António José Sá Barreto

BsC in Computer Science and Engineering

Conflict-Free Replicated Data Types in Dynamic Environments

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Hervé Miguel Cordeiro Paulino, Assistant Professor, NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2019

Conflict-Free Replicated Data Types in Dynamic Environments

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my family and friends.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Hervé Paulino, for his patience, availability, supervision and the numerous insightful discussions provided during the development of this thesis, allowing me to better understand and build a concrete solution. It is with much gratitude and recognition, that I thank Prof. Nuno Preguiça and João Silva for the opportune contributions and suggested implementation details that helped improve the overall quality of the developed solution.

Evidently, I am thankful for being a part of the great family that is the Department of Informatics of the NOVA University of Lisbon, composed by its employees, students and teachers for providing me with the knowledge, working installations and tools that made possible my education, surely shaping my future in meaningful ways I can't even begin to comprehend. In addition, this work was partially supported by the Faculty of Sciences and Technology of the NOVA University of Lisbon in the context of the DeDuCe project (PTDC/CCI-COM/32166/2017) and the NOVA LINCS research center (UID/CEC/04516/2013), which I would like to thank for providing me with a thesis topic.

A very special thanks goes towards my parents, Timóteo Barreto and Lúcia Barreto, for the support given to me through this whole stage of my life and for the countless times I felt down and they encouraged me to keep going, despite being distant and only being with them a handful of days a year.

Last but not least, I want to express my sincere gratitude to my close friends, some of whom I've met through my academic life. Every one of them, contributed by making me more comfortable in times of need and showing camaraderie in the many sleepless nights made throughout this academic course. To everyone that contributed directly or indirectly, thank you!

Abstract

Over the years, mobile devices have become increasingly popular and gained improved computation capabilities allowing them to perform more complex tasks such as collaborative applications. Given the weak characteristic properties of mobile networks, which represent highly dynamic environments where users may experience regular involuntary disconnection periods, the big question arises of how to maintain data consistency. This issue is most pronounced in collaborative environments where multiple users interact with each other, sharing a replicated state that may diverge due to concurrency conflicts and loss of updates.

To maintain consistency, one of today's best solutions is Conflict-Free Replicated Data Types (CRDTs), which ensure low latency values and automatic conflict resolution, guaranteeing eventual consistency of the shared data. However, a limitation often found on CRDTs and the systems that employ them is the need for the knowledge of the replicas whom the state changes must be disseminated to. This constitutes a problem since it is inconceivable to maintain said knowledge in an environment where clients may leave and join at any given time and consequently get disconnected due to mobile network communications unreliability.

In this thesis, we present the study and extension of the CRDT concept to dynamic environments by introducing the developed P/S-CRDTs model, where CRDTs are coupled with the publisher/subscriber interaction scheme and additional mechanisms to ensure users are able to cooperate and maintain consistency whilst accounting for the consequent volatile behaviors of mobile networks. The experimental results show that in volatile scenarios of disconnection, mobile users in collaborative activity maintain consistency among themselves and when compared to other available CRDT models, the P/S-CRDTs model is able to decouple the required knowledge of whom the updates must be disseminated to, while ensuring appropriate network traffic values.

Keywords: CRDT, Eventual Consistency, Dynamic Environments, Publish/Subscribe, Data Propagation, Mobile Networks.

Resumo

Ao longo dos anos, os dispositivos móveis tornaram-se cada vez mais populares e adquiriram capacidades computacionais acrescidas, o que lhes permitem realizar tarefas cada vez mais complexas tais como aplicações colaborativas. Tendo em conta as redes móveis, que representam ambientes altamente dinâmicos onde os utilizadores podem incorrer em períodos regulares de desconexão involuntários, levanta-se a grande questão de como manter a consistência dos dados. Este problema é mais pronunciado em ambientes colaborativos onde vários utilizadores interagem entre si, compartilhando um estado replicado que pode divergir devido à conflitos de concorrência e perda de atualizações.

Para manter a consistência, uma das melhores soluções dos dias hoje é os *Conflict-Free Replicated Data Types* (CRDTs), que fornecem baixos valores de latência e resolução automática de conflitos, garantindo a consistência eventual dos dados compartilhados. No entanto, uma limitação frequentemente encontrada nos CRDTs e nos sistemas que os empregam, é a necessidade do conhecimento das réplicas para qual as mudanças de estado devem ser disseminadas. Isto constitui um problema, pois torna-se inconcebível manter o referido conhecimento em ambientes em que os clientes podem sair, entrar e serem desconectados a qualquer momento, devido à fraca natureza das redes móveis.

Nesta dissertação, apresentamos o estudo e a extensão do conceito de CRDT a ambientes dinâmicos através da introdução do modelo P/S-CRDTs, em que os CRDTs são acoplados ao esquema de interação publicador/subscritor e mecanismos adicionais para garantir que os utilizadores possam cooperar e manter a consistência enquanto lidam com os consequentes comportamentos voláteis das redes móveis. Os resultados experimentais mostram que em cenários de desconexão, os utilizadores móveis em atividades colaborativas conseguem manter consistência entre si e que quando comparado com outros modelos disponíveis de CRDT, o modelo P/S-CRDTs permite desacoplar o conhecimento necessário de quem as atualizações deve ser disseminadas para, ao mesmo tempo garantindo valores de tráfego de rede apropriados para redes móveis.

Palavras-chave: CRDT, Consistencia Eventual, Ambientes Dinamicos, Publish/Subscribe, Propagação de Dados, Redes Móveis.

Contents

List of Figures

List of Tables

Listings

Acronyms

δ-CRDT	Small delta-based CRDT.
Δ-CRDT	Big delta-based CRDT.
CmRDT	Operation-based CRDT.
CRDT	Conflict-Free Replicated Data Type.
CvRDT	State-based CRDT.
MCC	Mobile Cloud Computing.
MEC	Mobile Edge Computing.
P/S-CRDT	Publish/Subscribe CRDT.

Chapter 1

Introduction

1.1 Context & Motivation

Throughout the years, mobile devices have become more portable, smarter and affordable, leading to a surge of users embracing such devices as smartphones, tablets and smartwatches which nowadays consists the mainstream way of accessing the internet for mundane activities, such as accessing social networks, communicating with friends and checking emails. Due to the growth, 13.1 billion mobile devices were registered worldwide in 2019 and a projected value of 16.8 billion mobile devices worldwide by 2023 (Figure 1.1).

