency, reduced server load, and the ability to operate disconnected (from the server).

This work supports the claim that instances of the application on the client-side working together is a good approach for user-centric applications, especially if users interact with each other. We thus envision a world where peer-to-peer systems at the client-side are intrinsically connected with such applications.

Legion was built with a modular approach allowing the chosen backend services to be exchanged trivially by only implementing a small integration module. A concrete example is switching long-term storage between local-storage (on the client device), Legion's servers, some centralized/cloud backend, or a combination of these. This approach allows for easy replacement of each of the networking, data handling, and security stacks of the system, depending on the application's needs.

1.2.4.2 Consistency guarantees and partial replication

In Chapter 4 we discuss enabling client-side replication and how Legion implements causal consistency using Conflict-Free Replicated Data Types (CRDTs [31]). Legion provides a data storage that spans a potentially huge number of clients and also the centralized infrastructure – including a high turnover of dynamically spawning and removing replicas (user's devices).

The design of Δ-CRDTs is detailed where the proposed communication model allows for a broad range of implementations – not only does it allow for implementing the partial replication algorithms as referenced next, it allowed to implement the secure models referenced in the next section.

The base design of Legion provides causal consistency in a full replication model – altough full-replication is partitioned per network of clients, within such a network each replica is unable to dynamically choose which objects to replicate, even if only interested in a very small subset of the data.

In Chapter 6 we investigate how to support partial replication at client devices operating under weak consistency models. We prove the impossibility of providing genuine partial replication in our system model – as we must account for ephemeral client-side replicas – and discuss practical alternatives which aim to provide causal consistency without forcing full-replication.

1.2.4.3 Weak consistency semantics for insecure settings

In many applications users have incentives for misbehaving. When reasoning about the previous application examples (in particular multiplayer games), it is clear that users would attempt to misbehave to gain an advantage, but most users whould only attempt such actions if they would be able to do so without being discovered (or if it would be impossible to prove these users did anything wrong).

Although Legion has security mechanisms for privacy and access control, it depends on the non-malicious behaviour of its authenticated users. Legion works well in a setting where groups of users work together to achieve a common goal, such as, for instance, collaborative document editing. Users can easily cheat on any application running on the base Legion system, by altering or circumventing the protocols the application depends on.

We explore the impact of malicious behaviour on the consistency guarantees of weak consistency models in Chapter 5. We discuss possible attacks and explore in detail (malicious) circumventions to the guarantees provided by causal consistency. We detail how we designed, implemented, and evaluated the algorithms which provide various levels of guarantees to explicitly deal with misbehaviour from client-side replicas.

1.3 Summary of contributions

The design of the cloud-edge applicational model led to the following specific contributions:

Communication – connecting client devices using peer-to-peer connections (Chapter 3):

- networking among client-side peers with modular choice of network overlay (providing implementation of cliques, randomized graphs, trees, and DHTs);
- an additional topology-aware overlay-network optimized for client location to improve communication latency and efficiency by reducing redundant paths;
- the combination of a leader election mechanism and data aggregation and message compression modules to reduce redundant communication with the server;
- lightweight security mechanisms enforcing data privacy and integrity.

Replication – allowing for efficient client-side data replication:

- transparent (to the user) client-side replication, using CRDTs, while allowing for the choice of synchronization based on state or operations (Chapter 3);
- the design of Δ-CRDTs for efficient synchronization in settings with dynamically changing networks (Chapter 4);
- a study on the impossibility of providing causal consistency with genuine partial replication, resulting in theorems on the minimum requirements for the proposed system model (Chapter 6);

- the algorithms that explore the edges of the impossibility and which aim to provide causal consistency without forcing full-replication at all times or at every client (Chapter 6).

Security – enforcing privacy and integrity while dealing with client misbehaviour (Chapter 5):

- a systematic study on how client misbehaviour can impact the guarantees of causal consistency;
- the definition of secure consistency models, variants of causal consistency and also eventual linearizability, preventing multiple types of misbehaviour;
- the algorithms for implementing the secure consistency models.

Prototypes and experimental evaluation – the implementation of the proposed systems and evaluations, in particular a comparison of our system with existing cloud-based solutions (Section 3.3) and an extensive evaluation of the security aspects (Section 5.5).

1.3.1 Publications

This work has led to the following publications.

Key publications

WWW'17 [1] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 283–292, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee [overlay network, replication, security based on trusting authenticated users]

PaPoC'16 [2] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-CRDTs: Making δ-CRDTs Delta-Based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, New York, NY, USA, 2016. Association for Computing Machinery [replication]

PaPoc'20 [3] Albert van der Linde, Diogo Serra, João Leitão, and Nuno Preguiça. On Combining Fault Tolerance and Partial Replication with Causal Consistency. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery [replication (partial replicas)]

PaPoC'20 [4] Albert van der Linde, Pedro Fouto, João Leitão, and Nuno Preguiça. The Intrinsic Cost of Causal Consistency. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery [replication]

VLDB'20 [5] Albert van der Linde, João Leitão, and Nuno Preguiça. Practical Client-Side Replication: Weak Consistency Semantics for Insecure Settings. In *Proceedings of the VLDB Endowment*, volume 13 (12), page 2590–2605. VLDB Endowment, July 2020 [security]

Other publications and communications

Parts of the thesis were also presented in the following.

- (Doctoral workshop) [6] Albert van der Linde. Edge-cloud hybrid model for distributed apps. In *Eurosys Doctoral Workshop*, 2018 [discussion on security and partial replication].
- (Presentation) [7] Albert van der Linde, João Leitão, and Nuno Preguiça. Secure causal delivery with client-side replication. In *Presentation at the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, 2019 [first approach on security].
- (Presentation) [8] Sara Simões, Albert van der Linde, and Nuno Preguiça. Composition of CRDTs Using References in Key-value Stores. In *Comunicações do 11º Inforum.*, 2019 [on composing crdts by value or by reference].
- (Book chapter) [9] Georges Da Costa, Alexey L. Lastovetsky, Jorge G. Barbosa, Juan C. Díaz-Martín, Juan L. García-Zapata, Matthias Janetschek, Emmanuel Jeannot, João Leitão, Ravi Reddy Manumachu, Radu Prodan, Juan A. Rico-Gallego, Peter Van Roy, Ali Shoker, and Albert van der Linde. Programming models and runtimes. In *Ultrascale Computing Systems*, pages 9–63. Institution of Engineering and Technology, 2019 [of Legion and others].
- (National conference) [10] Tiago Costa, Albert van der Linde, Nuno Preguiça, and João Leitão. Controlo de Acessos em Sistemas com Consistência Fraca. In *Actas do 8º Inforum.*, 2016 [approach of adding access control into crdts directly].

1.4 Outline

The remainder of the dissertation is organized as follows:

Chapter 2: Fundamental Concepts and Research Context – explains how client devices can interact (WebRTC), target applications, and related work.

Chapter 3: Peer-to-peer and applications: Legion – details the Legion framework to create web applications, the implementation of the proposed cloud-edge model;

Chapter 4: Client-side replication – details Legion's consistency guarantees, the synchronization model, and implementation of Δ-CRDTs;

Chapter 5: Securing causal consistency – details how replicas can maliciously circumvent the guarantees of causal consistency, proposes secure consistency models with additional guarantees against those attacks, and details the design and implementation of algorithms providing those guarantees;

Chapter 6: Client-side partial replication – discusses the difficulty in providing weak consistency models when considering partial replicas at the client, and proposes practical solutions;

Chapter 7: Final considerations – presents the conclusions on this work and topics for future research.

Chapter 2

Fundamental Concepts and Research Context

Interactive applications run in user devices, either in a web browser or as a standalone application (desktop or mobile), and access Internet services to read and modify application state. These services are typically supported by servers running in a data center, often running within a cloud infrastructure.

We intend to dillute the clear boundary that exists between client-side instances of an application and its cloud counterparts. As discussed in the introduction, the focus of this work is on creating client-side replicas with direct connections between client devices, letting client-side instances communicate directly among each other to synchronize application state.

We begin by explaining the main enabling technologies (Section 2.1), followed by a brief overview of target applications (Section 2.2). The remaining sections cover the state-of-the-art which applications can leverage, namely in terms of data storage (Section 2.3) and peer-to-peer (Section 2.4).

Moving application state to the client-side requires special care in terms of security, especially if considering malicious behaviour from end-users. As security related aspects are mostly covered in Chapter 5, we leave the related work for Section 5.6.

2.1 Enabling technologies

Real time communication is used by many web services supporting applications (such as Skype and Zoom), but requires large downloads, the use of native apps, or plugins. Downloading, installing, and updating plugins can be complex for both the developer and end user. Additionally, it is often difficult to persuade a person to install plugins (or browser extensions), which impacts the adoption of applications with this requirement.

Web browsers, in particular, restricted the ability to establish direct communication channels among client devices. The Web Real Time Communication (WebRTC) initiative has solved this limitation by enabling direct communication between browsers. WebRTC [32, 33] was specifically designed to support plugin-free, realtime video, audio, and data communication, directly between

browser instances (i.e., HTML pages with enabled JavaScript). Additionally, HTML5 makes it possible for web applications to locally store data that persists across sessions on the same browser, further motivating client-side replicas with weak consistency models.

WebRTC (and most HTML5 features such as local storage and multithreading support) has increasing adoption by major browsers, being already supported my most.[1] Besides support for multiple browsers in different devices (e.g., mobile), interoperability is possible (connections between any pair of browsers such as Chrome and Firefox).

Firewalls and NAT boxes are also an important factor which restricted connectivity among client devices. This limitation can currently be circumvented by relying on widely available techniques, namely leveraging STUN [34] and TURN [35] services.

2.1.1 WebRTC API

WebRTC is an API specification standardized by the W3C to allow web browsers to communicate over peer-to-peer connections. To acquire and communicate streaming data, WebRTC offers the following APIs:

- MediaStream, to get access to multimedia data streams, such as the user's camera and microphone;
- RTCPeerConnection, for audio or video calls, which facilitates encryption and bandwidth management;
- RTCDataChannel, for peer-to-peer communication of generic data, using the same API as WebSockets (full-duplex communication channels over TCP).

WebRTC audio and video engines dynamically adjust bitrate of each stream to match network conditions between peers. When using a DataChannel this is not true, as it is designed to transport arbitrary application data. When using DataChannels it is the application developer who is responsible to compress data before sending. Similar to WebSockets, the DataChannel API accepts binary and UTF-8 encoded application data and, in contrast to WebSockets, it gives the developer choices on message delivery order and reliability. There is no choice on security: channels are always end-to-end encrypted using DTLS [36].

Altough WebRTC is designed for peer-to-peer connections, applications using WebRTC rely on servers in order for each of the following interactions, mediated by a centralized server, can happen:

- before any connection can be made, WebRTC clients (peers) need to exchange network information (the signaling protocol as detailed in the next section);
- for streaming media connections, peers must also exchange data about media such as video format and resolution;

[1] http://iswebrtcreadyyet.com

- additionally, as clients often reside behind NAT gateways and firewalls, these may have to be traversed using STUN [34] (Session Traversal Utilities for NAT) or TURN [35] (Traversal Using Relays around NAT) servers.

2.1.2 Signaling

Signaling is the process of communicating network information between future peers in WebRTC. In order for a WebRTC application to set up a 'call' (i.e., a connection between two peers), two devices first need to exchange information: i) session control messages used to open or close communication channels; ii) error messages; iii) media metadata, such as codecs and codec settings, bandwidth, and media types; iv) key data, used to establish secure connections; v) and network data, such as a host's IP address and port as seen by the outside world.

Figure 2.1 depicts the operation of the signaling protocol to establish a WebRTC connection. A signaling channel can be any medium that allows messages to go back and forth between clients. This channel is not implemented by the WebRTC APIs – it has to be implemented by the application developer. As it is only required to exchange text initially to bootstrap the connection, it can be as rudimentary as using e-mail or an instant messaging application (any medium that allows an exhange of text). Ideally this process is done via a centralized server hosted by the application developer to control connections that clients establish. Interestingly, WebRTC's DataChannels themselves can be used to further establish connections among other connected clients (which we explore in this work).

As depicted in the figure, when peers reside behind firewalls or NATs they have to make use of STUN [34] or TURN [35] services to establish connections. STUN is used to obtain the public address for a peer to pass along via the signaling mechanism. If no connection can be made between two peers (due to, for example, non-permissive firewall rules), WebRTC can resort to the use of TURN. TURN servers are used to relay encrypted data between peers as if a peer-to-peer connection exists, but underneath (and transparent for both clients and application developer) a client-server architecture is used.[2]

2.2 Applications

Applications where users interact among each other are typically implemented with a client-server communication model (being it a web, mobile, or desktop application). Although this eases reasoning about the application logic, and simplifies the communication paths among clients, it promotes contention on the centralized component.

In this section we focus on user-centric applications – applications where users interact and share information – which may benefit from our proposal of including a local data replica and allowing client-side replicas to synchronize directly. In order to explore the design considerations for a system that provides peer-to-peer interactions among client devices, we must first discuss

[2] Altough TURN services are freely available, in our experiments (Section 3.3 and Section 5.5) we disabled the use of TURN servers.

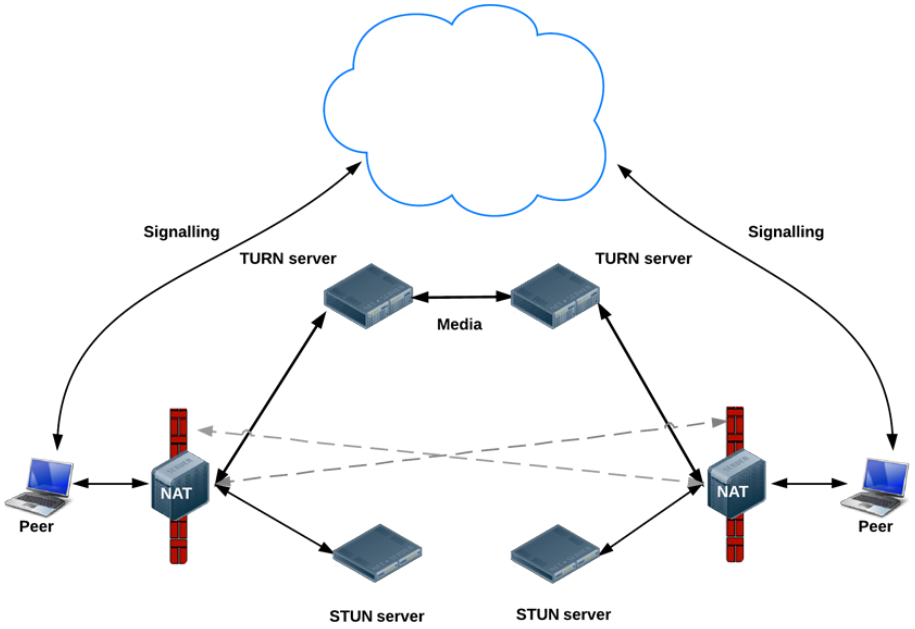


Figure 2.1: Overview of WebRTC signaling, image from www.html5rocks.com/en/tutorials/webrtc/infrastructure/. To establish a connection devices use signaling to propagate the required data to establish a connection (such as addresses, keys, and encoding for media). Devices behind NAT devices use STUN (which enables peer-to-peer connections for clients behind NAT) or TURN (to communicate through a relay server when NAT/firewalls disallow such connections).

the target application's requirements. Note that for many applications only including a local data replica already enables many benefits, such as the ability to work offline and very fast application response time.

2.2.1 Information dissemination

Many systems and applications operate based on small messages being delivered to the user's device. The application then shows these messages to the user or fetches additional data based on these (typically small sized) notifications. The providing system needs to ensure all clients receive all updates, by guaranteeing that all messages are delivered and that delivery itself is timely. The services providing the building blocks for such applications seem to be well suited to use peer-to-peer dissemination among client devices.

Such a service, when leveraging peer-to-peer dissemination, must ensure that all replicas where information is generated are also able to propagate this information to all other interested replicas. This means a connected network has to exist with efficient communication, while being resilient to the presence of replica churn. This is especially important as client-side replicas are ephemeral in comparison to a traditional server. Note that security, namely data privacy (users may can only access authorized data) and data integrity (the data is valid and accurate) must be enforced.

2.2.2 Collaborative applications

Several applications and frameworks support collaboration across the Internet by maintaining replicas of shared data at the client's device [24, 37, 38].

A collaborative editor (such as shared text or video editors) is a piece of software that allows several people to edit files using different client-devices, working together through individual contributions. In collaborative editing the main challenge is to figure out how to apply updates from remote users, who produced these updates on versions of the document that possibly never existed locally, and that can potentially conflict with the user's own updates. Users can coordinate to write on previously decided sub-parts of the document, reducing chances of conflicting changes, or, on the other end of the spectrum, users can work simultaneously together on the same task. For example, Etherpad [24] allows clients to collaboratively edit documents while ShareJS [37] and Google Drive Realtime [38] are generic frameworks that are able to manage concurrent modifications to different types of objects.

The basic need for such systems is the possibility for collaborative (possibly in real time) editing of objects while preserving user intent. Some approaches to manage concurrent access to objects include:

Turn taking, where one participant at the time 'has the floor'. This approach lacks in concurrency but is easy to comprehend, trivially preserving user intent.

Locking based techniques, where concurrent editing is trivially possible as users work on different objects. Pessimistic locking (similar to turn taking but at smaller granularity) introduces delays and optimistic locking introduces problems when the lock is denied, which can lead to an user's updates being rolled back to a previous state, leading to work being reverted or lost.

Serialization can be used to specify a total order on all operations. Non-optimistic serialization delays operations until all preceding operations have been processed while in optimistic serialization, executing operations on arrival is allowed, but there might be a need to undo/redo operations to repair out-of-order executions (possibly using programatically defined merge procedures). Such solutions may require user input to come up with a correct result (for example, merging two concurrent overlapping commits in `git`).

Operational Transformation or commutative operations can be leveraged to address the challenge of collaborative editing systems. By using such operations a high degree of concurrency can be achieved while capturing and preserving user intent (the most commonly used techniques are detailed in Section 2.3.1).

A service supporting collaborative applications has to be able to maintain, besides some method to communicate or propagate data and ensure secure interactions, also local replicas on the users' devices. This is important to ensure operations can be generated locally, as to minimize operation latency and to allow for offline work (from the server, or from other replicas).

Additionally, this allows client-side replicas to synchronize directly minimizing user-to-user interaction latency.

Such local replicas, leading to high concurrency of all changes generated at many replicas, lead to the necessity of efficient synchronization mechanisms among a large number of replicas, while keeping data consistent and fresh at all times.

2.2.3 Multiplayer games

Services [39, 40] providing multiplayer games typically divide the application into client and server to provide a clear boundary on who does what – the centralized component, the server or cloud, is used to execute and verify write operations on data, while restricting what clients can read.

This separation ensures that the client component can provide the user only with what he is allowed to see, serving mostly as thin client – an interaction proxy between the user and the server. This separation is not only important due to the sensitive access to data (privacy and integrity) but also as development is much easier when reasoning about correctness of application data (consistency).

In fact, one major issue when creating a game is reasoning about, and eventually dealing with, cheating players [41]. Altough for the previous application examples simple access control seems enough, in games the users can cheat by tampering with the code of the game – one example is modifying the rendering code in First-Person Shooters so that walls of a game are transparent, making it easy to spot other players [42].

The main additional requirement is ensuring that users are not able to tamper with the previously discussed requirements, namely connectivity and propagation latency, while also having to deal with users being able (and having the strong incentive) to act maliciously.

Discussion From the previous examples, we can summarize the properties needed:

- low propagation latency among clients;
- scalable solution, implying low load increase on the components of the system as the system scales to a large number of clients;
- ability to work offline or disconnected from the main server;
- the ability to continue interaction with other clients when the server becomes unreachable, if a subset of connected peers are contributors of information;
- very fast application response time;
- addresses malicious clients.

There are many requirements to be able to provide these properties:

- allow for client-side replicas, where changes can be created and applied locally without coordination;
- ensure that any pair of replicas can synchronize their local data while keeping state consistent at all times;
- ensure all replicas are connected and, when failures happen, that they are able to reconnect and re-synchronize efficiently;
- ensure propagation latency is low, especially for close-by clients where interaction latency is noticeable;
- provide a secure system, ensuring not only data privacy, integrity, and non-repudiation, but also resilience against general malicious behaviour.

2.3 Data storage systems

Interactive services often store client data on geographically distributed data-centers, trying to provide low latency and high availability for interacting with the data. Typically, replication and distribution of state across geographically separated data centers is required to ensure low latency and fault tolerance.

In such systems, a problem arises, formally captured by the CAP theorem [43]. The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three of the following: Consistency (linearizability of all operations, where every request must act as if it were executed atomically at a single replica), Availability (every request receives a response about whether it succeeded or not), and Partition tolerance (the system continues to operate despite arbitrary message loss, partial failure of the system, or unavailability due to network partitions). Due to systems being distributed, and as network and device failures will eventually happen, partition tolerance cannot be precluded, leading developers the choice about whether to sacrifice consistency or availability.

More recent works detail that CAP isn't as strict as stated – availability can range from 0 to 100 percent, partitions are rare and nuanced, and consistency can provide many different guarantees [44]. Nevertheless, our system model aims to provide an as high as possible availability while partitions, especially taking into account client-side replicas, are expected to be frequent. Therefore we must consider weakening the provided consistency guarantees. We can broadly classify consistency as follows:

Strong consistency [45, 46]. A system is said to provide strong consistency if all accesses to data are seen by all clients in the same order (sequentially). These approaches simplify the development of applications, as replicas do not diverge, at the price of requiring coordination among replicas for executing operations. Operations may execute concurrently, but concurrent write operations on the same data items are not allowed. This leads to

distributed systems providing strong consistency coming to a halt if enough replicas become network-partitioned, as coordination is necessary to ensure conflicts do not happen. Some examples include linearizability [46], parallel snapshot isolation [45], and serializability [47] where concurrent (conflicting) updates are not allowed without some form of coordination. Coordinating replicas for executing all updates is prohibitively expensive for high throughput and large numbers of clients (manipulating the same set of data objects). Consistency can be maintained but the system must sacrifice availability.

Eventual consistency [48]. Distributed computing systems which aim to achieve high availability must weaken their consistency guarantees – for example, eventual consistency informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the same value. This allows these systems to, even during network partitions, always serve read and write operations over data – write operations may execute concurrently. Eventual consistency is thus not suitable for every application. Consider a message board, where user A creates a post to which user B replies. Due to network latency, user C may receive B's reply before receiving A's initial post. Although eventually all replicas receive all updates and converge to a common state, this example shows that using eventual consistency can lead to confusion and error-prone application development.

Causal consistency [49, 50]. A system provides causal consistency if potentially causally related operations are seen by every replica of the system in an order that respects their causal dependencies (i.e., the happens before relations between operations as defined by Lamport [49]). Concurrent writes (write operations which are not causally related) may execute at different replicas in different order. When a replica performs a read followed by a write, even on different objects, the newer write is said to be causally dependent on the write that originated the read value, because the result of the read value may have had an impact on the write value. Intuitively, returning to our previous message board example, such a system would never show B's reply before A's initial post as the former is causally dependent on the latter – a replica may only present B's reply after all of its causal dependencies, in this case A's post, have been applied.

Using weak consistency models (eventual or causal consistency) usually comes with a cost: state divergence. To address state divergence, conflict resolution techniques must be used such as the ones discussed in the next section.

2.3.1 Conflict resolution techniques

Relaxing from a strong consistency model to a weaker model, such as causal consistency, minimizes the amount of coordination required among replicas at the expense of having to deal with concurrent updates. When multiple replicas can concurrently write to the same data, some

control on object versioning has to be done using, for example, logical clocks or version vectors [49, 50] and conflict resolution techniques have to be applied. Common conflict resolution techniques include:

Last writer wins In this case, the last write based on a total order (typically using physical clocks) will overwrite older ones. Besides potential problems that may arise due to clock skew, this approach leads to lost updates, where the effects of one update are silently overwritten by a concurrent update with a larger timestamp.

Programatic merge In this case the programmer can decide what to do when conflicts arise. As an example, an application maintaining shopping carts can choose to merge the conflicting versions by returning a single unified cart. This conflict resolution technique requires replicas to be instrumented with a merge procedure (e.g., Coda [51]), or alternatively, requires replicas to expose diverging states to the client application, which then reconciles and writes a new value (e.g., Dynamo [48]).

Commutative operations If all operations are commutative, conflicts can easily be solved. Independently of the order, when all operations have been received (and applied), the final outcome will be the same. An always incrementing counter, where each operation is uniquely marked by the writing replica, is a simple example: independently of the order of operations, the final result reflecting all operations will be the same. Commonly used commutative operation techniques are:

Operational Transformation (OT) [52, 53] The idea of OT is to transform the parameters of an operation to take into consideration the effects of previously executed concurrent operations, so that the outcome converges to a common and consistent state. Consider the example depicted in Figure 2.2. Two users concurrently edit a text document that initially contains 'abc'. User A inserts 'x' at position 0 and user B deletes 'c' from position 2. If both users execute their operation locally and later receive the operation of the other user (due to network latency), the final states diverge at user A and at user B, respectively, to 'xac' and 'xab'. Transforming the operations when receiving them resolves this problem – when user A receives the delete, it is transformed to increment one position and when user B receives the insert, it remains unchanged. Both outcomes then become 'xab', independently of the order in which operations are applied.

Although many algorithms for implementing OT have been proposed [54–56], it was shown that most algorithms proposed for decentralized architectures are incorrect [57].

Conflict-free Replicated Data Types (CRDTs) [31, 58, 59] CRDTs are replicated data types that are guaranteed to eventually converge towards a common state (that is, when all updates are received by all participating replicas). An example is a CRDT counter, which converges because its increment and decrement operations are commutative.

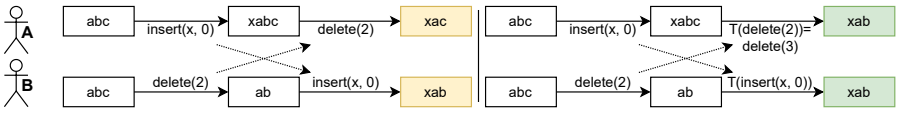


Figure 2.2: Operational Transformation. If user A and user B directly apply the remote operations, they obtain diverging state. By applying a transformation function T before executing operations, the state of both users converges.

> Most CRDTs use some form of metadata to ensure that all operations on an object, for example concurrent writes on the same key in a map, are commutative.
>
> No coordination is required to ensure convergence, so updates always execute locally (immediately), un-affected by network latency, faults, or disconnections.

In Chapter 4 we explore common CRDT designs, discussing the drawbacks when applying them direct within our system model, and explore an alternative design (Δ-CRDTs, in Section 4.3).

The proposed cloud-edge systems (described in Chapter 3 and Chapter 5) apply the usage of commutative operations with programatic merge for cases where the CRDTs are unable to decide a value.

2.3.2 Data storage systems

A number of data storage system have been designed for supporting distributed applications. We give a brief overview of some of these systems.

PNUTS [60] One way to avoid state divergence, as achieved in Yahoo!'s PNUTS, is to funnel all state changing operations through a per record chosen primary site and lazily propagating updates to other replicas. This increases latency and reads can return stale data, but data exposed to users is always consistent. The problem of this approach is availability as the primary site is a potential single point of failure.

Spanner [46] is a system which provides scalable data storage and synchronous replication. Spanner provides strong consistency using 2-phase commit and the Paxos algorithm as part of its operation to replicate data across data-centers. It also makes use of hardware-assisted time synchronization using GPS clocks and atomic clocks to ensure global consistency.

One server replica is elected as the Paxos leader for a replica group, which will become the entry point for all transactional activity for that group. Groups may include read-only replicas, which do not vote in the Paxos algorithm and cannot become group leaders.

Furthermore all transactions in Spanner are globally ordered as they are assigned a hardware assisted commit timestamp. These timestamps are used to provide multi-versioned consistent reads without the need for taking locks. A globally safe timestamp is used to ensure that reads at the timestamp can run at any replica and never block behind running transactions.

Dynamo [48] is a highly-available key-value storage system. To achieve high availability, consistency is sacrificed using object versioning and application-assisted conflict resolution, exposing data consistency issues and reconciliation logic to the developers. During network partitions operations are allowed to continue, and update conflicts are detected using a vector clock scheme where client-side conflict resolution is preferred.

Data is partitioned and spread over different replicas using consistent hashing. Dynamo ensures that adding and removing replicas can be done without manual effort by using a gossip based failure detection and membership protocol, creating a decentralized system. Each replica is aware of the data being hosted at its peers and each replica actively gossips the full routing table with other replicas in the system.

Gemini and RedBlue consistency [61] build on the premise that while a system can be leveraged to use eventual consistency for higher performance, strong consistency may be necessary to ensure correctness of some operations.

RedBlue consistency labels each operation as red or blue. Blue operations are to be fast (eventually consistent) while red operations are slow (strongly consistent) – blue is used whenever possible and red only when needed. Gemini is a system implementing RedBlue consistency, with experimental results showing that RedBlue consistency provides high performance while being able to maintain application invariants. The downside is that transactions have to be individually modified and correctly labelled.

Saturn [62] is a metadata service for geo-replicated data management systems that enforces causal consistency. It builds on the idea of decoupling the metadata path from the data path. Saturn uses labels and tree-based metadata dissemination aiming to provide high throughput and data visibility, implementing partial replication [63]. Updates are serialized within the metadata path and transmitted in FIFO order ensuring that operations are delivered (and executed) in an order that respects causality. The methods for building an optimal metadata-path limit scalability, making the system impractical if new replicas are dynamically added and removed.

Bayou [64] provides a weakly consistent replicated database where updates are first tentatively executed at the replica which first receives the write, and later committed after being totally ordered by a primary replica. Access control is checked at the tentative execution and, again, at the final commit, possibly rejecting previously accepted operations. To handle concurrent updates, individual write operations can have rules for application-specific conflict detection and resolution. Bayou also introduces the notion of session guarantees, ensuring additional properties, such as read-your-writes.

SwiftCloud [65] is an eventual consistency data storage system with low latency that relies on CRDTs to maintain client caches, providing fast reads and writes at the expense of data staleness. The main focus of this work is to integrate client and server-side storage.

Responsiveness is improved when accessing objects which are locally available at the client cache, allowing also for disconnected operation (on cached objects).

In the presence of infrastructure faults, a client-assisted failover solution allows client execution to resume immediately and access consistent snapshots without blocking. Additionally, the system supports merge-able and strongly consistent transactions that target either client or server replicas and provide access to causally-consistent snapshots efficiently.

By keeping client-side caches, updates can be applied locally and later sent to datacenters. As soon as updates are visible to a number of datacenters to ensure no data-loss, they become visible to clients with the datacenters pushing notifications to clients. SwiftCloud internally uses CRDTs for convergence and conflict resolution.

Simba [66] is a system designed to remove the complexities of network and data management from the development path of applications. The motivation is for allowing developers to focus on interface and features, with the system offering data synchronization with flexible policies while handling failures and efficiently utilizing mobile resources. Simba lets developers choose the desired consistency of data that is stored separately as binary objects (e.g., images) and of tabular data, enforcing the consistency semantics on both types of data and on the relation between them. It proposes consistency abstractions for application data, allowing applications to choose among various consistency models (strong, causal, and eventual consistency).

Diamond [67] is a data management platform that aims to support *reactive* applications. Applications created to be reactive must make it seem that devices and cloud are continuously being synchronized among each other, which is a complex distributed data management problem for programmers. This is achieved by using reactive data types, which are synchronized between client and cloud, and reactive transactions, which are re-executed as data changes.

This work demonstrates that a system, which hides the complexity of distributed data management, can greatly simplify and expedite the design of applications.

2.3.3 Client-side replication

At the client-side, many applications (especially web applications) use a stateless approach, where data is fetched from the servers whenever necessary. A number of applications store data at the client-side to provide faster response times and to allow continued local operation when the device is disconnected from the network. For example, Google Maps can be used in offline mode and Facebook supports offline feed access [68].

A large number of data storage systems for mobile computing have been designed [69]. Some of these systems, such as Parse [70], Coda [51], and Rover [71], support disconnected operation by caching data in clients and by synchronizing clients with servers, but leverage eventual consistency models using last-writer wins to merge concurrent updates.

The previously discussed SwiftCloud [65] systen also caches objects in the client machines. SwiftCloud allows updates while disconnected and client replicas are notified of changes to update cache contents. The system supports highly available transactions [72] that enforces causality, merging concurrent updates using CRDTs.

Simba [66] stores data on client machines, allowing applications to select the level of observed consistency: eventual, causal, or serializability. Applications must provide functions for resolving conflicts that may arise when operating under weaker consistency guarantees (programatic merge).

Parse [70] is a system where data can be stored in client devices. Objects can be read and modified while disconnected, with updates being uploaded to the server when connectivity is (re-)established. Parse adopts an eventual consistency model, with the last write operation prevailing as the final state.

Cimbiosys [73] and Bayou [64] are systems where clients hold data replicas and that exploit decentralized synchronization strategies (either among clients [73] or servers [64]). Cimbiosys replicates data across multiple devices of a user, which are expected to be connected intermittently. This system only provides static data items, with increasing version numbers for an updated item – the only guarantee is that a connected device eventually receives the newest version of items of interest. Conflicts are solved automatically or by letting the user decide which to keep, creating a new item with a higher version number.

Although our work shares some of the goals and design decisions with the previous systems, the difference on letting user devices share mutable data directly among each other (through peer-to-peer interactions). Our focus is on reducing latency of update propagation and permitting user-interactions even when disconnected from servers.

2.4 Peer-to-peer systems

Peer-to-peer systems typically have a high degree of decentralization, using the resources available at each available replica. Each replica, commonly referred to as node in peer-to-peer systems, implements both client and server functionality to distribute bandwidth, computation, and storage across all of the participants of a distributed system [74]. This is achieved by allocating state and tasks among peers with few, if any, dedicated peers.

Nodes are initially introduced to the system and typically little or no manual configuration is needed to maintain a connected network (a network is connected if there is at least one communication path from each node to every other node).

Peer-to-peer systems are interesting due to their low barrier to deployment, their organic growth (as more nodes join, more resources are available), resilience to faults/ malicious attacks, and the abundance/diversity of systems [75]. Popular peer-to-peer applications include sharing and distribution of files, streaming media, telephony, and volunteer computing. Peer-to-peer technologies were also used to create a diversity of other applications, for example Amazon's Dynamo storage system [48].

It is important to note that the chosen network topology has a high impact on the performance of peer-to-peer services and applications as, for nodes to be able to cooperate, they need to be aware of the network and methods of communication. The typical approach is to create a logical network of links on top of the underlying network, called an overlay network. To achieve an efficient and robust method of delivering data through a peer-to-peer approach an adequate overlay network is necessary – it is very important to study the mechanisms for creating and managing overlay networks that match the application's requirements.

2.4.1 Overlay networks and communication models

An overlay network is a logical network of nodes, built on top of another network. Links between nodes in the overlay network are virtual links, possibly composed of various links on the underlying network. Overlay networks can be constructed on top of the Internet, with each link being a connection between two peers. When designing an application, the programmer must first decide on the overlay network to deploy and use, choosing between degrees of centralization as well as on structured or unstructured network designs.

Considering the use centralized components, a peer-to-peer (or overlay) network can be classified as being:

Partly centralized networks – these networks typically leverage components of dedicated nodes or a central server to control and index available resources. The centralized components can effectively be used to coordinate system connections, facilitate the establishment of communication patterns, and coordinate node co-operation. As an example, when client nodes want to execute a specific query only the central component is contacted, which in turn can return the set of nodes that match the query (and possibly facilitate in establishing a peer-to-peer connection). These systems are relatively simple to design but come with the drawback of a potential single point of failure and bottleneck – this approach is therefore not as resilient and scalable as a fully decentralized system. Well known examples include Napster [76], Bittorrent using trackers [77], BOINC [78], and Skype [79].

Decentralized networks – these networks aim to avoid the use of dedicated nodes. All network and communication management is done locally by each participating node, using decentralized coordination mechanisms. This way a singe bottleneck and point of failure is avoided, increasing the potential for scalability and resilience.

In this type of architecture, nevertheless, a few selected nodes may act as supernodes, as to leverage potential higher CPU or bandwidth available, gaining additional responsibilities such as storing state or even becoming the entry point for new nodes. Example protocols include Gnutella [80] and Gossip [81].

Besides the degree of centralization, one aspect that defines an overlay network is the kind of structure it forms. For example, some may form a random graph or mesh network, while others may result in an efficient tree like network to reduce redundancy:

Structured overlays – these typically force nodes into specific positions in the overlay structure, often determined by the node's identifier. Identifiers are chosen in a way that peers are usually uniformly distributed at random over the key space. This allows to create a structure similar to a hash table, a distributed hast table (DHT). This type of overlay graph is typically chosen when efficient (logarithmic complexity) key-based routing is required – a lookup method used in conjunction with distributed hash tables that enables to find the node that has the closest identifier to the key being searched. Structured overlays typically use more resources to maintain the overlay, but in return get efficient queries at the cost of poor performance when churn is high. Well known examples are Chord [82] and Pastry [83].

Unstructured overlays – these are used when no particular structure in the network is required, and where queries may be propagated by flooding the network. There is a (small) fraction of all peers in the system with whom each participant can interact directly and queries are typically disseminated among the connected peers. Each peer keeps a local index of its own resources and, in some cases, the resources of its neighbours. To ensure that a query returns all possible results, the query must be disseminated to all participants. Examples include Cyclon [84], Scamp [85], HyParView [86] and PlumTree [87].

Choosing between structured and unstructured overlays depends mostly on the usefulness of key-based routing algorithms and the amount of churn that the system is expected to be exposed to. For example, churn, the participant turnover in the network (the amount of nodes joining and leaving the system per unit of time), has a major impact in both how to maintain data and how to coordinate among nodes [75]:

Maintaining data – in partially centralized systems data is typically stored at the nodes uploading and downloading data. The central component maintains metadata, typically an index of the stored data, including where it is located. In decentralized systems queries are either flooded or some specific network structure must allow for efficient lookups.

In unstructured systems, data can be stored on uploading and downloading nodes but to locate data the queries are flooded through the network. For faster and more efficient queries, nodes may distribute metadata among neighbours.

In structured overlays distributed state can be maintained using, for example, distributed hash tables. Primitives are similar to any hash table, and easily implemented when a key-based routing function is available. When churn is high it becomes very inefficient to store large amounts of data at peers responsible for the keys, therefore indirection pointers are commonly used, pointing to the node (or nodes) that effectively holds the data.

Coordination – in partially centralized systems, the centralized component can be used to achieve coordination among nodes.

In unstructured overlays, epidemic techniques are typically used because of their simplicity and robustness to churn. Information tends to propagate slowly throughout the whole

network and scaling to very large overlays is costly. Spanning trees – a logical tree connecting all nodes in the network – can increase efficiency but maintaining the tree structure requires maintenance which becomes costly when churn is high.

In contrast, when using structured overlays, key-based routing algorithms allow for fast coordination and good efficiency among large amounts of nodes as long as the overhead of churn is small.

2.4.2 Examples of peer-to-peer overlay networks

Structured overlay networks such as Chord [82], offer high network efficiency as all requests are routed efficiently to the right nodes. Efficient lookups can be implemented using distributed hash tables, for example Amazon's Dynamo storage system [48] which internally heavily relies on DHTs. Unstructured network overlays typically flood the network, reducing efficiency, but create tolerance to network churn. Such networks can be combined with partial centralized services enabling efficient indexing, resource management, and access control (Napster [76], Bittorrent using trackers [77], BOINC [78], and Skype [79]).

A large suite of reliable broadcast gossip [81, 88] protocols exist. For example, Cyclon [84], Scamp [85], and HyParView [86] each show the importance of updating network knowledge periodically, of reacting to changes of the network, of being able to discover such changes timely, and of using random walks to provide balanced network overlays. By declaring some nodes as dedicated to the network (Gnutella [80]) or by creating efficient broadcast trees (ADCMCST [89] or PlumTree [87]) network traffic can be greatly reduced by eliminating redundancy.