Thus, according to studies conducted by Cisco [31], mobile traffic accounts for 51% of the global communication traffic, generating 19.01 exabytes per month in 2018 and expecting mobile data traffic to reach 77.5 exabytes per month in 2022 at an annual growth rate of 46%. Furthermore, researchers reveal that the exponential mobile traffic growth is directly linked to the increased consumption of media streaming services, representing the largest chunk of mobile user interactions.

Hence, services such as online shopping, social networks and media streaming have millions of concurrent users accessing on a daily basis from these mobile devices, as a consequence applications that revolve around these services must meet clients expectations of availability, scalability and low latency. To assure that services are able to scale, replication techniques are employed where services are replicated across multiple replicas around the world. Having user's data scattered among multiple replicas enables services to provide better fault-tolerance, however since mobile users grow at an unprecedented rate, centralized network systems observe an increased resource consumption generating expensive network loads that not only becomes harder for the services to manage but also resulting in high latency for mobile users.

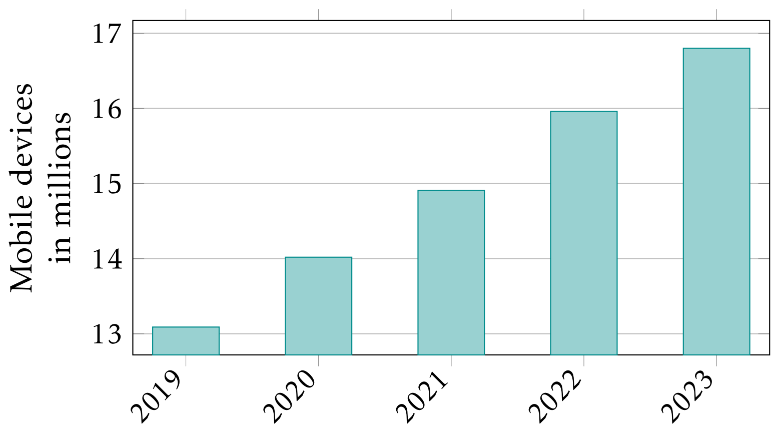


Figure 1.1: Projection for the number of mobile devices worldwide from 2019 to 2023 in billions. Based on [33].

Moreover, as users demand network systems for better performance and overall better user experience, whilst such systems must deal with large amounts of data traffic, traditional approaches such as Mobile Cloud Computing (MCC) [49] incur new noticeable challenges such as security vulnerability, low coverage, lagged data transmission and revealing unacceptable performance values as communication between mobile users and remote cloud centers is often over a long distance, adding to the latency in cloud computation. Moreover, MCC is not a suitable approach for scenarios involving real-time applications and guaranteeing high quality of service.

Therefore, Mobile Edge Computing (MEC) [18] is introduced as a way to address the challenges that are raised by MCC systems. By expanding the traditional cloud architecture with additional datacenter layers, edge computing provides computation and storage closer to the users (Figure 1.2), offering cloud computing capabilities within the radio access network at a considerable bandwidth cost decrease, compared to MCC systems. By adding datacenters closer to the client device, edge computing makes possible next generation mobile and IoT applications that require low latency or that produce large volumes of data. In addition, end users get more powerful computing, energy efficiency, storage capacity, mobility, location and context awareness support for end users.

In order to communicate, mobiles nodes in MEC environments exchange messages through a wireless medium, using standard technologies such as Wi-Fi, WiFi-Direct, Bluetooth, or a combination of these [50, 54, 55]. Unfortunately, by nature, mobile network communications are unreliable, being prone to limited bandwidth, regular periods of disconnection and overall poor connectivity, thus entailing new dilemmas around the trade-offs between high availability, data consistency and performance.

As it is impossible for a distributed system to provide availability, partition tolerance and consistency simultaneously [26], some services opt to provide two of the above characteristics and discard the third. For instance, more robust services, require data to be

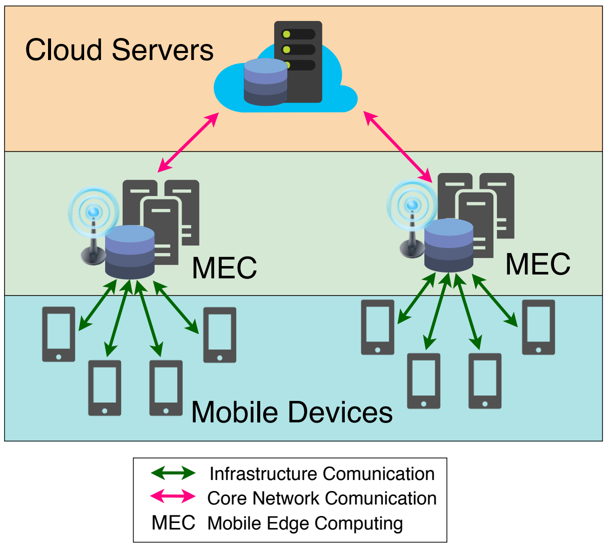

Cloud Servers
MEC
MEC
Mobile Devices
Infrastructure Comunication
Core Network Comunication
MEC Mobile Edge Computing

Figure 1.2: Architecture of Mobile Edge Computing (MEC).

consistent at all times, thus providing strong consistency guarantees alongside with availability, at the cost of not ensuring partition tolerance. On the other hand, some services rather guarantee partition tolerance and high availability, at the cost of having a weaker form of consistency.

Thus, in mobile environments where the network may be composed by a large amount of clients and subject to the unreliable side of over-the-air communication channels, often a debate is created around strong and weak forms of consistency. Since, over-the-air communication channels are characteristically unreliable, thus inducing various common erratic behaviors such as network disconnection and unpredictable message loss. In collaborative scenarios where mobile users share a common state throughout the activity, said prominent erratic behaviors of mobile environments create some pragmatic side effects.

A user in a collaborative activity, such as a shared to-do list application may momentarily suffer a disconnection period, whilst the remaining intervenients of the application are still operating. Once the user reconnects with the network, some data inconsistencies may be noticeable, such as missing items comparatively to the remaining intervenients who kept issuing modifications during the user's disconnection, meaning that the user has an out of date version of the state, hence diverging in correlation the correct and up to date state as the activity continues. Additionally, a very common issue found in collaborative applications are concurrency conflicts, originated from concurrent activity within the collaborative application. Thus, if a conflict occurs and the system is unable to determine the correct order of the operation, application users will experience a permanent state divergence, as different operation orders are applied to different users.

In order to to eliminate the side effects originated in dynamic environments, some solutions were formulated, such as rollback mechanism that when detected a conflict, allows the system to revert to prior state where no conflicts are present and apply the resolution algorithm for the detected conflicting operations. Although being able to solve conflicts, rollback mechanisms suffer from severe problems either by relying on a primary replica to run the conflict resolution as well as some systems that employ the rollback technique requiring all replicas be frozen whilst the conflicts are solved [45], thus representing a non optimal approach for mobile users, as it doesn't meet the availability expectations of mobile users.

Ultimately, ensuring data consistency in dynamic environments where communications are unreliable, whilst meeting mobile users expectations of low latency and high availability is an ongoing struggle. Notably in times where the number of mobile devices increases exponentially and with the introduction of new emergent technologies such as 5G, guaranteeing data consistency is of paramount importance.

1.2 Problem & Challenges

As the dilemma to be studied in this thesis revolves around the characteristic dynamic environment originated by mobile networks, where users may join as well as willingly leave the network, or unwillingly due to over-the-air communication channels being susceptible to frequent periods of disconnection, thus representing an environment where users show highly volatile behaviour. Due to the unpredictability and unreliability of over-the-air communication channels, problems arise and are more noticeable in applications based on collaborative activity among users, where a state defined by the application is shared and managed among users.

As a result, users who experience a disconnection in such applications are prone to inconsistencies, since during the time out of the network, the user's state diverges from the common shared state among users. Not only that, but concurrent operations among users, showcases the same effects by creating conflicts, to which networks alone are not capable of solving.

Thus, one of today's best approaches to obtain conflict resolution at relatively low performance impact are the CRDTs [52], which are data types that allow replication over replicas with no coordination, guaranteeing eventual state convergence. CRDTs have proven to be a suitable approach to conflict resolution, ensuring that replicas state eventually converge and offering minimal latency compared to other solutions. This, allows CRDTs to have low requirements from the network, which in turn demonstrates why CRDTs are suitable for distributed storage systems such as Riak [1] and AntidoteDB [2], collaborative applications like Treedoc [48] and LSEQ [41], and peer-to-peer networks like Legion [39].

Although, CRDTs present themselves as a way to achieve conflict resolution, existing work on eventually consistent systems employing CRDTs are still heavily focused in centralized approaches, where the knowledge of participating nodes is taken for granted. The same cannot be told about the dynamic environments in question which are loosely coupled networks, where maintaining such a knowledge of the system is inconceivable.

Therefore, the main questions we intend on asking ourselves is how to guarantee that nodes will be able to continue to interact and view a consistent state of the storage when participating nodes in collaborative activity show volatile behavior, by being capable of leaving and joining the network at any given time and how to enable replicas to disseminate local updates without holding any reference to the receiving replicas.

1.3 Proposed Solution

The objective of this thesis, as means of solving the previously presented challenges, is to take the concept of CRDTs and extend it to dynamic environments by coupling the CRDT objects with a system capable of using the publish/subscribe interaction scheme, with the intent of taking advantage of the decoupling characteristics of publish/subscribe systems.

Thus we intend to implement a new design of CRDTs, the Publish/Subscribe CRDTs (P/S-CRDTs), where mobile users may enjoy a collaborative application deployed in a peer-to-peer network. Whereas the result of coupling these CRDTs on top of a publish/-subscribe system, users are able to maintain a shared common state among its peers, being able to perform volatile scenarios such as disconnection and eventual re-entry in the network, without feeling the repercussions.

To do so, P/S-CRDTs specify the manner of how updates must be disseminated, how update's content are built according to the employed synchronization models, as well required mechanism to ensure that mobile users who experience disconnection and late entry events are able to regain consistent, by retrieving the needed missing information required to converge. Furthermore, by using the publish/subscribe interaction scheme as the medium for update dissemination, we aim to decouple the knowledge of whom updates must be sent to, and even make use of publish/subscribe time, space and synchronization decoupling properties, enabling disconnected work for users.

1.4 Contributions

As the solution presented, comprises a new design of CRDTs capable of using the publish/subscribe interaction scheme to disseminate updates and provide the decoupling between senders(publishers) and receivers(subscribers), this thesis presents the following contributions:

1. Proposal of an extension of the CRDT concept to dynamic environments (P/S-CRDT);
2. Implementation of P/S-CRDTs on top of a topic-based publish/subscribe system for mobile networks;
3. Development of a mobile collaborative simulations;

4. Evaluation of the developed of the CRDT concept comparatively with other solutions;

5. Implementation of techniques for storage optimization;

1.5 Communications

Throughout the work done in this dissertation, the developed solution in this thesis was presented in the following publication:

- **CRDTs em Ambientes Dinâmicos** [24] António Barreto, Hervé Paulino, João Silva and Nuno Preguiça. Actas do décimo primeiro Simpósio de Informática, Guimarães, Portugal, September, 2019.

1.6 Document Outline

The remainder of this document is organized as follows. In Chapter 2 we introduce the major areas of interest regarding the context of this thesis, state-of-the-art implementations of CRDTs, eventually consistent mobile systems and a brief introduction to the generic publish/subscribe interaction scheme. Followed by the model design used as the proposed solution for this thesis, the P/S-CRDT model in Chapter 3.

Afterwards, in Chapter 4, we showcase the implementation of the proposed solution with an existing publish/subscribe system as proof of concept. In Chapter 5 we detail the developed testing environments and show the results obtained in our evaluation process, describing the results comparatively to other systems. Finally, arriving at Chapter 6 where all conclusions about the developed work are made, along with possible future work derived from the developed work.

Chapter 2

State-of-the-Art

The purpose of this chapter is to exhibit concepts, implementations and related work of multiple technologies and network systems adopting replicated data structures, that are pertinent to the development of this dissertation. To successfully develop the proposed solution, the research was heavily centered on specific areas of interest such as the eventual consistency model, conflict-free replicated data types and preexisting mobile edge systems in a collaborative environment that benefit from conflict-free replicated data structures.

To further understand the shortcomings of a strong consistency model, the uprising of weaker forms of consistency and how should one approach to achieve conflict resolution in said consistency model, a brief explanation is presented in Section 2.2.2.

Moreover, given that the goal is to extend the CRDT concept to dynamic environments a research was conducted on the various types of CRDTs in order to gather the advantages, disadvantages, and limitations, followed by a concise analysis of the proposed objective and which of the CRDTs shall bring the most to the table, showcasing the possible improvements deriving out of the extension of the CRDT concept to dynamic environments in Section 2.3.

To conclude the Chapter, in Section 2.4, we present several cases of applications in a collaborative environment, employing an eventual consistency scheme. Furthermore, in Section 2.5 a succinct overview of the publish/subscribe interaction scheme is made, detailing its roles and properties.