We now give a brief overview of some peer-to-peer overlay networks that influenced our work:

Chord [82] was designed as an efficient distributed lookup protocol, enabling peer-to-peer systems to efficiently locate nodes that store a particular data item. It only offers one primitive: given a key, return the nodes responsible for that key. Keys are distributed over the nodes using consistent hashing and replicated over succeeding nodes. Nodes typically store their successor nodes, forming an ordered ring (considering node's identifiers), making it easy to reason about the overlay structure. For fault-tolerance a list of successor nodes is kept and for efficient lookups a finger table, shortcuts to nodes over the graph, is used to jump over nodes in the graph.

Gnutella [80] is a decentralized peer-to-peer file sharing protocol. When a node is joining the network, it tries to connect to the nodes it was shipped with, as well as nodes it receives from other clients.

Queries are issued and flooded from the client to all connected nodes, and then forwarded to any nodes these know about. Forwarding ends if the request can be answered or the request's Time-To-Live expires. The protocol in theory doesn't scale well, as queries increase network traffic exponentially at each hop. Additionally, the system can be unreliable as

queries are routed through nodes that run on regular computer user's devices, which may connect or disconnect at arbitrary intervals.

A revised version of the gnutella protocol is a network made of leaf nodes and ultra peers. Each leaf node is connected to a small number of ultra peers, while ultra peers connect to many leaf nodes and ultra peers. Leaf nodes send a table containing hashed keywords to their ultra peers, which merge all received tables. These tables are distributed among ultra peer neighbours and used for query routing, by hashing the query keywords and matching table contents.

Cyclon [84] is a membership management framework for large peer-to-peer networks. The membership protocol maintains a fixed length partial view managed through a cyclic strategy (updated every T time units). This partial view is updated by each node through an operation called shuffle. When shuffling, a node selects the oldest node in its partial view and exchanges some elements of its local partial view with it. When nodes initially join the overlay a random walk is used, ensuring that the number of connections of all nodes remains balanced. This work achieves an overlay topology with some very important traits: low diameter and low clustering coefficient with highly symmetric node degrees and high resilience to node failures.

Scamp [85] is a membership management framework for large peer-to-peer overlays. The Scamp protocol maintains two views, an out view to send gossip messages and an in view from which nodes receive messages. The out view is not of fixed length and it grows to a size logarithmic in scale to the number of nodes in the network, without any node being aware of the precise number of actual nodes in the network. In contrast to Cyclon, the protocol uses a reactive strategy, in the sense that the partial views are updated when nodes join or leave the system. Periodically nodes send heartbeat messages as to detect and recover from isolation due to failures. Not receiving any heartbeats allows the node to assume that it is isolated, triggering the join mechanism to effectively rejoin the overlay.

HyParView [86], Hybrid Partial View, is a reliable gossip-based broadcast protocol that builds on Cyclon and Scamp and maintains a small symmetric active view (managed through a reactive strategy) for broadcasts and a larger passive view (managed through a cyclic strategy) to recover timely from faults. TCP is used as a reliable transport and to detect failures. This work shows the importance of each reactive and cyclic strategies to maintain views of the network, and that the use of a reliable transport mechanism, like TCP, can greatly improve results by timely discovering network failures.

PlumTree [87] builds on HyParView's resilience a tree-based network to propagate messages to reduce redundancy of network flooding. Messages are sent over the tree overlay while the remaining links of the gossip overlay are used to enable fast recovery, i.e., tree healing.

2.5 Final remarks

Applications are created with a client-server communication model in mind, or have as goal being fully decentralized. In contrast, we claim that applications can benefit from replicating data at the client devices, especially if those client-side replicas may synchronize among themselves – while still leveraging a client-server model to support those clients (namely in finding each other, providing durability, and enforcing security).

Our approach leads to a different set of tradeoffs from the discussed systems. In the next chapters we discuss how leveraging the server together with a peer-to-peer network among clients can support applications (Chapter 3), how consistency can be provided in such a system model (Chapter 4), as well as security (Chapter 5), and present findings in allowing for partial replication at the client-side (Chapter 6).

Chapter 3

Peer-to-peer and applications: Legion

In this chapter we present Legion, a framework to enrich web applications with client-side replicas and peer-to-peer synchronization, materializing the cloud-edge hybrid model envisioned in our work.

Unlike systems that cache objects at the client and still only let clients communicate with a server [38, 65, 66, 70], Legion clients can synchronize directly among each other as well, using a peer-to-peer interaction model. To support these interactions, (subsets of) clients form overlay networks to propagate objects and updates among them. This ensures low latency for propagating updates and objects between nearby clients.

We designed Legion to support web applications where groups of up to a few hundred of users can collaborate by manipulating a set of data objects – each client maintains a local data store with a subset of the shared application objects. Legion adopts an eventual consistency model where each client can modify its local replica without coordination and updates are propagated asynchronously to other replicas. To guarantee that all replicas converge to the same state despite concurrent updates, Legion relies on Conflict-free Replicated Data Types (CRDTs) [58, 59]. CRDTs and Legion's consistency model are explored in Chapter 4.

Unlike uniform overlay networks [84, 85], Legion adopts a non-uniform design where a few selected (active) nodes act as bridges between the client network and the servers that store data persistently. These active nodes upload updates executed by clients in the network and download new updates executed by clients that have not joined the overlay (including both legacy clients, those that do not use the Legion framework, and clients unable to establish direct connections with other clients). This design reduces the load on the centralized component, which no longer needs to broadcast every update to all clients (nor track these clients).

While leveraging direct client interactions brings significant advantages, 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 builds on the access control mechanism of the central infrastructure to securely distribute keys among clients. We focus on security, in particular the misbehaviour of authenticated clients, in Chapter 5.

Client-side modules, adapters, allow Legion to, instead of using its own standalone servers,

leverage existing web infrastructures for storing data and assist in several functions of the framework, including peer discovery, overlay management, and security management. As a showcase, we describe our adapters for Google Drive Realtime (GDriveRT), a service provided by Google for supporting collaborative web applications similar to Google Docs [38].

The GDriveRT adapters allow Legion to: (i) store data in GDriveRT, while exposing an API and data model compatible with GDriveRT; (ii) support the interaction between Legion-enriched clients accessing local object replicas and legacy clients accessing the same objects through GDriveRT; (iii) resort to GDriveRT to assist in establishing initial peer-to-peer connections among clients.

Our evaluation shows that porting existing GDriveRT applications requires changing only a few lines of code (2 lines in the common case), allowing these applications to benefit from direct interactions among clients. We also show that the latency to propagate updates is much lower in Legion when compared with the use of a traditional centralized infrastructure, as in GDriveRT. Additionally, clients can continue to interact when the server becomes (temporarily) unreachable. Updates are stored locally and can be made durable by any active client when the server becomes available, either in the context of the same session or a future session. Since we avoid continuous access to the centralized infrastructure by all clients, the network traffic induced on the centralized component is lower, improving the scalability of the system. We also show that our security mechanisms have minimal overhead.

In summary, we present the design of Legion, a novel framework to enrich web applications through client-side replication and (transparent) direct peer-to-peer interactions. To achieve this, and besides introducing the design of the Legion architecture, we make the following contributions:

- a topology-aware overlay-network core that uses WebRTC and promotes low-latency links between clients (Section 3.1.1);
- a data storage service for web clients, providing causal consistency and using CRDTs (Section 3.1.2);
- a lightweight security mechanism that protects privacy and integrity of data shared among clients (Section 3.1.3);
- a set of client adapters (Section 3.1.4) that integrate Legion with existing cloud-services to support the system. In particular we show the GDriveRT adapters, enabling data storage in the GDriveRT service for durability, providing a seamless API and support for interoperation with legacy clients (Section 3.1.4.1);
- the implementation (Section 3.2) and evaluation (Section 3.3) of a prototype that demonstrates the benefits of our approach in terms of latency for clients and reduced load on servers.

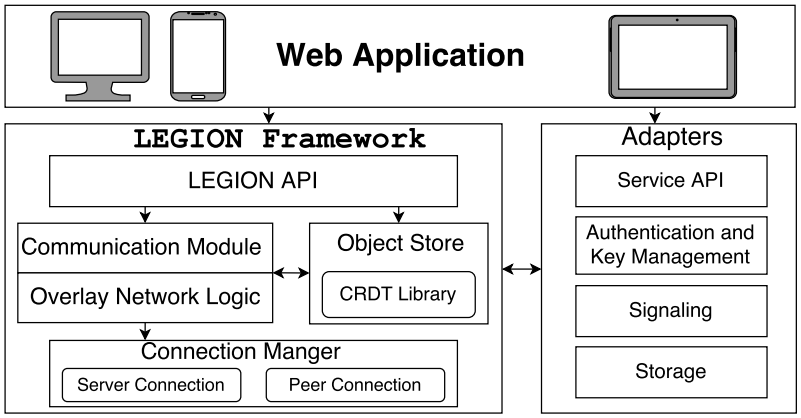


Figure 3.1: The Legion framework architecture.

3.1 System design

Legion is a framework for data sharing and communication among web clients. It allows programmers to design web applications where clients access a set of shared objects replicated at the client's devices. Clients can synchronize local replicas directly with each other. For ensuring durability of the application data as well as to assist in other relevant aspects of the systems operation (discussed further ahead), Legion resorts to a set of centralized services. We designed Legion so that different Internet services (or a combination of Internet services and Legion's own support servers) can be employed. These services are accessed uniformly by Legion through a set of adapters with well defined interfaces.

By replicating objects in clients and synchronizing in a peer-to-peer fashion, Legion reduces dependency and load on the centralized component (as the centralized component is no longer responsible to propagate updates to all clients), and minimizes latency to propagate updates (as they are distributed directly among clients). Furthermore, it allows already connected clients to continue interacting when connectivity to servers is lost.

Figure 3.1 illustrates the client-side architecture of Legion with the main components and their dependencies/interactions:

Legion API - This layer exposes the API through which applications interact with our framework.

Communication Module - The communication module exposes two secure communication primitives: point-to-point and point-to-multipoint. Although these primitives are available to the application, we expect applications to interact using shared objects stored in the object store.

Object Store - This module maintains replicas of objects shared among clients, which are grouped in containers of related objects. These objects are encoded as CRDTs from a pre-defined (and extensible) library including lists, maps, strings, among others. Clients use the communication module to propagate and receive updates to keep replicas up-to-date.

Overlay Network Logic - This module establishes a logical network among clients that replicate a shared container. This network defines a topology that restricts interactions among clients, meaning that only overlay neighbours maintain WebRTC connections among them and exchange information directly.

Connection Manager - This module manages connections established by a client. To support direct interactions, clients maintain a set of WebRTC connections among them. Some clients must also maintain connections to the central component, as discussed below.

Legion additionaly requires centralized infrastructure, which, at the client-side, can be accessed through adapters for:

- access control to the network and network eviction when access is revoked;
- durability of application state and support for interaction with legacy clients (storage);
- assisting clients to initially join the system (signaling);
- a service API adapter exposing an API similar to the server API, simplifying porting existing applications to our system.

Our prototype includes adapters for GDriveRT and a Node.js [90] implementation for the server-side.

In the remainder of this section we discuss in more detail the design of each of the modules that compose the Legion framework.

3.1.1 Networking and communications

3.1.1.1 Communication module

The communication module exposes an interface with point-to-point and multicast primitives, allowing a client to send a message to another client or to a group of clients. In Legion, each container has an associated multicast group that clients join when they start replicating an object from the container. Updates to objects in some container are propagated to all clients replicating the container.

Messages are propagated through the overlay network(s) provided by the Overlay Network Logic module. The multicast primitive is implemented using a push-gossip protocol – new or received messages are sent to all overlay neighbours (excluding the sender).

Messages exchanged among clients are protected using a symmetric cryptographic algorithm, using a key (associated with each container) that is shared among all clients and obtained through the centralized component. Clients need to authenticate towards the centralized component to obtain this key, ensuring that only authorized (and authenticated) clients are able to observe and manipulate the objects of a container. We provide additional details about this mechanism further ahead.

3.1.1.2 Overlay network logic

Legion maintains an independent overlay for each container, defining the communication patterns among clients (i.e., which clients communicate directly). The overlay is used to support the multicast group associated with that container.

Our overlay design is based on a randomized topology composed by symmetric links (similar to Scamp [85], Cyclon [84], and HyParView [86]). Each client maintains a set of K neighbours (where K is a system parameter with values typically below 10 for the scale that we target with this work). Overlay links change in reaction to external events (clients joining and leaving the network, or failing).

In contrast to the referenced works, we have designed our overlay to promote low latency links. Each client connects to K peers, with $K = K_n + K_d$, where K_n denotes the number of nearby neighbours and K_d denotes the number of distant neighbours.

Each client must maintain a small number of distant neighbours when biasing a random overlay topology to ensure that the global overlay maintains connectivity and to yield better (lower)dissemination latency, while retaining the robustness of gossip-based broadcast mechanisms [88].

This requires clients to classify potential neighbours as being either nearby or distant. A common mechanism to determine whether a potential neighbour is nearby or distant is to measure the round-trip-time (RTT) to that node. However, in Legion, since clients are typically running in browsers, it is impossible to effectively measure round trip times among them before establishing a WebRTC connection for this purpose. Since executing the signaling protocol has non-negligible overhead we have to estimate distance by some other metric.

We rely on the following strategy that avoids clients to perform active measurements of RTT to other nodes. When a client starts, it measures its RTT to a set of W well-known web servers through the use of an HTTP HEAD request (the web servers employed in this context are given as a configuration parameter of the deployment, see Section 3.2.1). The obtained values are then encoded in array of size W which is included into messages related to network management. These values are then used as coordinates in a virtual cartesian space of W dimensions. This enables each client to compute a distance function between itself and any other client.

3.1.1.3 Connection manager

This module manages all communication channels used by each Legion client, namely server connections to the centralized infrastructure, and peer connections to other clients. We now discuss how these connections are managed.

A server connection offers a way for Legion clients to interact with the centralized infrastructure. We have defined an abstract connection that must be instantiated by the adapters that provide access to the centralized services. The connection for the Legion Node.js server uses websockets. For the GDriveRT adapter, connections are established and authenticated for each container (a document in GDriveRT). Independently of the employed centralized component, server connections are only kept open by active clients (Section 3.2.2).

A peer connection implements a WebRTC connection between two clients. To create these connections, clients have to execute the signaling protocol, an exchange of information necessary to circumvent firewalls or NAT boxes using STUN/TURN servers.[1] This requires using some method to propagate messages before connections are established among two (future) peers.

Legion uses the centralized infrastructure for supporting the signaling protocol between a client joining the system and its initial overlay neighbours (that have to be active clients – those with active server connections). After a client establishes its initial peer connections, it starts to use its overlay neighbours to find new peers. In this case, the signaling protocol required to establish these new peer connections is executed through the overlay network directly among client devices. If a client becomes isolated and needs to rejoin the overlay, it relies again on the help of the centralized infrastructure. This greatly simplifies fault handling at the overlay management level.

The signaling adapter for GDriveRT stores information on a hidden document associated with the main (data) document. Alternatively, clients can use the Legion native Node.js signaling server while using GDriveRT documents only for durability.

3.1.2 Object store

The object store maintains local replicas of shared objects, with related objects grouped in containers. Client applications interact by modifying these shared objects. Legion offers an API that enables an application to create and access objects.

3.1.2.1 CRDT library

Legion provides an extensible library of data types, which are internally encoded as CRDTs. Objects are exposed to the application through object handlers that hide the internal CRDT representation.

The CRDT library supports the following data types: Counters, Registers, Strings, Lists, Sets, and Maps. Our library by default uses Δ-based CRDTs [2], which are very flexible, allowing replicas to synchronize by using Deltas (Δ) with the effects of one or more operations, or the full state. These were specially designed to allow efficient synchronization in epidemic settings, by avoiding, most of the times, a full state synchronization when two replicas connect for the first time. Each data type includes type-specific methods for querying and modifying its internal state, and generic methods to compute and integrate Deltas (i.e., updates received from the network).

The design and usage of Δ-CRDTs in Legion is detailed in the following chapter, in Section 4.3, where we evaluate their usage compared to the traditional operation and state based CRDTs.

3.1.3 Security mechanisms

Allowing clients to replicate and synchronize among them a subset of the application state offers the possibility to improve latency and lower the load on central components. However, it also

[1] Our experiments have shown that WebRTC connections can be established even among mobile devices using 3G/4G connectivity.

leads to concerns from the perspective of security, in particular regarding data privacy and integrity.

Privacy might be compromised by allowing unauthorized users to circumvent the central system component to obtain copies of data objects from other clients. Integrity can be compromised by having unauthorized users manipulate application state by propagating their operations to authorized clients.

We assume that an access control list is associated with each data container, and that clients either have full access to a container (being allowed to read and modify all data objects in the container) or no access at all. While more fine-grained access-control policies could easily be established, we find that this discussion is orthogonal to the main contributions of this work. We also assume that the centralized infrastructure is trusted and provides a secure authentication mechanism to authenticate authorized clients. Finally, Legion does not address situations where authorized clients perform malicious actions.

Considering these assumptions, and to deal with the security aspects discussed above, Legion resorts to a simple but effective mechanism that operates as follows. The centralized infrastructure generates and maintains, for each container C, a symmetric key K_C for that container. Due to the authentication mechanism with the centralized infrastructure, only clients with access to a container C can obtain K_C. Every Legion client has to access the infrastructure upon bootstrap, which is required to exchange control information required to establish direct connections to other clients. During this process, clients also obtain the key K_C for the to be accessed container C.

K_C is used by all Legion clients to encrypt the contents of all messages exchanged directly among clients for container C. This ensures that only clients that have access to the corresponding container can observe the contents and operations issued over that container (as they must have previosuly authenticated themselves towards the centralized component), addressing data privacy related challenges.

Whenever the access control list of a container is modified to remove some user, the associated symmetric key is invalidated, and a new key for that container is generated by the centralized infrastructure. Each container has an associated increasing version number for each key. This version number is attached to every message, so that clients, on reception, can verify which key was used to encrypt the message.

If a client receives a message encrypted with a different key from the one it knows, either the client or its peer have an old key. When the client has an old key – a key with assiciated version number smaller than the version number of the key used to encrypt the message – the client contacts the centralized infrastructure to obtain the new key. Otherwise, the issuer of the message has an old key: the receving client discards the received message and notifies the peer that it is using an old key. This will lead the sender of the message to connect to the central infrastructure (going again through authentication), update its local key, and re-transmit any messages that were invalidated.

To enable clients to detect when the key is updated in a timely fashion, the centralized component periodically generates a cryptographically signed message containing the current

version of the key, and a nonce (the signature is created using the asymmetric keys associated with the certificate of the server, used to support the bootstrap SSL connections). This message is sent by the server to active peers, that disseminate the message throughout the overlay network.

Note that clients that have lost their rights to access a container are unable to obtain the new key and hence, unable to modify the state of the application directly on the centralized component, send valid updates to their peers, or decrypt new updates.

While there might be a small increase in communication with the centralized infrastructure when a user's access is revoked (as a new key has to be generated and distributed), we believe that removing user permissions in collaborative web applications is not a frequent task. Furthermore, several access revocations can be compressed into a single update of the access control list as it would require only generating and distributing a single key.

Altough these mechanisms ensure access control and privacy for this setting, there are many actions malicious users can attempt. For example, active clients (those that connect to the server) can omit the key-update message from being propagated to other clients, ensuring they are not removed from the network. In Chapter 5 we addess the situations where authorized clients can perform malicious actions and explore how malicious users can attempt to circumvent the restrictions imposed by causal consistency.

3.1.4 Adapters

Adapters can be used to integrate Legion with existing systems. The parts that Legion clients use to interact with the base Legion centralized service can easily be replaced by an adapter implementing the required API. This lets applications to leverage existing web infrastructure for storing data and to assist in several functions of the framework, including peer discovery, overlay management, and authentication.

To simplify our prototyping, we have implemented the adapters each as its own stand-alone component. This enables programmers to configure which adapters should be enabled (when an adapter is disabled, the functionality is by default provided by the Legion Node.js server). Next we discuss the most relevant aspects related with the design and implementation of these adapters which cover the specific challenges that a programmer faces when integrating Legion with an existing service.

3.1.4.1 GDriveRT

We describe our effort to create adapters for Google Drive Realtime (GDriveRT), a service provided by Google for supporting collaborative web applications similar to Google Docs [38].[2] To integrate Legion with GDriveRT (see Figure 3.1), we have implemented 4 distinct adapters with the following purposes:

[2]The GDriveRT API was shutdown in September 2019 but the general outcome remains the same. Porting our adapters to work on the new Firebase Realtime Database is trivial. In Section 3.5.1 we list additional adapters created for Legion, namely overlay alternatives and for integrating with additional storage backends.

- a storage adapter enables Legion to outsource storage of application state and to support GDriveRT legacy clients (those clients accessing GDriveRT directly instead of using Legion);
- a signaling adapter enables the use of GDriveRT to support signaling for establishing WebRTC connections;
- an authentication and key management adapter enables Legion to outsource to GDriveRT both user authentication and key management and distribution;
- a service API adapter that exposes to client applications an interface similar to the GDriveRT API.

Data model Our GDriveRT storage adapter supports the same data model as GDriveRT, in which collaboration among users is performed at the level of documents. A document contains a set of data objects and is mapped to a Legion container. Each object inside a document is mapped to an object of a similar type in Legion. The adapter transparently performs this mapping.

The associated service API adapter provides applications with an API similar to the GDriveRT API. The main functions of this API include a method to load a document, which in our case, initializes the Legion framework. This method gives access to a handler for the document, which can be used by the application to read and modify the data objects included in the document state.

By exposing the same API of GDriveRT, this adapter enables any web application written in JavaScript that uses the GDriveRT API to be (easily) ported to Legion through the manipulation of a few lines of JavaScript code. A developer has only to:

- add an include statement to the script file with the code of Legion (line 5 and 6 of Listing 3.1, the former to include Legion and the latter to include the GDriveRT adapter);
- replace the function call to load a document by the equivalent function of the Legion GDriveRT service API adapter (replace line 2 with line 4 of Listing 3.2).

Listing 3.2 details how both variants (Legion and GDriveRT) can be used. To initialize and use each respective framework (obtain the handler, `doc` or `model`, for a document) for an application made for the GDriveRT API, the following is required:

- initialize the framework (lines 2 or 4, respectively for GDriveRT and Legion);
- a call to `createRealtimeFile` (line 7) to create the file (if the file did not exist), which calls `onFileCreate` on success (line 10);
- a call to `load` (line 8) to locally load the file, which calls `onFileLoaded` (line 18) if the document was previously setup or `onFileInitialize` (line 14) on the first client which is responsible to ensure all objects are correctly initialized;

```
1  // GDriveRT
2  <script src="apis.google.com/js/api.js"></script>
3  <script src="www.gstatic.com/realtime/client-utils.js"></script>
4  // Legion
5  <script src="legion.js"></script>
6  <script src="legion-adapter-GDriveRT.js"></script>
```

Listing 3.1: GDriveRT and Legion adapter import.

```
1  // GDriveRT init
2  const realtimeUtils = new utils.RealtimeUtils({clientId: CLIENT_ID});
3  // Legion init
4  const realtimeUtils = new LegionRealtimeUtils({clientId: CLIENT_ID});
5
6  // API for usage of (either) realtimeUtils:
7  realtimeUtils.createRealtimeFile("filename", onFileCreate);
8  realtimeUtils.load("id", onFileLoaded, onFileInitialize);
9
10 function onFileCreate(documentID) {
11   // call realtimeUtils.load
12 }
13
14 function onFileInitialize(model) {
15   // create lists, maps, strings using <model>
16 }
17
18 function onFileLoaded(doc) {
19   // Applications may use objects (lists, maps, strings) existing in doc,
20   // i.e., modify directly and attach handlers for remote updates.
21 }
```

Listing 3.2: Simplified GDriveRT and Legion adapter APIs for initialization and usage.

- implement `onFileInitialize(model)` (line 14) – use `model` to initialize objects, for example `map = model.createMap(data)` to create a Map object and `model.set(someID, map)` to attach it to the document;
- implement `onFileLoaded(doc)` (line 18) – use `doc` to access objects for use in the application, for example `map = doc.get(someID)`.

With the handler for the document model (`model`) to setup objects initially and the loaded document (`doc`), the application can use exactly the same function calls using either GDriveRT as in the original application or when using the `realtimeUtils` generated by Legion.

Legion functionality Our Legion storage adapter can, in addition to providing a similar API, leverage the GDriveRT infrastructure to:

- serve as a gateway between partitioned overlays that replicate the same GDriveRT document (in our case, seperate groups of clients which are unable to create WebRTC connections);
- reliably store application state (i.e., durability of documents and associated objects).

For serving as a gateway between partitioned overlays, for each document, the adapter maintains in GDriveRT the list of deltas of the CRDTs of the document. As discussed before, in each overlay, a set of active clients is responsible to upload modifications executed by clients in the overlay and to download and disseminate new changes throughout the overlay. If more than one client executes this process in each overlay, this does not affect correctness, as changes received in a client are discarded if it is already reflected in the state of the replica (due to the commutativity of CRDT operations).

Support for legacy applications While Legion allows web applications to explore peer-to-peer interactions using the Legion framework, it is also possible to allow legacy client applications to continue accessing data using the original GDriveRT interface. This is done by enabling a special flag when initializing our storage adapter. Note that this support, as we show in the evaluation (specifically, in Section 3.3.2.4), incurs an overhead due the differences between both systems in encoding of state and metadata required.

When supporting legacy clients, for each data object, Legion keeps two versions: the version manipulated by all Legion replicas and the version manipulated by legacy applications. The key challenge is to keep both versions synchronized, a process executed by a Legion replica (a client).

Applying operations executed in Legion clients to the GDriveRT object is a straightforward process that requires converting the list of executed changes to the corresponding GDriveRT operations and executing them.

Applying operations executed in a GDriveRT object to the Legion object is slightly more complex because it is necessary to infer the newly executed operations. To this end, the client executing the synchronization process records the version number of the GDriveRT document in which the process is executed. In the next synchronization, the client infers the updates produced by legacy clients by comparing the state of the current version of the document against the state of the version of the last synchronization (using a diff algorithm). The updates are converted into changes that Legion's CRDTs accept, and added to the log of executed changes, which guarantees that these are applied to the Legion objects.

Both synchronization steps need to be executed by a single client to guarantee an exactly-once transfer of updates from one version to the other. We implement an election mechanism for selecting the client relying on a GDriveRT list. When no client is executing this process, a client willing to do it checks the version number of the document and the current size of the list, and then writes in the list the tuple $< id, n, t >$, with id being the client identifier, n the observed size of the list, and t the time until when the client will be executing the process (for periods in the order of seconds or minutes). The client then reads the following version of the document, which guarantees that its write has been propagated to GDriveRT servers. If the tuple the client has written is in position $n + 1$, the client is elected to execute the process. This is correct, as when two clients concurrently try to be elected, the tuple of only one will be in position $n + 1$ of the list in the new version of the document.

Security in GDriveRT When using GDriveRT, we can leverage on the existing authentication mechanism of GDriveRT to perform access control, which defines which user can access which document. The security mechanism presented previously (shared key among peers) had to be slightly adapted as to ensure compatibility with the authentication and key management adapter due to the fact that GDriveRT only provides storage. GDriveRT exposes no computational capabilities, being therefore unable to generate symmetric keys, nor generate signed messages periodically to speed up the notification of clients of key changes.

To address these challenges we made the following modifications. First, when a new container C is created, the symmetric key K_C associated with that container is created by the first Legion client that accesses the container. As clients can only access the container after being granted access by GDriveRT, the key is generated by a client with access.

Additionally, when a client removes some user's access to a container, it also generates a new key for that container. Using the GDriveRT authentication and key management adapter, clients monitor any changes to the key (to verify that the known key is still valid).

This step can be performed infrequently because, as soon as a single client becomes aware of a new key, the knowledge that a new key exists is epidemically propagated throughout the overlay network using the previous protocol – new messages will be encrypted with the new key, leading the receiving clients to fetch the new key from the centralized infrastructure. As only clients that still have access to the document are able to obtain the new key, only clients with access will be able to maintain membership in the network.

3.2 Implementation details

We now provide a few implementation details of our prototype.[3]

We have used `Count Lines of Code` [91] and verified that the code for our GDriveRT adapters has 1.768 JavaScript lines of code, while the whole implementation of Legion (including the server for the centralized component) has 4.639 JavaScript lines of code.

3.2.1 Overlay networks

To achieve the threshold of K neighbours we do the following. Upon joining the system, a client resorts to the centralized component (either the Legion server or another web service accessed through a specialized adapter) to obtain the identifiers of nodes that currently have an open connection to the centralized infrastructure. Using this information, the client establishes connections to nearby neighbours and (few) distant neighbours. For these connections the centralized infrastructure is leveraged to perform the WebRTC signaling protocol. For further connections the existing overlay is used – we apply random walks through neighbours to find other nearby or distant neighbours, to fill the required parameters of K_n and K_d.

[3]The code is available at: https://github.com/albertlinde/Legion.

To classify peers as being either nearby or distant we resort to the previously described protocol (HTTP HEAD ping – see Section 3.1.1.2). In our experiments we used 4 distinct sites through endpoints of Amazon EC2 Web API (scattered over 4 different AWS regions).

While different distance functions can be employed over the virtual coordinates associated with each client, in our prototype we use a function that categorizes a client to be distant if the difference between at least two coordinates in the 4D virtual space are equal or above 70ms, and nearby otherwise (we have experimentally asserted that this strategy yields adequate results).

3.2.2 Selection of active clients

In our design, we use a small subset of clients (active clients) to upload and download updates over objects to and from the centralized infrastructure and to monitor the cryptographic key associated with each container.

To select these clients, we use a bully algorithm [92] where initially all clients act as an active client, and periodically, every T ms, sends to its nearby overlay neighbours a message containing its unique identifier – in our experiments we set $\mathsf{T} = 7000$. Whenever a client receives a notification from a neighbour whose identifier is lower than its own, it switches its own state to become a passive client, and stops disseminating periodic announcements (effectively being bullied). To address the departure or failure of active clients, if a passive client does not receive an announcement for more than $3 \times \mathsf{T}$, it switches its own state back to become an active client (the factor of 3 is used to avoid triggering this process unnecessarily).

The result of executing this algorithm is that only a subset of non-neighbouring clients remain active clients. Passive clients disable their connection to the centralized infrastructure, leading to a reduction in server connections. As detailed in Section 3.3.2.4, a reduction in overall server load can be obtained using this method as less often the same operations have to be sent to clients – these share them directly.

3.2.3 Security

For the symmetric cryptography algorithm, we used AES operating in block cipher mode, using a key of 128 bits. We use RSA, configured with a key of 2048 bits, for generating and verifying the signature of the messages issued by our Node.js server. Our implementation resorts to the Forge [93] JavaScript library to implement all cryptographic operations.

If Legion is used as a standalone system (without any adapters such as GDriveRT), access control at the centralized server has to be implemented by the application provider as there is no service to depend on. We provide callback handlers to enable integration with any existing backend or database.

3.2.4 Networking

To circumvent firewalls and NAT boxes when establishing connections among clients a set of STUN and TURN servers need to be available (or some clients might not be able to communicate

directly).

This is a configurable aspect in our prototype, and can easily be modified to use privately owned and managed servers if an application operator desires. By default our prototype relies on Google's publicly available STUN servers and does not make use of TURN – we already rely on the centralized server to mediate interactions between not connected clients.

3.3 Evaluation

This section presents an evaluation of Legion. We showcase the operation of Legion when using the adapters to inter-operate with the GDriveRT infrastructure (except if specifically stated in our experiments, we ran Legion with the GDriveRT adapters enabled and with support for legacy clients disabled).

The evaluation mainly focusses on two aspects. We start with an analysis of our experience in adapting existing GDriveRT applications to leverage Legion. Then, we present an experimental evaluation of our prototype, comparing it to the centralized infrastructure of GDriveRT regarding the following practical aspects:

- What is the impact on update propagation latency?
- What is the impact on application performance?
- How does the system behave when the central server becomes (temporarily) unavailable?
- What is the impact of using Legion in terms of load imposed on the central component and on individual clients?
- What is the overhead for supporting seamless integration with legacy clients?

3.3.1 Designing applications

In this section, we describe a set of web applications that we have ported to Legion using the GDriveRT adapters.

Google Drive Realtime Playground The Google Drive Realtime Playground [94] is a web application showcasing all data-types supported by GDriveRT. We ported this application to Legion by changing only 2 lines in the source code (see § 3.1.4.1).

Multi-user Pacman We adapted a JavaScript version of the popular arcade game Pacman [95] to operate under the GDriveRT API with a multiplayer mode. We also added support for multiple passive observers that can watch a game in real time. In our adaptation up to 5 players can play at the same time, one player controlling Pacman (the hero) and the remaining controlling each of the four Ghosts (enemies).

The Pacman client is responsible for computing, and updating the adequate data structures, with the official position of each entity. Clients that control Ghosts only manipulate the

information regarding the direction in which they are moving. If no player controls a Ghost, its direction is determined by the original game's AI, running in the client controlling Pacman.

In this game, we employed the following data types provided by the GDriveRT API:

- a map with 5 entries, one for Pacman and the remaining for each Ghost, where each entry contains the identifier (ID) of the player controlling the character (each user generates its own random ID);
- a list of events, that is used as a log for relevant game events, which primarily includes players joining/leaving the game, a Ghost being eaten, and Pacman being captured.
- a list representing the game map, used to maintain a synchronized view of the map between all players. This list is modified, for instance, whenever a pill is eaten by Pacman;
- a map with 2 entries, one representing the width and the other the height of the map. This information is used to interpret the list that is used to encode the map;
- a map with 2 entries, one used to represent the state of the game (paused, playing, finished) and the other used to store the previous state (used to find out which state to restore to when taking the game out of pause);
- 5 maps, one for each playable character, with the information about each of these entities, for maintaining a synchronized view of their positions (this is only altered when the corresponding entity changes direction, not at every step), directions, and if a ghost is in a vulnerable state.

Along with extending and porting this application to use the GDriveRT API, we also implemented the same game (with all functionality) using Node.js as a centralized server for the game through which the clients connect using web-sockets (this implementation does not leverage Legion). This enables us to investigate the effort in implementing such an interactive application using both alternatives. The Node.js implementation of the game is approximately 2.200 LOC for the client code, and 100 LOC for the server. In contrast, the implementation leveraging the GDriveRT API has approximately 1.620 LOC for the client code, and 40 lines of code for the server-side (used to run multiple games in parallel). This shows that an API such as the one provided by GDriveRT and Legion simplifies the task of designing such interactive web applications.

Creating the Legion version (using the GDriveRT adapters) required to change only two lines of code of the GDriveRT version (as described before). From a user perspective, the Legion version runs much smoother, which is also shown by our evaluation presented further ahead.

Spreadsheet We have also explored an additional application: a collaborative spreadsheet editor. Each spreadsheet represents a grid of uniquely identifiable rows and columns, whose intersection is represented by an editable cell. Each cell can hold numbers, text, or formulas that can be edited by different users.

A prototype of the spreadsheet web application was built using AngularJS and supporting online collaboration through GDriveRT. The spreadsheet cells were modeled using a GDriveRT map. Each cell was stored in the map using its unique identifier (row-column) as key. Porting this application to the Legion API only required the change of 2 lines of code (as discussed previously).

Our experience with porting these applications to leverage Legion shows that doing so is simple, as the programmer can easily use our GDriveRT adapters. Furthermore, this shows that carefully designing our framework to expose (through adapters) APIs that are similar to existing Web infrastructures is paramount to promote easy adoption of our solutions.

3.3.2 Experimental evaluation

In our experimental evaluation, we compare Legion, with and without the use of adapters, against GDriveRT, as a representative system that uses a traditional centralized infrastructure.

In our experiments, we have deployed clients in two Amazon EC2 datacenters, located at North Virginia (us-east-1) and Oregon (us-west-2). In each DC, we run clients in 8 m3.xlarge virtual machines with 4 vCPUs of computational power and 15GB of RAM. Unless stated otherwise, clients are equally distributed over both DCs. The average round-trip time measured between two machines in the same DC is 0.3 ms and 83 ms across DCs.

3.3.2.1 Latency

To measure the latency experienced by clients for observing updates, we conduct the following experiment. Each client inserts in a shared map a key-value pair consisting of his identifier and a timestamp. When a client observes an update on this map, it adds to a second map, as a reply, another pair concatenating the originating identifier and the replier's identifier as the key, and as value an additional timestamp. When a client observes a reply to his message, it computes the round-trip time for that reply, with latency being estimated as half of that time. All clients start by writing to the first map at approximately the same time and reply to all identifiers added by other clients. Thus, this simulates a system where the load grows quadratically with the number of clients.

The results are presented in Figure 3.2 and Figure 3.3 (as empirical Cumulative Distribution Functions). Both figures present the latency observed by all clients for both Legion and GDriveRT varying the number of clients from 4 to 64.

Figure 3.2 presents the results of running all clients within the same datacenter. The results show that client-to-client latency using Legion is much lower than using GDriveRT with very

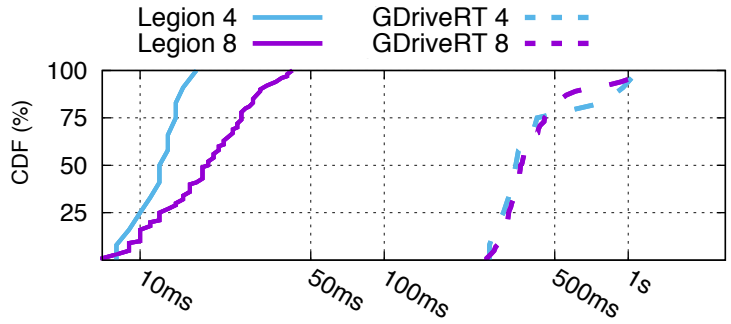


Figure 3.2: Latency for the propagation of updates: all clients within the same datacenter.

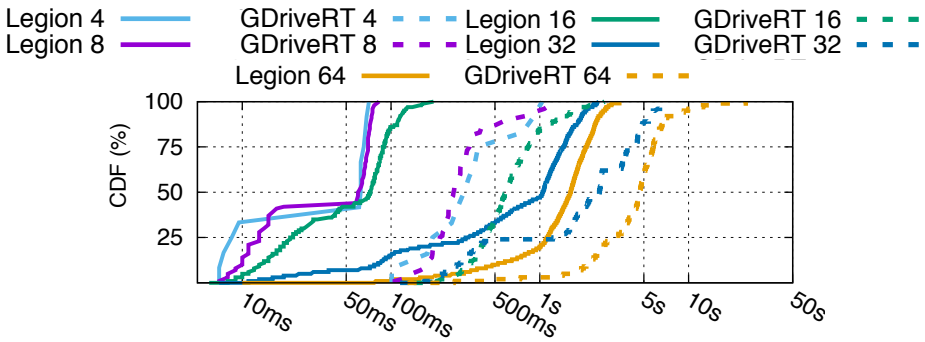


Figure 3.3: Latency for the propagation of updates: clients distributed over 2 datacenters.

close clients. The main reason for this is that the propagation of updates does not have to incur a round-trip to the central infrastructure in Legion.

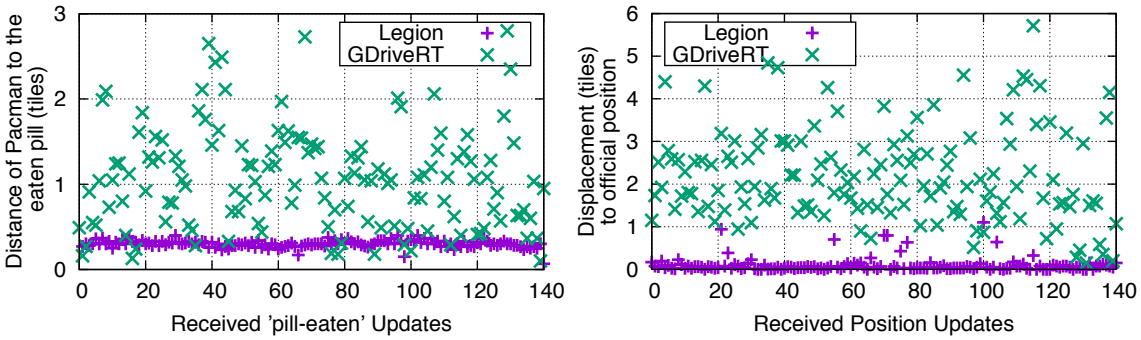
Figure 3.3 presents results when running clients across 2 datacenters. Note the very visible inflexion point in lower amounts of clients – the points is close to the 50% of updates as this is the amount of nearby clients. For 64 clients, the 95^{th} percentile for GDriveRT is almost an order of magnitude greater than Legion, suggesting that Legion's peer-to-peer architecture is better suited to handle higher loads than the centralized architecture of GDriveRT.

3.3.2.2 Multiplayer Pacman performance

We now show the impact of Legion on the performance of applications in the context of the multiplayer Pacman game.

To that end we conducted an experiment with volunteers, where we had five users playing the same game (one player controlling Pacman, and four players for each of the ghosts). This experiment was conducted using five machines, in a local area network. Machines were running Ubuntu and clients executed in Firefox.

We focus our experiments in measuring the displacement of entities in relation to their official position. As explained before, each client updates an object with the direction of its movement. The Pacman client computes and updates the official position of each entity periodically. Each client independently updates its interface based on the known direction of movement and the latest official positions (extrapolation). Displacement captures the difference between the position computed (extrapolated) by a client and the received official position upon receiving an



(a) Pacman displacement to eaten pills.

(b) Pacman and Ghosts displacement of extrapolated to actual positions.

Figure 3.4: Muti-User Pacman performance.

update. When displacement is high, users see entities jumping on the game map as these must be repositioned to their correct locations.

Figure 3.4 reports the obtained results where the displacement is measured in tiles (the square unit that forms the interface). The board size of Pacman was 19×22 tiles featuring approximately, 59 turning points. The Pacman and all Ghosts move at approximately 3.33 tiles per second. In particular we measure, at all clients controlling Ghosts:

- the displacement of Pacman in relation to an eaten pill when an update reporting a pill being eaten is delivered. Figure 3.4a shows that when using Legion, Pacman is visible by other players much closer to the eaten pill than when using the GDriveRT version of the game, meaning that the interface is much more closely updated to the real state of the system (i.e., less stale data);

- the displacement of Pacman and Ghosts when a client controlling a ghost receives an update for a position. Figure 3.4b shows that when using Legion the displacement of entities in the game interface is significantly lower when compared with the game version that only uses GDriveRT, which is unable to send updates to all clients at an adequate rate.

These results are the practical effect on application usage (in this case, game playability) of staleness observed when computing extrapolated game state. Lower propagation latency of Legion (Section 3.3.2.1) results in lower staleness and leads to a better user experience.

3.3.2.3 Effect of disconnection

We study the effect of disconnection by measuring the fraction of updates received by a client over time. In the presented results, clients share a map object, and each client executes one update per second to the map (the same results were observed with other supported objects). We simulate a disconnection from the GDriveRT servers, by blocking all traffic to all Google domains using `iptables`, 80 seconds after the experiment starts. The disconnection lasts for 100 seconds, after which rules in iptables are removed so that connections can again be re-established.

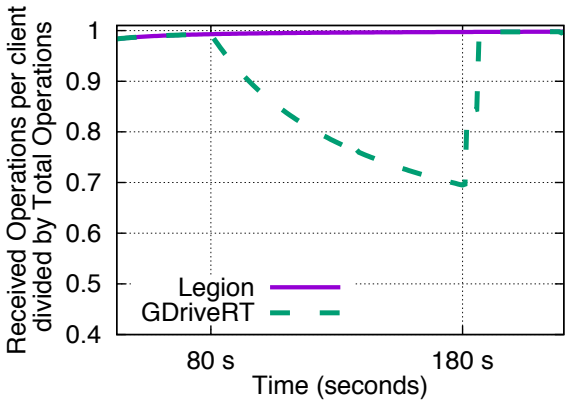


Figure 3.5: Effect of server disconnection on update propagation.

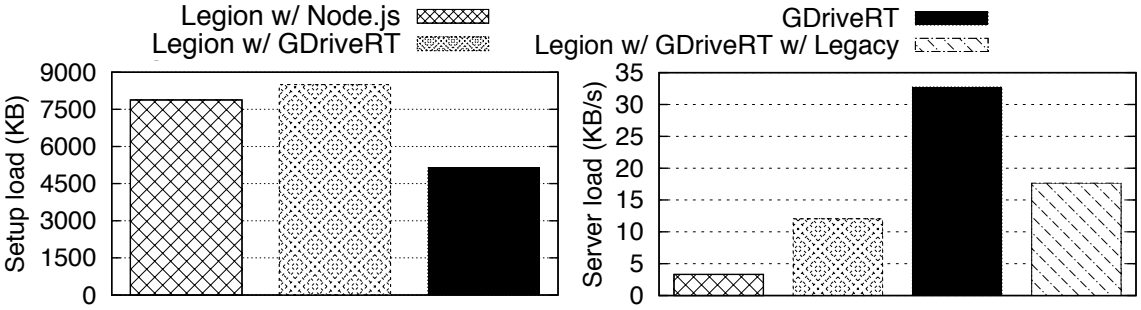
Figure 3.5 shows, at each moment, the average fraction of updates observed by clients since the start of the experiment – the average number of updates received divided by the total number of updates executed (including those of network partitioned clients). As expected, the results show that during the disconnection period, GDriveRT clients no longer receive new updates, as the fraction of updates received decreases over time. When connectivity is re-established (as `iptables` rules are removed), GDriveRT is able to recover as clients can again synchronize with the servers. With Legion, as updates are propagated in a peer-to-peer fashion, the fraction of updates received is always close to 100% as client-server-client is no longer the only path operations can take.

We note that while servers remain inaccessible, new clients cannot join the network. However, when leveraging Legion, clients that are active when the server becomes unavailable can continue operating as normal without noticing the server unavailability – nevertheless the application is notified of this disconnection and, depending on the application, can opt to inform the user of such events.

3.3.2.4 Network load

We now study the network load induced by our approach. To this end, we run experiments where 16 clients share a map object and where each client executes one update per second. The workload is as follows: 20% of updates insert a new key-value pair and 80% replace the value of an existing key selected randomly. The keys and values are strings of respectively 8 and 16 characters. We measure the network traffic by using `iptraf`, an IP network monitor. In these experiments, we used the following configurations to obtain the results presented in Figure 3.6:

- Legion w/ Node.js: uses our own Legion server as backend for both signaling and durability, also leveraging peer-to-peer connections;
- Legion w/ GDriveRT: uses GDriveRT documents as backend for durability and signaling through the adapters instead of running our own servers, while also leveraging the peer-to-peer connections provided by Legion;



(a) Server load during setup.

(b) Server load during operations.

Figure 3.6: Server network load comparing Legion to GDriveRT.

- GDriveRT: uses the unmodified GDriveRT API and original documents as backend.

Figure 3.6a shows the total aggregated network load of the setup process, which entails making the necessary connections to the infrastructure and peer-to-peer connections. The incurred load using our own backend server is due to clients requiring to use this component to connect to each other initially (the overhead from WebRTC signaling – note that all traffic, including that to STUN servers, is captured). Legion using GDriveRT as a backend has a slightly higher cost due to the overhead of performing signaling through the GDriveRT infrastructure (encoded within documents), which is less efficient than performing this through our server implementation. In both cases using Legion, only few clients obtain the initial object which is then propagated to other clients using the established peer-to-peer connections. Finally, in GDriveRT each client downloads the shared data directly from the infrastructure but as signaling is not executed (as no peer-to-peer connections exist), the overall bandwidth usage is lower.

Figure 3.6b shows the continuous network load on the server without considering the initial setup load (computed by adding the traffic of all clients to and from the centralized infrastructure) for all competing alternatives. The results show that the load imposed over the centralized component is much lower when using Legion with GDriveRT as backend than when using only the non-modified GDriveRT. This is expected, as only a few clients (active clients) interact with the GDriveRT infrastructure, being most udaptes propagated among clients using the established client-to-client connections. Interestingly, the use of our server leads to an even lower load on the centralized component, this happens because the signaling mechanism used to establish new WebRTC connections among clients and the process for replica synchronization with the server is much more efficient.

For Figure 3.6b we run an additional configuration (Legion w/ GDriveRT w/ Legacy) that not only uses GDriveRT documents as backend but also synchronizes with the original document to support legacy clients – those working with the original document without using Legion. Supporting legacy clients incurs a non negligible overhead because the mechanism used requires a large number of accesses to the centralized infrastructure as to infer which operations should be carried from legacy clients to the Legion clients and vice versa. We execute this process every 5 seconds – there is a tradeoff when using smaller intervals to reduce update propagation latency as

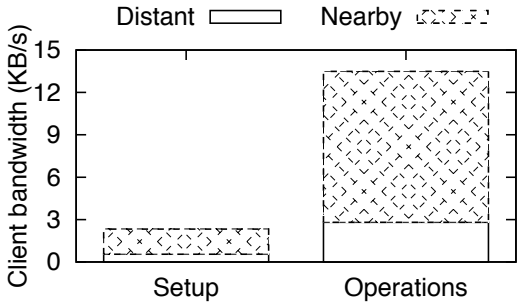


Figure 3.7: Client-to-client bandwidth usage (average).

it has a direct impact on used bandwidth. However, even with support for legacy clients enabled, Legion induces lower load on the centralized component when compared with GDriveRT.

Figure 3.7 reports the average peer-to-peer communication traffic for each client during the setup of WebRTC connections (Setup) and while clients issue and propagate operations (Operations). The results show that the traffic at each client is larger than the traffic at each client interacting with only GDriveRT (which can be approximated by dividing the server load – in Figure 3.6 – by the number of clients). This happens not only because clients use one another for signaling – which becomes negligible during Operations phase as the network is mostly stable – but because our dissemination strategy has inherent redundancy due to multiple propagation paths. In GDriveRT there are no redundant transmissions between each client and the centralized infrastructure. We note that an average of under 14KBps does not represent a significant fraction of available bandwidth nowadays for any kind of client device (even mobile devices can easily deal with multiple MBps, even when using mobile data instead of Wi-Fi).

We additionally measured the difference in network bandwidth usage between distant or nearby peers. The use of our location aware overlay leads to a network usage pattern where the amount of data sent to distant nodes is significantly lower that that sent to nearby nodes. This allows us to obtain a significant reduction in used bandwidth among clients while not having a major impact on latency. The reason for this is due to local area network latency being under one ms which allows for very fast propagation, where additional redundant propagation to far away replicas would not have a great effect on overall latency – shaving of a couple of ms from a 83 ms average (the average measured using `ping` during our experiments) is not significant considering the cost (additional network load due to redundancy).

3.4 Related work

This work has been influenced by prior research in multiple areas:

Internet services Internet services often run in cloud infrastructures composed by multiple data centers, and rely on a geo-replicated storage system to store application data [46, 60, 67, 96]. Storage systems often adopt strong consistency models, such as parallel snapshot isolation [45] and linearizability [46], where concurrent (conflicting) updates are not allowed without some

form of coordination. In our context the algorithms used to coordinate access to data storage for executing each update are prohibitively expensive for high throughput and large numbers of clients (manipulating the same set of data objects).

In contrast, some storage systems provide weaker consistency models allowing for high-availability under network partitions, such as eventual consistency [48] and causal consistency [62, 96–98] – this allows for clients to apply updates on different replicas concurrently and without coordination. In our case, to provide causal consistency, we must adapt to a setting where we can have a very large number of replicas writing on the data (replicas at each client and propagating changes in a peer-to-peer manner) – in this setting replica failures and reconnections are very common, which the referenced works do not address efficiently.

Collaborative applications Several specialized applications support collaboration across the Internet by maintaining replicas of shared data in client machines, such as Etherpad [24] which allows clients to collaboratively edit text-documents. Google Drive Realtime [38] and ShareJS [37] are generic frameworks that manage data sharing among multiple clients – these allow to create applications for realtime collaboration on shared objects. Similar to these systems, Legion keeps replicas locally but allows both for continued operation even when disconnected and for client-side replicas to synchronize directly.

The referenced works [24, 37, 38] rely on centralized infrastructure to mediate interactions among clients and internally use operational transformation [52, 53] for guaranteeing eventual convergence of replicas. In contrast, our work relies on CRDTs [59] for guaranteeing data convergence while allowing clients to synchronize directly among them. Collab [99] is an example allowing for peer-to-peer but relies on browser plugins installed by the user to allow clients to synchronize among each other, while being limited to connections on the same local network. Our work uses standardized and widely supported techniques for supporting collaboration over the Internet, requiring no installation by the end user, and allowing interaction with existing Internet services.[4]

Replication at clients While many web applications are stateless, fetching data from servers whenever necessary, a number of applications cache data on the client for providing fast response times and support for disconnected operation. For example, Facebook supports offline feed access [68] and Google Maps may be used offline as long as maps are locally cached – such applications cache state from the server and do not allow the user to update state. In contrast, the previosuly discussed collaborative systems mostly allow for local replicas which may be edited.

Several systems that replicate data in client machines have been proposed in the past. In the context of mobile computing [69], systems such as Coda [51] and Rover [71] support disconnected operation relying on weak consistency models. Parse [70], SwiftCloud [65] and Simba [66] are recent systems that allow applications to access and modify data during periods of disconnection. While Parse provides only an eventual consistency model, SwiftCloud additionally supports

[4] From http://iswebrtcreadyyet.com, all major browsers support Legion's requirements when this was written.

highly available transactions [72] and enforces causal consistency. Simba allows applications to select the level of observed consistency: eventual, causal, or serializability. In contrast to these systems, our works allows clients to synchronize directly with each other, thus reducing latency of update propagation and allowing collaboration when disconnected from servers.

Bayou [64] and Cimbiosys [73] are systems where clients hold data replicas and that exploit decentralized synchronization strategies (either among servers [64] or clients [73]). Although our work shares some of the goals and design decisions with these systems, we additionally focus on the integration with existing Internet services. This poses new challenges regarding the techniques that can be used to manage replicated data and the interaction with legacy clients, namely because most of these services can only act as storage layers (i.e., they do not support performing arbitrary computations).

Peer-to-Peer systems and Fog Computing Extensive research on decentralized unstructured overlay networks [84–86] and gossip-based multicast protocols [81, 86] have been produced in the past (as detailed in Section 2.4). Although our design for supporting peer-to-peer communication among clients builds on previous designs, so far it mostly differs in the way we promote low latency links among clients and leverage the centralized infrastructure to effectively handle faults. Additionally, our system effectively uses the peer-to-peer network to increase server capacity (Section 3.2.2).

Fog Computing [100, 101], a variant of cloud computing where the cloud is divided into smaller cloud infrastructures located in the user vicinity, is a close topic to our research. Most research on Fog Computing approaches Internet of Things as their use case (i.e., networks of sensors and actuators, often wireless) – the aspects not explored in our work, such as device heterogeneity and battery management, can easily be adapted. Nevertheless, many of these aspects go hand-in-hand with our work and can be applied together to acomplish the same goals we aim for – for example, bringing 'cloudlets' closer to end users can drastically reduce latency for security related aspects, as we will discuss in Chapter 5.

3.5 Final remarks

In this chapter we presented the design of Legion, a framework that materializes the proposed cloud-edge hybrid model, allowing the development of web applications with seamless support for replication at the client's device leveraging peer-to-peer interactions to propagate operations among clients.

Furthermore, we presented the design of adapters that enable Legion to leverage existing internet services. We showcase an implementation of adapters for Google Drive Realtime (GDriveRT), namely to provide: a) a storage backend; b) WebRTC signaling; c) authentication and key management; d) exposing an API akin to that of GDriveRT; e) a mechanism to support the co-existence of legacy clients.

The evaluation of our prototype shows that latency for update propagation is much lower using Legion when compared with the use of GDriveRT. We show the impact this can have on

an application usability, using a multiplayer Pacman game. Furthermore, load to the centralized infrastructure can be greatly reduced by leveraging peer-to-peer interactions. Finally, we show that clients are able to interact while servers are temporarily unavailable.

3.5.1 Derivative works

As detailed in Section 3.1.4, Legion supports adapters which allow for the usage of previously existing infrastructure or for modifications to the internal behaviour of Legion. Additionally, Legion's CRDT library can trivially be expanded with additional implementations. This modular approach of Legion allowed for the following works.

Networking Adapters to change the default networking of Legion, either by implementing different overlay networks (1, 2, and 3) or by modifying client-server connection frequency (4).

1. Rafael Seara (2015/2016) – Research program for bachelor students – Structured peer-to-peer overlays (Chord) on Legion;
2. Francisco Magalhães (2015/2016) – Research program for bachelor students – Unstructured peer-to-peer overlays (HyParView) on Legion;
3. Frederico Aleixo (2019/2020) – Research program for bachelor students – Location aware peer-to-peer networks and an initial approach to partial replication on Legion;
4. Filipe Luis (2017/2018) – MCs Thesis – Coordination of clients to lessen server load (e.g., flash-crowds) using Legion's peer-to-peer networks.

Integrating with other backend systems Adapters to replace or work in addition to Legion and GDriveRT's storage adapaters.

1. André Rijo (2015/2016) – Research program for bachelor students – Integrating Legion with Redis;
2. Pedro Durães (2016/2017) – MCs Thesis – Integrating Legion with AntidoteDB;
3. João Martins (2019/2020) – Research program for bachelor students – Integrating Legion with PotionDB.

Other Implementation of Legion subsystems in Java for usage in mobile, including also composable CRDTs (1). Adapters for both storage and networking to allow for storage and dissemination of static objects (2). CRDT implementations with access control, including changes to the data propagation adapters to apply additional security checks (3).

1. Sara Simões (2018/2019) – Research program for bachelor students – WebRTC between mobile and browsers and designing composable CRDTs in java, integrating with Legion's JavaScript versions;

2. Francisco Fernandes (2017/2018) – MCs Thesis – Caching built on client-side peer-to-peer replication (Legion);

3. Tiago Costa (2016/2017) – MCs Thesis – Secure eventual consistency between Legion client replicas.

Chapter 4

Client-side replication

Client-to-client connections allow for disconnected operation from the server and increase system scalabilty as clients can coordinate directly reducing server load. In Chapter 3 we put this idea in practice and obtained experimental results (the Legion system).

In particular, client-side replicas allow for o-latency local operations and low latency for interacting with other nearby replicas, as we allow for peer-to-peer replica synchronization among client devices. This chapter explores highly-available client-side replication.

One important aspect of such a system is managing consistency of the data. Consider applications which use cloud infrastructure backends. These may use geo-replication for providing high availability and low latency to clients and, to be able to continue operating during network partitions, these systems must adopt weakly consistent data replication protocols [43]. Such protocols allow replicas to be modified concurrently, requiring some reconciliation mechanism to merge concurrent updates.

Our system model, based on client-side replicas, must also somehow deal with the CAP theorem. We expect network partitions to be frequent, possibly over extended periods of time, and highly irregular in terms of which (groups of) replicas are affected. Additionally, a major aspect is that the client-side replicas must be able to act on their own, without coordination with other replicas being necessary to apply changes to data. In this chapter we explore causal consistency, which can allow for high availability under the presence of network partitions.

Causal consistency can be described, at a high level, as enforcing clients to always observe a state that respects happens before relationships among operations [49]. Causal consistency is an important consistency model as it was proved that it provides the basis for the strongest semantics that do not compromise both availability and convergence [102, 103].

We start the chapter by discussing causal consistency (Section 4.1) with a focus on how it is typically implemented (Section 4.1.1) and its associated costs (Section 4.1.2).

We then explore conflict-free replicated data types (CRDTs) (Section 4.2) and detail how we designed (Section 4.3) and evaluated (Section 4.3.1) Δ-CRDTs to be used with Legion (Section 4.3.2).

Note that this chapter has no specific focus on partial replication, which is explored in Chapter 6. Additionally, managing user (mis) behaviour under weak consistency is explored in Chapter 5.

4.1 Causal consistency

Causal consistency is a consistency model that can be described, at a high level, as enforcing all replicas to always observe a state that respects the happens before relationships among operations [49]. Considering any two operations o_1 and o_2 such that $o_1 \prec o_2$, where $\prec$ is the partial order that encodes the happens before relationship, causal consistency forbids any replica to observe the effects of o_2 without having already observed the effects of o_1. We say that an operation o_1 happened before operation o_2, $o_1 \prec o_2$, iff o_2 was generated in some replica r while o_1 had already been executed in r. Furthermore, we say that $o_1 \prec o_3$ if there exists an operation o_2 such that $o_1 \prec o_2$ and $o_2 \prec o_3$.

Many algorithms have been proposed to enforce causal consistency (or implement causal dissemination) [43, 44, 49, 50, 58, 62, 64, 66, 96, 97, 102–130].

Causal+ [96, 102] consistency is an extension to causal consistency. A system providing causal+ consistency not only respects the causal dependencies between operations, but also determines that a conflict handling component must be present to ensure that replicas converge. Conflicting updates to a data item are dealt with in such a manner that clients see a causally-correct, conflict free, and always progressing data state.

As we see later, Legion provides Causal+ consistency (Section 4.3.2).

4.1.1 On version vectors and direct dependencies

To be able to characterise dependencies among operations, the overall metadata size depends on the number and connectivity of the replicas in the system [114] – we refer the reader to [131] for a general discussion on the costs and benefits of different methods to track causality. As characterising causality is often required, two of the most popular techniques to implement causal consistency consist in using version vectors [50, 104, 105, 108] and direct dependency graphs [109, 117].

When using version vectors, the dependencies of each operation are summarized in a vector that states which operations have happened before the operation was generated – each position in the vector is assigned to a replica which allows it to encode which operation has been included from that replica. Using direct dependencies, each operation instead includes only the information (such as identifiers) on the concurrent operations that have been executed before their generation. As dependencies are transitive, it is possible to build the complete dependency graph of operations using only direct dependencies.

Figure 4.1 depicts how both version vectors and direct dependencies can be assigned to operations. Mattern [108] presents a detailed exploration of the happens before relation, realtime, and vector-time (version vectors) and Baquero and Preguiça [132] has a more recent discussion on compressing causal histories.

Notice that the choice of technique used has a direct impact in the metadata size of an operation. For example, if the number of replicas is very high then using version vectors becomes impractical, where direct dependencies would allow for less metadata usage. Direct dependencies

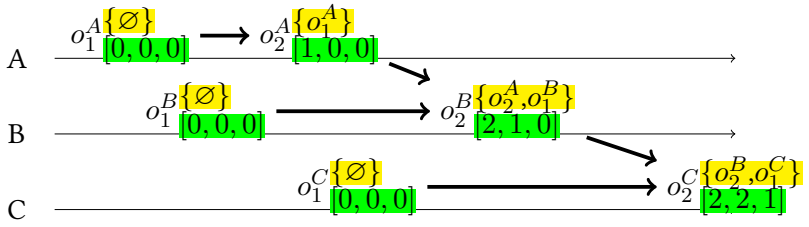


Figure 4.1: Common techniques to implement causal consistency using version vectors [50, 104, 105, 108] and direct dependency graphs [109, 117]. o_N^R B A means operation number N from replica R would be created with either version vector A or direct dependencies B. For example, the operation o_2^C may be executed at a replica when the replica's local clock is greater or equal to [2,2,1], meaning it executed at least two operations from both replicas A and B, and one operation of replica C. When using direct dependencies, o_2^C may execute as soon as both o_2^B and o_1^C were executed. The condition for execution is effectively the same, only represented differently.

are also easier to reason about when considering dynamic membership changes – as the associated metadata has no specific structure. As a counter-point, a version vector does summarize all of the dependencies, where with direct dependencies the causal graph has to be traversed to obtain the same information.

4.1.2 On characterising versus providing causal consistency

In 1991, Charron-Bost [114] proved that to provide the ability to check for concurrency in causal consistency (i.e., characterising causality or verifying causal independence of operations), the necessary metadata attached to any operation is on the order of the amount of replicas which can apply writes to the system state, assuming all replicas can synchronize pairwise. An intuitive example is that in a system with N replicas, N is the maximum amount of concurrently created operations, without dependencies among each other, that have to be tracked at any given time.

This result may lead to the impression that causal consistency is intrinsically costly and non scalable. – the algorithms proposed to enforce causal consistency typically associate with each operation metadata, which is used to guarantee that an operation is not executed if its execution would break causality.

Interestingly, if the system simply aims to provide causal consistency, being able to characterise causality is not at all important, as long as operations are delivered respecting their causal order. Enforcing causal consistency is equivalent to enforcing that operations are delivered (and executed) and that delivery (or execution) respects the causal order across all replicas.

In a system that aims to provide causal consistency, a property that typically also needs to be enforced is reliability: guaranteeing that every operation submitted is eventually executed in every replica. To enforce reliability, algorithms also need to use some metadata.

4.1.2.1 Centralized networks

A simple way to enforce causal consistency is to use a central or specialized replica for propagating operations – this approach is used, for example, in CVS [133] and subversion [134]. A simple

algorithm follows.

Every replica has a FIFO channel with the central replica and every operation generated at each replica is propagated to the central replica using the FIFO communication channel. The central replica receives the operations from each channel in order, and adds it to the outgoing queue of every other replica atomically. Operations are propagated to replicas asynchronously using their respective FIFO channels.

This algorithm enforces causal consistency because: (i) any two operations generated at a replica execute in the same order across all replicas, as they are propagated through FIFO channels and processed in the order in which they are received; and (ii) when an operation is generated at a replica, all operations which are locally known (the happens before operations) are already in the outgoing queues to the other replicas, thus guaranteeing that they will be executed before the operation that is currently being generated.

Note that, in a run without faults, no metadata is necessary at all to enforce causal consistency. Considering that replicas and channels can fail, we would need some metadata to guarantee causal consistency. Independently of the failure recovery algorithm, it seems clear that the recovery process would need to determine if a given operation had already been propagated or not, for which it would need that each operation can be identified with some unique identifier. Lamport clocks [49] can be used to create unique identifiers – for example, composed of the pair (`replica:timestamp`) – which would allow to enforce reliability. Recovering from faults can then be performed as follows.

For faults in communication channels, when creating a replacement channel, replicas start by exchanging the identifier of the last message they have received from the remote replica. Each replica resumes sending messages in the queue for the remote replica starting with the message following the one that the remote replica sends initially. Note that some care has to be given to remove messages from the queue – messages may be removed when they are acknowledged by the remote replica.

For faults in replicas, in a crash-recovery model, when a replica recovers with its previous state, it only needs to resume the propagation of channels by executing the previous channel recovery process for every channel it has. For recovering from a definite fault of the central replica, each replica can replay its log – when receiving an operation, all replicas (including the central replica) discard operations they have already received (duplicate checking is trivial with uniquely identified operations).

These mechanisms allow for causal consistency to be enforced without any additional metadata over that already required to enforce reliability.

When reasoning why using a central replica is sufficient to enforce causal consistency, we can conclude that it is due to the fact that when an operation goes through the central replica, all of its dependencies had already been propagated by that central replica to all other communication channels (or added to the respective queues). We can extend this idea and, instead of using a central replica connected to every other replica for propagating operations, use a dissemination

tree connected by FIFO channels.[1]

Similarly as the previous algorithm, each replica receives operations from each of its channels in order. When a replica receives an operation, it atomically both delivers the operation locally and puts it in the outgoing queues of every other channel. Additionally, the creation of a new local operation leads to it being atomically added to the outgoing queue of every channel which that replica has.

This guarantees that when an operation is added to a channel's queue, all of its dependencies have already been propagated through the channel (in one or the other direction) or are queued ahead of that operation in the channel. This approach is used by Saturn [62] to enforce causal consistency in partially replicated databases, by propagating operations through all channels that will reach replicas interested in the operations.[2]

In this algorithm, again, there is no need for any metadata to enforce causal consistency in a run without faults – the way messages are propagated guarantees that they will be received in causal order. Recovering from faults would be more complex than in the central replica scenario, but the same techniques as before can be adapted, while relying on Lamport clocks for uniquely identifying operations.

Given this, we can conclude that no additional metadata overhead is necessarily imposed to provide causal consistency when already enforcing reliability – we only require messages to be sent and received in a specific order.

The next step is reasoning on system models without specialized replicas or fixed network topologies. Ideally, causal consistency is provided in any graph the network forms among replicas.

4.1.2.2 Decentralized networks

We now consider the more general case where any pair of replicas can communicate with each other to propagate operations. Two classical approaches are used to enforce causal consistency in this setting.

In the first, proposed by Lamport [49], operations are tagged using a Lamport clock and an operation can only be executed after it is known that there is no operation to be received with a smaller Lamport clock. This knowledge is called stability – an operation is deemed stable once is it globally safe to be executed and no causality violations would occur.

The approach based on every replica communicating with every other replica [49, 115] is as follows. If every replica communicates with every other replica directly, and replicas propagate local operations in order, when a replica r_1 receives an operation o with clock t from r_2, it knows that it has already received all operations which could be causal predecessors with clocks smaller than t from r_2. A replica r_1 can execute operation with clock t from replica r_2 after it has an operation with clock larger or equal to t from all other replicas (with operations executed in clock order). This approach does not need any specific information to enforce causal consistency, but it requires every replica to communicate with every other replica to execute the stability process –

[1] The overhead of maintaining a tree, possibly per partition, is orthogonal to the metadata cost of reliability.

[2] Saturn actually only propagates unique identifiers of the operations through the specialized channels, and propagates operations directly among replicas, separating operation data and metadata.

this is not only costly and time-consuming, but impractical in large or unreliable networks where not every replica can communicate directly with every other replica.

To not require all replicas to communicate to all other replicas, one approach is to use vector clocks, where every operation includes a vector clock that records the exact operations an operation depends on [50, 105–110, 112–114]. When receiving an operation, a replica can locally verify if all dependencies are satisfied and, if not, it knows exactly which operations are missing. When compared with the previous approach, this trades having specific metadata to enforce causal consistency for being faster in determining when it is safe to execute an operation.

Direct dependencies [96, 109, 117] can be used as a compressed history instead of using version vectors which can become large when multiple replicas are able to create operations. Each operation is tagged with its direct dependencies – the last operations to have been locally applied which are concurrent among each other. Attaching direct dependencies to operations allows for the same guarantees, but requires replicas to keep the causal graph in memory for efficient recovery.

Operations list We now show that it is possible to avoid both executing a complex stability processes or having additional metadata to enforce causal consistency. The only metadata cost, or overhead from the algorithm, is the one already required to provide reliable delivery.

We start by presenting a non-optimized version of an algorithm similar to the CBCAST protocol (proposed by Birman and Joseph [106]) and then discuss possible ways to optimize it.

In our algorithm, every replica keeps an ordered list of operations it has previously executed. The key idea is that the list of operations maintained in each replica respects causality, i.e., all dependencies of an operation o appear before o in the list.

When an operation is created at a replica, the operation is appended to the list. This maintains the list causally ordered with respect to locally created operations.

One replica communicates with any other replica by sending it the full ordered list of operations. When receiving a list of operations from a remote replica, the replica iterates through the list in order and for each operation, if it is not in the local list, it appends the operation to the local list. The local list thus remains causally ordered as, when an operation is added to the local list, all operations which appear before in the received list are already in the local list. Thus, all dependencies of newly applied operations are always satisfied.

For executing this algorithm, it is only necessary to be able to check if an operation is already in a list. As before, we only need to assign an unique identifier to each operation – which would be also necessary to enforce reliable delivery of operations.

This algorithm trivially tolerates network faults (albeit very inefficiently) and also allows replicas to recover from failures if operation lists are kept in durable storage.

Thus, causal consistency is adding a grand total of zero additional metadata over that required for reliability. This algorithm has another interesting property: it is possible to remove or add new replicas to the network, at any moment and in a decentralized way, which is not at all the case for algorithms that need to execute a stability process.

Minimizing the list Although from the theoretical point of view the previous algorithm has interesting properties, propagating every single operation in every communication step is not acceptable in practice. It is clear that when a replica propagates its full list of operations to a remote replica, it actually only needs to propagate the operations that are still not known by the remote replica (as already known operations are ignored when they are received). Several techniques can be used to minimize the operations to be sent.

First, when a replica sends operations to a remote replica, and the reception is acknowledged, it can locally record that information – for each remote replica, it would suffice to maintain the last position of the local list that was acknowledged remotely. Thus, each replica will send each operation only once to a remote replica (in the absence of failures). Second, when a replica receives an operation from a remote replica, if it already knows the operation, it can also record that the remote replica already has that operation.

Previous works with similar mechanisms [106, 110] require every replica to store information about every other replica. Global knowledge about every replica is impractical and a solution which permits any pair of replicas to efficiently communicate is preferred. Our algorithm only keeps such information for every currently connected replica, but still needs to efficiently handle new connections (or recover from failures).

Alternatively, when two replicas synchronize, they can start by propagating a summary of locally known operations – e.g., propagating checksums of the ordered list as in anti-entropy epidemic communication [135]. This last optimization is typically more interesting when synchronising with a replica for the first time (or after a long period without direct communication).

Note that large amounts of operations, if uniquely idenfitied with a `(replica, counter)` pair, with counter starting at 1 and incrementing with each generated operation, can efficiently be summarized into version vectors. Version vectors can then trivially be used to verify from which position in the list the operations have to be propagated, while possibly skipping those operations already present in the vector.[3]

As we discuss further ahead, Legion uses version vectors to synchronize when stablishing new connections. Detailed in Section 4.3.2, these vectorts are used to ensure replicas synchronize objects efficiently among them and afterwards, using the established connection, propagate individual operations without metadata overhead.

In systems with a large number of replicas, it is also possible to use mechanisms to minimize the size of vectors transmitted [112, 116, 136], reducing the overhead of initially sending the vector. We note that if we want to enforce reliability in the same setting, similar techniques must be used.

Thus, when aiming to enforce reliability, and by only carefully deciding the order in which operations are propagated, we can enforce causal consistency without any additional metadata overhead. This is a very important result as, when reasoning on client-side replication, any associated costs that scale up with the order of replicas in the system has a major impact – the

[3]Some care has to be taken into which data-structures are used to ensure computational complexity remains low, but that discussion is orthogonal to the point we wish to make.

amount of client-side replicas is multiple orders of magnitude higher than that of server-side replicas.

4.2 Conflict-free replicated data types (CRDTs)

When reasoning about causal consistency, one aspect that has a major impact on applications is how concurrent and conflicting operations are dealt with. One simple example is a shared map, where two replicas concurrently write a different value to the same key. When aiming to provide causal+ consistency, the system must guarantee that replicas converge in the presence of such concurrent and conflicting writes.

There are many ways to handle conflicting writes emitted concurrently at remote replicas. Some conflict resolution techniques require replicas to be instrumented with merge procedures (Bayou [64] and Coda [51]), or alternatively, require replicas to expose diverging states to the client application which then reconciles and writes a new value (Dynamo [48] and SwiftCloud [65]). The programmer must decide what to do when conflicts arise, for example, in an e-commerce application, concurrent updates to a shopping cart can be merged into a single unified shopping cart.

Conflict-free replicated data types (CRDTs) have been proposed as an approach for providing general purpose replicated data types that guarantee eventual convergence leveraging commutativity [31, 58, 59]. For example, a counter CRDT converges because the increment and decrement operations commute. No coordination is required to ensure convergence and thus updates always execute locally and immediately, un-affected by network latency, faults, or disconnections. CRDTs can typically be divided in two classes: state-based and operation-based, where both guarantee to eventually converge (when all updates are received by all participating replicas).

CvRDTs, state-based Convergent Replicated Data Types, are CRDTs where replicas synchronize by exchanging their state. This approach requires only eventual communication between pairs of replicas. The successive states of an object should form a monotonic semi-lattice and replicas merge state by computing the least upper bound.[4] Replicas exchange their full local state (including metadata) when synchronizing with any replica. This is inefficient when the size of these data objects grow significantly (for instance, in a large Set, the full Set needs to be propagated whenever a single element is added).

CmRDTs, operation-based Commutative Replicated Data Types, are CRDTs where replicas synchronize by exchanging operations. This approach requires a reliable broadcast communication mechanism to provide causal delivery for commutative operations. Operations are commutative resulting in a single unified state when all operations are executed at every replica, independent of execution order. Instead of exchanging the full state, replicas only propagate among them the operations that mutate their state. As operations have to be propagated respecting the causality of operations, which not only introduces additional

[4] A monotonic semi-lattice is graph or partially ordered set of states which are monotonically increasing.

overhead (to keep track of causality) this also fits poorly in scenarios where there are large number of replicas, especially if communication patterns among these replicas are highly dynamic, for instance, due to poor connectivity among these replicas.

Both classes (state and operation based) have large associated costs that, again, scale up with the order of replicas in the system.

An alternative, named δ-CRDTs [137], has been proposed as a middle ground between the two approaches to try to mitigate the overhead introduced by the tracking of causality. δ-CRDTs assume that communication is mostly pairwise, with each replica maintaining a communication buffer for each of its peers where it stores all operations that have not been propagated to (and acknowledged by) the remote peer. These buffers are used to compress multiple operations into a single delta (δ), enforcing FIFO communication semantics between each pair of replicas. Whenever a new synchronization path is established between two replicas, the whole state of both replicas has to be synchronized by resorting to a mechanism similar to that employed in state-based CRDTs. Thus, this approach works well, although only in settings with continuous and static synchronization patterns among replicas.

When communication patterns are highly dynamic the existing designs of CRDTs incur in excessive communication overhead. This is because originally neither of CvRDTs, CmRDTs, nor δ-CRDTs were designed to cope with our system model: client-side replicas with possibly high network churn, leading to inefficient CRDT usage.

This is clearly an issue that comes from the system model – client-side replicas come in large numbers and their communication patterns are highly dynamic. Thus, the existing classes of CRDTs provide the required guarantees (causal+ consistency), but are not suitable (efficient) for our system model.

4.3 Δ-CRDTs

We proposed a new design for CRDTs which we call Δ-CRDT, and experimentally show that under dynamic communication patterns this design achieves better network utilization than the existing alternatives.

Δ-CRDTs were specifically designed to support dynamic communication patterns among a potentially large number of replicas, and removes the assumption that pairs of replicas are continuously communicating to synchronize their state. Additionally, Δ-CRDTs do not resort to specialized pairwise communication buffers, minimizing the space overhead imposed over each individual replica. Instead, we use the internal CRDT metadata to compute a minimal Delta that needs to be propagated to a remote replica, based on a causal context (usually, a vector clock) that replicas exchange.[5] Due to this, Δ-CRDTs are well suited to be used in decentralized dissemination protocols, such as gossip protocols.

Practical use of CRDTs shows that they can become inefficient over time. For example, many CRDT designs that aim to provide collections, such as Lists or Maps, accumulate tombstones

[5] Riak support for big sets uses a similar idea for efficiently identifying removed elements [138].

leading to internal data structures becoming unbalanced [59]. To mitigate this issue garbage collection can be performed using a weak form of synchronization, outside of the critical path of client-level operations.

Δ-CRDTs also maintain additional metadata – the tombstones of removed elements of collections (e.g., Lists, Maps, and Sets). Contrary to the alternatives, in Δ-CRDTs, this metadata can be garbage collected locally at any time – and thus the local storage overhead imposed on replicas can be cleared without any coordination – at the price of being unable to synchronize by sending only a Delta when the garbage-collected information is needed for computing the Delta. If this happens, the full state needs to be exchanged (as is always the case when starting a new connection in δ-CRDTs).