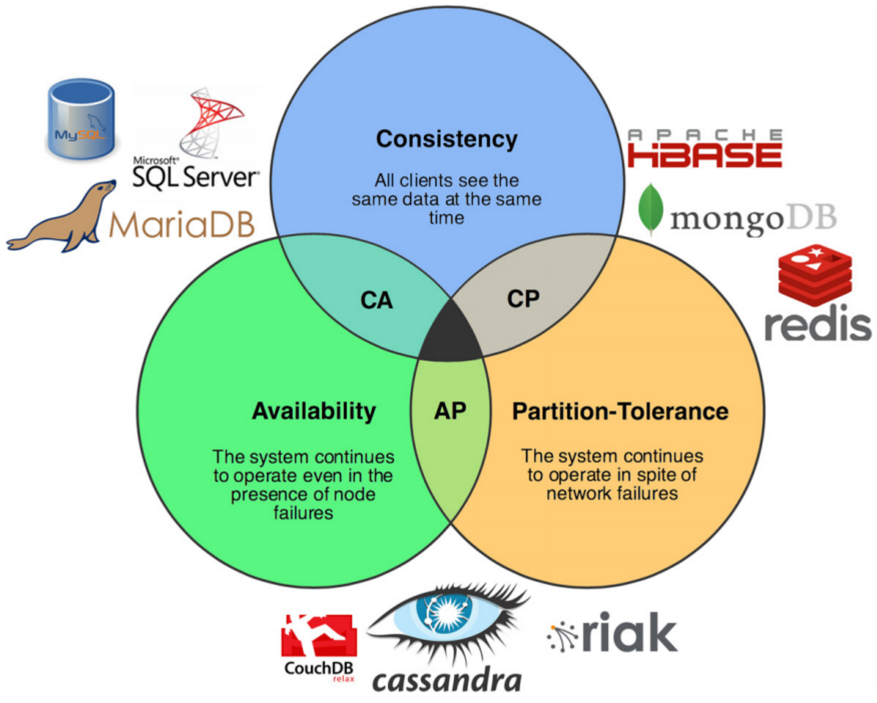

Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
MySQL
Microsoft SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB relax
cassandra
riak

Figure 2.1: CAP Theorem. Taken from [40].

2.1 CAP theorem

In this day and age, large scale infrastructure services such as Amazon's S3 (Simple Storage Service) [3] and Riak® KV [1] that provide resources for a great array of applications, should present guarantees in the areas of security, availability, scalability and performance, whilst continuously serving millions of customers around the globe. Given the scale, additional challenges arise. By processing enormous amounts of requests, these systems design and architecture must take into account that data inconsistencies occur in concurrent environments and should not be visible to the clients [57]. To that end, various replication techniques are used to guarantee a consistent state across multiple nodes, grant higher performance and availability.

It would be naive to think that when an update operation takes place all observers will immediately see the effect of that same update since as we know through advances in technology, new larger internet systems have emerged and new ways of accessing the network like mobile phones have become mainstream. Thus, entailing new dilemmas around the trade-offs between high availability and data consistency, in order to conceal the high latency, regular periods of disconnection and poor connectivity of mobile network communications.

The *CAP theorem* presented by Eric Brewer [26], states that the properties of availability, data consistency and network partition tolerance, only two can be achieved at any given time (Figure 2.1).

In large scale distributed systems, network partitions are presumed to happen, hence consistency and availability cannot be reached simultaneously at any given time. Thus bringing us to the conclusion that either a consistency relaxation must take place granting the system high availability or on the other hand making consistency a priority leaving the system unavailable under certain conditions. Considering the goal of overcoming the obstacles provided by a dynamic environment, such as a mobile edge network, where availability is not always granted, it seems rather obvious that a consistency relaxation is the route to take in order to tackle the obstacles.

2.2 Consistency Models

2.2.1 Strong Consistency

A *strong consistency* model, implies that after an update is completed, any following access by a process yields the return of the updated value, therefore granting a consistent state across all replicas. Regrettably, it comes at a cost for guaranteeing such strong characteristics, requiring an increased communication cost. This communication requirement becomes problematic whenever connections are slow or unavailable. Hence, any system that provides strong consistency is susceptible to a state of low *availability*, for example, if the network should become partitioned making communications not viable, then multiple clients may become ineffective since they can no longer perform updates or read data. Moreover,*Performance* additionally sees a fall-off under a strong consistency model, for example, if each update requires a round-trip to some central unit and communication is slow due to the geographical distance between the client and the server [28]. Also when taking into consideration the characteristics of edge networks, such as over-the-air communication congestion and unreliability, a stronger consistency would be impractical considering a large number of nodes requesting and editing a huge number of data in the same network.

2.2.2 Eventual Consistency

Due to the limitations of a strong consistency scheme, we adopt a weaker form of consistency frequently designated as eventual consistency in order to achieve consensus over operations performed on the data structure. The *eventual consistency* model does not ensure that following accesses yield the updated value, in turn, it guarantees that if no new updates take place then eventually all accesses will yield the last updated value, thus effectively making a trade-off between consistency and availability [22]. There is also an abundant number of variations that derive from the eventual consistency model, that are important to consider[28, 34, 57]:

- **Causal Consistency**: Model based on the *happens before* relation, distinguishing between events that are causally related and those that are not. *If process* A *has communicated to process* B *that it has updated a data item, a subsequent access by process*

B *will return the updated value, and a write is guaranteed to supersede the earlier write* [21]. An access made by another process *C* that has no causal relationship to process *A* is subject to consistency irregularities. Since in this model there is no necessity for collaboration with other nodes to obtain the correct order of updates, it is regarded as a fast consistency model according to *Attiya and Friedman* [21]. It is regarded as a weak model since it can allow access to stale data and different replicas may have different state values due to concurrent operations if no reconciliation techniques applied.

- **Session consistency**: Model where a process accesses a node in the context of a session. Session being an abstraction for the sequence of read and write operations performed during the execution of an application. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session must be created and the guarantees do not overlap the sessions.

- **Read-your-writes consistency**: Established upon the concept that natural order of operations for a user is to be able to read what they write in the very same sequence. Therefore, as long as the session lasts, clients will access the last value they have updated. Considering process *A* after having updated a data item, all subsequent accesses operate on the updated value and never sees an older value. Read operations within the same session occasionally see other writes that are performed outside the session, since there may be other users performing writes to other servers and these updates could be seen in the present session and server. Thus, a user might perceive the interaction with the system by other users.