Δ-CRDTs are replicated by propagating a Delta (Δ) of the current state that is missing the peer replica. To compute the Δ, a `getDelta` function is called with the causal context of the replica which initiated the communication (which might be missing updates). This causal context can be sent by a requesting replica (pull model) or, when local operations are performed, sent to other replicas (push model).

Figure 4.2 shows how Δ-CRDTs can be used to create a synchronization protocol. A replica r_1 receives a causal context (typically a version vector) from replica r_2 and computes a Δ that is to be shipped back. A replica r_1 can receive a version vector from replica r_2 where there is no *is newer than* relationship between the received and its own context (i.e., the relationship is bidirectional). This means that both replicas r_1 and r_2 have (concurrent) operations that the other has not yet seen, and thus both a Δ and a causal context have to be shipped (as to ensure replica r_2 also computes and sends a Δ back to replica r_1). In the case that one replica is strictly newer than another an empty Δ is computed and instead only a causal context has to be shipped (in order to fetch missing operations).

To create a Δ-CRDT, the following methods have to be implemented:

- a `delta` function must be implemented to be able to compute a Δ from a given point in time (from a causal history, typically in the form of a version vector);
- an `applyDelta` function must be implemented which applies a given delta to the current state.

For some data types, implementing these functions might require to store significant amounts of metadata. Hence these functions should be carefully crafted to avoid such pitfalls.

In the case of container like data types, such as Sets and Maps, CRDTs typically associate a unique timestamp to each data item. To avoid concurrent add-remove anomalies, these data-types can use a remove-set of unique timestamps, which are called tombstones. In our Δ-CRDTs we use as unique timestamp pairs of `replicaID` and `operationNumber`. This ensures that each existing data item and tombstone can be compared (through the identifiers) to any given version vector (as to be before or after that point).

Causality is maintained by the same principle associated with shipping the whole state when using state-based CRDTs. `getDelta` always returns a complete Δ and thus all missing operations

```
1:  upon onVersionVector(VV, REPLICA) do
2:      Δ ⟵ self.state.getDelta(vv)
3:      if Δ.size() > 0
4:          REPLICA.send(Δ)
5:      optionally do (push model)
6:          if vv after self.versionVector
7:              REPLICA.send(self.versionVector)
8:  upon delta(Δ) do (atomically)
9:      self.state.applyDelta(Δ)
10:     self.versionVector.update(Δ)
11: periodically do (pull model)
12:     r ⟵ randomReplica()
13:     r.send(self.versionVector)
14: on local operation do (push model)
15:     r ⟵ randomReplica()
16:     r.send(self.versionVector)
```

Figure 4.2: Δ-CRDT replication mechanisms. Here, and in our implementation for the evaluation, version vectors are used for simplicity. Other forms to encode causal histories [132] could be used to the same effect.

on the other replica are sent in a single message. A Δ is always added to the local state in a single execution step (during which no other methods are able to access the internal data-structures), and thus causality is maintained. Note that when two replicas are synchronized, or when a replica receives a causal context that is in its future, the generated Δ will be empty.

To be able to compute the Delta from a given causal context, Δ-CRDTs need to maintain metadata about deleted elements (note that δ-CRDTs maintains such information in the pairwise communication buffers). As the internal state of a CRDT grows due to application operations, the amount of accumulated tombstones typically rises continuously, using storage space but not contributing to useful application data. In order to keep the amount of wasted space small we remove old metadata periodically (we provide a mechanism to garbage collect old tombstones).

A `garbageCollection` function is required to be implemented by CRDTs which should remove old metadata associated with all operations that happened before a given point in time (the causal history is the only argument – a version vector in the common case).

When garbage collection occurs, the previously described `applyDelta` function has to be able to infer if some portion of the current local state is outdated (i.e., removed data items whose's tombstones have been garbage collected). Using a Δ, the included version vector of the Δ, and the garbage collection point of the other replica (which is also included within the Δ), the `applyDelta` functions are able to correctly infer which data items locally exist, but were removed on the other replica.

The `getDelta` function is also adapted and must be able to handle the (typically rare) case where the local replica's garbage collection point is further ahead in time than the sender's causal

context. In this case, a Δ-CRDT falls back to a state-based CRDT merge procedure, where the whole state, including the causal context of the last garbage collection step, has to be shipped and integrated by the remote replica. Note that this returns a regular (albeit bigger) $Delta$ and can be applied normally at the receiving replica.

The main drawback of using Δ-CRDTs as defined in Listing 4.2 is expected to be an increase in latency for replicas to receive operations. Typically state-based CRDTs and operations-based CRDTs use a push model to propagate local changes to a replica. Though these data-types are able to immediately send the changes, Δ-CRDTs need an additional communication step between replicas. A causal history is sent first – typically a version vector – and then a Δ is sent back which can be locally applied. A version vector can also be piggy-backed along with the delta, as to ensure the initiating replica also ships any locally applied changes that the remote replica has not yet received. Note that when Δ-CRDTs are used with stable communication patterns, the additional communication step can be paid only when establishing the connection and then a model similar to operation or δ based propagation suffices.

When used in a scenario with dynamic communication patterns and compared to δ-CRDTs, Δ-CRDTs have the following advantages: (i) Δ-CRDTs do not require each replica to maintain a buffer for each of its connections; (ii) by using the information initially exchanged, a replica will only send the minimal Delta needed by the remote replica, instead of sending all the information stored in the CRDT (that might have arrived to the remote replica through a different communication path).

In summary, our reasoning on fully using Δ-CRDTs in Legion is the expectation of better network usage and lower metadata overhead. We adapted Δ-CRDTs for usage in Legion, and the results presented in Chapter 3 benefit directly from the modifications on how replicas synchronize (detailed further ahead in Section 4.3.2).

4.3.1 Δ-CRDTs evaluation

We have evaluated the use of Δ-CRDTs in comparison to state-based or operation-based CRDTs in Legion.

For these results we implemented an Observe-Remove Set (Specification 15: ORSet of [59]) in each form: Δ-CRDT, state-based CRDT, and operation-based CRDT.

We run multiple Legion clients (each client owns a replica of a replicated Set). The interactions between peers is dynamic, i.e., replicas communicate with a random subset of all existing replicas at each synchronization step. This means connections are not stable as the expected usage in Legion – we address how to leverage continued use of the same connection in Section 4.3.2.

4.3.1.1 Implementation

Communication between replicas happens every T seconds. In a synchronization step, a random subset of the currently connected neighbours are selected by a peer. At this point, using Δ-CRDTs, the causal context of the initiating replica is sent to those peers. In contrast, when using state-based CRDTs the whole state is shipped to the randomly selected peers.

When using operation-based CRDTs the timestamp (a version vector) is also first sent instead of the operations themselves. This was required for this experimental setting as there is no continuous flow of messages between pairs of replicas (i.e., the communication patterns change at each synchronization step). The alternative would require each replica in the system to maintain information about all operations which have previously been sent and acknowledged. The remote replica will use this vector clock to send back missing operations (and its own version vector).

This means, for these results, we employ a push model when propagating state based CRDTs and a pull model for operation and Δ based CRDTs.

4.3.1.2 Experimental setup

We run 8 clients where each client continuously issues operations over a single replicated CRDT Observe-Remove Set. Each client runs in its own Google Chrome instance, on a local machine (MacBook Pro Retina 2013, 2.6 GHz Quad-Core Intel Core i7, 16GB RAM). All reported results are the mean result of three independent runs.

The Set is updated, by each peer, twice per second. Each peer, per update, has a 30% chance to remove an existing data item and 70% chance to add a new data item (a string with 14 characters, with 2 bytes per char resulting in 28 bytes per data item). Each peer contacts 2 randomly selected peers, every 5 seconds, as to begin state reconciliation between them (as discussed previously).

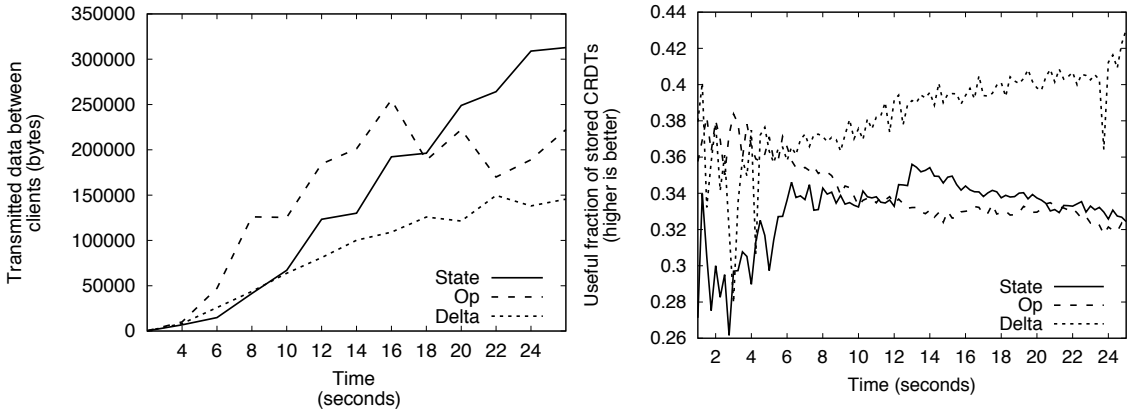
4.3.1.3 Results

We compare the sizes of messages sent between clients when using Δ, state, and operation-based CRDTs. Figure 4.3a reports the obtained results showing the aggregate bandwidth used, in bytes, of all messages exchanged between replicas, with a sampling interval of one second.

We only include object related messages, including state, operations, Δs, and version vectors when applicable (i.e., networking, overlay management, and other messages are ignored).

As expected, state-based CRDTs have an always growing load on the network. This is due to when more operations are executed, more state has to be exchanged between replicas. The currently implemented operation-based CRDTs are not optimized for the employed communication model and thus incur an initial load penalty. Nevertheless, as only the required operations are sent over the network (along with an initial version vector in each step), eventually the network usage becomes lower than state propagation. Δ-CRDTs propagate less data over the network as, when the total amount of applied operations increases, what is shipped between clients is always a Δ where this Δ is much smaller than the whole state of the object, and smaller than individual operations.

Figure 4.3b shows the ratio of useful state to total state. This is computed by dividing useful application state without any metadata by the whole CRDT, including application data and internal metadata. As our implementation of state-based CRDTs and operation-based CRDTs do not garbage collect, the fraction of tombstones increases over time. Our Δ-CRDTs with garbage collection enabled ensures that the fraction of useful state increases over time compared to the alternatives.



(a) Total bandwidth usage in bytes aggregated at per second intervals.

(b) Average storage overheads of metadata.

Figure 4.3: Comparison of state, operation, and Δ based CRDTs. Results for 8 replicas sharing an Observe-Remove Set. Each replica issues an operation every 500ms. The workload is composed of 70% inserts and 30% removes.

4.3.2 Legion's CRDT usage and causal propagation

The results that we have reported in the previous section show that, in a scenario with dynamic communication patterns, Δ-CRDTs outperform, from the standpoint of network usage, the competing alternatives.

Δ-CRDTs were initially implemented in Legion (Chapter 3), but required various changes to utilize them efficiently, as we detail next. The need for these changes comes from to the cost of establishing new connections due to WebRTC's signaling protocol – ideally when a connection is established it should be kept for as long as it is useful, and not closed/reopened when peers aim to propagate individual operations. Additionally, the presented mechanisms for Δ-CRDT synchronization adds latency – to propagate a single update over an established connection, first a metadata exchange is required.

Here we aim to efficiently use Δ-CRDTs when synchronizing a new pair of replicas, while also being able to use the established connection to continuously propagate operations without the initial metadata exchange being repeated for every operation. The remainder of this section explores how we accomplished this detailing how a CRDT can be implemented.

Legion's object store was built for the interaction among pairs of replicas. To explain how the system works we show how a Last-Writer-Wins register can be implemented. The original state-based version is taken from [59] (Specification 8: LWW register). Other CRDTs from [59, 139, 140] can be adapted similarly.

Listing 4.1 shows a minimal implementation (note we omitted code not relevant for this discussion such as error checking). The base idea of a LWW register is to keep a single value associated with the last write. A register accepts a set method to update the value and a query method to obtain the current value. Sequential sets trivially preserve user intent whereas concurrent updates are overwritten.

We associate with each newly set value a timestamp which must be unique, totally orderable,

```
function LWWCRDT (replicaID) {
  this.value = null // start with empty state
  this.ts = [null, Date.now()]
  this.set = function (value) {
    this.value = value
    this.ts = after(this.ts)
  }
  this.get = function () {
    return this.value
  }
  function after(ts){
    return [replicaID, ts[1] + 1]
  }
}
Legion.objectStore.defineCRDT(LWWCRDT);
```

Listing 4.1: State for LWW register Δ-CRDT.

and consistent with causality. For this we use as timestamp the pair `(replicaID, timestamp)`. The `replicaID` is used to disambiguate among concurrently set operations with equal timestamps. Replica identifiers are uniquely atributed to replicas when they initially connect to the server to be able to connect to the peer-to-peer network – the server uniquely assigns identifiers to replicas (Section 3.1.1.3).

To be able to synchronize with other replicas, the following methods are required for each CRDT object:

`getContext()` obtains, in the common case, a version vector. Some CRDTs might require less metadata, such as an LWW register that only needs to send the last timestamp (assuming it is uniquely atributed and consistent with causal order [49, 59]);

`getDelta(context)` obtains a Delta to be propagated to the remote replica, computed from the given context. In the case of an LWW register, the value and its timestamp;

`applyDelta(Delta)` applies the Delta to the object.

Listing 4.2 shows how these methods can be implemented for the LWW register. Notice that `getContext` is typically very simple: in this case it is just the timestamp, which suffices to compute if it is necessary to return the value or not, and in the common case the associated version vector.

As we aim to allow for individual operations to be propagated after replicas have called `applyDelta`, we require the application to notify the supporting system of such operations. To this effect, implemented CRDTs must take into account the following:

`callChangeHandler(change)` must be called every time a replica makes a change directly to the object, so these can be propagated to every connected replica;

`applyChange(change)` will be called for each locally created operation and also with further changes received from other clients (after having already called `applyDelta`).

```
this.getContext = function () {
  return this.ts
}
this.getDelta = function (context) {
  if(context <[a] this.ts) // compare timestamps and disambiguate on replicaID
    return {v:this.value, ts:this.ts}
  else
    return null
}
this.applyDelta = function (Delta) {
  if(Delta.ts > this.ts)
    this.value = Delta.value
    this.ts = Delta.ts
    return Delta
  else
    return null
}
```

Listing 4.2: Δ-CRDT behaviour for LWW register.

[a] Note that the code for < and > were simplified for clarity.

```
this.set = function (value) {
  callChangeHandler({
    value: value,
    ts: after(this.ts)
  })
}
this.applyChange = function (change) {
  if(change.ts > this.ts)
    this.value = Delta.value
    this.ts = change.ts
    return change
  else
    return null
}
```

Listing 4.3: Change propagation for LWW register Δ-CRDT.

The new implementation of `set` is presented in Listing 4.3. We no longer change data directly, the change handler is instead responsible to apply the change to the local replica. It is important that a set only executes after the previous set has completed. Our implementation ensures that `callChangeHandler` applies the change locally before returning (by calling `applyChange`).

Notice that both `applyDelta` and `applyChange` have a return value. This is important as the system needs to know if the changes were already applied previously or if they are new and need to be propagated to other neighbours. In the case of LWW register, we can simply return the full change. When the change has no effect we return null so that is it not propagated to other replicas (as it would have no effect). Other CRTDs, such as collections, can propagate a subset of the changes if some were already observed. These should be removed in `applyDelta` or `applyChange` as propagating them would be redundant.

To maintain causal delivery of operations, we require the Legion system to enforce an order on message delivery. For this we use a similar approach as the algorithm from Section 4.1.2.2

(including optimizations to minimize the cost of the initial synchronization).

To send individual operations after the initial synchronization Legion maintains, for each connection, a queue. For the object store to send a change to every peer, it enqueues the change to all queues. We ensure all connections between clients are FIFO, i.e., messages are received in the same order they have been sent.

In the actual implementation the propagation queue is shared among all connections – this allows for some optimizations:

- a single list of changes / deltas;
- peers still in the initial synchronization phase store no individual changes as these would be redundant to any to be computed Δ;
- subsequent operations on the same object can be merged. Using the LWW register counter example, multiple `set` operations can be merged into a single operation, where only the last operation (the one with the highest timestamp) is kept;[6]
- operations can be grouped and sent with higher compression ratios than indivual messages for each operation (simple data compression).

This has some overhead on tracking which replica must receive which Δ or individual operation (`change`), but overall the benefits far outweigh the costs.

To ensure every replica eventually observes every operation one additional aspect is required: every replica must be connected (directly or indirectly) to every other replica, as to allow information to flow among them. For this, as described in Section 3.1.1, Legion maintains a single connected overlay for a container – a set of objects – ensuring that every replica eventually observes every operation for each container it joins.

Our use of Δ-CRDTs along with the previous delivery mechanism ensures casual+ consistency for objects within a single container.

First, causal delivery is ensured due the per-replica synchronization mechanism and following propagation of individual changes (see Section 4.1.2.2). On network failures, synchronization is re-executed. As the (Legion) system ensures all replicas form a connected graph, all operations are eventually delivered to every replica. The CRDT specifications allow for convergent behaviour, thus providing causal+ consistency.

Some of the implemented CRDTs include:

- LWW register;
- Increment counter [59];
- Counter [59];

[6] To maintain causality the system does not reorder or merge operations which would invalidate causality. For example, subsequent increments and decrements may be merged if on the same object, but when another operation is in between them, the overall order may not be altered.

- MV-LWW register using version vectors [59];
- ORSet (add-wins), using version vectors [59];
- ORMap (add-wins), expands on Set (similar internals but an additional indirection);
- List (Treedoc [139]), using version vectors.

We use programatic merge for cases where the CRDT is unable to decide a value. For example, in the Map CRDT, concurrent writes do not overwrite concurrently added values to a single key. Instead, we allow for methods such as `get(key)` in collections to return a list of values. The application can decide to automatically overwrite with a new value or let the user decide which to keep.

Most of our implementations use version vectors for efficient computation of which changes (the Delta) to return. Note that this means careful thought has to be given into what data structures to use to implement more complex data-types, such as collections.

As a final note, Δ-CRDTs can be used to implement both operation and state based CRDTs. For state based, propagating the whole state can trivially be accomplished with the `getDelta` and `applyDelta` methods. For operation based, each CRDT can maintain a causally ordered list of operations which is fully propagated in the `getDelta` method and, afterwards, can individually be sent with the individual changes methods. This allows for implementing CRDTs following existing specifications and later including optimizations where needed.

4.4 Final remarks

In this chapter we explored client-side replication focussing on providing causal consistency.

We provided a discussion on causal consistency, detailing typical implementations (Section 4.1.1) and include a discussion on the intrinsic cost of causal consistency (Section 4.1.2).

In Section 4.2 we detailed CRDTs, followed by our proposal of Δ-CRDTs in Section 4.3. Δ-CRDTs, which improve networking and storage overheads over state and operation based CRDTs, provide causal+ consistency using well defined data types and synchronization mechanisms.

Related work is detailed in the previous chapters (Section 2.3 and Section 3.4).

Chapter 5

Securing causal consistency

In this chapter we focus on dealing with client-side replicas attempting to misbehave by circumventing the data-layer's properties.

The Legion middleware offers a peer-to-peer communication mechanism providing causal delivery (Chapter 3). Legion gives clear benefits when the application is used by small groups of cooperating clients (Section 3.3.2.1). We also used Legion to create peer-to-peer realtime games, leading to much faster interaction among users.

The Legion framework supports these applications, using secure channels and providing access control, but is unable to handle cases where malicious users actively try to defeat the system after joining. There is no way to specify the expected system and user behaviour and no way to verify or guarantee correct execution. While running a game with real users (computer science students), many cheat attempts were successful:

- `players[player_id].score+=10` – increasing own score (or decreasing opponents') without any direct cause (a causal dependency, such as capturing an item);
- `game.constants.shoot_interval*=0.5` – leading to the player being able to shoot twice as fast – altough to shoot there is no strict cause-effect order, there is a violation in execution speed. Editing such constants also allows for faster movement, among others;
- `players[player_id].kill()` – leading to the immediate death of a player – this function should only be called when an overlap with a previously shot bullet is found. Curiously, this cheat was followed up with spawning bullets directly on top of enemy players. Both tricks lack clear cause-effect events.

Moving application state to clients and allowing peer-to-peer synchronization poses multiple security challenges.

First, it is necessary to guarantee that unauthorized accesses do not compromise confidentiality and integrity of the system. This problem has been addressed resorting to standard security techniques [141, 142], where previous proposals on client-side replication (e.g., SwiftCloud [65] and Diamond [67]) resort to server-side security checks.

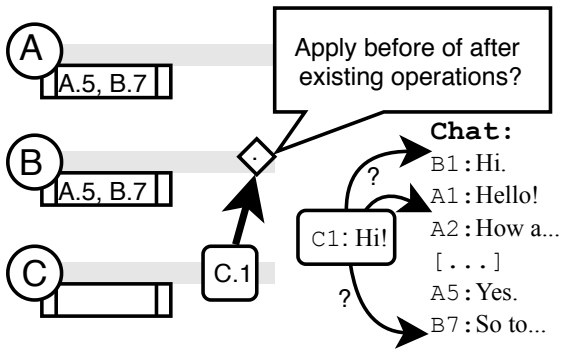


Figure 5.1: Chat application with clients A, B, and C using causal delivery of messages. As message C.1 is concurrent to the history of A and B, any result (show on the right) is causally correct.

Second, it is necessary to address client misbehaviour, which can be characterized by the Byzantine [143] and BAR [144] models. Several algorithms for providing functionality under these models have been proposed, such as reliable dissemination [144–146] and BFT state machine replication [147–149]. Decentralized replication algorithms, such as secure causal BFT [150–152] and blockchain-based replication [153–156], which enforce a total order on all operations, impose a high latency on writes.

We focus on a different problem: how to address client misbehaviour, that deviates from correct behaviour in a way that cannot be detected, in systems with client-side replicas and peer-to-peer synchronization and that adopt weaker consistency models to promote availability and low latency. This is an important issue as many applications (e.g., games) require high availability and low latency for a smooth user experience, and users have incentives for misbehaving (to gain an unfair advantage), but only if it is not possible to prove that they are misbehaving. See Figure 5.1 for an example – not only will incorrect (or un-intuitive) merge policies result in incoherent behaviour for end-users, malicious replicas may generate operations in such a manner as to ensure the final outcome is to their own benefit.

We analyze the possible effects of misbehaving replicas in causal consistency (Section 5.2).[1] From this analysis, we derive secure variants of causal consistency that preclude different types of misbehaviour (Section 5.3). We propose practical algorithms for implementing these models in a setting where clients communicate directly (Section 5.4).

Other forms of consistency are required for many use-cases, double spending being a concrete example for stronger requirements. If users do not coordinate, in a game application with a restricted amount of resources each could spend all resources concurrently, leading to erroneous behaviour on the application after they communicate their operations with each other and attempt to merge application state.

To address this we also propose a secure version of eventual linearizability [64, 118, 157], as a way to provide stronger guarantees when required (Section 5.3.3.2).

[1] Causal consistency itself is introduced in Chapter 4.

We have designed and evaluated a system that provides the proposed secure consistency models. Our evaluation (Section 5.5) shows that adopting the secure consistency models imposes low overhead when compared with their insecure counterparts, while providing low user-to-user latency and server load compared to traditional client-server architectures. The latency gains are more expressive for interactions among nearby clients, which maps the expected use in many applications, such as augmented reality games. We show that providing multiple secure consistency models can be important, as it allows developers to select a different point in the trade-off space between application guarantees, latency, and communication overhead.

The secure consistency models can be used to enrich server-based architectures with fast and secure peer-to-peer interactions. In summary, in the work presented in this chapter, we make the following contributions:

- a systematic study on how client misbehaviour impacts the guarantees of causal consistency (Section 5.2);
- the definition of secure consistency models, variants of causal consistency and also eventual linearizability, preventing multiple types of misbehaviour (Section 5.3);
- algorithms for the secure consistency models (Section 5.4); and
- an experimental evaluation (Section 5.5) of our prototype.

5.1 Attacker model

In a tradiational client-server communication model the server can verify any action each client attempts to apply to the state which is kept at the server. In our model, as clients manage local replicas and, instead of communicating only with the server also synchronize directly, this is no longer true.

As in Legion, we consider multi-user applications where users interact using their own devices. Clients execute operations on their local replicas and synchronize directly among themselves by propagating operations. A subset of these clients synchronize with the server to ensure durability and to allow clients that cannot communicate directly with other clients to participate.

The attack surface is increased compared to a client-server model as each client manages a local replica and generates and propagates changes to the shared state not only to the server but also to other clients.

We depart from the assumption that works for Legion and reason on applications which operate with untrusted clients. We consider an attacker model focused on clients (i.e., servers and other infrastructure nodes are trusted).

Our attacker model is based on a trusted centralized server and trusted infrastructure nodes, where clients follow the BAR model [144]. Malicious clients can behave in a fully byzantine mode (arbitrarily deviating from their prescribed behaviour) or be rational, meaning they will deviate from the prescribed protocol (to attempt to gain some advantage) only if the misbehaviour cannot

be detected by correct clients or servers. Malicious replicas can affect only their local state, we assume that malicious clients cannot prevent correct clients from establishing secure channels with a server or among each other.

Correct clients will always follow the prescribed protocols (Altruistic in BAR).

Byzantine clients, by not focusing on hiding misbehaviour, provide to their peers either a demonstration or a proof of their malicious actions. For example, sending unsigned messages is a demonstration of misbehaviour but doesn't lead to a proof, while a correctly signed message that contains a falsehood is a proof-of-misbehaviour.

Rational clients pose the most risk in our model. These clients may attempt to manipulate the generation and propagation of operations in a way that benefits them. This problem is important, for example, for the gaming industry, where peer-to-peer approaches are attractive in terms of latency and availability if cheating can be avoided. In this case, clients (players) have interest in being rational (to gain an advantage) if they cannot be discovered as being rational (to avoid being banned due to cheating).

5.2 Attacks on causal consistency

A malicious replica, if left unchecked, can easily disrupt the properties of a replication algorithm for causal consistency. In this section we systematically identify possible attacks by both byzantine and rational replicas.

5.2.1 Generating operations under causal consistency

Causal consistency is described in detail in Chapter 4. In summary, for a replicated system, we say that operation o_1 happened before operation o_2, $o_1 \prec o_2$, if o_2 was generated in some replica where o_1 had already been executed. For a set of operations O, this generates the partial order $(O, \prec)$. An algorithm that enforces all replicas to apply operations respecting this causal order, enforces causal consistency.

For any operation o, generated at replica r, we can consider three disjoint sets of operations, as shown in Figure 5.2:

- $P(o)$ is the set of past operations that happened before o – this is also known as the causal history of o, $H(o)$;
- $C(o)$ is the set of operations that are concurrent with o, i.e., $\forall o_c \in C(o), \neg(o_c \prec o) \wedge \neg(o \prec o_c)$;
- $F(o)$ is the set of future operations that happened after o, i.e., $\forall o_f \in F(o), o \prec o_f$.

The way each operation has its dependencies encoded depends if a system uses either version vectors or direct dependency graphs. In the former, the dependencies of each operation are

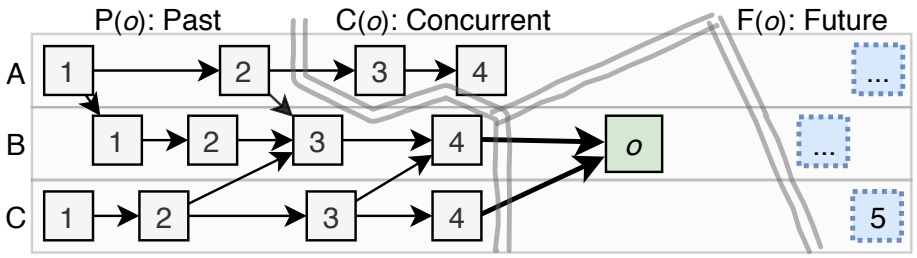


Figure 5.2: Operation dependencies in causal consistency.

summarized in a vector that states which operations generated at each site happened before a given operation. In the example of Figure 5.2, the dependencies of the new operation o would be `[2,4,4]`, stating the dependency on operations up to 2 from replica A, and up to 4 from both replicas B and C. Using direct dependencies, each operation includes only information on the concurrent operations that have been executed before its generation. In the example, operation o would depend on `{B:4, C:4}`. Since each operation includes its dependencies, it is possible to build the whole dependency graph.

5.2.2 Tampering with other replica's operations

This class of attacks comprises actions that a malicious replica can perform regarding operations created by other replicas – this includes tampering with the integrity of messages in transit (such as modifying causal dependencies of operations), generating operations in the name of other replicas, and creating malformed operations.

A simple example is altering the overall order of events by creating new operations and set other (already existing) operations to depend on them. In a game where players shoot each other, one can make a shot depend on a later created moving operation, making the shot miss instead of hit the malicious player. In general, such attacks can be addressed by having replicas sign the operations they generate, as discussed in Section 5.4.1. Attacks on message propagation (e.g., not propagating a subset of operations) are discussed in Section 5.4.7.

5.2.3 Attacks on operation generation

This class of attacks consists in manipulating the creation of new operations by attaching incorrect dependencies. In contrast to Figure 5.2 which illustrates the correct dependencies of a new operation o, Figure 5.3 shows possible attacks, discussed next.

5.2.3.1 Omitting dependencies

A malicious replica may create an operation that contains only a subset of the actual dependencies. Given the set of operations executed in replica r, P_r, this attack consists in setting the causal history of a new operation to the set ${P_r}^{rem}$, such that ${P_r}^{rem} \subsetneq P_r$ (i.e., ${P_r}^{rem} \subset P_r \wedge \exists o \in P_r : o \notin {P_r}^{rem}$).

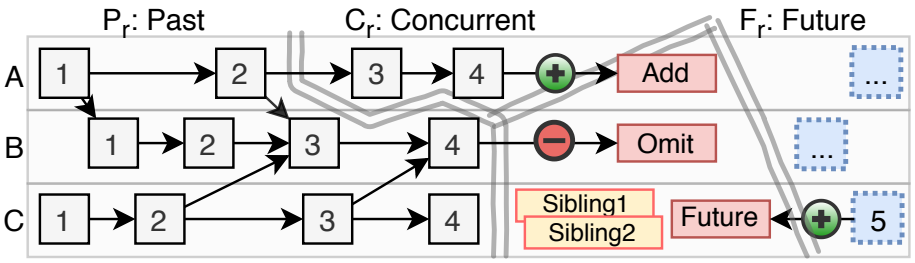


Figure 5.3: Attacks on operation generation under causal consistency.

By including only a subset of the known operations in the dependencies, a malicious replica can forge an operation that is concurrent to operations that it already knows, creating an operation that occurred in its logical past. In Figure 5.3 this is shown as *Omit*, with the omission of known operations from replica B in the dependencies of the new operation.

Just as the previous attack of manipulating dependencies of existing operations, this attack can also be used for moving away from another user's shot. The difference is that the resulting state will depend on the application's conflict resolution policy, as the shot and movement operations will be concurrent.

We note that the possible attacks to causal consistency are similar when using version vectors or direct dependencies. Consider the example of Figure 5.3. When generating o, operations 1-2 from replica A and 1-4 from replicas B and C were received, leading to a version vector `[2,4,4]` or to the direct dependencies `{B:4; C:4}`. When using version vectors, a malicious replica can selectively remove a suffix of operations from any replica – e.g., vector `[0,4,4]` would remove the dependencies from replica A. However this has no impact when enforcing causal consistency, as only the direct dependencies are important – executing operation 3 (and 4) from B requires that operation 2 from A has been executed already.

5.2.3.2 Depending on unseen operations

A malicious replica may create an operation that depends on an operation that has not been executed locally and possibly does not even exist yet. Given the set of operations executed in replica r, P_r, this attack consists in setting the causal history of a new operation to the set ${P_r}^{add}$, such that $P_r \subsetneq {P_r}^{add}$ (i.e., $P_r \subset {P_r}^{add} \wedge \exists o \in {P_r}^{add} : o \notin P_r$). In Figure 5.3, this is represented as *Add* or *Future* (for depending respectively on an operation not yet received or on a future operation). Such an attack allows a malicious replica to create operations that do not respect the real time order, potentially counter-acting the actions of other replicas even before they take place.

Consider a multiplayer game where players fight some monster, which drops a specific item when defeated. The item is available as soon as it is dropped, but only the first player to react will pick it up. A malicious player can create a `pick-up` operation that depends on a future drop operation. This ensures an unfair advantage over other players, as the pick-up will execute immediately (and only) after the drop operation is created and the dependency is met.

Depending on the application, such an attack may be detectable – the dependency may never

be executed (e.g., if the creature is never defeated) or might reach the replica that will create the dependency before its execution.

5.2.3.3 Combining omit and add

A malicious replica can create an operation that combines the previous two attacks, altering dependencies to both omit and include unseen operations. Given the set of operations executed in replica r, P_r, the attack consists in setting the causal history of a new operation to the set ${P_r}^{add+rem}$, such that $(\exists o_1 \in {P_r}^{add+rem} : o_1 \notin P_r) \wedge (\exists o_2 \in P_r : o_2 \notin {P_r}^{add+rem})$.

5.2.3.4 Sibling generation

In any replicated system, an operation typically includes an identifier. A malicious replica may generate two different operations with the same identifier (which we call sibling operations). In a system that is not prepared to deal with malicious replicas, different replicas may execute different versions of the operation, leading to a permanent state divergence.

A simple example is a replica creating, within a chat application, two different messages with the same identifier, leading to different users seeing a different chat history. Sending a praise to some users and an insult to others can lead to comical results, while the system assumes that both messages, having the same identifier, are the same message.

5.3 Secure consistency models

In this section we propose consistency models that deal with the presence of malicious replicas, by addressing the attacks discussed in the previous section.

5.3.1 Secure causal consistency

We start by deriving a secure form of causal consistency by defining a set of properties that must be enforced. Our first property precludes tampering with the causal history of an operation, after it is generated:

Secure Causal Property 1. *(Immutable History) If H_o is the causal history of operation o at generation, o is delivered with H_o at every correct replica.*

We now define properties concerning the dependencies on unseen operations. To disallow *Future* from Figure 5.3, the *No Future Dependencies* property precludes having dependencies on operations that have not been executed yet in any replica.

Secure Causal Property 2. *(No Future Dependencies) Given P_{all}, the set of all operations generated in any replica of the system when operation o_{new} is created, o_{new} can only depend on such operations, i.e., $\nexists o \notin P_{all} : o \prec o_{new}$.*

Note that this disallows not only *Future* – depending on operations which will depend on the newly created operation, creating circular dependencies – but also a subset of *Add*: the operations that are concurrent but which have not yet been created.

As for creating operations that depend on existing operations which were generated in some other replica but that have not been executed locally (*Add* in Figure 5.3), we note that verifying this seems counter-intuitive – the situation is equivalent to synchronizing the local replica before issuing the operation. Instead we define a property that enforces all correct replicas to execute operations respecting their defined dependencies:

Secure Causal Property 3. *(Causal Execution) All correct replicas execute an operation respecting the dependencies defined in the operation. Given the causal history of an operation o, $H(o)$, the causal serialization $O_r = (Ops, <)$ in every correct replica r is such that $\forall o_i \in H(o), o_i < o$.*

We now address the problem of having a malicious replica issuing two operations with the same identifier, which can lead correct replicas to execute different versions of the operation (*Siblings* in Figure 5.3). This could be avoided by executing a consensus step to certify the operation associated with each identifier [150–152]. However, such an approach goes against our objective on availability – our goal is allowing replicas to both generate operations and immediately execute received operations. Instead, we require this situation to be eventually detected and reported to correct replicas, by locally executing a $fault(o)$ operation:

Secure Causal Property 4. *(Eventual Sibling Detection) Given two operations o_1 and o_2 with the same identifier, for any replica r that has executed the set of operations Ops_r, the following conditions will apply: (i) if $o_1 \in Ops_r$, then eventually $fault(o_1) \in Ops_r$, with $o_1 < fault(o_1)$; and (ii) if $o_2 \in Ops_r$, then eventually $fault(o_2) \in Ops_r$, with $o_2 < fault(o_2)$.*

5.3.1.1 Omitting dependencies

We now consider the attack where a malicious replica generates an operation that omits some of the locally executed operations from the set of dependencies (*Omit* in Figure 5.3). It is impossible for a correct replica r_c, receiving an operation o from a malicious replica r_m, to verify if o includes all dependencies it should or not. Even if r_c has previously sent to r_m some operation o_o, the fact that o_o is not included in the dependencies of o can be due to o being generated before o_o was received and executed. The correct behaviour due to delays is indistinguishable from an incorrect behaviour where o_o is purposely omitted from the dependencies.

Let $O^{real} = (Ops, \prec^{real})$ be the happens before partial order as registered by an external observer, which perceives all dependencies within the system. When generating a new operation o, a malicious replica may omit some of its real dependencies, leading to the partial order $O^{omit} = (Ops, \prec^{omit})$, that omits some of the dependencies defined in O^{real}, including direct dependencies of o and indirect dependencies among other operations established through o.

A malicious replica cannot omit dependencies in an arbitrary way without being detected. This can be exemplified in Figure 5.3 where, if operation A2 is omitted from the dependencies, then all operations that depend on it – A3, A4, B3, and B4 – must also be omitted. When using

version vectors, as discussed before, setting the dependencies as `[0,4,4]` has no actual effect, as B3 has itself A2 as its dependency and so will only execute after A2 has executed. Moreover, it allows the detection of the misbehaviour by analyzing the graph of dependencies. When using direct dependencies, it is also only possible to omit (a suffix of) immediate dependencies.

In general, to avoid detection, a malicious replica cannot omit from the dependencies of an operation o it generates any operation o_p that happened before an operation o_m included in the dependencies of o:

Secure Causal Property 5. *(Limited Omission) Given P_r, the set of operations executed in replica r, a malicious replica can only omit dependencies for a new operation without being detectable, by setting the causal history of the new operation to be ${P_r}^{rem}$, such that ${P_r}^{rem} \subsetneq P_r \wedge \nexists o_p \in P_r \setminus {P_r}^{rem}, o \in {P_r}^{rem} : o_p \prec o$.*

We now formally define secure causal consistency:

Definition 5.1. ***Secure causal consistency*** *is a model which ensures that any correct replica r executes operations according to a serialization order $O_r = (Ops, <)$, such that:*

a) no operation with tampered dependencies is executed (Property 1: Immutable History);

b) no operation that depends on a future operation is executed (Property 2: No Future Dependencies);

c) O_r is a valid serialization of $(Ops, \prec^{omit})$, i.e., given the set of dependencies of operation o, $H(o), \forall o_p \in H(o), o_p < o$ (Property 3: Causal Execution, Property 5: Limited Omission);

d) if two different operations with the same identifier are generated, any correct replica that executes any of such an operation o will also eventually execute $fault(o)$, with $o < fault(o)$ (Property 4: Eventual Sibling Detection).

5.3.2 Secure strict causal consistency

Ideally, a replica should be forced to set the real dependencies to the operations it generates:

Secure Causal Property 6. *(Real Dependencies) The dependencies of an operation o, $H(o)$, are the real dependencies iff $\forall o_p, o_p \prec^{real} o \Rightarrow o_p \in H(o)$.*

This leads all operations to be created with the dependencies according to $\prec^{real}$ – we discuss practical implementations of such a property in Section 5.4.3, which requires trusted software or hardware. From this property we can derive:

Definition 5.2. ***Secure Strict Causal Consistency*** *is a consistency model that ensures that any correct replica r executes operations according to a serialization order $O_r = (Ops, <)$, such that O_r is a valid serialization of $(Ops, \prec^{real})$, i.e., given the dependency set for an operation o, $H(o)$, $\forall o_p \in H(o), o_p < o$ (Causal Execution). Note that a), b), and d) from Secure Causal Consistency are enforced by this model.*

5.3.3 Tolerating collusion

Even if replicas are unable to generate operations with incorrect dependencies, two colluding replicas are able to communicate through a side-channel, circumventing any existing mechanisms to enforce secure strict causal consistency.[2]

A possible solution to tackle this challenge is to use a consistency model based on recency, requiring new operations to depend on all existing operations at all replicas (e.g., Natural [102] or External [127] causal consistency). Implementing such an approach requires a form of synchronization among all replicas for generating new operations which, again, goes against our goal of remaining available in the presence of network partitions.

We adopt a different approach: operations are generated and executed without coordination, and a replica is eventually notified if an operation o_2, that might externally depend on operation o_1, was executed before o_1.

We define a total order, $<^{ext}$, that guarantees that if an operation o_2 might depend on operation o_1, then $o_1 <^{ext} o_2$. The ordering of $<^{ext}$ must respect the following properties:

Extended Causal Property 1. *(Total Order) Given the set of operations Ops, $O_{<^{ext}} = (Ops, <^{ext})$ is a total order (i.e., $\forall o_1, o_2 \in Ops : o_1 <^{ext} o_2 \vee o_2 <^{ext} o_1$).*

Extended Causal Property 2. *(External Causal Visibility) Given two operations o_1 and o_2, if some replica has observed (in realtime) o_1 before the generation of o_2 in any replica, then $o_1 <^{ext} o_2$. i.e., $\forall r_1, r_2, \forall o_1, o_2 : observed_{r_1}(o_1) <^{obs} generate_{r_2}(o_2) \Rightarrow o_1 <^{ext} o_2$, with $<^{obs}$ the total order of events as observed by an external omniscient observer).*

We now define two consistency models that use this total order ($<^{ext}$).

5.3.3.1 Secure extended causal consistency

Secure Extended Causal Consistency is a model which extends Secure Causal Consistency by notifying applications of out of order (according to $<^{ext}$) executions:

Definition 5.3. ***Secure Extended Causal Consistency*** *is a consistency model that ensures that any correct replica r executes operations according to a serialization order $O_r = (Ops, <)$, such that:*

a-d) equal to Secure Causal Consistency (Definition 5.1);

e) if an operation is executed in an order that violates $<^{ext}$, the application is notified when executing the operation that should have been executed earlier using `signal`*, i.e., if $\exists_{o_2} : o_1 <^{ext} o_2 \wedge o_2 < o_1$ then the execution of o_1 is replaced by the execution of* `signal`*(o_1, O_s), with O_s the set of operations that should have been executed after o_1 according to $<^{ext}$ (i.e., $O_s = \{o : o < o_1 \wedge o_1 <^{ext} o\}$).*

[2] Side channels might not be discoverable, they can range from locally installed software which automatically propagates data and applies operations directly into the application on the other device, to video-calling or users simply sitting next to each other.

Note that signaled operations are not necessarily a causality violation due to collusion, and should be handled by the application in an application-dependent way.

A chat application is a good example as messages can be trivially ordered by $<^{ext}$. When a message is received out of order, and `signal` is called, the user interface can explicitly annotate such messages (with read/unread labels), leading to an intuitive user experience. In Section 5.4.5 we further detail how this model can be used in practice.

5.3.3.2 Secure eventual linearizability

We define Secure Eventual Linearizability as to enforce the execution of received operations according to $<^{ext}$:

Definition 5.4. ***Secure Eventual Linearizability*** *is a consistency model that ensures that any correct replica r executes the operations it has received, Ops, according to the serialization order $O_r = (Ops, <^{ext})$.*

Note that this definition allows to execute operations immediately after they are received – only the serialization order must be respected while delivery of operations may be delayed.

Eventual linearizability is an interesting model as it can provide strong guarantees while remaining highly available [64, 118, 157]. In short, the serialization of operations at each replica gravitates, over time, towards a total order shared among all replicas, depending (in our case) on which operations were delivered.

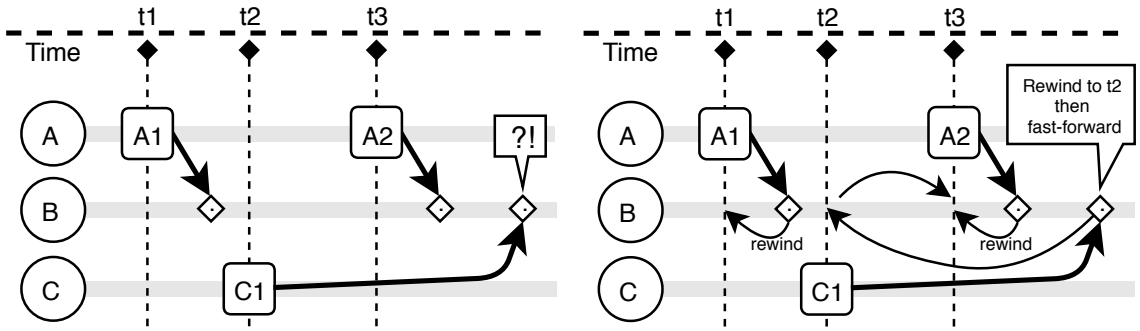
Just as in previous works [64, 118, 157], to enforce the order $<^{ext}$, when a new operation is received it might be necessary to undo/redo operations. In contrast to these works our definition of the order $<^{ext}$ is final when an operation is generated and may not be altered later (even if at each replica not all operations are yet known). Additionally, we do not reason on operations which the referenced works name *strict* or *forced* – these disallow an undo/redo execution model and would require a coordination step among all replicas which we wish to avoid.

The undo/redo behaviour can be summarized as rolling back operations (undo phase) when a previously unknown operation arrives which should precede the already executed ones. After the newly received operation is executed a redo phase happens: the rolled-back operations can be re-executed (in order). See Figure 5.4 for a detailed example.

5.4 Algorithms and implementation

This section discusses possible implementations of the secure consistency models, which we used in our prototype. Figure 5.5 presents excerpts of the proposed algorithms.

We assume that each replica authenticates with the centralized service when joining the system (i.e., when a user is starting its session), receiving a certificate signed by the server that can be used to prove the replica's identity. All replicas trust the server, being able to locally validate certificates of other replicas. Replicas use the associated private key to sign information. We assume that the cryptographic primitives such as digital signatures and hash functions



(a) Out of order reception of operations – the application's conflict resolution mechanism must ensure the final state provides an outcome which is both consistent and intuitive for the user.

(b) An undo/redo execution model ensures operations can be executed as soon as they are received, rewinding time (application state) and re-executing operations when needed. See Section 5.4.6 for implementation details.

Figure 5.4: Execution model of Secure Eventual Linearizability. Replica B receives two operations from replica A, followed by an operation from replica C which was created, in realtime, in between A's operations but has its propagation delayed.

cannot be undermined. We further assume that replicas communicate through secure channels, authenticating each other by leveraging the certificates obtained when joining the system.

5.4.1 Authenticity, non-repudiation, and integrity

An application can issue a new operation by calling the *NewOperation* procedure (line 5). This function creates an operation record that includes the operation with its identifier (the pair (c_r, id_r)) and metadata specific for each consistency model. The record has a signature of this information ($record.signature$). The operation record (or simply operation where no confusion can arise) is sent to the replica's neighbours.

The signature is used to ensure that operations propagated among replicas are originated in a valid replica (authenticity), that operations can be associated to its creators (non-repudiation), and that they are not modified in transit by malicious replicas (integrity).

Upon receiving an operation record (line 22), a replica first verifies that the signature is correct (VERIFYSIGNATURE). If the operation's signature cannot be validated, the operation is discarded. If the signature is valid, the metadata is verified (VERIFYMETADATA) and, if valid, processed according to the chosen secure consistency model.

If the previous verifications end successfully, the operation contents can still be invalid, for instance when a Byzantine replica issues invalid operations according to application logic or impersonates other replicas [158–160]. In this case, a proof-of-misbehaviour is produced for the replica that generated the operation and sent to the centralized infrastructure which will disseminate the proof among all clients to expel the Byzantine client from the system (Section 5.4.7).

For misbehaviour that does not allow to produce a proof (e.g., sending unsigned records), clients will simply disconnect from its sender. Continuous erroneous execution thus leads to

1: **Local State (replica r):**
2: id_r : *identifier of local replica*
3: c_r : *counter used for identifying operations*
4: $cert_r$: *local replica certificate*
5: **procedure** $NewOperation(op)$
6: $c_r \leftarrow c_r + 1$
7: $record.op \leftarrow AddMetadata_{model}(op, (c_r, id_r))$
8: $record.signature \leftarrow \text{SIGN}_r(record)$
9: $\text{SEND}(p, record), \forall p \in neighbours$
10: **function** $AddMetadata_{SecureCausal}(op, op_{id})$ ▷ Section 5.4.2
11: $deps \leftarrow LatestDependencyIds()$
12: $hashDependencies \leftarrow Hash(LatestDependencyRecords())$
13: **return** $<op, op_{id}, random(), deps, hashDependencies>$
14: **function** $AddMetadata_{StrictSecureCausal}(op, op_{id})$ ▷ Section 5.4.3
15: $deps \leftarrow LatestDependencyIds()$ ▷ Run both lines in secure module
16: **return** $\text{ENCODE}(<op, op_{id}, random(), deps>)$
17: **function** $AddMetadata_{SecureExtended/EventualLinearizability}(op, op_{id})$ ▷ Section 5.4.4
18: $deps \leftarrow LatestDependencyIds()$
19: $hashDependencies \leftarrow Hash(LatestDependencyRecords())$
20: $(ts, signature_{TiS}) \leftarrow TiS\_OpTs(<Hash(op, op_{id}, deps, hashDependencies)>)$
21: **return** $<op, op_{id}, ts, deps, signature_{TiS}>$
22: **upon receive** $record$ **by** p **do:**
23: **if** $\text{VERIFYSIGNATURE}(cert_p, record)$ **then**
24: **if** $\text{VERIFYMETADATA}_{model}(p, record)$ **then**
25: $Process_{model}(record)$

Figure 5.5: Algorithms for secure consistency models.

a malicious replica to be restricted to pure client-server model as correct replicas will deny connections to it.

5.4.2 Secure causal consistency

To track causal dependencies, the metadata of each operation includes the identifiers of its direct causal dependencies (line 11). For a newly generated operation, the direct causal dependencies includes any locally executed operation o for which there is no operation $o_n : o \prec o_n$ (Section 4.1.1). When compared with version vectors, this approach has the advantage of not requiring one entry per replica, being more suitable for handling large and dynamic memberships.

We enforce the properties of secure causal consistency (Definition 5.1):

Immutable History As replicas sign the operations (line 8), the causal histories of operations cannot be manipulated by malicious replicas, thus enforcing Secure Causal Property 1.

No Future Dependencies To disallow depending on operations that have not yet been generated, including those not yet observed, the metadata of an operation includes a cryptographic summary (hash) of all direct causal dependencies (line 12) and a random number (line 13).

This makes it impossible for a malicious replica to create a dependency on an operation that it has not yet observed, as it is unable to compute a valid hash of the dependencies. Replicas must validate the hash before executing the operation (in the VERIFYMETADATA step). If an invalid hash is detected, a proof-of-misbehaviour for the replica that generated the invalid operation is issued which, as previously stated, will lead the replica to be excluded from the system. This technique allows to enforce Secure Causal Property 2.

Causal Execution Secure Causal Property 3 (causal execution) is achieved by verifying, before executing an operation, that its dependencies have already been executed.

Our prototype uses the Δ-CRDT synchronization protocol proposed in Section 4.3.2, where operations are disseminated through a communication overlay and each replica only propagates an operation to a peer after propagating the operation's dependencies (or knowing that the peer already received them). This guarantees that, when receiving an operation, causal dependencies are satisfied and the operation can execute immediately.

When a replica detects that a remote replica is not following the protocol (when verifying metadata), it produces a proof-of-misbehaviour. This proof can be generated when discovering an out-of-order propagation as all messages sent between two replicas are hash-chained: each message (containing operations) sent between two peers is signed by the sender and includes the hash of the previous message (omitted in the code for simplicity). Note that the first message between two peers is always a certificate exchange, followed by the Δ-CRDT synchronization protocol and subsequent propagation of individual changes (operations).

Limited omissions As the metadata includes only direct dependencies, it is impossible by design for a replica to introduce causal gaps in the dependency graph (as it can only omit dependencies in a suffix of the dependencies), thus guaranteeing Secure Causal Property 5.

Eventual Sibling Detection We use several techniques to detect when multiple operations with the same identifier are created, as to be able to guarantee Secure Causal Property 4. First, a replica that receives two siblings from different paths creates a proof-of-misbehaviour and informs the server. Second, each operation includes in its metadata the hash of its dependencies – when receiving an operation, if this hash does not match the hash computed locally for the same dependencies, the replica signals a potential sibling by informing the server. Finally, the server periodically sends a summary of its state, containing the hashes and identifiers of the last observed operations at the server which replicas can use to verify if they have received the (exact) same operations. If the verification fails, the client connects to the server to verify the hashes of each individual operation, leading to a proof-of-misbehaviour for the replica that generated them.

The server's state summaries also follow causal propagation (respecting Δ-CRDT propagation) and is included in each pairwise hash-chain. A replica found to propagate a summary which includes some operation it has not previously sent, is proven to be malicious.

While these mechanisms cannot prevent Byzantine replicas from exhibiting arbitrary behaviour, they are enough to prevent rational replicas (that want to avoid exclusion from the system) to perform such attacks. These mechanisms together allow to provide **Secure Causal Consistency**.

Intuitively, there seems to be no straightforward mechanism to disallow omitting operations from causal dependencies or delaying propagation of operations. As any two operations generated by the same replica always have an implicit dependency from higher to lower identifier, a replica is unable to selectively hold back its own operations. Nevertheless, replicas can collude to omit operations from causal dependencies. The following sections discuss how to provide additional guarantees by leveraging trusted components, being it within trusted servers or secure hardware modules.

5.4.3 Secure strict causal consistency

Secure Strict Causal Consistency requires a replica to record the exact causal dependencies. This can be implemented by delegating the reception and generation of operations to a trusted service. An instance of the service can use trusted hardware if available at the replica, such as Intel's SGX [161, 162]. When the replica has no trusted hardware, the same function can be delegated to instances of the trusted service running nearby the replica or to trusted infrastructure nodes.

The service is responsible to receive operations before delivering them at the client to track all received operations (to guarantee that causal dependencies are faithfully assigned). When new operations are created, the service assigns the correct causal dependencies. When compared with secure causal consistency, the metadata does not need to include the random number as the trusted module will never include incorrect dependencies.

To prevent the application from accessing an operation before it being processed by the secure module, each operation is ciphered (ENCODE on line 16) by the secure module with a key shared only among instances of the trusted service. As only instances of the trusted service have access to the shared key, it is guaranteed that operations are correctly created and can only be accessed after being deciphered by the secure module (for simplicity, the DECODE step is omitted in the code).

An immediate issue that arises when using an external service is ensuring correct client-handover and maintaining causal dependencies when such a handover is done. Applying earlier works [117, 163, 164] is not practical as no thought has been given to replicas attempting to circumvent causal relations – intuitively, any reconnection by a client from one instance of the service to another always requires some form of coordination among instances of the service. A malicious client actively *hopping around* can introduce a great overhead in such a system. Due to this costly operation, we do not further explore this direction, leaving it for future work.

5.4.4 Secure timestamps to prevent collusion

Even if a single replica is unable to forge an operation with incorrect dependencies, two replicas, r_1 and r_2, can collude create an operation o_2 that is concurrent with some known operation o_1

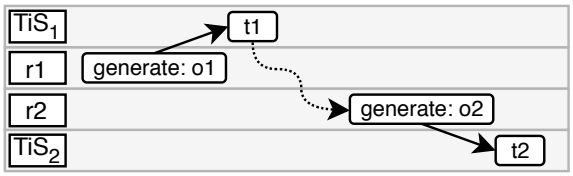


Figure 5.6: Causal dependencies and timestamps associated to operations.

(by having replica r_1 sending o_1 through an external channel to r_2, and having r_2 submitting o_2). In Section 5.3.3, we proposed to address this problem by using a total order $<^{ext}$ on operations, such that if o_2 might depend on o_1, then $o_1 <^{ext} o_2$ – we synthesized the underlying properties as Extended Causal Property 1 and 2.

Consider realtime timestamps uniquely attributed to each operation by an omniscient entity. Such timestamps would allow to order operations in a way that respects the required conditions, as any externally visible happens before relations are captured by this global realtime order. Having a single server assigning such timestamps would provide the required total order, but would introduce a single point of failure – not being able to communicate with this server means no progress can be made, defeating the goal of remaining available.

5.4.4.1 Practical external visibility

We propose the use of a decentralized timestamping service (TiS) of which all instances have synchronized clocks with a small divergence of up to δ. Replicas communicate with this service to obtain timestamps for their operations.

There is a total order on all operations generated at a single instance of the service, as we restrict it to only emit one timestamp per time unit. With multiple instances, we order operations with colliding timestamps by sorting on a hash of the whole operation. This provides a total order among all operations, thus providing Extended Causal Property 1.

For enforcing Extended Causal Property 2, we must guarantee that, if any replica observes an operation, and after (in realtime), any replica creates a new operation, the latter must appear after the former in the total order.

Consider Figure 5.6, where replica r_2 generates o_2 after observing replica r_1's o_1. If clocks and execution were perfect, the smallest possible timestamp for o_2 would be: $t_2 = t_1 + (TiS_1 \rightarrow^T r_1) + (r_1 \rightarrow^T r_2) + (r_2 \rightarrow^T TiS_2)$.[3] As clocks are not perfectly synchronized, we need to take into consideration clock divergence, leading to: $t_2 = (t_1 + \delta_{TiS_1}) + (TiS_1 \rightarrow^T r_1) + (r_1 \rightarrow^T r_2) + (r_2 \rightarrow^T TiS_2) + \delta_{TiS_2}$.

As in a distributed systems it is impossible to exactly measure the minimum value for $(TiS_1 \rightarrow^T r_1)$,$(r_1 \rightarrow^T r_2)$, and $(r_2 \rightarrow^T TiS_2)$, and as we have no control over malicious replicas using external communication channels, the safe approach is to assume they are 0. This lets us simplify to $t_2 = t_1 + \delta_{TiS_1} + \delta_{TiS_2}$.

[3] $A \rightarrow^T B$ is the minimum time it takes for a message to be sent from A to B, by any means, internal or external to the system.

For guaranteeing that $t_2 > t_1$ (as to enforce External Causal Visibility), we force all TiS instances to wait $TWait$ before returning a newly generated timestamp to a client (i.e., the timestamp generated at t_i is returned at $t_i + TWait$), with $TWait > |\delta_{TiS_1} + \delta_{TiS_2}|$. As δ_{TiS_1} is the clock divergence of TiS_1 to a global reference, $\delta_{TiS_1} + \delta_{TiS_2}$ is the clock divergence between TiS_1 and TiS_2. To guarantee that the condition for $<^{ext}$ holds for timestamps generated in any pair of TiS instances, we need to wait for more than the maximum clock divergence between any pair of TiS instances, i.e., $max(\{|\delta_{TiS_i} + \delta_{TiS_j}| + 1, \forall_{TiS_i,TiS_j} \in T\})$.

5.4.4.2 Discussion on TiS implementation

The TiS is a lightweight service that executes across multiple and potentially geographically distributed nodes (e.g., servers at the network infrastructure or cloud-based), allowing a replica to request a signed timestamp for a given operation. To avoid that a malicious replica requests multiple timestamps that are then used at its convenience to issue arbitrary operations at points in the past, these timestamps must be issued linked to a particular operation. To this end, when a replica requests a timestamp for an operation, it sends to the TiS a cryptographic summary of the operation, its identifier, and dependencies (line 20). The TiS will then issue a verifiable and trusted timestamp in the form of a tuple $(ts, signature_{TiS})$, where $signature_{TiS}$ is the signed operation summary (hash, including ts) and ts is the timestamp generated by the TiS. These timestamps can be validated by any replica using the certificate of the TiS, which is signed by the centralized server. In summary, the service can be implemented by leveraging well known timestamping protocols [165, 166].

The deployment of the timestamping service is an interesting research question on its own. The lightweight nature of the TiS, in contrast to full application servers, makes it easy to deploy and scale. We envision the following scenarios: $i)$ collocated with the application's centralized service; $ii)$ geo-distributed at multiple cloud points of presence; $iii)$ at edge locations such as ISPs and 5G towers; or $iv)$ on client devices within Trusted Execution Environments.

A client that cannot contact a TiS instance directly or through other replicas must stop generating operations (and possibly notify the user). Given the different alternatives to deploy the TiS, we expect this situation to be rare (and less frequent than unavailability in client-server architectures).

On synchronizing TiS instances TiS instances should execute a clock synchronization protocol (e.g., NTP [167] or PTP [168]), whose precision will impact $TWait$. This work does not focus on synchronizing the clocks of TiS instances, and we assume that: (i) TiS clock synchronization cannot be tampered by clients [169–171]; (ii) under normal conditions and in most common deployments scenarios, $TWait$ will be in the order of single-digit milliseconds [172–176]. We note that even if $TWait$ is up to double-digit milliseconds, it still allows for faster progress than resorting to a client-server model.

In most applications only specific subsets of clients interact with each other, for example when a game divides players in rooms or users are collaborating on separate documents. This has as

a direct result that TiS instances need not be synchronized globally, and neither has $TWait$ to be calculated among all globally available instances. For example, a multiplayer game providing multiple regions (e.g., Europe, Asia, and Americas), can trivially partition the TiS instances ensuring synchronization is easier and $TWait$ can be lower – not only due to less connected instances, but as the overall distances are smaller.

5.4.5 Secure extended causal consistency

Applications that use **Secure Extended Causal Consistency** will be notified when an operation is delivered in an order that is correct to the system's observed causality, but that does not respect the external observer's order, $<^{ext}$. The system itself does not reject or re-order any operation – it is up to the application to use this information to perform suitable actions in accordance with the application logic. A simple example, discussed in Section 5.3.3.1, is a chat application where the user-interface can be updated with this information for an intuitive outcome.

As reported in other systems [66, 67], using this information can lead to complex application code – it typically requires applications to manage metadata and merge and recompute the final state, while guaranteeing that all replicas converge to the same state which itself should provide an intuitive result for the user. To help programmers, our prototype includes a set of CRDT data-types that take advantage of this information. We provide semantics that are not typically available in systems that provide causal consistency, some examples include:

- a `list` object where concurrent insertions on the same position are ordered by insertion time – e.g., this can be useful in a chat application;
- a `map` with a first-to-write-wins conflict resolution policy – e.g., this can be useful for ordering bids of the same value in an auction.

If using this information is too complex, or stronger guarantees are required (e.g., invariants), the application programmer should resort to Secure Eventual Linearizability, in which operations are executed according to the external observer's order.

5.4.6 Secure eventual linearizability

Secure eventual linearizability is implemented by executing the operations in timestamp order, thus guaranteeing that each replica executes the received operations according to $<^{ext}$, as defined by the timestamps obtained from the TiS. As operations can be received out-of-order, our implementation uses an undo-redo execution model – when an operation is received, operations that were executed out-of-order are undone and reapplied in the correct order – see Figure 5.4. Although this consistency model only uses the timestamps, the metadata of an operation includes its dependencies (line 18) as this information is often useful for managing the application state and their hashes (line 19) are used to efficiently detect sibling operations.

With this approach, the final outcome of an operation is only determined after its execution order becomes stable, i.e., when no operation with a smaller timestamp can arrive – until then,

the operation's execution is considered tentative (similar to previous systems providing eventual linearizability [64, 118]). Our prototype includes an optional mechanism for establishing the stability of operations, that works as follows. The server periodically defines the stability timestamp, t_s (up to some seconds in the past of the current clock at the server), and determines the set of operations that become stable: the operations it has received with a timestamp t, such that $t \leq t_s$. This information is broadcasted to all clients (we note that it is only necessary to propagate the identifiers of operations that have no operation that happened after – clients can compute causal dependencies). The operations with a timestamp $t \leq t_s$ that are not included in the set of stable operations are undone forever. When a replica finds out that an operation it has generated is in this situation, it may resubmit the operation by first obtaining a new timestamp from the TiS.

5.4.7 The need for a server: reliability

For implementing the proposed secure consistency models, it is also necessary to guarantee that all correct replicas will receive all valid operations. In our propagation model some clients are responsible to propagate operations of other clients, hence we must ensure eventual delivery at the client-to-client level.

We build on the mechanism to detect sibling operations to achieve this property (Eventual Sibling Detection in Section 5.4.2). Recall that the centralized service periodically disseminates a summary of its state, including a hash of the last observed (concurrent) operations. If after some time, a replica sees that its own operations have not been reported by the centralized service (either as a recent operation or a dependency of a recent operation), the replica contacts the server directly to synchronize its state. The replica also contacts the server when it does not receive an updated server report for some time.

This approach, besides ensuring that network anomalies among replicas do not impact reliability, also prevents Eclipse Attacks, where malicious replicas attempt to create a barrier between correct replicas. When this happens, correct replicas will communicate resorting to client-server-client interaction. Several decentralized protocols for providing secure broadcast [144, 177, 178] and preventing Eclipse attacks [179, 180] have been proposed in literature and could have been adopted to provide such guarantees without resorting to the centralized service.

Since in our system the server is used to connect to the peer-to-peer network, as well as to obtain a certificate for new replicas, it also acts as a protection against malicious replicas appearing with multiple identifiers (Sybil attacks [181–184]) or from reappearing with a new identifier after a proof-of-misbehaviour has been generated.

A correct replica that detects any form of Byzantine behaviour immediately disconnects from the malicious one, never to connect again. If a malicious replica misbehaves with all correct peers, it is eventually removed from the peer-to-peer network and limited to interact through the server. If a proof can be obtained, then the correct replica submits the proof to the server which propagates a revogation on that replicas' id and certificate pair. This is similar to the eviction protocol in Legion, leading Byzantine replicas to have no effect on correct ones.

A malicious replica can follow the protocol but submit invalid operations according to the

application logic. As each operation includes its dependencies, other replicas and the server can use this information to verify the validity of the operation. The complexity of this process depends on the application. When verification is complex (e.g., requiring recomputing the state of the replica when the operation was issued), this can be done in the background by the server.

This opens the opportunity for attacks, but there is little incentive for a single rational client to emit an invalid operation, as it will be expelled from the system and it will not be able to reenter. However, when users compete in teams, if the gain/loss ratio is high enough, there might be an incentive to sacrifice some members by issuing an invalid operation that will only be detected much later – simple solutions such as banning the whole team can be applied but such decisions depend on the application. This is an open problem that is not specific to our decentralized approach, but is inherent also to geo-replicated (server-side) systems running under weak consistency.

Our system effectively mitigates attacks on the consistency model but it is not able to completely eliminate all of the attacks that have been discussed. Although eventual delivery is enforced, as we show in the evaluation (Section 5.5.6.2), it still requires a 50% ratio of correct replicas to ensure interactions among correct replicas are not delayed. This is in line with the work on BAR-tolerant protocols [144].

5.5 Experimental evaluation

The evaluation demonstrates that security guarantees can be provided even when clients replicate data and communicate directly among each other. The highlights of our results are the following:

- Our secure consistency models provide considerable reduction in user-to-user latency when compared to client-server, with interactions between nearby clients exhibiting the larger improvements (Section 5.5.2).
- Our solution scales to a large number of clients with a modest increase in latency (Section 5.5.3).
- Low-latency is crucial for effective Eventual Linearizability since its execution is affected by staleness (Section 5.5.4).
- Deployment of the timestamping service should consider the geographic distribution of clients, ideally by exploiting dynamic placement in client vicinity (Section 5.5.5).
- The proposed algorithms disallow discoverable Byzantine misbehaviour, mitigating the effect of such behaviour on correct clients (Section 5.5.6.1). Rational clients (remaining undiscovered) can impact correct clients most when they are a majority of the replicas (Section 5.5.6.2). Resorting to the TiS and the centralized component mitigates the actions of colluding rational replicas (Section 5.5.6.3).

5.5.1 Experimental setup

5.5.1.1 Prototype

Our prototype allows applications running in browsers to use the proposed secure consistency models, namely secure causal and secure extended causal consistency, and for applications that require stronger guarantees, secure eventual linearizability (EvtLin).

Our prototype extends Legion (Chapter 3), a causally consistent system that includes a library of CRDTs for merging concurrent updates. The secure models were implemented by adapting Legion's propagation mechanisms, which were susceptible to the attacks discussed in Section 5.2. Additionally we extended the CRDT library to implement the CRDT semantics leveraging realtime timestamps as defined in Section 5.4.5. A non-secure Legion version is used in the evaluation as a comparison, providing unsecured causal consistency (Causal unsecured).

The prototype was written in JavaScript, with test applications running in browsers or as NodeJS applications. The latter were primarily developed to avoid the overhead of graphical user interfaces in our experiments. Our prototype has an abstract communication layer that enforces the required security abstractions and FIFO channels on top of the network layers in both environments: WebSockets (TLS) in NodeJS and WebRTC (DTLS) in browsers. The implementation of our algorithms uses SHA-256 and RSA-2048 for hashes and signatures, relying on the forge library [93].

5.5.1.2 Baseline

User-facing applications typically rely on client-server architectures. To serve as a baseline, we added support in our prototype for strict-serializability, using client-server interactions with a single master (Client-Server in the plots). Switching from EvtLin to Client-Server requires changing a single configuration variable, ensuring a fair comparison as there are no hidden language overheads or implementation optimizations that can affect results.

When using Client-Server, replicas communicate with the server using standard techniques: WebSockets over TLS with an initial client authentication (user login). Unlike the secure models, operations are transmitted over the secure channel without any further cryptographic processing. The server verifies if an operation is valid before propagating it to other replicas, by checking if it can be applied in the current state.

5.5.1.3 Application

For the evaluation, we created a game where players move on a 2D map with obstacles and gather coins to obtain points. Initially a coin is located at the center of the map, and players are spawned near the edges. When a player touches the coin, it gains a point and a new coin is spawned at a random location on the map (computed deterministically from the hash of the catching replica's operation). The movement and gathering operations can be verified based on the last movement from that player (this gives a verified trace as players start in deterministic positions).

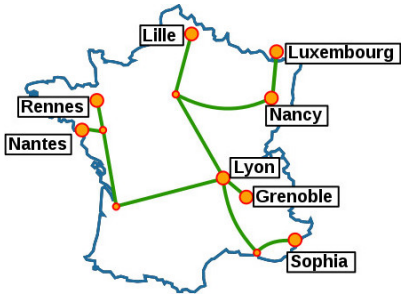


Figure 5.7: Grid5000 clusters map (France).

5.5.1.4 Deployment Setup

Our experiments were run on the Grid'5000 [185] platform (G5k), complemented with AWS EC2. Clients instances (1792 by default, with 256 at each cluster) are spread over G5k's clusters (except Grenoble), shown in Figure 5.7. Each client has one CPU core and 1 GB RAM available. Unless stated otherwise, the server runs in Ireland (AWS EC2: t2.xlarge, 4vcpu, 16GB) and the TiS instances are deployed in Ireland, Paris, and Frankfurt (also on AWS EC2).

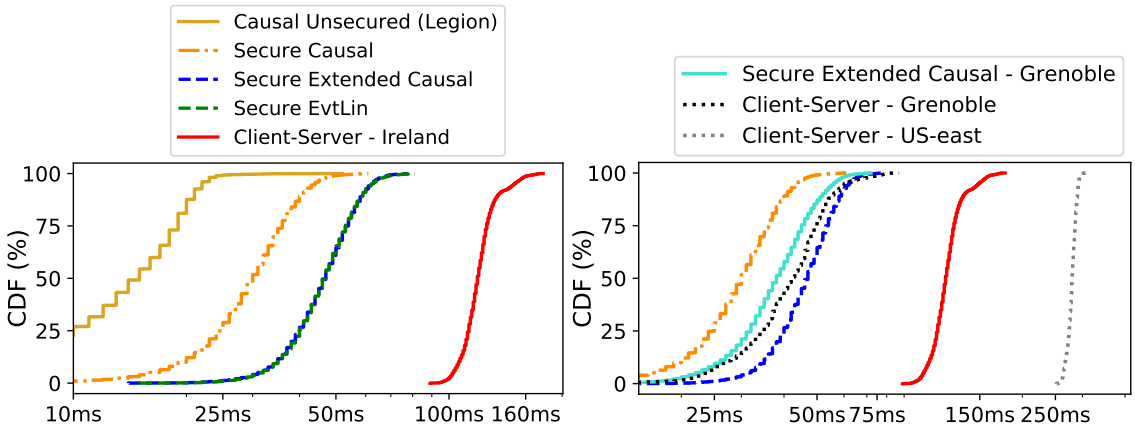
With this setup, clients and servers are distributed across different geographic locations, with varying latency among clients and varying latency from clients to the servers.

5.5.2 Latency evaluation

Client-to-client latency measurements (of Figure 5.8) consider the time since the call to create a new operation (*NewOperation*) until its reception at each client. All clients generate, apply locally, and propagate a new operation every 5 seconds. This leads every *client* to receive, verify, and apply $1792/5 = 358.4$ operations per second. A typical (single room) multiplayer game has significantly less players and operations being executed.

Client-to-client latency Figure 5.8a reports the latency (as an empirical Cumulative Distribution Function) observed for the secure variants of our system compared with both the unsecured and client-server variants. Among the secure variants, Secure Causal provides the lowest latency between clients as new operations only have to be signed before being propagated (and verified upon reception). Extended Causal and EvtLin require a client to obtain a timestamp from the TiS for each operation, leading to an additional delay before propagation. As a consequence, results are very similar (due to this reason, we omit the results of EvtLin in Figures 5.8b, 5.9, and 5.10). Client-Server presents the highest latency due to the time required for operations to be sent to the server and back to other clients. When comparing with the unsecured implementation of causal consistency, we can observe that the secure variants exhibit additional latency – this is due to the use of cryptography and, when required, communicating with the TiS.

Impact of server and TiS location Figure 5.8b shows the effect of server and TiS location. We considered three server locations: local to G5k (in Grenoble), in Ireland, and in the US (east). For servers in AWS locations (Ireland and US), latency of Client-Server increases as the latency to



(a) Default variants of the application.

(b) Additional server and TiS deployments.

Figure 5.8: Client-to-client delivery latency (ms) with clients spread over Grid5000.

the data centers increases, as expected. For the Grenoble server, latency becomes better than for the Extended Secure model using the TiS at AWS locations, as obtaining the timestamp from the TiS is more expensive compared to sending the operation to the Grenoble server (located close to the center of the G5k network). With a TiS server in Grenoble (S. Extended Causal – Grenoble), the latency of the Extended Secure model became slightly better than that of Client-Server with a Grenoble server, as it is faster to propagate messages throughout the client-side overlay network than having the server responsible for sending every message generated at each client to every other client. The Secure Causal model is the only one that is not impacted by the latency to the server/service, as operations are propagated through the overlay without requiring any coordination.

Impact of data-locality Figure 5.9 shows the difference of latency for operations received from nearby clients (running in the same G5k DC) and from remote locations (running in remote DCs). As expected, the Client-Server solution shows practically no difference, as operations always have to be propagated through the server, even for operations from nearby clients. For secure consistency models (and Causal Unsecured), there is a noticeable difference between operations from local and remote clients, which results from the underlying latency (distance) among clients. Secure Causal Consistency (and unsecured) has considerably lower latency for local propagation compared to the secure variants as these depend on the latency to reach the closest TiS instance to generate operations.

Discussion The results suggest that the proposed decentralized secure consistency models are preferable to traditional cloud-based solutions when the latency among clients is lower than the client-server-client latency. We expect that this is the common case in cloud-based deployments, where an application is deployed in a small number of data centers.

When servers can be deployed close to clients, leading to client-server latency lower than the latency between distant clients, the decentralized models still exhibit considerable advantage for

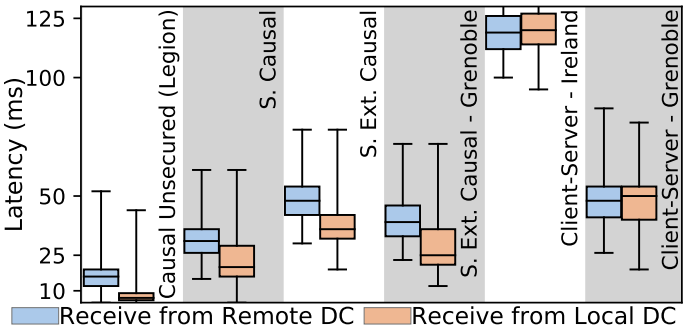


Figure 5.9: Locality effect of secure consistency models.

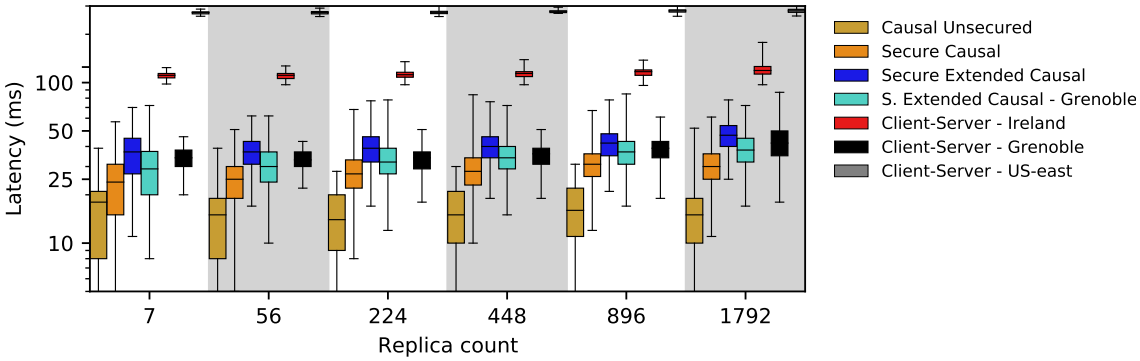


Figure 5.10: Latency results when varying the number of client replicas.

interactions among nearby clients. This makes the proposed models attractive for applications dominated by interactions among nearby clients, such as location-dependent information sharing and augmented reality games.

If an application is dominated by interactions among distant clients, and needs to use a consistency model stronger than Secure Causal Consistency, the advantage of the decentralized models is reduced (and depends on the location of TiS servers). In this case, the additional complexity of the proposed models might not be worth the benefit. We note that the cost and overhead of deploying and maintaining a fleet of full application servers is much higher when compared with TiS instances, which should also be taken into consideration when deciding which approach to adopt.

5.5.3 Scalability

Figure 5.10 reports latency between clients as a function of the total number of clients. The results show that, unlike the Causal Unsecured model, the penalty in latency grows modestly with an increasing number of clients for all secure consistency models and for the Client-Server solution. The reason for this lies on the processing overhead in the replicas for the decentralized models (cryptography) and in the server for Client-Server (message propagation). We have not scaled beyond 1792 clients due to lack of available hardware.

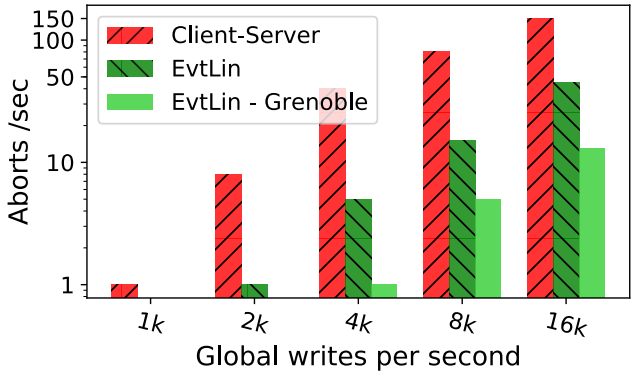


Figure 5.11: Effect of operating on stale views using EvtLin.