- **Monotonic read consistency**: From the assumption that when requesting more than one read in the same session, the user expects to see more operations over time and not fewer. The users can see the evolution of the system as they read through time increasingly within the session, thus giving the impression that the system is growing and changing through time. Then if a process has seen a particular value for the data item, any subsequent accesses will never return any previous values.

- **Monotonic write consistency**: For a user of the current session and other users from outside the session, a write is only updated in the server if the copy includes all the previous session writes. As a result, the user session can be sure of the order between two write operations, taking into account the cost of coordination among replicas and requiring users to agree in a write order. Systems that do not guarantee this level of consistency are notoriously difficult to program.

- **Strong eventual consistency**: Eventual consistency systems will execute an update immediately, finding later that it conflicts with another, requiring rollback mechanisms to resolve this conflict. This develops a waste of resources, demanding consensus to ensure that all replicas conciliate conflicts in the same way. To avoid this, a stronger condition is required, saying that an object is strongly eventually consistent if it is eventually consistent and guarantees strong convergence, by validating that correct replicas that have delivered the same updates have equivalent state [52]. Summing up, Strong Eventual Consistency (SEC) is a model that reaches a compromise between strong and eventual consistency, requiring a system to be eventually consistent and its replicas to have a recipe to solve conflicts automatically, hence not needing consensus, allowing n-1 nodes to be down.

Numerous combinations of these variations are possible, for instance, one may have monotonic reads with session-level consistency, which are very desirable in an eventual consistency system by making it simpler for developers to build applications, whilst allowing the storage system to provide high availability and relax consistency [57]. A few application scenarios are possible yet it depends on the particular application and whether or not one can deal with the consequences.

Given the context of this thesis, a Strongly Eventually Consistent system is desirable since it is possible to implement it in a decentralized setting without any central server or leader, and allowing local execution at each node to proceed without waiting for communication with other nodes, unlike strongly consistent systems [34]. Moreover, large scale deployments of strong eventual consistency include storage services and collaborative editing applications such as Google Docs [4].

Additionally, the method of resolving conflicts within an eventually consistent model comprises many challenges such as what and when to resolve as well as who should address the conflict resolution. GitHub [5], a hosting service for version control, is an example of a collaborative environment where manual conflict resolution is employed by the system. For instance, when a user commits the code changes and then pushes it to some remote node where other users can access it, whilst another user concurrently changes the same files, the system will prompt the user for a manual resolution of the conflict in order to maintain a single consistent copy across all users in the collaborative environment.

There are different plans of attack to handle these conflicts, a number of systems let the user manually resolve them, some choose one update as the winner and throw away the other concurrent updates and other systems merge concurrent updates automatically such as Riak [1] that uses CRDTs to perform the merge operation [27]. Conclusively acquiring a consistent state requires two properties, *convergence* and *validity*. To ensure convergence, a system must reconcile differences between multiple copies of distributed data, requiring that all replicas eventually agree and validity that they agree upon a sensible resolution.

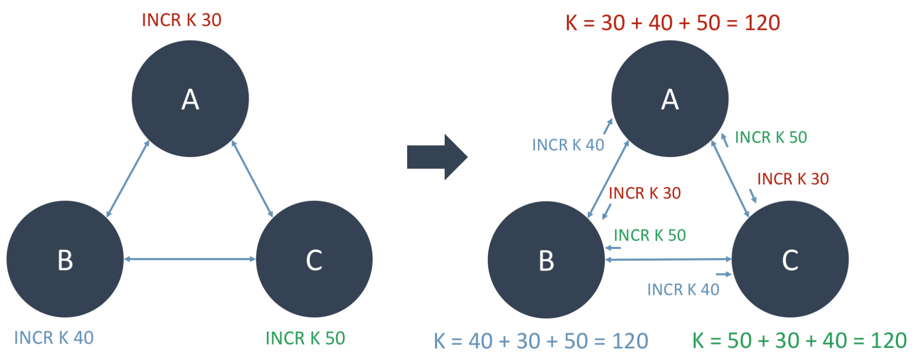


Figure 2.2: Example of CRDT-based conflict resolution.

2.3 Conflict Free Replicated Data Types

The *Conflict-Free Replicated Data Types* (CRDTs) [52] are a family of abstract data types designed for highly available systems, that can be updated without the expensive synchronization by allowing replicas to be modified without the need of coordination, avoiding complex roll-back and conflict resolution mechanisms (Figure 2.2). CRDTs represents a suitable solution for replication in eventual environment given their asynchronous nature, promising eventual convergence and remaining responsive, available and scalable despite high network latency, faults or disconnection [46].

Considering the relatively new genesis, the CRDTs have proven to be very useful, seeing the increasing amount of applications and storage services adopting CRDTs, such as Bet365 [6], SoundCloud [7], GitHub [5], AntidoteDB [2] and Redis [8].

2.3.1 State-based CRDTs

State-based CRDTs (CvRDTs) employ a state-based(or passive) synchronization model, where replicas synchronize by periodically exchanging their full local state, along with metadata information. For the payload of the data type a semilattice is used, hence all operations must comply with associative, commutative and idempotent properties.

When a replica receives the state from another replica, it merges the received state with its local state with the help of a merge function defined by the data type. Updates generally are first performed locally at the source, and then later sent to an arbitrary replica, where it will be merged with the state of that replica using the merge function (Figure 2.3).

Every update eventually reaches every replica and converges states if [46]:

- **(i)**: The possible states of the CRDT are partially ordered forming a join semilattice.
- **(ii)**: An update mutates the state of a replica by an inflation, producing a new state that is larger or equal to the original state.

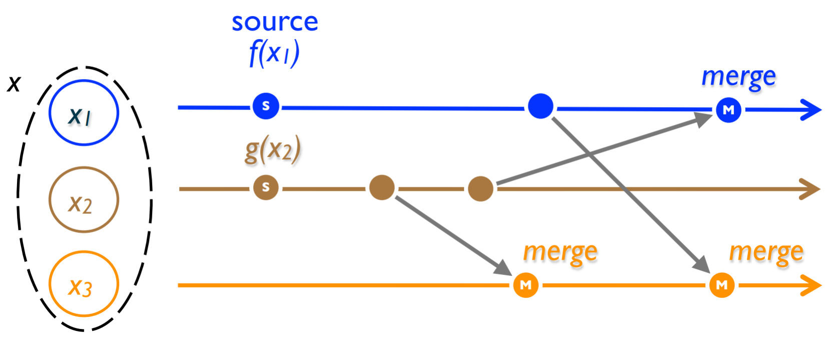


Figure 2.3: State-based replication. Taken from [51].