5.5.4 On data staleness

Games typically use some form of extrapolation to show expected object positions based on current object movement, but this can lead to objects jumping on the screen when the extrapolated value is stale due to actions of other players (which did not arrive before calculating expected positions). Decisions are thus made (by the players) on stale data. In this experiment we measure the effect of data staleness (due to propagation latency) on both EvtLin and Client-Server.

To increase contention, we used the game application but reduced the play-area and enabled player collisions. When clients operate over local stale data they possibly grab a coin concurrently with other players. Using the Client-Server communication model, only the first grab operation to arrive at the server is accepted, with all concurrently created operations being aborted. In EvtLin, concurrently created operations must be undone and reapplied in the correct order when operations are delivered. This is because locally at each client the coin may still exist, and the client can successfully generate and execute a coin grab operation. When these operations are ordered this leads to all but one to become aborted.

We vary the number of operations per second to tune the amount of conflicts that occur. The results, presented in Figure 5.11, show that with few operations the contention among players grabbing the coin is low (few aborts in Client-Server). As contention increases, due to the increasing number of operations per second, staleness increases which leads to additional aborts. EvtLin has significantly less aborts as clients operate over fresher data when compared with Client-Server. This is confirmed by the even lower number of aborts in EvtLin-Grenoble, which, as discussed in the previous sections, further reduces the latency to propagate operations.

The take away from these experiments is that enforcing application invariants on client-side replicas, with a low number of aborting operations, requires avoiding data staleness as much as possible. This means a scheme has to be chosen which lowers, as much as possible, the propagation latency of updates among clients.

5.5.5 Impact of TiS deployment

Figure 5.12 shows the effect on latency of adding TiS instances closer to clients. Clients are scattered throughout the G5k clusters and continuously issue operations. Initially there is a

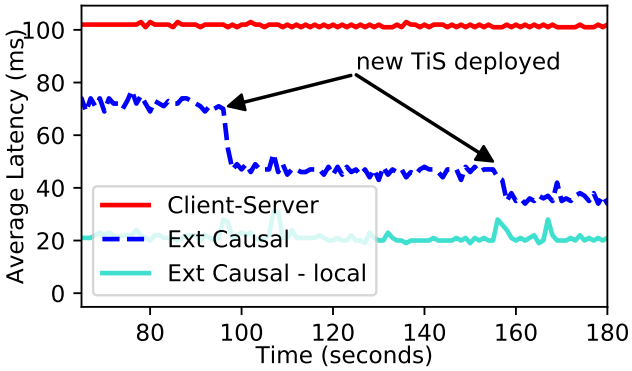


Figure 5.12: Latency when adding TiS servers (mean value of 1s windows).

single TiS instance in AWS EC2 Frankfurt. At the 90 and 150 second marks we deploy additional instances in AWS EC2 Paris and in G5k Grenoble, respectively. We also report the average latency for Client-Server (with the server in AWS EC2 Ireland) and Extended Causal with a TiS instance deployed at each G5k cluster (Ext Causal – local).

The results show that adding new TiS instances can have a significant positive effect in the latency experienced by clients. This effect is more noticeable when the TiS is closer to clients. This process can be done in an autonomic fashion by, for instance, leveraging work on database management on auto-tuning and commissioning replicas [186, 187]. As TiS instances are lightweight, it is practical to deploy them in the edge of the network, placing them very close to the clients.

5.5.6 Impact of rational and arbitrary behaviour

We now discuss how client misbehaviour affects latency.

5.5.6.1 Discoverable (Malicious) behaviour

Figure 5.13 reports the latency observed by correct and incorrect clients in a scenario with multiple malicious clients, for which a proof-of-misbehaviour can be produced. In this experiment, incorrect clients (one third of all clients) follow the protocol up to the 27s mark, at which point they start propagating incorrect messages (trash, incorrectly signed, and tampered contents). This behaviour makes correct clients disconnect from such misbehaving clients. The latency perceived by incorrect clients grows beyond the round-trip-time to the server (Server RTT), as incorrect clients need to communicate through the server. Correct clients continue experiencing low latency, as they restrict peer-to-peer interactions among them. Our experiments in varying the amount of malicious clients (5% to 95%) and secure consistency models (causal, extended causal, and eventual linearizability) all presented similar results.

5.5.6.2 Undiscoverable (Rational) behaviour

Rational clients can selectively delay operation propagation to other clients or the server, while being fast to disseminate to other rational clients (i.e., a form of collusion). We focus on delaying

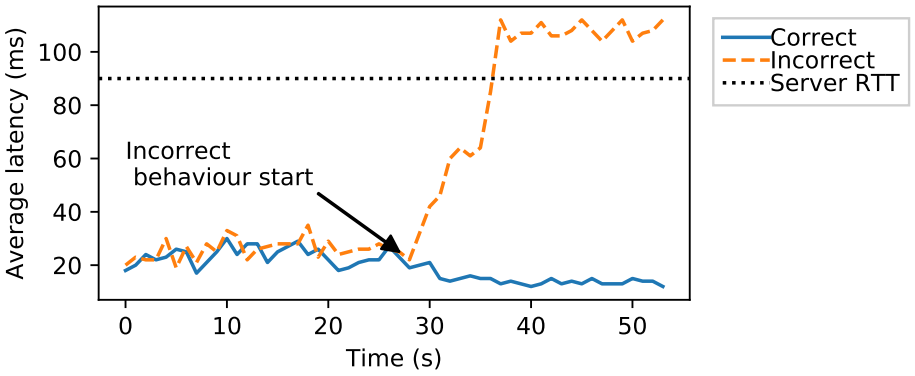


Figure 5.13: Effect of discoverable incorrect behaviour on latency.

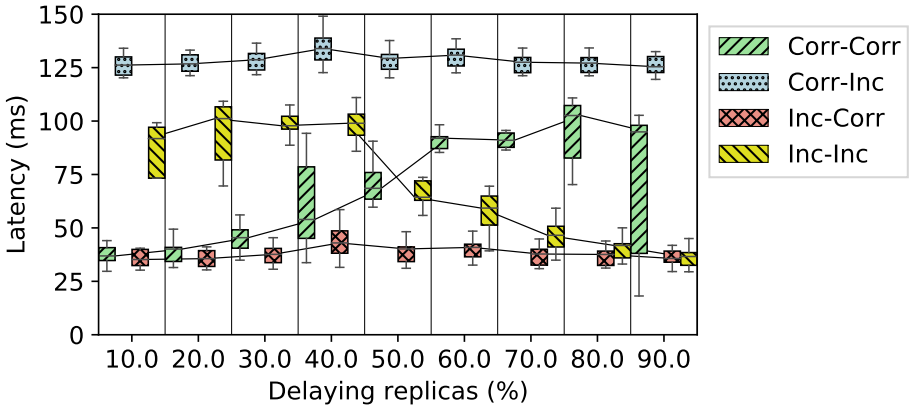


Figure 5.14: Effect of rational replicas on latency by selectively delaying propagation.

propagation, as other attacks either have the same or smaller effect, or lead to the discovery of such actions. Figure 5.14 reports the latency with an increasing fraction of rational clients colluding to delay messages sent to correct clients. The results show that rational clients always observe operations of correct clients fast (Inc-Corr) and correct replicas observe operations from rational clients with a high delay (Corr-Inc).

Interestingly, correct clients start to perceive a noticeable effect among operations generated by themselves (Corr-Corr) when the amount of colluding rational clients grows over 50%. Similarly, at that point, the latency perceived by rational clients for operations issued by other rational clients (Inc-Inc) significantly drops. This is related with the probability of a correct (respectively rational) client having another correct (respectively rational) client as a direct neighbour, which depends on the fraction of rational clients, as neighbours are mostly selected at random. This implies that in our system, correct clients will benefit the most as long as they are the majority of participants (this was observed already in [144]).

5.5.6.3 Attacking the speculative execution of eventual linearizability

As discussed previously, EvtLin is significantly impacted by local data staleness when creating new operations (Figure 5.11) – delayed operations can lead to increased data staleness and force frequent undo/redos. We designed an experiment to measure the capacity of rational clients

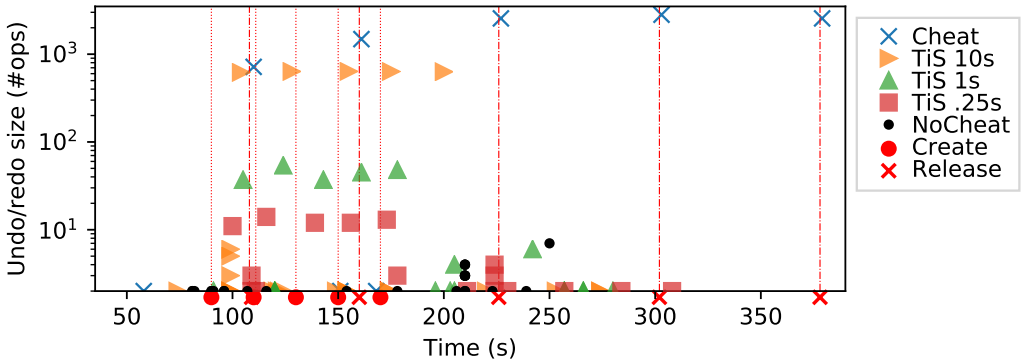


Figure 5.15: Effect on EvtLin undo/redo sizes as rational clients hold back operations.

to disrupt a system using EvtLin by delaying operation propagation. In our experiment clients manipulate a bounded counter concurrently.

The counter bounds and increment/decrement values were selected to force contention on client operations – we limited the counter's the value to be between 0 and 10, and all clients continually generate operations to decrease the counter by 1. If a client observes the counter with a value of 0 it issues an operation to increment the counter by 4. As clients apply their operations locally when generating the operations, these might be set to be aborted later if, when re-ordered, they would violate the invariant on the counter value.

Figure 5.15 shows the moment at and the number of operations undone/reapplied at each client. As baseline, we show the results when no client delays propagation (NoCheat) – in this case, contention over the counter already leads to some undo/redos. For disrupting the system, a rational client obtains from the TiS timestamps for 5 operations (identified by Create at the x axis) propagating these operations much later (at the following Release). This leads to a high number of undo/redos (Cheat), as a large number of operations with greater timestamps have to be undone/reapplied by each client.

One way to address this issue would be to use the servers' stability mechanism (Section 5.4.6). For greater effect, stability should be chosen as close as possible to the server's actual time. Not only does this lead to clients which have high latency to have their operations being rejected more often, it effectively returns to client-server communication model as rational replicas can trivially delay correct replica's operations just enough as not to be included in time, leading them to communicate with the server directly to have their operations included.

We instead investigated if the problem could be addressed by having TiS instances storing timestamped operations. In this case, clients poll the TiS for operations that they might be missing – we show the results for a polling time of 10s, 1s, or 0.25s (Figure 5.15). The late release of an operation does not lead to a large number of undo/redos, as clients obtain the (on purpose) delayed operations directly from the TiS service, showing that keeping operations at the TiS mitigates this attack. Naturally, enabling clients to poll the TiS more frequently increases the mitigation effect at the expense of additional resource consumption.

5.5.6.4 Discussion

It is clear that rational replicas cannot be removed altogether – users can always communicate with means external to the system to gain a latency benefit. This is also true in the client-server model, as nothing disallows such replicas from running additional software. Our approach in reducing inter-replica latency mitigates this by letting correct clients observe operations from other correct clients with low latency.

5.5.7 Discussion on network performance

When running the experiments we also measured the network usage and concluded that, when using our system, the server makes a lower use of the network (about one third) when compared with the classic Client-Server model.

This is in line with previous results of Legion (Section 3.3.2.4) as just as in Legion, our system elects a fraction of clients to maintain communication with the server, having the remaining clients only interacting with other clients. This not only enables the server to support a larger number of clients but also reduces server bandwidth usage.

Comparing secure and non-secure consistency variants, there is an overhead when a client bootstraps since cryptographic keys must be generated and exchanged with the server to obtain a leased certificate – this is a one-time cost for each replica (for the duration of the lease). The overhead on the network due to secure mechanisms among clients depends on the application's message sizes. Experiments using small messages among clients incur an increase of bandwidth usage of up to $\times 2.5$, due to the additional signatures and timestamps, which can have a considerable size for small data sizes (e.g., 256 bytes for signatures with key sizes of 2048 bytes). This additional cost tends to $\times 1$ (minimal overhead) when the size of application messages increases.

5.6 Related work

In this section we explore related work with respect to the security aspects of the system. A general overview of the research context can be found in Chapter 2.

5.6.1 Storage systems

Byzantine fault tolerance (BFT) is designed to cope with malicious replicas that can perform arbitrary actions [147, 188]. In the context of replicated systems, existing solutions typically focus on strong-consistency models among a limited numbers of replicas [145–147]. Works on secure causal BFT define secure causal in addition to SMR, and require a total order among replicas with a causal order for interacting with the system [150–152]. Our work addresses a different problem, requiring no total order, leading to the possibility of using weaker consistency models. This leads to different requirements and properties for the proposed models and algorithms that were used.

Approaches based on blockchain also enforce a final unified total order on all operations but additionally impose high latency on writes due to the use of majority voting algorithms [153–156].

Such systems are also sensitive to a majority of replicas acting against the rest, which is often the case in a game setting. For example, in games where the current leaderboard is clearly divided into a few winning users and the remaining clients (which are losing) a co-operating majority is easy to accomplish. Our work practically mitigates the effects of a majority overtaking the peer-to-peer network by use of a trusted fallback – the central server.

A very important aspect that we accomplished, which the referenced works do not address at all, is to allow for islands of replicas to evolve concurrently.

Bayou [64] uses a form of eventual linearizability by relying on primary replicas to assign global orders among all operations. An operation is tentatively executed in each replica until its global order is defined. Note that this permits major reordering of operations – an operation o_1 created before operation o_2 (in realtime), may later be ordered as $o_2 < o_1$. Our implementation of eventual linearizability orders operations based on the timestamps assigned by the TiS service, effectively mitigating this issue. Furthermore, our algorithm provides protection against malicious replicas.

5.6.2 Peer-to-peer middlewares

Most peer-to-peer message propagation and content delivery schemes [189–191] only allow clients to share static content and do not cope with simple man-in-the-middle attacks to provide malicious or fake content.

Several decentralized protocols for providing secure broadcast [144, 177, 178] and preventing Eclipse attacks [179, 180] have been proposed in literature and could have been adopted to provide some of the guarantees we provide resorting to the centralized service. Since in our system the server must be used to connect to the peer-to-peer network, it also acts as a protection against malicious replicas appearing with multiple identifiers or from reappearing (Sybil attacks [181–184]).

These protocols led to the creation of many peer-to-peer systems which were designed with some thought to malicious behaviour in replicas, for example:

The Atum [192] middleware offers resilient group communication for large and dynamic networks. Although using cryptography (signatures and MACs) to authenticate messages, there is no consideration for replicas actively attempting to subvert the system itself.

BAR Gossip [144] proposes an algorithm for reliable message dissemination in the BAR model by adopting a peer-to-peer gossip protocol. The BAR model introduced explores the notions of Byzantine, Altruistic, and Rational replicas.

S-Fireflies [193] is a system for data dissemination robust to Byzantine faults. In contrast to our system, it relies on global knowledge of all replicas at all times.

FlightPath [194] is a reliable peer-to-peer data streaming application that tolerates up to 10% of replicas behaving maliciously and the remaining peers acting rationally. The focus

is on having a single designated source replica while others are solely used for reliable dissemination.

Our work is complementary to these works and could rely the proposed solutions to achieve reliable dissemination.

5.6.3 Secure hardware

Trusted Execution Environments can be used locally at the client to provide additional security guarantees for user-executed code [161, 195–197]. Using trusted hardware at each client possibly provides a secure execution environment for a given application or web-page [198] or even disallow unauthorized users from accessing sensitive information [199]. TrustJS [198] is comparable to our use-cases as it allows for JavaScript to run, in a secure-fashion, at the client-side.

These works focus on protecting the integrity and confidentiality of code and data of, for example, a single player game (i.e., DRM), whereas we focus on direct user-to-user interaction (i.e., multiplayer games where players themselves interact in a peer-to-peer fashion).

Although our proposed models could leverage such hardware to be implemented, the modules are not widely available and recent works [200–205] show possible attacks to these modules.

These solutions also do not suffice to counter delaying propagating (to progagate messages they have to be exported from the trusted hardware for propagation) or colluding to circumvent the guarantees of causal consistency (as the data itself has to be shown to the user for it to be used, and the user can run additional software or hardware). We address these issues with the use of a lightweight and highly-available trusted service, controlled by the application provider.

In our work we also do not want to restrict an application to require the existence of a TEE, but to only use it when needed and if available at the client-side. In cases where a trusted component is indeed necessary, we want to be able to choose from various options depending on what is available and acceptable: i) use the locally existing TEE; ii) use an available TEE existing at a neighbouring peer; iii) use an available TEE or trusted node at the network infrastructure (e.g., 5G [206]); iv) use trusted instances running at multiple cloud points of presence (the lightweight nature of TiS instances allow for much wider deployments – see Section 5.4.4.2); v) use the (trusted) server if all else fails or when it is simply the only acceptable choice latency wise.

5.7 Final remarks

This chapter addresses the impact of replica misbehaviour on the guarantees of available consistency models, namely causal consistency and eventual linearizability.

We analyzed possible replica misbehaviour based on causal consistency as, depending on the application, a client may gain an unfair advantage by executing a different type of misbehaviour. We derived secure consistency models that prevent different types of misbehaviour and, by selecting the secure consistency model that prevents such misbehaviour, an application may prevent such attacks. We proposed techniques for implementing these models and implemented them in

our prototype, a middleware providing a peer-to-peer data-service with secure variants of causal consistency and eventual linearizability.

Our evaluation shows that the proposed algorithms to enforce the secure consistency models impose a modest overhead when compared with unsecured variants, while keeping much lower user-to-user latency and a reduced server load when compared to solutions relying on client-server architectures. Additionally, our algorithms effectively mitigate the effects of malicious clients (even when colluding). Reducing latency among correct clients has an additional advantage: it reduces the benefit that colluding replicas could have by using fast (peer-to-peer) connections external to the system – this is an unsolvable problem in pure client-server communication models.

The fact that applications have different requirements, and preventing different types of misbehaviour leads to different overheads, justifies the interest of providing different consistency models.

Chapter 6

Client-side partial replication

6.1 Introduction

In the previous chapters, Chapter 4 and Chapter 5, the definitions on causal consistency and supporting algorithms consider causality as defined in a system where each individual replica locally maintains a copy of all objects – full replication.

In this chapter we explore partial replication. We start by defining consistency models with respect to individual objects, with the overall goal to provide partial replication. We explore a system model with client-side replicas supporting weak consistency models, in particular, we focus on causal consistency.

6.2 Preliminaries

6.2.1 System model

Objects The system is composed by a set of objects Obs.

Replicas We consider a system comprised of $\mathcal{R}$ replicas. Each replica $r \in \mathcal{R}$ has an associated unique identifier, $r.id$. Each replica also has an associated clock, r_c, which starts at 0 and is incremented with each operation generated by that replica.

Operations An operation o encodes a set of effects on an object. Each operation has an associated unique identifier, composed of the identifier of the originating replica and its operation count (clock), i.e., if r generated o, $ID(o) = (r, C)$ where $C = r_c + 1$ at the time of generation (note that r_c is updated).

Each operation o is generated respective to an object $ob = Object(o)$, i.e., with O as the set of all operations, $\forall o \in O, Object(o) \in Obs$.

Metadata Objects and operations can have associated metadata, $ob.metadata$ and $o.metadata$. Thus, an operation is defined as $o = (oid, data, metadata)$, with oid defined as $oid = ID(o)$, $data$ as any data releveant for the execution of the operation (such as operation

name and parameters), and $metadata$ any information required by the system's mechanisms to ensure correct behaviour.

Failures We consider two types of failures:

Temporary or recoverable failures These failures consiste of networking and replica failures which are recoverable. Eventually, the replica recovers without losing any data.

Permanent failures In a permanent failure a replica fails permanently without recovering. This is a common occurence when considering client-side replicas, as users may uninstall applications, refresh their browsers, or somehow destroy their devices.

For this work we assume that replicas do not behave arbitrarily – all replicas follow system specification without any deviation.

6.2.2 Consistency models

We start by defining a generic model for consistency that can be instantiated with different relations among operations.

Definition 6.1 (Model). *A model $\mathcal{M}_{\lhd}$ is the set of relations among distinct operations following a specific set of rules as defined by $\lhd$, i.e., $o_1 \lhd o_2 \Rightarrow (o_1, o_2) \in \mathcal{M}_{\lhd}$.*

A model is transitive ($[(o_1, o_2) \in \mathcal{M} \wedge (o_2, o_3) \in \mathcal{M}] \Rightarrow (o_1, o_3) \in \mathcal{M}$) and acyclic ($(o_1, o_2) \in \mathcal{M} \Rightarrow (o_2, o_1) \notin \mathcal{M}$).

Altough similar to happens before of Lamport [49], this generalizes to other models that have relations among operations, such as explicit dependencies (Section 6.5.5.1) and total order (Section 6.6).

6.2.3 Serialization

Each replica r applies the set of operations it has received, O_r, in its local state. As each replica may not have received all operations, we can assume the following relation: $O_r \subseteq O$.

We define the local state of a replica r as the execution of a serialization, S_r, of all operations in O_r to an initial state. A serialization is defined as follows:

Definition 6.2 (Serialization). *We say that for a set of operations R, S is a serialization $(R, <)$ of all operations in R if it is a total order of R.*

To be able to define consistency models, a stricter definition is required:

Definition 6.3 (Correct serialization by model). *We say that for a set of operations R, $S \because \mathcal{M}_{\lhd}$ is a correct serialization $(R, <_{\lhd})$ of all operations in R according to $\mathcal{M}_{\lhd}$, iff $\forall o_1, o_2 \in R, [(o_1, o_2) \in \mathcal{M}_{\lhd} \Rightarrow (o_1 <_{\lhd} o_2)]$.*

In other words, $S \because \mathcal{M}$ means the serialization S conforms to any restrictions imposed by $\mathcal{M}$.[1]

[1] $\because$ is read as *by* or *because*.

6.2.4 Distributed execution

A distributed execution D is the set of all serializations at every replica, i.e., $D = \{S_r : r \in R\}$.

A distributed execution is valid towards $\mathcal{M}$ iff every replica is able to execute all operations by following $\mathcal{M}$:

Definition 6.4 (Valid distributed execution). *A distributed execution D is valid towards $\mathcal{M}$ iff, for every replica r, O_r is executed as $S_r \because \mathcal{M}$, i.e., $\forall S_r, S_r \because \mathcal{M} \Rightarrow D_{\mathcal{M}}$.*

6.2.5 Possible serializations

At any point each replica may have received a set R of operations, such that $R \subseteq O$. As each replica may receive a different R, and the operations in R can be received in different order (and possibly serialized in a different order), we can define the valid set of serializations (or executions) of O under $\mathcal{M}$, $\mathcal{S}(O, \mathcal{M})$, as such: $\mathcal{S}(O, \mathcal{M}) = \{(R, <) : R \in \mathcal{P}(O) \wedge (R, <) = S \because \mathcal{M}\}$, i.e., the set of all possible serializations of any subset of O which are valid under $\mathcal{M}$.

6.3 Partial replication

6.3.1 Interest set

To introduce partial replication we assume each replica has a set of objects it is interested in. Each replica replicates all objects in its interest set.

Definition 6.5 (Interest set). *The interest set I_r of a replica r is the set of objects the replica replicates, such that $I_r \subseteq Obs$.*

Definition 6.6 (Full replica). *A replica r is a full replica iff $I_r = Obs$.*

Definition 6.7 (Partial replica). *A replica r is a partial replica iff $I_r \subsetneq Obs$.*

A partial replica r includes in its serialization S_r only operations that are respective to objects in its interest set I_r. As an example, we have two operations, o_1 and o_2 such that $(o_1, o_2) \in \mathcal{M}$ and $Object(o_1) = \triangle$ and $Object(o_2) = \square$. If replica r_1 has interest set $I_{r_1} = \{\square\}$, it ony stores in O_{r_1} the operation o_2 as $Object(o_1) \notin I_{r_1}$. In contrast, replica r_2 with interest set $I_{r_2} = \{\square, \triangle\}$ stores both operations, and $S_{r_2} = (o_1, o_2)$.

To ensure we can reason on serializations impacted by partial replication we explicitly show the non-serialization of operations at a given replica with $\neg o$. In the previous example, the serialization of O_{r_1} ($O_{r_1} = \{o_2\}$) becomes $S_{r_1} \because \mathcal{M} = (\neg o_1, o_2)$. Altough $\neg o_1$ is shown explicitly, only o_2 has an effect of the replica's state.

The goal of partial replication is not only to ensure replicas do not store information they have no interest in, but also on reducing network overhead – not every replica should receive every operation. This leads to the following definition:

Definition 6.8 (Genuine partial replica.). *A system supports genuine partial replicas iff any replica r only receives o and any metadata related to o iff $Object(o) \in I_r$. Additionally, a genuine partial replica only receives metadata related to object ob iff $ob \in I_r$.*

Intuitively, genuine partial replicas with intersecting but not equal interest sets may not be able to synchronize correctly due to missing information – we present the impossibility to provide for such replicas later in the chapter (Section 6.7).

The impossibility depends on the chosen consistency model – it works for any model that has relations among pairs of operations in different objects. For example the happens before relation (Definition 6.15) for causal consistency in many cases requires storing some metadata to be able to enforce the chosen model.

To ensure such relations may be kept, we define replicas which may receive, propagate, and store some part of any control information (metadata).

Definition 6.9 (Genuine partial replica with full control.). *A replica r is a genuine partial replica with full control iff r only receives o if $Object(o) \in I_r$. Additionally, r receives metadata for every $o \in O$.*

The previous definition enables two replicas to exchange metadata to ensure relations among pairs of operations are captured and stored, even if the bulk of the data (the operation contents themselves and additional metadata required to apply these operations) is not. The following restricts the shared metadata to only include that which is strictly necessary to ensure information on relations is tracked.

Definition 6.10 (Genuine partial replica with genuine control.). *A replica r is a genuine partial replica with genuine control if r only receives o if $Object(o) \in I_r$. Additionally, r may only receive metadata associated to operation o_i if $Object(o_i) \in I_r$ or if there exists an operation o_j such that $(o_i, o_j) \in \mathcal{M}$ and $Object(o_j) \in I_r$.*

6.4 Progress

Progress is loosely defined as the possibility of two replicas being able to synchronize their local states with each other, in a finite number of steps and without coordinating with other replicas.

We build on the notion commonly refered as partition recovery in the CAP theorem [44], i.e., at the end of the (network) partition recovery step, "the state on both sides must become consistent" and "there must be compensation for the mistakes made during partition mode". The first part can be seen as simply ensuring operations are shared among replicas, but the second part is what gives guarantees on the expected outcome for users.

To explain progress in our model we depict a distributed execution in Figure 6.1. There are three replicas, A, B, and C. The interest sets are $I_A = I_C = \{\triangle, \square\}$ and $I_B = \{\square\}$. There are three points in time depicted by τ_0, τ_1, and τ_2.

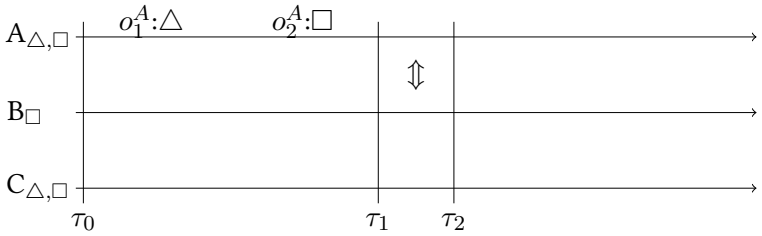


Figure 6.1: Diagram of a distributed execution.

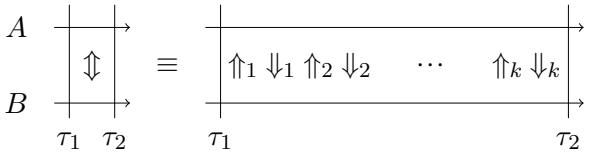


Figure 6.2: Diagram of a pairwise synchronization.

Somewhere between τ_0 and τ_1 two operations, o_1^A and o_2^A[2], were generated at replica A, with $Object(o_1^A) = \triangle$ and $Object(o_2^A) = \square$. Between τ_1 and τ_2 replicas A and B synchronize ($\Updownarrow$), i.e., at τ_2, B is consistent with A and B compensated for any changes A made in its local state, and vice-versa.

A synchronization ($\Updownarrow$) is detailed in Figure 6.2. It contains two points in time, τ_1 and τ_2 in which a finite number of messages are exchanged by two replicas. At the end of the exchange both replicas have serialized the same set of operations, with respect to their interest sets. In the example of Figure 6.1, at τ_1, $S_A = (o_1^A, o_2^A)$ and $S_B = (\,)$. At τ_2 (after $\Updownarrow$ between replicas A and B), $S_A = (o_1^A, o_2^A)$ and $S_B = (o_2^A)$. S_B will not contain o_1^A as $(Object(o_1^A) = \triangle) \wedge (\triangle \notin I_B)$. Depending on the mechanisms to ensure progress for the used model, replicas may need to exchange additional information to ensure the model's guarantees (as we will show later).

From here on we use the abbreviated notions $A \Updownarrow B$ or $A \Updownarrow_{\tau_1}^{\tau_2} B$. Formally, using Figure 6.2, $\Updownarrow$ can be defined between two replicas A and B as:

Definition 6.11 (Synchronization). *Two replicas, A and B, synchronize between the moments τ_1 and τ_2, $A \Updownarrow_{\tau_1}^{\tau_2} B$, iff $\forall o \in S_A^{\tau_1}, Object(o) \in I_B \Rightarrow o \in S_B^{\tau_2}$, and $\forall o \in S_B^{\tau_1}, Object(o) \in I_A \Rightarrow o \in S_A^{\tau_2}$.*

An algorithm which aims to provide synchronization among replicas makes sense if it can provide progress when restricted to a consistency model – i.e., an algorithm should not only ensure a valid distributed execution, but also allow for any pair of replicas to communicate.

Ideally, two replicas should be able to synchronize (correctly) in a finite number of steps. Less formally this means that, at any point in time two replicas may attempt to synchronize, and such synchronization eventually completes.

[2]We write o_n^r as the nth operation generated at replica r.

Definition 6.12 (Algorithm). *An algorithm $\mathcal{A}$ is a set of steps which ensures synchronization for any pair of replicas for a given model. Formally, for any pair of replicas A and B, $\forall_{\tau_1} \exists_{\tau_2} : A \Updownarrow_{\tau_1}^{\tau_2} B \wedge S_A^{\tau_2} \ddots \mathcal{M} \wedge S_B^{\tau_2} \ddots \mathcal{M}$.*

Note that this does not disallow for eventual progress or using a third party. As there are no bounds set on τ_1 and τ_2, τ_2 might, in theory, be years into the future. In that direction we also define progress:

Definition 6.13 (Progress). *An algorithm $\mathcal{A}$ provides progress iff, in the case both replicas A and B do not fail and their shared communication channel does not fail, between τ_1 and τ_2, replicas A and B can synchronize without exchanging information with any other replica.*

Informally, an algorithm ensures any pair of replicas can at any point in time attempt to synchronize and eventually the state will be correctly merged among them. Additionally, an algorithm provides progress if this is possible without the synchronizing replicas requiring any outside help – synchronization finishes in a finite number of steps without any external intervention being required.

6.5 Causal consistency

6.5.1 Reliability

Now that we can define how operations are executed at any replica (and in which order), we need to ensure operations are delivered at every replica.

Definition 6.14 (Reliability). *A system provides reliability, or eventual delivery, iff for every operation o executed in any replica, every other (interested) replica eventually executes o, i.e., $\forall o, o \in S_{r_1}^{\tau_1} \Rightarrow [\forall r_2, Object(o) \in I_{r_2} \Rightarrow \exists_{\tau_2}, o \in S_{r_2}^{\tau_2} \wedge \tau_2 < \infty]$.*

$\tau < \infty$ means that τ is within finite time but with no strict bound. Informally, the previous definition states that if any replica executed o, eventually every other replica also executes o (respecting interest sets).

A system which only provides reliability has no restrictions on serialization order. Thus, in a system which provides reliability, no information on ordering is required. Typically keeping some ordering among operations is important, for example to ensure preservaton of user intent.

6.5.2 Happens before

In 1978 Lamport defined happens before as an irreflexive partial ordering on the set of all events in the system [49]. In contrast, we do not consider events not related to objects, such as messages being sent/received. In relation to objects we consider only writes – we assume reads are immediate, atomic, and local at each replica, and need not be propagated to any other replica. This allows for the following definition:

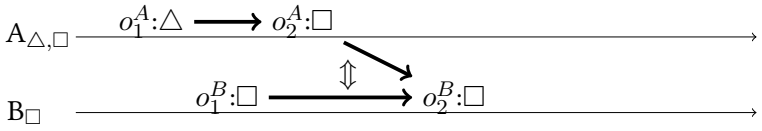


Figure 6.3: Diagram sholding happens before relations among operations.

Definition 6.15 (Happens before). *We say operation o_1 happens before operation o_2, $o_1 \prec o_2$, iff when operation o_2 was generated at some replica r, r has already executed o_1.*

In other words, happens before can be thought of as running a `read(ALL_LOCAL_STATE)` function before generating an operation, and the new operation will depend on everything in the local state.

In Figure 6.3 we show 2 replicas that each generate two operations, executing a synchronization in between. In this case, the following dependencies are established:

- $o_1^A \prec o_2^A$,
- $o_2^A \prec o_2^B$, (due to the synchronization step),
- $o_1^B \prec o_2^B$.

Some dependencies are derived by transitivity: $a \prec b \wedge b \prec c \Rightarrow a \prec c$. In this case, we have:

- $o_1^A \prec o_2^B$.

Based on the model $\mathcal{M}_\prec$ and Definition 6.3, we can define a causal serialization:

Definition 6.16 (Causal serialization). *A causal serialization of a set of operations R is the serialization $S \because \mathcal{M}_\prec$ of all operations in R with respect to $\mathcal{M}_\prec$.*

Note that $\mathcal{M}_\prec$ enables multiple distinct serializations for a set of operations, which in practice typically depend on the order in which operations are delivered.

Let us consider again Figure 6.3. With $O = \{o_1^A, o_2^A, o_1^B, o_2^B\}$ and $\mathcal{M}_\prec = \{(o_1^A, o_2^A), (o_1^A, o_2^B), (o_2^A, o_2^B), (o_1^B, o_2^B)\}$, all possible serializations of these operations are $\mathcal{S}(O, \mathcal{M}_\prec) = \{(\,), (o_1^A), (o_1^A, o_2^A), (o_1^B), (o_1^A, o_1^B), (o_1^B, o_1^A), (o_1^A, o_2^A, o_1^B), (o_1^A, o_1^B, o_2^A), (o_1^B, o_1^A, o_2^A), (o_1^A, o_2^A, o_1^B, o_2^B), (o_1^A, o_1^B, o_2^A, o_2^B), (o_1^B, o_1^A, o_2^A, o_2^B)\}$.

$\mathcal{S}(O, \mathcal{M}_\prec)$ is similar to the notion of consistent cuts defined by Friedmann Mattern in 1988 [108]. If operation b depends on operation a and b is included in a consistent cut, a must also be included in that cut – see Figure 6.4. In our notation, O_c is a consistent cut of O iff $O_c \subset O \wedge (\forall (a, b) \in \mathcal{M}_\prec, (b \in O_c \Rightarrow a \in O_c))$. The set of all possible consistent cuts of a set of operations O with respect to $\mathcal{M}_\prec$ equals to $\mathcal{S}(O, \mathcal{M}_\prec)$.

In the absense of partial replication, the notion of consistent cut maps directly to our definition of synchronization: a synchronization between two replicas is the same as joining two consistent cuts. Using Figure 6.4, *sup* is defined as the union of the two cuts C1 and C2 – this is similar as having two replicas, A (C1) and B (C2), synchronizing to a single unified state (*sup*).

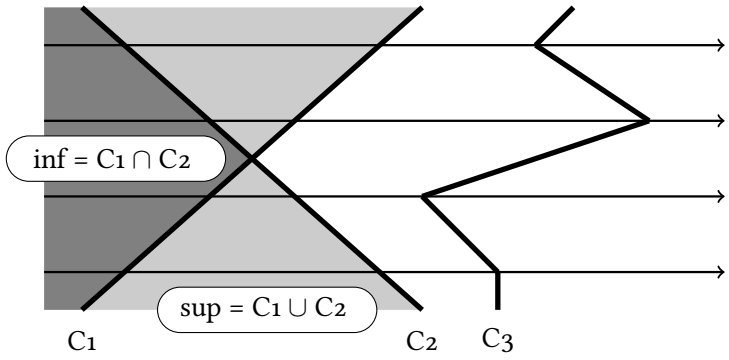


Figure 6.4: Consistent cut (diagram and quote from [108]): "A lattice is a partially ordered set any two of whose elements have a greatest lower bound inf and a least upper bound sup. Obviously, inf = C1 ∩ C2 and sup = C1 ∪ C2 for any two cuts C1, C2".

6.5.3 Objects and causal consistency

Causal consistency is typically defined with respect to relations (happens before) among operations, messages, or events in the system [43, 44, 49, 50, 58, 62, 64, 66, 96, 97, 102–127]. In these works, the definition of causal consistency depends on how happens before is defined, and typically does not include reliability.

Considering our previous definitions of reliability (Definition 6.14) and causal serialization (Definition 6.16), we define live causal consistency as follows:

Definition 6.17 (Live causal consistency). *A system enforces live causal consistency iff it is reliable and, for every replica r, r executes all operations according to a causal serialization.*

This definition, similar to previous works, has no explicit consideration for partial replication and individual objects. We define inter-object live causal consistency next, which is equal to live causal consistency but simplifies the distinction with what follows.

Definition 6.18 (Inter-object live causal consistency). *A system enforces inter-object live causal consistency iff live causal consistency is provided among all operations, independent of which objects these belong to.*

For a system with full replicas the second definition is redundant. For partial replicas this is not true, as operations for non-replicated objects will not be included in the happens before relation, which includes only previously executed operations (i.e., from objects contained in its interest set).

Note that transitive dependencies still remain. For example, if replica A generates o_1 and o_2 and replica B generates o_3, such that $o_1 \prec o_2 \wedge o_2 \prec o_3$, then $o_1 \prec o_3$ independent of the target objects of those operations and interest sets of the replicas.

We now define live causal consistency with respect a subset of objects.

Definition 6.19 (Per-object live causal consistency). *A system provides per-object live causal consistency if it is reliable and all operations are executed according to a causal serialization, where* $\prec$

is limited to operations from the same object, i.e., a happens before (Definition 6.15) relation $o_1 \prec o_2$ *can only be true iff* $Object(o_1) = Object(o_2)$.

This definition considers objects at a per-object basis, but can be easily extended to sets of objects. Per-object causal consistency with fixed replica-sets, such as some container or consistent-hashing based replication, reduces to per-object causal consistency by considering the object to be the container. Replicas may keep multiple groups of objects but do not consider dependencies among operations from different groups of objects.

6.5.4 Strong convergence

As there is no restriction on final state after applying operations, with causal consistency replicas can diverge in state. A simple example is two replicas writing some value to the same key concurrently, such that neither write dependends on the other. As both operations may be executed in any order, these two replicas will diverge just by applying the remote replica's operation.

In [58, 59], strong convergence is defined to address this issue, which we adapted:

Definition 6.20 (Strong convergence). *A system provides strong convergence if the same set of operations results in the same state, independent of execution order (serialization), i.e.,* $\forall_{S_1,S_2} \in \mathcal{S}(O, \mathcal{M}), ops(S_1) = ops(S_2) \Rightarrow State(S_1) = State(S_2)$.[3]

Intuitively, for every possible (valid) serialization of the same set of operations, the resulting state is the same. Eventual convergence – not for every set of operations, but for when no new write operations are generated – and reliability, are the base for eventual consistency [207]. Combining strong convergence with causal consistency (reliability and happens before) leads to causal+ consistency [96, 126]:

Definition 6.21 (Causal+ consistency). *A system provides causal+ consistency iff it provides causal consistency with strong convergence.*

The previous example of Figure 6.3 now reduces from 12 to 7 possible final states. This is because (o_1^A, o_1^B) and (o_1^B, o_1^A) must both result in the same state. The same applies to (o_1^A, o_2^A, o_1^B), (o_1^A, o_1^B, o_2^A), and (o_1^B, o_1^A, o_2^A) and to $(o_1^A, o_2^A, o_1^B, o_2^B)$, $(o_1^A, o_1^B, o_2^A, o_2^B)$, and $(o_1^B, o_1^A, o_2^A, o_2^B)$. Note that convergence has no effect on the size of $\mathcal{S}(O, \mathcal{M}_\prec)$, it only has an effect on the amount of possible (intermediate) states.

6.5.5 Comparing models

We now compare different consistency models by the amount of valid serializations each model supports for a set of operations.

[3] With $ops(S)$ a function that returns a set of operations for a given serialization, and $State(S)$ a function that returns the deterministic state of executing S in order.

Note that many specific settings can be devised where all models lead to the same execution. For example, a setting where a single replica generates one operation trivially equals all models as there is only one possible serialization, independently of the chosen model. We are not strict in the usage of $\subsetneq$ versus $\subset$ for cases such as single writer replica or single operation, but focus on the general case of many writing replicas, multiple writes per replica, and multiple and randomized synchronization steps between replicas. Thus, given the models:

$\mathcal{M}_r$: reliability (Definition 6.14);

$\mathcal{M}_{lcc}$: live causal consistency (Definition 6.17);

$\mathcal{M}_{\prec}$: causal consistency, differing from live causal consistency by not including reliability, see causal serialization (Definition 6.16);

$\mathcal{M}_{io-lcc}$: inter-object live causal consistency (Definition 6.18);

$\mathcal{M}_{po-lcc}$: per-object live causal consistency (Definition 6.19);

$\mathcal{M}_{c+c}$: causal + consistency (Definition 6.21).

The following is true for the general case:

- $\mathcal{S}(O, \mathcal{M}_{lcc}) \subsetneq \mathcal{S}(O, \mathcal{M}_r)$ – as live causal consistency builds on reliability and has additional restrictions (causal consistency);
- $\mathcal{S}(O, \mathcal{M}_{lcc}) = \mathcal{S}(O, \mathcal{M}_{\prec})$ – as live causal consistency builds on causal consistency and the additional restriction (reliability) has no impact on valid serializations;
- $\mathcal{S}(O, \mathcal{M}_{lcc}) = \mathcal{S}(O, \mathcal{M}_{c+c})$ – as convergence is a property on resulting state, and has no impact on serialization of operations;
- $\mathcal{S}(O, \mathcal{M}_{io-lcc}) \subsetneq \mathcal{S}(O, \mathcal{M}_{po-lcc})$ – as per-object is much more permissive than inter-object because many dependencies are not included, i.e., $\mathcal{M}_{po-lcc} \subsetneq \mathcal{M}_{io-lcc}$.

6.5.5.1 Explicit dependencies

Instead of assuming that when an operation is generated we execute a `read(ALL_LOCAL_STATE)`, dependencies among operations can be set explicitly. Allowing for dependencies among only a specified set of objects, or even individual operations, allows for greater concurrency when executing operations [96, 111, 120, 122].

There are many ways for dependencies to be set explicitly (and many pitfalls to avoid when letting developers deal with low level consistency semantics [66]). Examples range from letting the programmer provide operations to be dependend on when generating new operations, which forces reasoning about consistency for all possible operations, to configuring application semantics such that sets of objects are related (and hence only operations on these objects create dependencies among each other).

For example, consider that $S_A = (o_1^A, o_2^A)$, and replica A executes another operation, o_3^A. Under causal consistency ($\mathcal{M}_{\prec}$), we have $o_2^A \prec o_3^A$ (and $o_1^A \prec o_3^A$ by transitivity).

The model of explicit dependencies, $\mathcal{M}_{\rightarrow}$, allows for such relations to only exist if explicitly set. For example, even if there is no relation between o_1^A and o_2^A, the relation $o_1^A \rightarrow o_3^A$ can be set explicitly when o_3^A is generated. o_2^A will still be independent from both other operations.

Because $\rightarrow$ can only be set for operations that are visible when generating new operations, we know that $(o_1, o_2) \in \mathcal{M}_{\rightarrow} \Rightarrow (o_1, o_2) \in \mathcal{M}_{\prec}$ (i.e., $\mathcal{M}_{\rightarrow}$ can generate only a subset of $\mathcal{M}_{\prec}$). This leads to $\mathcal{M}_{\rightarrow} \subsetneq \mathcal{M}_{\prec}$ and directly the following result:

- $\mathcal{S}(O, \mathcal{M}_{\prec}) \subsetneq \mathcal{S}(O, \mathcal{M}_{\rightarrow})$.

Altough we focus on the size of the set of possible serializations, this agrees with related work [96, 111, 120, 122] in the sense that more efficient algorithms can be used (for example, by omitting dependencies greater concurrency can be achieved) while still giving a correct outcome in light of the applications' semantics. In our notation, a larger the size of $\mathcal{S}(O, \lhd)$ results in more possible executions and thus allows for higher concurrency.

The rest of this document focusses on causal consistency (i.e., the global state read happens) but also applies to any consistency model $\mathcal{M}_{\lhd}$ where $\lhd$ specifies an order among operations, such as explicit dependencies ($\mathcal{M}_{\rightarrow}$) or total orders as defined in the next section.

6.6 Total orders

Interestingly, causal+ consistency is often used to create causally consistent systems based on the claim that causal consistency is the strongest consistency model which remains available under network partitions [96, 102, 103, 120, 122, 123]. In practice the result is that, in our system model, causal+ consistency should be the strongest model we are able to support. The works that formalize this result rely on the following statements:

a) it may not be true that "the data store is pretending" that $o_1 \prec o_2$ even if $o_1 \not\prec o_2$, of observable causal consistency (OCC) [103];

b) "Time does not travel backward" of natural causal consistency (NCC) [102].

The first statement, a), can be translated to the system not being allowed to create some dependency among two concurrent operations to guarantee consistency. The second statement, b), loosely states that the system should not rollback operations, or reorder them.

In this section we show how we can create stronger models than causal+ consistency (reliability, happens before, and convergence), by not relying on those statements. We challenge statement a) by introducing dependencies not observed by causal consistency, and statement b) by allowing an execution model that explicitly uses rollbacks and re-executions to ensure a consistent outcome.

Considering Lamport's work [49] on creating a total order which respects the happens before relation among operations, we can create a consistency model stronger than just causal+

consistency but the data store will be pretending some order among operations exists. Lamport's approach to provide a total order is to ensure all operations can be sorted in an order that both respects causal ordering while also providing a single total order shared among all replicas. This approach requires the participation of all replicas, where every replica must communicate with every other replica, and where any failures impact liveness.

Alternatively, one can allow time to travel backward. Bayou [64] rolls back the database and replays the set of operations in a deterministic order to ensures all replicas reach the same state. Concurrent versioning systems use a similar approach when merging diverging state – start from a consistent state and merge updates creating a total order, resolving conflicting updates as they are applied.

6.6.1 Eventual linearizability

Eventual linearizability [118, 157] is typically defined as providing a partial order on operations that converges to a total order over time. An alternative definition is as providing a total order on all operations, but which may not include knowledge of all operations at all times.

This is in contrast to linearizability [208] where the total order is defined instantaneously (with knowledge of all related operations). This leads to the impression that all operations are executed at a centralized component in sequential order, consistent with the real time ordering of operations.

Eventual linearizability allows for reordering the sequence of operations, as long as all previous operations are always included (monotonic), and in two replicas with the same set of operations, the order is the same (converging).

The referenced works allow for replicas to operate on their local state without the necessity of coordinating with other replicas. In Chapter 5 we use a technique based on a total order $<^{ext}$ to obtain similar results. $\mathcal{M}_{<^{ext}}$ guarantees that if an operation o_2 might depend on operation o_1, then $o_1 <^{ext} o_2$. *Might depend* is formally defined in Extended Causal Property 2 – if some replica observes o_1 before o_2 is generated at any other replica, then $(o_1, o_2) \in \mathcal{M}_{<^{ext}}$.

Such a model can by implemented with realtime timetamps, by introducing a small delay before returning a timestamped operation to ensure any future operations capture the newly created one (defined as $TWait$ in Section 5.4.4).

This model was defined with the intent to deal with misbehaving users. If, alternatively, the system expects correct behaviour from replicas and synchronized clocks, the wait primitive is not necessary and timestamps can, instead of being generated at trusted servers or hardware, be generated locally. Additionally, the algorithms defined in Figure 5.5 can be simplified. This is because it is no longer necessary for eventual linearizability to include dependency identifiers and hashes for optimizations in discovering incorrect behaviour (namely Sibling detection, Section 5.4.2).

If the setting is trusted, then the additional metadata is not required, and a much more efficient implementation is possible. For details on how operations are serialized refer to Figure 5.4.

6.6.2 Extended causal consistency

If the happens before relations (dependencies) are included with operations, we can provide an extension to causal consistency which leverages realtime timestamps to make more informed decisions when operations are concurrent.

As defined in Section 5.3.3.1 for insecure settings, extended causal consistency can be defined as an extension to causal consistency which ensures that, if an operation is executed in an order that violates $<^{ext}$, the application is simply notified and may or not act on that information, i.e., if $(\exists_{o_2} : o_1 <^{ext} o_2 \wedge o_2 \in S_r \wedge o_1 \notin S_r)$, then, eventually, when o_1 arrives at replica r, the serialization of o_1 is replaced with $\texttt{signal}(o_1, \text{Ext}(o_1, S_r))$, with $\text{Ext}(o, S)$ the set of operations present in S, where $o \notin S$, that should have been executed after o according to $<^{ext}$ (i.e., $\text{Ext}(o, S) = \{o_2 : o_2 \in S \wedge o <^{ext} o_2\}$).

Note that no out of order operation is reordered as in eventual linearizability – the application developer must decide how to use the information provided to provide the user with an intuitive outcome.

For example, a chat application using eventual linearizability will reorder messages as they are received. Using extended causal consistency the application can chose how to show newly received messages by, for example, appending them to the current list of messages (which respects causality) and explicitly annotate such messages with read/unread labels, leading to an intuitive user experience.

These models were implemented for the secure consistency models defined in Chapter 5. Details can be found in Section 5.4.5 (Extended Causal Consistency) and Section 5.4.6 (Eventual Linearizability).

6.7 Genuine partial replication and causal consistency

In this section we prove that with genuine partial replication (Definition 6.8) it is not possible to provide causal consistency if replicas may fail permanently. This generalizes to any model which attempts to enforce relations among operations belonging to different objects, without further restricting which operations may relate to one another.

Intuitively, allowing replicas to store information on the objects not in their interest sets (Definition 6.9 and Definition 6.10), should allow for partial replication while encorcing causal consistency.

Unfortunately this is not true when replicas may fail permanently – we prove that no system is able to provide reliability under these conditions. This is trivial because when replicas storing the data fail permanently, and non-failing replicas store dependencies on that data, those dependencies can never be met.

This last statement provides an interesting outcome: if replicas may fail, and a system aims to provide partial replication, no consistency model can be used which, besides reliability, aims to provide some relation among operations (such as happens before for causal consistency).

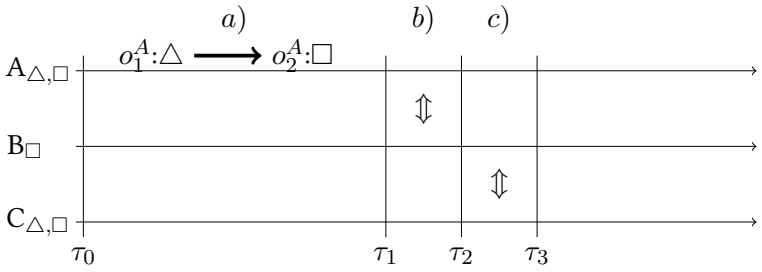


Figure 6.5: Diagram to detail the proof for Theorem 6.1.

6.7.1 On genuine partial replication

We begin with causal consistency, where the happens before relations among operations is the basic building block. Note that happens before relations (or dependencies) are tracked in metadata, and, to enforce genuine partial replication (Definition 6.8), replicas may not store nor receive dependencies for operations they do not hold.

This leads to the following theorem:

Theorem 6.1. *A system with arbitrary genuine partial replicas is unable to provide both causal consistency and progress.*

Providing only progress is simple as, if no guarantees are required at all, nothing is really necessary for progress to be possible. Providing only causal consistency is possible by arbitrarily enquiring other replicas about any missing operations, but this invalidates progress. The proof that providing both is impossible is based on the contradiction which arises between enforcing the happens before order on serialization (Definition 6.16, a requirement for causal consistency) and progress (ensuring a pair of replicas can synchronize: Definition 6.11):

Proof. Consider, the example of Figure 6.5, with three replicas: A, B, and C, and two objects, △ (a triangle) and □ (a square).

Replicas A and C are partial replicas with both objects, while replica B contains only one object, i.e., $I_A = I_C = \{\triangle, \square\}$ and $I_B = \{\square\}$.

Consider the following set of steps (in order):

a) Replica A generates two operations, o_1^A and o_2^A such that: $Object(o_1^A) = \triangle$, $Object(o_2^A) = \square$, and $o_1^A \prec o_2^A$;

b) Replicas A and B synchronize;

c) Replicas B and C synchronize.

After step a), at time τ_1, replica A has applied both operations while replicas B and C have not, i.e., $S_A = (o_1^A, o_2^A)$ and $S_B = S_C = (\,)$.

Note that as $O = \{o_1^A, o_2^A\}$ and $\mathcal{M}_\prec = \{(o_1^A, o_2^A)\}$, all possible correct serializations of these operations, are $\mathcal{S}(O, \mathcal{M}_\prec) = \{(\,), (o_1^A), (\neg o_1^A, o_2^A), (o_1^A, o_2^A)\}$. Note that $(\neg o_1^A, o_2^A)$ is included

as a correct serialization – replicas may serialize only o_2^A if they do not have $Object(o_1^A)$ in their interest sets.

At step b), replica B synchronizes its state with replica A. Per genuine partial replication, replica B may not receive any information about o_1^A, including the dependency $o_1^A \prec o_2^A$. At the end of this step, at time τ_2, $S_\text{A} = (o_1^A, o_2^A)$, $S_\text{B} = (\neg o_1^A, o_2^A)$, and $S_\text{C} = (\,)$.

This is still correct, as $S_\text{A} \in \mathcal{S}(O, \mathcal{M}_\prec)$ and $S_\text{B} \in \mathcal{S}(O, \mathcal{M}_\prec)$. Note that $\neg o_1^A$ being included in S_B does not invalidate partial replication – replica B has no information on o_1^A.

At step c), B synchronizes with C. Note that at time τ_2, $S_B \in \mathcal{S}(O, \mathcal{M}_\prec)$ and $S_C \in \mathcal{S}(O, \mathcal{M}_\prec)$. As we assume progress, B and C must synchronize, to a S_B and S_C where $S_C = \{o : o \in (S_B \cup S_C) \wedge object(o) \in I_C\}$ (see *sup* in Figure 6.4 for a visual aid, but note the restriction to the interest set).

As we have $S_B = (\neg o_1^A, o_2^A)$ (thus, not including o_1^A), this results in $S_C = (o_2^A)$. As $S_C \notin \mathcal{S}(O, \mathcal{M}_\prec)$, we conclude the proof. □

Interestingly, this proof trivially generalizes for any $\mathcal{M}_\lhd$ where there can be a relation of $\lhd$ among pairs of operations.

Generalization 6.1. *There exists no consistency model which enforces an order among operations which allows for both arbitrary genuine partial replication and progress.*

The proof, for any other model, can be constructed just as we did for causal consistency, replacing $\prec$ by $\lhd$ where needed.

Note that both the theorem and generalization only hold if the relation can be among operations which were made on different objects, and if replicas may replicate only the right-hand side of the objects in such a relation. From this we can derive a set of restrictions to enforce any model $\mathcal{M}_\lhd$ as such:

- $(o_1, o_2) \in \mathcal{M}_\lhd \Rightarrow [Object(o_2) \in I_r \Rightarrow Object(o_1) \in I_r]$ – if a dependency among a pair of operations exists, then any replica that serializes the second element of the pair must also serialize the first element of the pair;
- $[Object(o_1) \notin I_r \wedge Object(o_2) \in I_r] \Rightarrow o_1 \lhd o_2 \notin \mathcal{M}_\lhd$ – there may not be a dependency among a pair of operations, where in any replica the first element of the pair is not serialized and the second element is serialized.

These two restrictions are logically equivalent, and simplifies to:

- $(o_1, o_2) \notin \mathcal{M}_\lhd \vee Object(o_1) \in I_r \vee Object(o_2) \notin I_r$ – either the dependency among a pair of operations does not exist, the first element of the pair is always serialized, or the second element is not serialized.

We can leverage this restriction to explore directions to take when attempting to create algorithms for partial replication. To simplify the following we introduce origin and destination

objects – if we have the pair of operations, or relation, (a, b), then $Object(a)$ is the origin object and $Object(b)$ is the destination object.

To provide an algorithm which enforces $\mathcal{M}_\lhd$ for any $\lhd$, in a setting with genuine partial replicas (Definition 6.8), we must ensure that at least one of the following is true:

a) $(a, b) \notin \mathcal{M}_\lhd$ – relations may not exist;

b$_1$) $(a, b) \in \mathcal{M}_\lhd \Rightarrow \forall r : Object(a) \in I_r$ – every replica must keep all origin objects;

b$_2$) $\forall r \in R, I_r = Obs$ – every replica must keep all objects;

c$_1$) $(a, b) \in \mathcal{M}_\lhd \Rightarrow [\forall r : Object(a) \in I_r \vee Object(b) \notin I_r]$ – to keep a destination object a replica must also keep all related origin objects;

c$_2$) $(a, b) \in \mathcal{M}_\lhd \Rightarrow \forall r : Object(b) \notin I_r$ – replicas may not keep destination objects.

Note that a) is already true with an algorithm that only provides reliability.

A causally consistent system with full replicas ensures b$_1$) and b$_2$) as every object is included in every interest set. Attempting to ensure only b$_1$) would be interesting but we do not explore this direction further.

Note that c$_1$) is how Legion is implemented (detailed later in Section 6.8.3.1), as Legion provides partial replication with full replicas of fixed subsets of objects – it uses the concept of (exclusive) containers. Each container $C \in \mathcal{C}$ contains a set of objects, such that: $\bigcup_{C \in \mathcal{C}} C = Obs \wedge \bigcap_{C \in \mathcal{C}} C = \varnothing$. Additionally, there may be no dependencies among objects in different containers, i.e., $(a, b) \in \mathcal{M}_\lhd \Rightarrow [Container(Object(a)) = Container(Object(b))$.

If each replica keeps a single container this trivially allows for c$_1$). In Legion replicas may keep multiple containers but we disallow for dependencies among objects from different containers, as to ensure c$_1$).

The restriction of c$_2$) is not useful in practice. If a replica must keep $Object(b)$ to be able to create (a, b), then this forces $\mathcal{M}_\lhd = \varnothing$ as no such relation can be created. If a replica may create (a, b) without holding $Object(b)$, the replica can not be genuine by definition. Alternatively, keeping $Object(b)$ is a contradiction by itself.

6.7.1.1 Permanent replica failures

So far we did not discuss permanent failures of replicas. Intuitively, we know that even if progress may not be made in some cases, causal consistency can still be provided as long as operations are stored somewhere and may be fetched if needed (e.g., by leveraging some central repository that stores all operations that can be fetched on demand, only having replicas wait untill the repository responds to be able to apply operations). This leads to the definition of durability:

Definition 6.22 (Durability). *A system provides durability if it guarantees that operations when marked as* durable *by some mechanism are possible to be fetched (assynchronously), by every replica, independently of failures.*

Altough both durability (the previous Definition 6.22) and progress (Definition 6.13) are both welcome guarantees, they provide very different semantics. Durability ensures that an operation marked as durable may never be lost – for example, in a setting with client-side replicas a centralized storage service could be used which, assynchronously, provides durability. This is in contrast with progress as operations may effectively be removed from the system – for example, with the (permanent) failure of a large group of replicas from the system, some operations can be lost forever while all remaining replicas can still progress among each other.

Durability is not trivial to provide if replicas may fail and never return, which is a common occurence in client-side replicas: replicas may close their web-page, clean application caches, or even destroy their devices. Another impediment to durability in this setting is the cost of global coordination – intuitively, with *only* client-side replicas, an operation may only be marked as durable once every replica knows that every other replica has stored the operation. This leads to the following:

Theorem 6.2. *It is impossible to provide live causal consistency in a system where all of the following are true:*

a) arbitrary genuine partial replicas,

b) operations may always be locally generated without coordination among replicas,

c) replicas can fail permanently, and

d) communication is assynchronous.

Note that a) with live causal consistency is the end goal, and each other aspect is a side effect of our system model: b) is a necessity for immediate local reads and writes; and c) and d) originate from the system model with client-side replicas.

The impossibility follows directly from the example provided in the proof of Theorem 6.1 (Figure 6.5). If replica A fails permanently after step b) (i.e., at τ_2), then, B and C would no longer be able to progress in step c). Altough this does not invalidate live causal consistency by itself, the system will never be able to serialize operation o_2^A at replica C, nor any other future operation generated by replicas A and B (which would depend on that lost operation). This invalidates reliability (Definition 6.14) as required by live causal consistency (Definition 6.17).

Just as Theorem 6.1, this also generalizes for any model as such:

Generalization 6.2. *It is impossible to simultaneously provide reliability and any model $\mathcal{M}_\lhd$ which forces an order $\lhd$ on pairs of operations, in a system where all of the following are true:*

a) arbitrary genuine partial replicas,

b) operations may always be locally generated without coordination among replicas,

c) replicas can fail permanently, and

d) communication is assynchronous.

At this point we can derive what has most impact on creating algorithms to implemente a consistency model:

Corollary 6.1. *For any consistency model which besides reliability forces an order $\lhd$ on pairs of operations, an algorithm must either provide progress or durability.*

Note that the corollary doesn't explicitly mention full replicas or which objects partial replicas must, or not, keep – this stems from progress being trivially possible with full (or containerized) replicas.

In the previous proof we build on operations being serialized and violating the happens before relations. In contrast, here we do not violate the (happens before) relation as the relations may be known, but, as they restrict which operations may be serialized, reliability cannot be provided. For this proof we first detail why the corollary holds with either progress or durability, while with neither it does not.

Proof. Let us assume an algorithm $\mathcal{A}$ for $\mathcal{M}_{\lhd}$ (Definition 6.12).

If $\mathcal{A}$ provides progress then, by definition, we are always able to serialize the received operations when a pair of replicas communicate – given enough time and communication steps among pairs of replicas, eventually replicas serialize all operations they are interested in.

Similarly, if $\mathcal{A}$ provides durability then any synchronization between any pair of replicas can (eventually) complete as any required operations for the synchronization to finish can (assynchronously) be obtained.

To obtain reliability then a synchronization step between a pair of replicas must complete. This can be done either without coordinating with other replicas (i.e., providing progress) or by leveraging some process to eventually obtain dependencies (i.e., providing durability). If synchronization has no guarantee of completion then reliability is not provided, concluding the proof. □

6.8 Algorithms

An algorithm ensures that a pair of replicas can synchronize, in a manner that both their states evolve to a least upper bound on both their states (Definition 6.12).

Before discussing algorithms themselves, we must consider how replicas are selected to synchronize, and when such synchronization should start.

6.8.1 On the need for a server

A server (or some specialized replica) is needed to provide durability if other replicas may fail permanently or in a way that has them lose their state. In this work we aim to provide for a system with client-side replicas and possibly centralized servers which are used only when necessary. Namely, per Corollary 6.1, an algorithm must either provide progress or durability – in our setting if unable to provide progress, the server component can be leveraged to provide durability.

We assume a server is reachable for each client replica and that we can leverage it to address durability, bootstrapping client-side replicas to the network, and to act as a bridge among clients that are unable to establish direct network connections.

We interchange the use of the singlar term *server* with set of servers, as client replicas see a set of servers as a black-box of a single entity. We assume that each server can communicate with all other servers (possibly indirectly using other servers) and that each client replica can communicate with, at least, one server. This allows the server (or a path among servers) to also act as a communication bridge among client-side replicas. Altough servers and the communication channels among servers may fail, durability is provided. We assume that server failures are such that the server eventually becomes available and never loses any state, i.e., the server or centralized component provides durability.

6.8.1.1 When and where to synchronize

So far we assumed that replicas somehow are able and actually do communicate with one another. The important notion is that, somehow, communication should be done among all replicas after new operations are generated to ensure the knowledge of those operations is propagated throughout all replicas in the network. Again client-side replicas brings a challenge here – when, which, and how to connect (and synchronize) replicas.

The easy approach is to ensure every replica communicates to every other replica. This is non-scalable, does not consider that some connections cannot be established, and ignores that synchronization is transitive.

Instead, we propose that if an operation is generated with respect to some object, that operation must be propagated (by subsequent synchronizations) to every other replica that has that operation's object in its interest set.

Definition 6.23 (Propagation by transitive synchronization). *If there exists an operation o created by replica r_0 at time τ_1 and there exists a replica r which has interest in the object of o, then eventually an intermediate replica r_i synchronizes with r at τ_2 and $o \in S_{r_i}^{\tau_2}$, i.e.,* $\forall o \in S_{r_0}, [\forall_{r \in \mathcal{R} \setminus r_0}, object(o) \in I_r \Rightarrow (\exists_{r_i \in \mathcal{R}} \exists_{\tau_2, \tau_3} : \tau_1 \leq \tau_2 \leq \tau_3 \wedge o \in S_{r_i}^{\tau_2} \wedge r_i \Updownarrow_{\tau_2}^{\tau_3} r)]$.

Note that r_i may also include the server. A simple algorithm can be devised which trivially ensures propagation: every client replica, at randomized intervals, synchronizes with the server.

Alternatively, spanning trees can be used. For every object the server may compute a minimum tree, and many algorithms can be used for achieving this [209]. Computing optimal trees, such as done in [62] can be very costly – high replica churn makes such an approach impractical.

Another alternative is to construct a tree that minimizes operation delivery latency [87, 89]. These works could be leveraged instead of using our following approach which leverages the server as a specialized replica.

To ensure that clients with overlapping interest sets are able to find each other, to guarantee synchronization happens among them, for the remaining algorithms we assume the following

algorithm (adapted from Legion's bullying algorithm, detailed in Section 3.2.2).[4]

At every interval Δt, the server emits for every connected replica r the message $< IDS, C >$, with $IDS = \{id(ob) : ob \in I_r\}$ and C the value of a counter starting at 1 and incremented every Δt. This means every replica connected to the server receives, every interval Δt, a message containing the identifiers of all objects it is interested in.

Every replica r_1 receiving a message propagates to every other replica r_2 it is connected to IDS^2 where $IDS^2 = IDS \cap I_{r_2}$ (replicas exchange interest sets on establishing a connection). This allows for the construction of a latency-based spanning tree per object: as the same set of objects and C can arrive through multiple paths, the first time C arrives for some object defines that path (the replica that sent it) as the closest, lower latency, path to the server for that object.

Client replicas which receive from other client replicas messages with all identifiers in its interest set, may disconnect from the server as this means an (indirect) connection exists to the server. This allows for a protocol similar to [62] which uses the tree to propagate metadata, but, in contrast, we use the tree to propagate the data itself (including metadata) while the remaining connections are used to propagate only metadata. These inactive connections, when required to be used again for data, must first ensure a new synchronization is executed to ensure causal delivery of the data itself.

To ensure all replicas are connected we define a verification interval Δt_v. If for any object in its interest set a replica does not receive a message that includes that object's identifier, for a Δt_v, it should attempt to re-connect to the server directly. The parameters for the propagation interval Δt and the verification interval, Δt_v, should be carefully set as to allow operations to be propagated through the network, considering the delays these may incur.

This method also allows to discard connections to promote low-latency links (Section 3.1.1.2) and disrupt malicious replicas purposely delaying propagation (Section 5.5.6.2).

6.8.1.2 On the loss of operations

A replica that generates an operation and immediately fails, permanently, may invalidate previously discussed aspects. Reliability should thus only consider those operations that are known by non-failing replicas (as is common in distributed systems research). We consider a non-failing replica knowing of some operation if it does not fail and it either stores the operation directly or stores any operation that (possibly indirectly) depends on it.

For example, if all replicas that contain an operation o_1 and any dependency on o_1 fail, we assume o_1 does not exist – i.e., this is true if $\forall_{r \in \mathcal{R}}, o_1 \notin S_r \wedge [\nexists_{o_2} : (o_1, o_2) \in \mathcal{M} \wedge o_2 \in S_r]$.

Note that this has no impact on the previous theorems – for example the proof of Theorem 6.1 holds as the dependency still exists: even if all replicas storing o_1 may have failed, there are still replicas that store an o_2 such that $(o_1, o_2) \in \mathcal{M}$. Additionally, it has no impact on propagation by transitive synchronization (Definition 6.23) as the left-most hand of the formula would be false: $\forall_r, o \notin S_r$.

[4] The protocol is also similar to the protocols detailed in Section 5.4.7, which additionally includes information to ensure correctness and timely delivery (i.e., timestamps and signatures).

The notion of permanent loss of operations may have a major impact on application usage by users that continue to interact with the system even when failures happen (for example, with a simple page reload). Applications should inform the user of operations (or state) that might be lost when such errors happen. This leads to a new question: how to transit operations from this preliminary state to being durable?

With no specialized replicas the only way for an operation to become durable is to be serialized in every single replica – we refer to this as stability (detailed later in Section 6.8.4.1).

In our system model we can use a durability mechanism based on the server – and keep the user informed of those operations which the server has acknowledged to be durable. Also common in distributed systems research is the assumption that at most F replicas will fail or being F-resilient [210–212]. We could assume at most F replicas fail, and state that an operation o is durable iff $F + 1$ replicas store o.

Our system model considers client-side replicas. As estimating a realistic and useful value for F is intrinsically very hard with such replicas, we do not further explore this direction.

Note that we leverage the server instead, where server is a set of replicas S, we assume $|S| > F$, where at most F server replicas may fail, i.e., if the replicas in S are seen as a single server replica from the point of view of a client replica, that *single server replica* never fails. The works on geo-distributing the server-side into a fault-tolerant and highly available system are orthogonal to this work. The central server replicas can use multiple geo-replication mechanisms [62, 96, 111, 122, 123, 125], but here we simplify the notion of geo-replicated servers to a single centralized server for brevity and clarity. The server-side is seen as a black box and can be reasoned as a single entity, and every replica is able to connect to the server.

6.8.2 Full ordered list replication

The ordered list algorithm specified in Section 4.1.2.2 allows for causal delivery of operations without paying any additional metadata cost over providing only reliability.

In summary, each replica locally keeps a list of operations. When new operations are generated, these are appended to the end of the list. When synchronizing with a new replica, the whole list is sent in order and every operation received which is not in the local list, is appended to the end of the list. The only requirement for this algorithm is that operations can uniquely be identified. Unique identifiers are composed of a `(replicaID, opID)` pair, with `opID` a counter starting at 1 and incremented with every generated operation at the generating replica. This algorithm can have many optimizations.

Instead of propagating the whole list when synchronizing, keeping active connections allows for a full synchronization initially where afterwards only individual operations have to be propagated, using FIFO channels. Additionally, the initial state synchronization can also be done more efficiently:

- by keeping knowledge of which operations were previously sent to the remote peer, recovered connections do not incur into a full state transfer;

- using a protocol similar to that of Δ-CRDTs (Section 4.3.2) allows for major savings at the cost of sending metadata as a first step with size in the order of number of replicas;
- by globally ordering [49] operations (in a separate data-structure to the original list) and using Merkle trees [213], one can instead implement a mechanism that instead allow for logarithmic sized metadata overhead on the order of number of operations.

6.8.2.1 Reducing the list size

The proposed algorithm provides progress, as every replica keeps the full list of operations (i.e., full replication). To reduce the size of the operations list, by removing operations that have been applied in every replica, we require some mechanism which either provides durability or does not remove the progress property.

Using global stability would not impact progress, but would be very costly if all replicas are part of the mechanism. Existing algorithms [106, 110] based on Lamport's notion of stability [49] can be applied to garbage collect operations which are no longer required to ensure eventual synchronization. These algorithms give a notion of what is stable – which operations are known to be already propagated throughout the whole set of replicas – and thus clearly define which operations are no longer needed. Constructing a dependency-collection algorithm follows as the next step, but this comes at the cost of running a stability protocol. Such a stability protocol both becomes increasingly expensive as additional replicas join the system, as well as typically requires a fixed and globally known membership (which is impractical in our model).

An alternative is to ensure stability via some specialized replica. This goes well with our system model, where clients communicate peer-to-peer but also with a server to bootstrap the peer-to-peer network.

Thus in our case we leverage the central component, where the server periodically propagates a summary of all operations that can safely be removed. As we wish to only keep the `(replicaID, opID)` metadata, we store, at each replica, the execution history containing pairs of `(replicaID, lastOpID)`, where `lastOpID` is the idenfitier of the last operation executed originating at `replicaID`. Altough this does not require all client replicas to communicate with each other, this is still similar to the discussed notion of stability – it comes at the cost of metadata whose size, in the worst case, is in the order of the amount of replicas in the system [114].

The server propagates such history every ΔH_S seconds. Client replicas may safely delete from their list any operation that is contained within the server's propagated history.

For operations to be included into the server's history, the server needs the knowledge that such an operation has been executed in every single replica. This can be done by having every client replica, every ΔH_C seconds, send their local history to the server. The server, keeping track of the minimum of all such histories can trivially create a history that is globally safe to be removed.

Efficiency is important here – the previously described mechanism to ensure global connectivity (Section 6.8.1.1) can be used to create not only a tree path to communicate client histories, but to compress such messages. For example, at every propagation step, clients can merge their histories,

`(replicaA, historyA)` and `(replicaB, historyB)`, into a summarized history, `(replicaA, replicaB, min(historyA,historyB))`, where `min(h1, h2)` returns the minimum number for all pairs in both histories and omits any operation present in only one history.

Discussion The proposed algorithm, even if academically interesting, is not suitable for our goal of partial replication – replicas receive operations on objects they are not interested in, and are forced to keep them to provide causality among objects until safe to delete.

Additionally, clients which are not able to communicate with the server for extended periods of time have an impact on this algorithm – the server cannot distinguish between failed (not-returning) or slow/disconnected replicas – and therefore would never advance its history. One way to address this is to let the server assume, if it doesn't receive a client replicas's history in say, $5 \times \Delta H_C$, that the replica has (permanently) failed. Note that, when again reachable, the client may require operations that have been removed by all other replicas. Therefore if a client replica is assumed as to have failed, the replica must reset its state and obtain a new replica identifier, effectively making it a new replica. Albeit this approach may be harsh for the user, the system can keep both new and old state and allow the application to merge into the newly obtained state any missing changes (possibly requiring the user's input on changes the user wishes to save).

There are, however, some important properties which make the proposed algorithm interesting:

- not every replica is required to communicate with (or even know about) every other replica;
- any replica is able to correctly synchronize with any other replica (providing progress);
- ΔH_S and ΔH_C can be dynamically adjusted for the specific needs of the application at hand;
- an interruption on the durability protocol has no impact on safety or liveness – it only requires keeping operations until the mechanism restarts (assuming large ΔH_S and ΔH_C).

6.8.3 Full state replication

In contrast to the previous algorithm, which keeps operations, one can always compile changes directly into state and, instead of propagating operations, propagate only state [31, 58]. For a system based on CRDTs, it is possible to use state or Δ based CRDTs (in contrast of being able to directly use operation based CRDTs in the previous algorithm).

Assuming full replicas, where replicas keep every object, and apply all changes while respecting causality, it is not required to keep specialized replicas to ensure neither durability nor progress (assuming replicas are able to connect among each other). In our case, without global network knowledge, we would have to rely on the server only to ensure global connectivity to provide reliability.

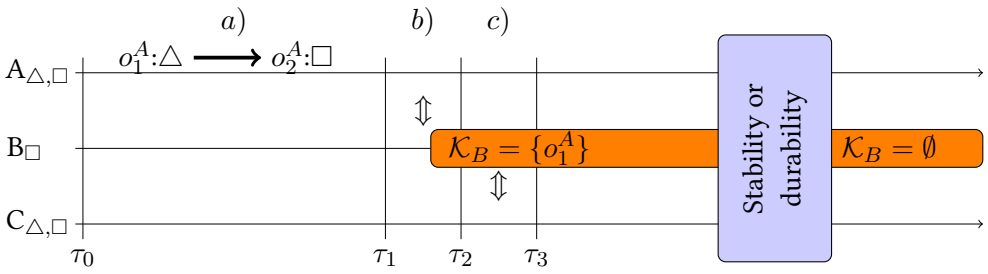


Figure 6.6: Diagram for the keeping dependencies family of algorithms.

6.8.3.1 Containerized state replication

In Legion (Chapter 3) all replicas are full replicas with respect to the containers they connect to. This means that a replica, when joining a container, will replicate every object of the container, i.e., if $Cont$ is the set objects belonging to the container replicated by r, then $I_r = \{o : o \in Cont\}$.

Altough replicas do replicate objects from multiple containers, Legion only provides causal consistency among objects of the same container. To this end, Legion leverages the work on Δ-CRDTs (Section 4.3). In summary, Legion replicas form a connected network leveraging the server (an overlay network per container, using a protocol similar as the one described earlier in Section 6.8.1.1). When new connections are established replicas synchronize using Δ-CRDTs synchronization for each container, and then keep using the same (FIFO) connections to send individual changes on data in causal order. For details on both Legion's causal propagation and Δ-CRDTs implementation refer to Section 4.3.2.

6.8.4 Partial replicas by keeping dependencies

It is clear from the previous sections that a compromise must be made in order to approach partial replicas without resulting, in the end, in requiring full replicas. For this we revisit the situation we used previously for the proof that besides the impossiblity of genuine partial replicas an algorithm must provide either progress or durability – Figure 6.6 will be used to discuss the initial approach and further optimizations.

The original problem originates in step $a)$ when replicas B and C synchronize, and their interest sets overlap but are not equal. In the example, replica B, which must be able to synchronize to other replicas, intuitively has to ensure that every predecessor of any operation op it applies is available to any other replica (replica C) when op is propagated (to C), i.e., replica B has to keep every dependency for op even if such dependencies are not in its interest set.

For this we define the set $\mathcal{K}_r$ of all operations whose object does not belong to the replica's interest set, but which have a successor operation whose object does belong to the replica's interest set, and will thus be serialized into S_r. Formally, $\mathcal{K}_r = \{op : Object(op) \notin I_r \wedge \exists_{op_2} : op \prec op_2 \wedge op_2 \in S_r\}$.

The set $\mathcal{K}_r$ includes all operations that any locally serialized operation might depend on. To ensure replicas are able to synchronize, besides the operations on objects in I_r, replicas keep the set $\mathcal{K}_r$. This leads to some operations being kept forever, as depicted in Figure 6.6 where, without

any further mechanisms, operation o_1^A is kept by replica B forever. In practice, as time goes on and operations are created beloning to all objects in the system, any operation (except very recent ones) will have predecessors from objects not stored at replicas. Given enough time $\mathcal{K}_r$ tends to $\mathcal{K}_r = \{op : Object(op) \notin I_r \wedge \cancel{\exists_{op_2} : op \prec op_2 \wedge Object(op_2) \in I_r}\}$, and eventually, $S_r \cup \mathcal{K}_r \cong Ops$ – the union of both serialized and stored operations in $\mathcal{K}_r$ equals the total set of operations and we are back at (almost) full replication.