- **(iii)**: The merge function produces the join (least upper bound) of two states.

One major downside of using this approach lies in the considerable communication overhead imposed by the propagation of entire data objects, notably 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.

On the other hand, state-based CRDTs are usually easier to implement, because additional mechanisms for replication are not required and all information is carried by the state. Moreover, they show a more flexible approach, by having less requirements regarding the underlying communication channel. Additionally, multiple updates can be combined to a single replication step [37, 46, 51, 52]. The predominant usage of CvRDTs is in file systems such as NFS, AFS, and Coda [51], as well as in key-value stores like Dynamo [9] and Riak [1].

2.3.2 Operation-based CRDTs

Operation-based CRDTs (CmRDTs) make use of the operation-based(or active) synchronization model, solving the issue of the previous synchronization model by simply propagating the update operation metadata, that mutated the local state, to every other replica. Additionally, CmRDTs require all operations to be delivered according to some specific order, causal order being the most common.

Furthermore, operation-based CRDTs need to define for each operation, a *prepare-update* function to be executed in the replica where the update is submitted (source), followed immediately by the *effect-update* function (effector), if this were not true, there would be no causality between successive updates [52]. The effector function has the purpose of encoding the side-effects of the update, responsible for the mutation of the state, so that it may be delivered to all replicas with the intention of accomplishing convergence.

Updates for CmRDTs are first carried out locally at the source and consequently sent to all other replicas, thus requiring a reliable delivery to guarantee convergence across all replicas. It then is possible to rely on any reliable broadcast(or multicast) communication

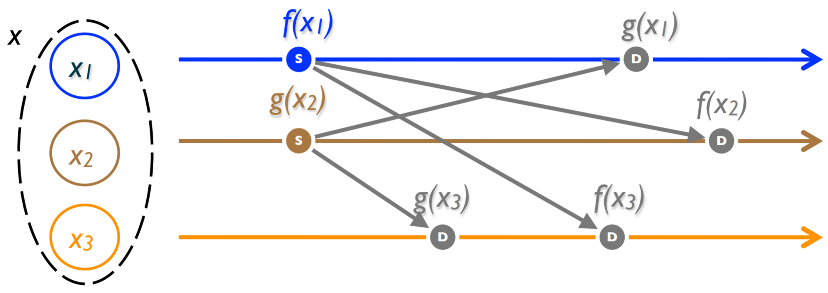


Figure 2.4: Operation-based replication. Taken from [51].

subsystem. Since this propagation is done through broadcasting from one to node to the entire system, there may be cases, such as concurrent updates, where operations reach in a different order to different replicas. Therefore, to converge to a correct state, operations need to be commutative.

By comparison with state-based CRDTs, operation-based CRDTs expose a greater efficiency, by only transmitting information about operations alongside their respective causal order, allowing larger states. However CmRDTs require operations to be transmitted in their causal order for replication, thus a causally ordered reliable broadcast channel is required, which requires additional guarantees from the communication channel. Currently, they are mainly used in cooperative systems such as Bayou and IceCube [47].

2.3.3 Pure Operation-based CRDTs

When necessary, op-based CRDTs in scenarios such as synchronization of replicas after failure, are obligated to propagate the full-state, and thus imposing propagation and storage overheads as well as making indistinguishable the distinction from state-based CRDTs [23].

In order to make a clear distinction from state-based CRDTs and diminish the impact of the imposed overhead, pure operation-based CRDTs that can only send unitary operations to other replicas, are introduced. Pure Op-based CRDTs, require effector functions(*effect-update* method) to be idempotent, in order to avoid delays originated by the *prepare-update* method building messages that duplicate the information already present in the middleware [59]. Therefore, to obtain idempotence, updates must propagate immediately at the cost of more complex operation representation and of having to store more metadata in the CRDT state. With pure op-based CRDTs, the *prepare-update* method is limited to returning the operation, not being able to inspect the state. Whilst the *effect-update* method delegates the entire logic of executing the operation in each replica, which is also made generic by not being data type dependent [46].

Additionally data types with commutative operations can simply be implemented as pure op-based CRDTs using standard reliable causal delivery, whilst data types with non-commutative operations are implemented using a partially ordered log of operations(*PO-Log*), making use of an extended API, such as Tagged Causal Stable Broadcast (TCSB), that provides extra causality information upon delivery and later informs when delivered messages become causally stable, allowing operations log compaction [23].

2.3.4 Delta-based CRDTs

By contrast with the previous synchronization models, if an update simply modifies a minor portion of the state, propagating the complete object's state to other replicas like in state-based synchronization is inefficient, since the replica already knows a major portion of said sate. On the other hand, if a large number of updates modify the same state, such as increment operations in a counter, propagating all update operations like in operation-based synchronization, imposes additional overhead over the communication channel, thus a more sensible approach would be to propagate the whole object's state once [46]. To solve the presented shortcomings, we present the delta-based CRDTs, that combine both state and operation synchronization models.

Small Delta CRDTs (δ-CRDTs)

Small delta-based CRDTs (δ-CRDTs) [19] make use of *delta-mutators*, that update the state according to the changes made since the last synchronization date. Each replica must contain a local communication buffer containing deltas, that have not yet been propagated to neighbor replicas. Taking into account the fact that deltas are not propagated right away when multiple changes are made to the same item, all deltas contained by the buffer are merged into a single delta entity before dissemination.

However, δ-CRDTs synchronization may only function in an environment where continuous and static synchronization patterns among replicas are employed. This constitutes a problem, since scenarios where clients have unreliable communication channels, exhibiting highly dynamic communication patterns such as mobile edge networks, it is rather hard to maintain a solution where replicas remain consistent with continuous and static synchronization patterns [38].

Big Delta CRDTs (Δ-CRDTs)

Because ensuring that replicas remain consistent in systems with unreliable communication channels is an arduous task, Big delta-based CRDTs (Δ-CRDTs) [38] have been introduced as an update over δ-CRDTs, in order to support highly dynamic environments for large scale systems.

Δ-CRDTs do not make use of pairwise communication buffers, reducing the space overhead upon each replica, instead the CRDT internal metadata is used to compute the minimal delta that needs to be propagated to another replica, based on a causal context exchanged between replicas, such as vector clocks. Hence, Δ-CRDTs are appropriate to be used in decentralized dissemination protocols, such as gossip protocols.

In contrast with δ-CRDTs, instead of having replicas periodically send deltas to other replicas, receiving nodes become responsible for disseminating *getDelta* request, in order to show the data's most recent version. When this request is sent, the origin node will send the version vector, which combines both the identifier of the replica and operation number, for a specific item being held and the destination node will then answer back with a fully merged delta if the received version vector is lower than what it holds. Otherwise, the delta will be empty and as a consequence, each replica is required to maintain a list of metadata describing the delta history.

Moreover, in order to save space, a *garbageCollection* function is periodically executed to delete old metadata associated with all operations that happened before a given point in time [38]. In the case where the origin node's version vector is much older than what the replica has, the framework will fall back to a state-based CRDT and send the full state.

2.3.5 CRDT applications

Table 2.1: CRDT data types and known implementations, based on [37].

Data Type	CRDT Specifications	Known Implementations
[52]Integer Vectors	Increment-only Integer Vector	
[51]Counters	G-Counter, PN-Counter	Riak [1], Eventuate [10]
[51]Registers	LWW-Register, MV-Register	Riak [1], Eventuate [10]
[51]Sets	LWW-Element-Set, PN-Set, OR-Set, U-Set	Riak [1], Eventuate [10]
[51]Maps	Dictionary, Map	Riak [1]
[51]Graphs	2P2P-Graph, Add-Remove Partial Order	

Table 2.2: CRDT use cases and known implementations, based on [37].

Use Case	Underlying CRDTs Structure	Known Implementations
Dynamic Vector Clocks	Increment-only integer vector [52]	
Collaborative writing	Add-Remove Partial Order [51]	Logoot [58], Treedoc [48], WOOT [42], LSEQ [41]
Logs	U-Set [52]	
Time-series Event Storage	LWW-Element-Set [51]	Roshi [11]
Shopping Carts	MV-Register [51]	Amazon [12]
Location Data	Combination of 2P-Set, LWW-element-Set [51]	NavCloud (TomTom) [13]
Betting Data	OR-Set [51]	bet365 [6]
Gameplay statistics	Sets of Counters [51]	League of Legends [14]

The previously displayed CRDTs can be used to implement several different basic data types (Table 2.1). Some data types are more tailored than others to certain types of applications. The applications have a varied range of use cases, varying from collaborative environments for text editing to web store shopping carts like the one in Amazon [12], and even betting applications such as Bet365 [6] (Table 2.2).

To select the most appropriate data types, the data model designer/developer is required to have an insight on the functionality provided by the data types, as well as taking into consideration their data model requirements to decide how to maintain the application state. Since CRDTs are data types designed to be modified concurrently, it seems rather obvious that the target will be applications in which data can be modified concurrently. Therefore, one of the most important aspects to consider is the manner how concurrent updates executed in different replicas are managed and subsequently propagated to all replicas, this is called the *concurrency semantics* [46].

Update operations defined in data types may or may not commute. For example, one of the most simple abstract data types is a counter, which supports increment and decrement operations, to increase and decrease by one unit its integer value. For a counter, the natural concurrency semantics is to have a final state that reflects the effects of all updates carried out, considering updates can be executed in any order yielding the same result(commutative). One way of reflecting the effects of all updates is to have the counter value be computed as the number of increments and subtracting the number of decrements, whilst verifying conditions, such as non-negativity that impose additional constraints or synchronization. Regrettably, for the majority of data types, having concurrency semantics as simple as the counter is not the case. For instance, when regarding a shared set object that supports add and remove update operations, there is no correct final state when concurrently adding and removing the same element. Thus, several concurrency semantics are acceptable, with different semantics being appropriate for different applications [51].

When defining concurrent semantics, it is helpful to understand the concept of *happens-before relation*, defining a relation that correlates the causality by saying that if an event a occurred before an event b in the same process or a is the event responsible for disseminating message m and subsequently b receiving the message or even if there is such an event c that correlates with both events a and b such that $a < c < b$, then *a happens-before b*. Another relation relevant for the definition of concurrency semantics such as the *last-writer-wins* semantics, is the *total order among updates*. By combining clock time along with a replica identifier, it is possible to create unique timestamps ensuring total order. By it itself, these timestamps do not respect the *happens-before relation*, but it can be achieved through the combination of physical and logical clocks [46].

Returning to the previous shared set object example, there are different approaches on how to tackle the fact that there is no correct final state when concurrent updates occur on the same element. The approaches differ mainly on how concurrent add and remove operations of the same element are managed, but ultimately it comes down to the application specification to mold the approach. For instance, a last-writer-wins (LWW) set incorporates the *total order among updates* relation by giving a timestamp for each value so that it can hold multiple add and remove operations, hence a value is in the set if the add operation supersedes the remove in the total order among updates. Another example is the grow only set (G-set), that circumvents the concurrent add and remove issue by only allowing add operations. There is also the 2P-set where addition and removal of elements are permitted, but addition becomes unavailable after a removal takes place. In order to function two sets are available, one for the values added and another set with the values removed (tombstone set). In the Observed-Removed Set (OR-set), an addition has precedence over removal, in contrast with the 2P-set.

There is a multitude of abstract data types supported by CRDTs and respective variants that integrate multiple concurrency semantics, but as one can notice they all serve their purpose and are perfectly tailored for their job, as Shapiro et al. have shown by implementing a web crawler that needed concurrent add operations to win over remove operations for the specific application [52]. A further discussion will be held at Section 2.3.6 on what requirements are needed and which CRDT brings more advantages to the table, in order to surpass the shortcomings arriving from dynamic environments.

2.3.6 Discussion

Table 2.3: CRDT implementations comparison.

	Message Size	**Propagation Model**	**Prior Knowledge**	**Late Entry**	**Churn Tolerance**
[51]CvRDT	Big	Push/Pull	Dependent	Converges, sends full state	Converges
[51]CmRDT	Small	Push/Pull	Dependent	Diverges, requires operations in causal order	Diverges
[20]δ-CRDT	Medium	Push	Dependent	Converges, reverts to state synchronization	Diverges
[38]Δ-CRDT	Medium	Push/Pull	Dependent	Converges, reverts to state synchronization	Converges
P/S-CRDT (Proposed)	Medium & Small	Push	Not Dependent	Converges	Converges

There are many characteristics to take into account when picking the right CRDT, one must contemplate the needs required by the system to maintain the state (Table 2.3). Such characteristics may come as a consequence of the synchronization model employed by the CRDT itself, for example, CvRDTs have a big message size since it is required the exchange of full local state among replicas, which may cause a considerable communication overhead. On the other hand, CmRDTs have a small message size since only the update operation metadata needs to be propagated, thus exposing a greater bandwidth usage efficiency.