Besides countering the primary objective, partial replication, it is impractical to store all operations forever – there needs to be a mechanism to minimize the size of $\mathcal{K}_r$.

6.8.4.1 Stability to clear $\mathcal{K}_r$

The alternative to storing operations forever is using stability. This can be done with a protocol similar to the one detailed in Section 6.6.

For this, any operation o stored at a replica r where $Object(o) \notin I_r$ can safely be deleted as soon as $Stable(o)$ becomes true. This is depicted in Figure 6.6 where, after the stability process ends, $\mathcal{K}_B$ would be empty as o_1^A would be stable – every replica has observed o_1^A at the beginning of the stability mechanism.

Such a mechanism is inherently not scalable – it requires every replica having to communicate with every other replica. There is not really a scalable solution to stability, especially in arbitrary dynamic networking models.

Nevertheless, any such algorithm is very promising, especially if interest sets among replicas mostly overlap or if networks are small. In such cases the overhead of stability might be worth the costs, as the given guarantees and properties show:

- it provides live causal consistency as both reliability (assuming a connected network graph) and happens before consistency are provided;
- it allows for progress – any replica is trivially able to synchronize with any other replica at any time;
- replicas are partial except for operations not yet collected by the stability mechanism;
- the stability protocol itself has no impact in overall execution, as it is not required for linevess or correctness. The only impact of any pause in the stability protocol is that some operations have to be kept for longer before being garbage collected.

Note that a stability algorithm is highly dependent on membership. If membership is highly dynamic, or if global membership knowledge is unfeasible, then the following approach might be preferred.

6.8.4.2 Durability to clear $\mathcal{K}_r$

In contrast to the previous solution, which aims to always provide progress, here we sacrifice progress in order to allow for partial replication leveraging some form of durability of operations. We discuss two approaches:

- assume up to F replicas fail, so that any operation which is serialized at $F + 1$ replicas can be assumed to be durable;
- assume there exists a specialized set of replicas which, when failures happen, do not lose any previously serialized operations.

The first approach assumes a failure model such that up to F replicas fail. As previously discussed, estimating or controlling a useful value for F when reasoning on client-side replication is very hard. In reality any replica may fail at any time, and thus the safe (or simple) approach is to assume F equals the number of replicas in the system. Altough the application can use a smaller value for F and explicitly present to the end user the current status of operations, we believe a better approach, as discussed next, is to leverage some form of specialized replica.

Leveraging the server In the goal to provide for partial replicas we will leverage durability provided by the server replica. We assume that when the server replica serializes some operation, that operation is not lost – recall we use the simplified notion of server replica assuming a geo-replicated service.

The goal is that the server replica is not necessary to be always available – we aim to minimize the overhead of storing $\mathcal{K}_r$ without compromising the ability of replicas to synchronize. As the server replica stores all objects, and thus all operations, then any client replica can fetch any missing dependencies from the server replica.

When a replica has knowledge of which operations have been serialized by the server replica, then garbage collection of these operations can be safely executed. This is because, when any dependencies are found to be missing when two client replicas attempt to synchronize, there is the possibility to fetch those dependencies from the server. Note that this approach does not provide progress (Definition 6.13).

In summary, the algorithm is as follows:

- all operations, which include dependencies in their metadata, are sent to all connected replicas, not taking into account any knowledge of the interest sets of the other replicas;
- operations on objects which the replica has no interest in but depends on ($\mathcal{K}_r$) are stored until marked as durable, after which they are removed;
- when two replicas attempt to synchronize and dependencies are missing, the server replica is reached to obtain those operations – client replicas may have to wait for the server to be reachable.

Similar to the algorithm to reduce the size of stored operations lists (in Section 6.8.2.1), to allow for garbage collection the server propagates, every ΔH_S, its history (a compressed summary) of its serialized operations.

The history of the server replica, H_S, is comprised of pairs of `(replicaID, opID)`, and formally defined as $H_S = \{(id(r), c) : \exists o \in S_S, id(o) = (id(r), c) \wedge c = max(x_i : [(id(r), x_i) \in$

$\{id(o_1) : o_1 \in S_S\}])\}$. Intuitively H_S contains, for every replica that created some operation, the highest (last) operation id that was serialized at the server. This is most often referred to as a version vector.

Simply stated, for every pair `(replicaID, opID)` included in H_S, every operation generated by replica with idenfitier `replicaID` with an identifier less or equal to `opID` is durable.

When receiving H_S, a client replica may exclude the operations present in H_S from its $\mathcal{K}_r$. At this point this leads a replica to keep $S_r \cup \mathcal{K}_r \approxeq S_r$. This is because, along with an increasing S_r, $\mathcal{K}_r$ will approach the empty set and have a negligible size when compared to S_r.

Note that, when the server is unreachable or if due to some other reason H_S is not propagated to some client replicas, the size of $\mathcal{K}_r$ may increase. This by itself has no impact on correctness or liveness, it only increases the time some operations have to be stored in $\mathcal{K}_r$ to ensure that dependencies can be provided when replicas synchronize.

There is a drawback in that client replicas may not be able to synchronize when the server is unreachable, as progress is not guaranteed (note that replicas with equal interest sets can always finish synchronization successfully). To mitigate this, client replicas may keep in $\mathcal{K}_r$ operations that are included in H_S but which are expected to be required in future synchronizations. Alternatively this can be done at the server level, not including recently received operations in H_S. Very recent operations, those that are probably not propagated to all clients, are good candidates for this behaviour.

In summary, this approach sacrifices progress and leverages the server replica's durability guarantees to provide causal consistency as any dependency can be obtained. Using this approach allows to create an algorithm where:

- replicas are partial, except for operations not yet collected with the durability mechanism;
- delays in the durability mechanism have no impact in algorithm correctness;
- progress can be provided among replicas that keep an extended $\mathcal{K}_r$ such that operations that are expected to be required in the near future are not collected (i.e., very recent operations);
- there is no need to reason about how often and how many replicas fail, how dynamic the network is, and which replicas belong to the network (no need for global membership knowledge).

6.8.5 On dynamic interest set changes

None of the previous algorithms considers that the interest set of a replica I_r can be dynamic. Consider that I_r is determined when replica r joins the system. To allow for later, dynamic, changes to I_r, there are many aspects that have to be considered:

- if any newly replicated object must be causally consistent with the local state then the replica may not execute or generate any operation while that object is not brought up to the same version as all other locally replicated objects;

- any optimization leveraging efficient propagation trees generated based on the interest sets are invalidated and may break causal consistency in continued use;
- if objects may be removed then algorithms leveraging membership (e.g., relying on $stability$) may be incorrect.

Therefore, any change in I_r possibly incurs a major overhead if causal consistency is to be maintained, and many edge-cases have to be considered.

We opt for a different approach: interest sets are fixed when replicas are created and, more importantly, replicas may not change their interest sets. To allow for dynamic changes in replicated objects at a client's device, we instead allow for multiple replicas to exist at the device – any required change in replicated objects is treated as the creation of a new replica.

Thus if replica r with interest set I_r aims to include (or exclude) object $\triangle$, it creates and runs a replica r_1 alongside r, with $I_{r^1} = I_r \cup \{\triangle\}$ (or $I_{r^1} = I_r \backslash \{\triangle\}$). This allows for replica r to remain available untill the state of r^1 is up to date with r.

Replica r will attempt to synchronize with replica r^1. When this process completes then replica r^1 can effectively be merged into r as it is no longer required, i.e., with $r \Updownarrow_{\tau_1}^{\tau_2} r^1$, at time τ_2 replica r executes, in an atomic step, $r = merge(r, r^1)$.

When excluding objects this process is immediate as there is no need to wait for remote operations to arrive – already serialized operations are moved into $\mathcal{K}_r$, the object is removed, and the merge procedure may complete immediately.

Such a system, being explicit to the application about the state of each local replica, allows for:

- local replicas to be used alongside each other while being explicit to the user about which part of the application is behind;
- allows the $merge$ step to resolve any conflicts between replicas (using, for example, CRDTs), where the application can provide the user with a correct and intuitive outcome;
- allow only changes on the up-to-date replica showing the user that part of the local state is currently behind and must be updated before being used – the merge step frees the usage and notifies the application on any significant changes that must be presented to the user.

6.9 Related work

Partial replication SwiftCloud [65] is a geo-replicated service which supports partial replication on the client-side (as a read/write cache). This allows clients to apply read and write operations locally and immediately on objects it has interest in, while being constantly updated on any changes though a FIFO channel with a server. SwiftCloud does not allow the interaction of such client-side partial replicas among each other – communication has to be done through full (server) replicas, effectively solving the problem that genuine partial replication imposes

which only occurs when partial replicas (without complete overlap in interest sets) attempt to synchronize.

Similar to our work, PRACTI [214] is a replicated system which aims to provide arbitrary partial replication and topology independence (along with ensuring stronger consistency when required, whereas we focus on causal consistency). Replicas may dynamically chose the objects in their interest sets. Operations are generated locally and immediately executed, and are tagged with the pair of `replicaID:clock` with `clock` the replica's Lamport clock value before propagated. Data and metadata is propagated seperately, allowing some freedom in how data flows through the system as data does not need to be causally ordered (the metadata flow ensures data is executed in correct order). The metadata flow is kept at each replica – this ensures causal consistency can be provided locally and when synchronizing with other replicas.

Genuine partial replication Atomic multicast is genuine if only the sender and receiver processes are involved in the protocol to propagate the message [215]. P-Store [216] aims to provide genuine partial replication but relies on a multicast primitive that is able to deliver a message to every replica that keeps a given object. Such a genuine atomic multicast for partial replicas primitive must restrict 'the destinations of multicasts to sets of disjoint process groups, each group behaving like a logically correct entity' [215].

Saturn [62] tracks causality metadata and assumes that data is propagated using an existing bulk-data mechanism in place. Saturn ensures operation identifiers are propagated through a global and fixed dissemination tree among datacenters. Datacenters act as leaves of the tree, where upper levels are servers which, using FIFO channels, enforce causal delivery of the identifiers of the operations among all datacenters – the order in which they are delivered is the order in which they are executed. There is global knowledge of all replicas and their interests in the system which is used to compute the propagation tree – the tree should be computed such as to minimize the amount of data sent to non-interested replicas (to ensure as close as possbile to genuine partial replication). Failures or changes in interest sets would require recomputing the tree – which additionally is a propagation bottleneck.

Karma [129] supports partial replication by separating replicas into disjoint replica sets using a consistent-hashing ring (in contrast to having the whole ring in each DC), where a ring is formed by geographically close DCs. Causality is guaranteed among the DCs of a single ring, and clients are blocked from reading from a different ring untill the system knows operations are stable, as update propagation between rings is asynchronous. Karma aims for static partial replication at the DC level where all objects are, at all times, nearby. In contrast, we aim for dynamic partial replicas at a much smaller granularity.

Opt-Track [128] provides partial replication for distributed shared memory, guaranteeing causal consistency. The granularity of partitioning data is at the data-center level, which itself allows for no partitioning internally. It relies on a multicast primitive [215] and it assumes a static system (fixed partial replicas) and every replica has complete knowledge of all other replicas and the objects they replicate.

PaRiS [130] is a causally consistent system which supports partial replicas, where clients additionally keep a local cache for faster application response time. Each DC may be partial, and each server in a DC keeps a fixed partition of the data. Replicas have global knowledge as misses (client operations on non-replicated objects) are directly forwarded to the correct server. Instead of blocking reads to ensure all updates have arrived, it presents stale data in the form of snapshots that consists of data that has been deemed as stable across all DCs.

Discussion Most referenced works impose no specific topology but require every replica (or datacenter) to be able to communicate with any other replica. Limiting the topology can create interesting algorithms, such as using the tree topology of Saturn (but these algorithms do not generalize to other network topologies). In contrast, we aim to neither require all-to-all communication neither to enforce some specific network topology among replicas. Most importantly, in our system model it is unfeasible for every replica (clients) to be aware of the global membership.

In contrast to the referenced works, we also aim to provide for partial replicas which do not build on a multicast primite which itself requires global knowledge of all of its replicas. Additionally, our definition of genuine partial replica is inherently different – the referenced works assume knowledge is restricted to the sending and receiving of operations whereas we aim to restrict also knowledge of other objects (i.e, any metadata associated to objects not in the interest set).

These goals and restrictions led to the proposed algorithms which are feasible for client-side replication.

6.10 Final remarks

In this chapter we discussed the impact that partial replication has on providing causal consistency.

In general, the outcome (Section 6.7) is that any algorithm that aims to allow for genuine partial replicas under a consistency model which ensures some order among operations, such as causal consistency, must also either provide progress – the ability to correctly synchronize pairs of replicas without the usage of any third party – or durability – using either a stability mechanism enforcing that every replica has observed the operation or extending the system model to include durable replicas (i.e., eventual recovery from crashes where data loss may happen).

Considering partial replicas we discuss algorithms that provide causal consistency, ranging from a no metadata overhead algorithm (Section 6.8.2) and full state replicas (Section 6.8.3), to an efficient algorithm focussed on storing dependencies only when necessary to ensure required operations for the happens before relations can be delivered (Section 6.8.4).

Chapter 7

Final considerations

In this chapter we conclude the dissertation and summarize the accomplished results. The extended scope of the work leads to many directions worth exploring in future research.

The research conducted was broadly aimed at providing lower latency to user-centric applications, by extending the system model with client-side replicas and the communication model with client-to-client interactions. This work explores how to bring most of the application logic to the client-side, using the centralized service only for storage, access control, or other aspects which are strictly necessary.

In summary, we proposed a hybrid of cloud-and-edge which provides lower user-to-user latency, availability under server disconnections, and improved server scalability – while being efficient, reliable, and secure.

A summary of the main contributions of this work are, by chapter:

Chapter 3 introduces the cloud-edge hybrid model. The Legion system establishes the base of the work, allowing for client-side replicas and direct client-to-client synchronization, enabling faster application response times, lower client-to-client latency, increased server capacity as fewer clients need to be connected at the same time, the possibility to work offline or disconnected from the server, and reduced server bandwidth usage. Briefly touching replication, consistency, and security, it defines the networking model and algorithms employed to ensure a connected and efficient network of clients. The implementation was evaluated comparing an existing cloud-based solution to our design.

Chapter 4 discusses the use of causal consistency in Legion. It introduces the algorithms for client-side replication and, in particular, details the design and implementation of Δ-CRDTs, including an evaluation compared to the standard operation and state based CRDTs. We show how Δ-CRDTs can be used and how Legion leverages Δ-CRDTs for efficient synchronization among replicas in dynamically changing networks.

Chapter 5 addresses security related aspects – enforcing privacy and integrity while dealing with client misbehaviour. We study how client misbehaviour can impact the guarantees of causal consistency and define secure consistency models preventing multiple types of

misbehaviour. An implementation of the proposed algorithms is also evaluated, considering user experience, focussing on latency, and the effectiveness of the algorithms in dealing with replica misbehaviour.

Chapter 6 explores supporting partial replication. We prove the impossibility of providing genuine partial replication in our system model – as we must account for ephemeral replicas – and discuss practical alternatives which aim to provide causal consistency without forcing full-replication at all times, or at every client.

7.1 Research directions

The scope of the work allows for many subjects worthy of further investigation.

7.1.1 Cloud-edge model

Δ-CRDTs replication model It seems that the *synchronize then propagate* communication and replication model can be used to implement almost any kind of replication algorithm. Chapter 6 already provides some algorithms that work, but it would be interesting to further explore this direction in building an implementation an expandable testing suit to compare different methods in pratice.

Such a system would also allow to systematically evaluate the secure consistency models by implementing maliciously behaving replicas.

Bullying functions Altough the bully algorithm employed in Legion (Section 3.2.2) is clear and does its job, it can be interesting to further explore improvements. It seems trivial to propagate bully messages multiple hops over the network to ensure a multiple order of magnitude drop in required client-server connections. Additionally, the employed protocol should take into account the type of users' devices (mobile or desktop) and what network it is on (mobile network or fiber line).

Leveraging network topology for partial replication The chosen implementation of the overlay – the network among replicas that is established to synchronize state and propagate operations – can have a great impact, not only on network usage but also on storage. If the overlay creates connections to other replicas in a way that promotes a large overlap on the interest set of replicas, then the amount of stored dependencies on non-replicated objects may be very small. In fact, if a given workload has clear boundaries on interest sets then a completely separate overlay can be created for each interest set, separating them at the network level without any change to the consistency mechanism.

An approach with strict topology rules (such as a causal propagation tree [62]) can be applied to reduce the size of the necessary metadata to track dependencies. In fact, techniques such as causal separators [217] – where specific topologies are used to create disjoint sets of replicas to reduce vector sizes – can easily be applied with the help of our centralized component.

7.1.2 Securing client-side replication

Is Secure Causal Consistency the limit? Secure causal consistency (Chapter 5, Section 5.3.1) is the only secure consistency model that we provide without the use of some form of trusted authority. The question is if it is the best that can be provided without a trusted authority (server or secure hardware).

As discussed in Section 6.6, natural causal consistency (NCC) [102] and observable causal consistency (OCC) [103] are the limit of what is achievable without considering maliciously behaving replicas.

It would be interesting to have a proof on being able to provide, or not, either of these consistency models.

The need for a server As pointed out by anonymous reviewers, in our work on securing the client-side there is an assumption that the punishment is greater than the reward. In some contexts this might not be true, such as in large games where the sacrifice of one for the benefit of many can easily be worth the cost.

The question is if this problem limited to our insecure setting (client replicas) or does a client-server approach using weak consistency suffer from the same problem?

Additionally, it seems that decentralized systems can be manipulated if many (or a majority of) malicious actors falsely state that an actually correct actor misbehaved. Using a server this can be trivially mitigated by having that client use the server instead. Is it at all possible to deal with such behaviour without a server?

Impossible forks One specific problem that secure causal consistency (Section 5.3.1) does not address is creating forks in the causal past that should not have happened due to membership.

For example, when a small group of replicas have created a graph of operations (and their dependencies), a newly joining replica should not be able to fork the operation's causal graph at a point before he joined. A clear example is a chat application, if two people have a conversation and reach a conclusion, a third person joining later should not be able to create a conclusion which, in the eyes of the consistency model, is concurrent to their existing operations. In other words, if the dependency graph $a \prec b$ exists and is shared between all replicas, no replica should later be able to create operation c where $a \prec c$ and $c||b$.

This seems to lead to an implementation which includes membership and some form of checkpointing, where new operations after a given checkpoint have to depend on that checkpoint (and, by transitivity, on all preceding operations).

Securely checkpointing for forcing progress In Chapter 5 we explore the effect of latency on eventual linearizability (Section 5.5.4) and how colluding rational replicas may impact it (Section 5.5.6.3).

As we show, resorting to a service or centralized component to store information of which operations were generated mitigates the problem. There is an un-explored alternative which

would be to let the timestamping system be not only poll-able on which operations exists (as we currently do), but be able to sign client-side version vectors to assign a timestamp. If some replica expects another to not be propagating operations, it requests from the suspected replica a service signed version vector – effectively requesting it to checkpoint its state towards a trusted component. The suspected replica must send its current version vector to the service, have it signed, and propagate to the requesting peer. Intuitively, this disallows withholding operations from the requesting replica.

This direction seems promising, and a direct follow up to the previous point, but careful thought has to be given into colluding replicas.

Small key rotation Our models use assymetric keys to both sign operations (Section 5.4.1) and all messages sent between replicas (Section 5.4.2).

Although to sign operations a large key is a necessity so it cannot be broken (for example, a 2048 bit key size, in current hardware, cannot be circumvented in useful time [218]), signing every single message with such a key is a non-negligible overhead – not only due to increasing message sizes but also in computation.

Ideally, as small as possible keys should be used (for example, 512bit RSA keys). This can be done by obtaining a large key from the server which is used to, every 10 minutes or so, rotate (sign) a smaller key that is then instead used to communicate with the connected peers.

This can easily bring a huge benefit if multiple hop propagation is in place, especially if at every hop multiple milliseconds can be shaved off.

On delivery latency and user notification As our evaluation of secure consistency models shows – specifically Section 5.5.6.2 and Section 5.5.6.3 – delivery latency is both susceptible to attacks and very important for correct application behaviour.

A guarantee that would have a major positive effect on applications would be as follows:

Definition 7.1 (Delay notification). *For any operation op, from the moment it is delivered at any correct replica r_1, then, for every other correct replica r_2, one of the following is always true: a) op is delivered at r_2 within a bounded interval, or, b) r_2 is aware of possible delivery delays.*

The proposed algorithms do not suffice to implement this guarantee. It seems that not only do we need a server summary to ensure eventual delivery at clients (as we use to disallow some attacks, Section 5.4.7), but that the inverse is also necessary: a client summary for each client, propagated to the server which tracks, for every client, any delays. This leads to interesting guarantees on forced propagation, but needs to be explored – intuitively the imposed overhead can easily outweigh the benefit of using the client-side replication model.

"The easiest way to solve a problem is to deny it exists."
– Isaac Asimov. The Gods Themselves (1972)

Bibliography

[1] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 283–292, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[2] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-CRDTs: Making δ-CRDTs Delta-Based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Albert van der Linde, Diogo Serra, João Leitão, and Nuno Preguiça. On Combining Fault Tolerance and Partial Replication with Causal Consistency. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Albert van der Linde, Pedro Fouto, João Leitão, and Nuno Preguiça. The Intrinsic Cost of Causal Consistency. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Albert van der Linde, João Leitão, and Nuno Preguiça. Practical Client-Side Replication: Weak Consistency Semantics for Insecure Settings. In *Proceedings of the VLDB Endowment*, volume 13 (12), page 2590–2605. VLDB Endowment, July 2020.

[6] Albert van der Linde. Edge-cloud hybrid model for distributed apps. In *Eurosys Doctoral Workshop*, 2018.

[7] Albert van der Linde, João Leitão, and Nuno Preguiça. Secure causal delivery with client-side replication. In *Presentation at the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, 2019.

[8] Sara Simões, Albert van der Linde, and Nuno Preguiça. Composition of CRDTs Using References in Key-value Stores. In *Comunicações do 11º Inforum.*, 2019.

[9] Georges Da Costa, Alexey L. Lastovetsky, Jorge G. Barbosa, Juan C. Díaz-Martín, Juan L. García-Zapata, Matthias Janetschek, Emmanuel Jeannot, João Leitão, Ravi Reddy Manumachu, Radu Prodan, Juan A. Rico-Gallego, Peter Van Roy, Ali Shoker, and Albert van der Linde. Programming models and runtimes. In *Ultrascale Computing Systems*, pages 9–63. Institution of Engineering and Technology, 2019.

[10] Tiago Costa, Albert van der Linde, Nuno Preguiça, and João Leitão. Controlo de Acessos em Sistemas com Consistência Fraca. In *Actas do 8º Inforum.*, 2016.

[11] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in Unreal Tournament 2003®. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151, 2004.

[12] Mark Claypool. The effect of latency on user performance in real-time strategy games. *Computer Networks*, 49(1):52–70, 2005.

[13] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. How sensitive are online gamers to network quality? *Communications of the ACM*, 49(11):34–38, 2006.

[14] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 14(2):8–es, 2007.

[15] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 297–304, 2000.

[16] Dan Pritchett. BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. *Queue*, 6(3):48–55, 2008.

[17] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on NoSQL database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[18] Sultana Kalid, Ali Syed, Azeem Mohammad, and Malka N Halgamuge. Big-data NoSQL databases: A comparison and analysis of Big-Table, DynamoDB, and Cassandra. In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*, pages 89–93. IEEE, 2017.

[19] Yan Huang, Tom ZJ Fu, Dah-Ming Chiu, John CS Lui, and Cheng Huang. Challenges, design and analysis of a large-scale p2p-vod system. *ACM SIGCOMM computer communication review*, 38(4):375–388, 2008.

[20] Sipat Triukose, Zhihua Wen, and Michael Rabinovich. Measuring a commercial content delivery network. In *Proceedings of the 20th international conference on World wide web*, pages 467–476, 2011.

[21] Siamak Azodolmolky, Philipp Wieder, and Ramin Yahyapour. Cloud computing networking: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 51(7):54–62, 2013.

[22] Niantic Inc. Ingress (video game). www.pokemongo.com, 2013.

[23] Niantic Inc. Pokémon Go (video game). www.ingress.com, 2016.

[24] Etherpad Foundation. Etherpad. etherpad.org, 2008.

[25] Mentimeter. Mentimeter (app) – Interactive Presentations, Workshops and Meetings. mentimeter.com, 2018.

[26] Geocaching. Geocaching(app) – The world's largest treasure hunt. geocaching.com, 2000.

[27] Nextdoor. Nextdoor (app) – Tap into your neighborhood. nextdoor.com, 2021.

[28] Meetup. Meetup (app) – Find events near you. waze.com, 2002.

[29] Google Waze. Waze (app) – GPS, Maps, Traffic Alerts and Live Navigation. waze.com, 2000.

[30] EC (European Comission). Mobile contact tracing apps in EU Member States. ec.europa.eu/info/live-work-travel-eu/coronavirus-response/travel-during-coronavirus-pandemic/mobile-contact-tracing-apps, 2021.

[31] Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. *arXiv preprint arXiv:0710.1784*, 2007.

[32] World Wide Web Consortium (W3C). WebRTC 1.0: Real-Time Communication Between Browsers. www.w3.org/TR/webrtc/, 2011.

[33] C. Holmberg, S. Hakansson, and G. Eriksson. Web Real-Time Communication Use Cases and Requirements. RFC 7478, RFC Editor, 3 2015.

[34] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, RFC Editor, 10 2008.

[35] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, RFC Editor, 4 2010.

[36] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, RFC Editor, 4 2006.

[37] Joseph Gentle. ShareJS API. github.com/share/ShareJS, 2011.

[38] Google. Google Drive Realtime API. developers.google.com/google-apps/realtime/overview, 2015.

[39] Ryan Shea, Jiangchuan Liu, Edith C-H Ngai, and Yong Cui. Cloud gaming: architecture and performance. *IEEE network*, 27(4):16–21, 2013.

[40] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. A survey on cloud gaming: Future of computer games. *IEEE Access*, 4:7605–7620, 2016.

[41] Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9, 2005.

[42] Peter Laurens, Richard F Paige, Phillip J Brooke, and Howard Chivers. A novel approach to the detection of cheating in multiplayer online games. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 97–106. IEEE, 2007.

[43] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[44] Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[45] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400, 2011.

[46] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[47] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[48] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[49] Leslie Lamport and Clocks Time. the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[50] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, 3:240–247, 1983.

[51] Mahadev Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.

[52] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120, 1995.

[53] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68, 1998.

[54] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(4):531–582, 2006.

[55] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[56] Tim Jungnickel and Tobias Herb. Simultaneous editing of JSON objects via operational transformation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 812–815, 2016.

[57] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. *Proving correctness of transformation functions in collaborative editing systems.* PhD thesis, INRIA, 2005.

[58] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th international conference on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes on Computer Science*, pages 386–400. Springer, 2011.

[59] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of convergent and commutative replicated data types.* PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[60] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[61] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–278, 2012.

[62] 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*, pages 111–126, 2017.

[63] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Optimistic algorithms for partial database replication. In *International Conference On Principles Of Distributed Systems*, pages 81–93. Springer, 2006.

[64] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5):172–182, 1995.

[65] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, pages 30–33. IEEE, 2014.

[66] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V Madhyastha, and Cristian Ungureanu. Simba: Tunable end-to-end data consistency for mobile apps. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[67] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M Levy. Diamond: Automating data management and storage for wide-area, reactive applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 723–738, 2016.

[68] Facebook Inc. Continuing to build News Feed for all types of connections. code.facebook.com/posts/1535185823471329/continuing-to-build-news-feed-for-all-types-of-connections/, 2015.

[69] Douglas B Terry. Replicated data management for mobile computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 3(1):1–94, 2008.

[70] Parse Community. Parse. parse.com, 2015.

[71] Anthony D Joseph, Alan F de Lespinasse, Joshua A Tauber, David K Gifford, and M Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 156–171, 1995.

[72] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[73] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of NSDI*, 2009.

[74] Dejan S Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing, 2002.

[75] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Communications of the ACM*, 53(10):72–82, 2010.

[76] P Krishna Gummadi, Stefan Saroiu, and Steven D Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. *ACM SIGCOMM Computer Communication Review*, 32(1):82–82, 2002.

[77] Dongyu Qiu and Rayadurgam Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. *ACM SIGCOMM computer communication review*, 34(4):367–378, 2004.

[78] David P Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM international workshop on grid computing*, pages 4–10. IEEE, 2004.

[79] Salman A Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[80] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings first international conference on peer-to-peer computing*, pages 99–100. IEEE, 2001.

[81] Kenneth P Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[82] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[83] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[84] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and systems Management*, 13(2):197–217, 2005.

[85] Ayalvadi J Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *International Workshop on Networked Group Communication*, pages 44–55. Springer, 2001.

[86] Joao Leitao, José Pereira, and Luis Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429. IEEE, 2007.

[87] Joao Leitao, Jose Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310. IEEE, 2007.

[88] Ken Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5):8–13, 2007.

[89] Shengdong Xie and Yuxiang Wang. Construction of tree network with limited delivery latency in homogeneous wireless sensor networks. *Wireless personal communications*, 78(1):231–246, 2014.

[90] Ryan Dahl. Node.js. nodejs.org, 2009.

[91] Al Danial. CLoC – count lines of code. *Open source*, 2009.

[92] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Computer Architecture Letters*, 31(01):48–59, 1982.

[93] DigitalBazaar. node-forge: an implementation of TLS and various other cryptographic tools in JavaScript. `github.com/digitalbazaar/forge`, 2020.

[94] Google Inc. Google Drive Realtime Playground. github.com/googledrive/realtime-playground, 2014.

[95] Dale Harvey. Original implementation of Pacman for browsers. github.com/daleharvey/pacman, 2010.

[96] 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*, pages 401–416, 2011.

[97] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414. IEEE, 2016.

[98] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 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)*, pages 453–468, 2017.

[99] Santiago J Castiñeira and Annette Bieniusa. Collaborative offline web applications using conflict-free replicated data types. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–4, 2015.

[100] 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*, pages 13–16, 2012.

[101] 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*, pages 37–42, 2015.

[102] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. Consistency, availability, and convergence. *Tech Report - University of Texas, Austin*, 11:158, 2011.

[103] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of Highly-Available Eventually-Consistent Data Stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 385–394, 2015.

[104] Barbara Liskov and Rivka Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 29–39, 1986.

[105] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*. Australian National University. Department of Computer Science, 1987.

[106] Kenneth P Birman and Thomas A Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

[107] Colin J Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed debugging*, pages 183–194, 1988.

[108] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms*, 1988.

[109] Larry L Peterson, Nick C Buchholz, and Richard D Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems (TOCS)*, 7(3):217–246, 1989.

[110] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *International Workshop on Distributed Algorithms*, pages 219–232. Springer, 1989.

[111] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 43–57, 1990.

[112] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.

[113] Mustaque Ahamad, Phillip W Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281. IEEE, 1991.

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

[115] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information processing letters*, 39(6):343–350, 1991.

[116] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, 1992.

[117] Ravi Prakash, Michel Raynal, and Mukesh Singhal. An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 744–751. IEEE, 1996.

[118] Fekete et al. Eventually-serializable data services. In *Theoretical Computer Science*, 1999.

[119] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[120] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–7, 2012.

[121] 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*, pages 1–14, 2013.

[122] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772, 2013.

[123] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 313–328, 2013.

[124] 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*, pages 1–13, 2014.

[125] 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*, pages 1–14, 2014.

[126] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):1–34, 2016.

[127] Manuel Bravo and Luís Rodrigues. Towards affordable externally consistent guarantees for geo-replicated systems. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–4, 2018.

[128] Ta-Yuan Hsu, Ajay D Kshemkalyani, and Min Shen. Causal consistency algorithms for partially replicated and fully replicated systems. *Future Generation Computer Systems*, 86:1118–1133, 2018.

[129] Tariq Mahmood, Shankaranarayanan Puzhavakath Narayanan, Sanjay Rao, TN Vijaykumar, and Mithuna Thottethodi. Karma: cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency. *IEEE Transactions on Cloud Computing*, 2018.

[130] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 304–316. IEEE, 2019.

[131] Hugo Guerreiro, Luís Rodrigues, Nuno Preguiça, and Nivia Quental. Causality tracking trade-offs for distributed storage. In *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE, 2020.

[132] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. *Communications of the ACM*, 59(4):43–47, 2016.

[133] Per Cederqvist, Roland Pesch, et al. Version management with CVS, 1992.

[134] C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version control with subversion: next generation open source version control.* O'Reilly Media, Inc., 2008.

[135] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[136] Richard Andrew Golding. *Weak-consistency group communication and membership.* PhD thesis, Citeseer, 1992.

[137] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-Based CRDTs by Delta-Mutation. In *Networked Systems - Third International Conference, Revised Selected Papers*, NETYS '15, pages 62–76, 2015.

[138] Russell Brown, Zeeshan Lakhani, and Paul Place. Big(ger) Sets: decomposed delta CRDT Sets in Riak. *CoRR*, abs/1605.06424, 2016.

[139] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403. IEEE, 2009.

[140] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

[141] Ted Wobber, Thomas L Rodeheffer, and Douglas B Terry. Policy-based access control for weakly consistent replication. In *Proceedings of the 5th European conference on Computer systems*, pages 293–306, 2010.

[142] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):1–38, 2011.

[143] LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[144] Harry C Li, Allen Clement, Edmund L Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. BAR gossip. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 191–204, 2006.

[145] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[146] HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

[147] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[148] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 253–267, 2003.

[149] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 163–176, 2008.

[150] Michael K Reiter and Kenneth P Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):986–1009, 1994.

[151] Sisi Duan, Michael K Reiter, and Haibin Zhang. Secure causal atomic broadcast, revisited. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 61–72. IEEE, 2017.

[152] Sisi Duan, Michael K Reiter, and Haibin Zhang. BEAT: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.

[153] Melanie Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, Inc., 2015.

[154] Christian Cachin et al. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310:4. Chicago, IL, 2016.

[155] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pages 45–59, 2016.

[156] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[157] Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. Eventually linearizable shared objects. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 95–104, 2010.

[158] Reza Curtmola and Cristina Nita-Rotaru. BSMR: Byzantine-resilient secure multicast routing in multihop wireless networks. *IEEE Transactions on Mobile Computing*, 8(4):445–459, 2008.

[159] Jing Dong, Reza Curtmola, and Cristina Nita-Rotaru. Secure network coding for wireless mesh networks: Threats, challenges, and directions. *Computer Communications*, 32(17):1790–1801, 2009.

[160] Linyuan Zhang, Guoru Ding, Qihui Wu, Yulong Zou, Zhu Han, and Jinlong Wang. Byzantine attack and defense in cognitive radio networks: A survey. *IEEE Communications Surveys & Tutorials*, 17(3):1342–1363, 2015.

[161] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (Intel® SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9, 2016.

[162] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.

[163] Sridhar Alagar. Causally ordered message delivery in mobile systems. In *1994 First Workshop on Mobile Computing Systems and Applications*, pages 169–175. IEEE, 1994.

[164] Sridhar Alagar and Subbarayan Venkatesan. Causal ordering in distributed mobile systems. *IEEE Transactions on Computers*, 46(3):353–361, 1997.

[165] American National Standards Institute, Accredited Standards Committee (ANSI, ASC). X9.95-2016 Standard for Trusted Time Stamps, 2016.

[166] Internet Engineering Task Force (IETF), Carlisle Adams, Pat Cain, Denis Pinkas, and Robert Zuccherato. RFC3161 - Internet X.509 public key infrastructure time-stamp protocol (TSP), 2001.

[167] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[168] National Institute of Standards and Technology (NIST) and Kang B Lee. Standard for A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, 2019.

[169] Markus Ullmann and Matthias Vögeler. Delay attacks — Implication on NTP and PTP time synchronization. In *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 1–6. IEEE, 2009.

[170] Bassam Moussa, Mourad Debbabi, and Chadi Assi. Security assessment of time synchronization mechanisms for the smart grid. *IEEE Communications Surveys & Tutorials*, 18(3):1952–1973, 2016.

[171] Omer Deutsch, Neta Rozen Schiff, Danny Dolev, and Michael Schapira. Preventing (Network) Time Travel with Chronos. In *NDSS*, 2018.

[172] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, 2018.

[173] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[174] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[175] David L Mills and Poul-Henning Kamp. The nanokernel. In *Proceedings of the 32th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 423–430, 2000.

[176] Diana Andreea Popescu and Andrew W Moore. PTPmesh: Data center network latency measurements using PTP. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 73–79. IEEE, 2017.

[177] Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

[178] Vadim Drabkin, Roy Friedman, and Marc Segal. Efficient byzantine broadcast in wireless ad-hoc networks. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 160–169. IEEE, 2005.

[179] Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, pages 21–es, 2004.

[180] Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

[181] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[182] Chris Piro, Clay Shields, and Brian Neil Levine. Detecting the sybil attack in mobile ad hoc networks. In *2006 Securecomm and Workshops*, pages 1–11. IEEE, 2006.

[183] Brian Neil Levine, Clay Shields, and N Boris Margolin. A survey of solutions to the sybil attack. *University of Massachusetts Amherst, Amherst, MA*, 7:224, 2006.

[184] Jochen Dinger and Hannes Hartenstein. Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration. In *First International Conference on Availability, Reliability and Security (ARES'06)*, pages 8–pp. IEEE, 2006.

[185] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding Virtualization Capabilities to the Grid'5000 Testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[186] Waheed Iqbal, Matthew N Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, 2011.

[187] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[188] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[189] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

[190] Yong Liu, Yang Guo, and Chao Liang. A survey on peer-to-peer video streaming systems. *Peer-to-peer Networking and Applications*, 1(1):18–28, 2008.

[191] Xiaosong Lou and Kai Hwang. Collusive piracy prevention in P2P content delivery networks. *IEEE Transactions on Computers*, 58(7):970–983, 2009.

[192] Rachid Guerraoui, Anne-Marie Kermarrec, Matej Pavlovic, and Dragos-Adrian Seredinschi. Atum: Scalable group communication using volatile groups. In *Proceedings of the 17th International Middleware Conference*, pages 1–14, 2016.

[193] Danny Dolev, Ezra N Hoch, and Robbert Van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *International Conference on Principles of Distributed Systems*, pages 343–357. Springer, 2007.

[194] Harry C Li, Allen Clement, Mirco Marchetti, Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Mike Dahlin. FlightPath: Obedience vs. Choice in Cooperative Services. In *OSDI*, volume 8, pages 355–368, 2008.

[195] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[196] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 245–260. IEEE, 2016.

[197] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[198] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. Trustjs: Trusted client-side execution of JavaScript. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[199] Erick Bauman and Zhiqiang Lin. A case for protecting computer games with SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, pages 1–6, 2016.

[200] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.

[201] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[202] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[203] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[204] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[205] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.

[206] Akhil Gupta and Rakesh Kumar Jha. A survey of 5G network: Architecture and emerging technologies. *IEEE access*, 3:1206–1232, 2015.

[207] Sebastian Burckhardt et al. Principles of Eventual Consistency. *Foundations and Trends® in Programming Languages*, 1(1-2):1–150, 2014.

[208] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[209] Ronald L Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[210] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978.

[211] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[212] Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory*, pages 127–140. Springer, 1983.

[213] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[214] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI Replication. In *NSDI*, volume 6, pages 5–5, 2006.

[215] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316, 2001.

[216] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *2010 29th IEEE Symposium on Reliable Distributed Systems*, pages 214–224. IEEE, 2010.

[217] Luís Rodrigues and Paulo Verissimo. Causal separators and topological timestamping: an approach to support causal multicast in large-scale systems. In *Proceedings of the 15th International Conference on Distributed Systems*. Citeseer, 1995.

[218] Arjen K Lenstra and Eric R Verheul. Selecting cryptographic key sizes. *Journal of cryptology*, 14(4):255–293, 2001.