Putting into context, when examining the necessities of a mobile edge network, two challenges immediately come to mind, the entrance of new nodes and the re-entry of nodes (churn rate). Additionally, it is in our interest to avoid the requirement of prior knowledge of all nodes currently on the system, which is hard to accomplish for mobile edge networks. These challenges come as a consequence of highly dynamic environments, where temporary unavailability is to be expected. Therefore, it is in our interest to discuss and underline which CRDTs best satisfy the needs of said environment.

At first glance, assuming a scenario where replicas may join at any given time, CvRDTs and both delta-based CRDTs (δ-CRDTs/Δ-CRDTs) provide the best approach. These CRDTs allow replicas to synchronize with other replicas by receiving their full state, thus combining multiple operations into a single replication step. On the contrary, CmRDTs allow replicas to converge by propagating updates to every other replica, which constitutes a problem since replaying all operations from the beginning in their causal order will cause considerable overhead.

When, tackling the second challenge, where nodes may re-enter the network, both δ-CRDTs and Δ-CRDTs present the best approach. If only a small part of the state was modified while the node was out, then a delta-mutator is propagated, with the changes made since the last communication. On the other hand, if a large portion of the state was modified while the node was out, then delta-based CRDTs send the full state since it is more efficient than propagating all update operations. Additionally, Δ-CRDTs comes as an improvement over δ-CRDTs, by helping with the minimization of the state that needs to be transferred. Although operation and state-based CRDTs are capable to deal with node re-entry, they do so in an inefficient manner, by not differentiating whether the state has changed too much or just a small portion. Hence when a small modification is made, operation based is more efficient than state-based, by either using the node's local state or copying the state from another node and replaying the missing operations, which is far better than sending the full state when in turn only sending the last couple of operations is sufficient. Moreover, when the state modification is so significant that it makes more adequate sending the full state, then the state-based CRDTs become more profitable in terms of bandwidth usage than operation based CRDTs [46].

Furthermore, some CRDTs require extra precautions, such as operation-based CRDTs that require compaction mechanisms to minimize identifier size and remove storage overhead, also CmRDTs rely on reliable communication subsystems to guarantee exactly-once and causal delivery. Since we aim to introduce CRDTs to cooperate with a topic-based publish/subscribe system for mobile edge networks and subsequently develop a collaborative simulation, it is in our interest to employ a push propagation model with one-to-many communication, enabling us to formulate an asynchronous and scalable solution, where clients are decoupled from servers in time. There are various available CRDT libraries, for instance Akka [15] offers an implementation of delta-CRDTs and Riak [1] provides a client library with multiple data types, these and other open-source libraries will require further research, so that it may so that we may extend various features.

2.4 Eventually Consistent Mobile Systems

2.4.1 Bayou

Bayou [43]: is an example of a replicated, weakly consistent storage system designed for mobile computing environments. Bayou provides support for collaborative applications, by managing conflicts generated by concurrent activity whilst relying on the weak connectivity provided by mobile network communications, where frequent disconnection and high latency is expected. Such situations may lead to cases where participants of the collaborative system may never be all connected simultaneously, thus Bayou's design requires only occasional, pair-wise communication between participants [56]. This behavior is the result of Bayou's partition tolerance mechanism, where a disconnected workgroup may be away from the rest of the system and yet capable of remaining connected with the rest of the system through pair-wise communication.

To maximize availability, Bayou employs a model where clients can read and write any accessible replica without the need for explicit coordination with other replicas. Furthermore, it is made certain that every replica eventually collects updates from every other, through a chain of direct or indirect pair-wise interactions.

In collaborative systems, conflicts among client update operations are bound to happen, to that end Bayou has focused on supporting application-specific mechanisms to detect and resolve the conflicts, ensuring that replicas move towards eventual consistency. Additionally, Bayou includes two mechanisms to achieve automatic conflict resolution, the dependency checks and merge procedures, in which clients indicate for each write operation, how the system should behave in order to settle detected conflict. To this end, Bayou has to schedule operations in causal order, thus a tentative timestamp is assigned to each operation as it is accepted by a primary replica. Subsequently, these operations are executed following their tentative order, but before an operation can be executed, the dependency check mechanism certifies whether the operation is valid or not. If valid, then the execution is allowed, otherwise a conflict is detected and the merge procedure provided by the application comes into action.

One of the major takeaways from Bayou is that to guarantee eventual consistency, excluding cases where all operations are commutative, servers must be capable to rollback the modifications caused by previously executed writes and redo them according to a global order. Furthermore, to arrange operation in their global order, Bayou relies on primary replicas, which may create a congestion point.

2.4.2 IceCube

IceCube [45]: is another weakly consistent system for optimistic replication, that supports mobile computing and collaborative work, by letting users write to shared data with no synchronization. That being said, in mobile environments, users expect to interact with shared data through disconnection periods, which may cause replicas to diverge. Therefore, a reconciliation mechanism guaranteeing convergence is compulsory. Hence, IceCube proposes a general-purpose reconciliation engine that is parameterized by application semantics, instead of using syntactic mechanisms such as timestamps and dropping actions to avoid conflicts, which are used by systems like Bayou [43].

The system stores logs with the preformed tentative updates and combines those concurrent logs into sequential executions, called schedules. These schedules, in turn must obey to application semantics, which are expressed through constraints, representing dependencies and invariants. IceCube uses these constraints, to apply correctness checks on schedules, which are the means to detect conflicts. Constraints can either be static, creating an unconditional relation between two operations, or they can be dynamic, representing the success or failure of a single action, depending on the current state [45].

As dynamic constraints violations can lead to costly roll-backs, IceCube devises a technique of dividing the work into joint sub-problems, where actions belonging to any sub-problem commute with actions in all other sub-problems, resulting in more efficient approach by limiting roll-backs to a small number of actions.

To satisfy these constraints and thus guarantee conflict resolution, a heuristic search is made by the scheduler to select the best action from a set of candidates, given a merit unit, measured by the number of other actions that can be scheduled after the selected action. However, if the execution of the action leads to a dynamic constraint violation, the scheduler dismantles its execution through roll-back, removing any action that conflicts statically with the selected action.

IceCube produces a more powerful approach than Bayou, by reconciling in cases where Bayou's merge procedures would find a conflict. Additionally, IceCube provides a bad solution to peer-to-peer networks, since concurrent operations are sent to a single replica for merging and reaming replicas waiting for the merged log, must be frozen until reconciliation is completed.

2.4.3 Telex

Telex [25]: is a system that provides solutions for sharing mutable data in collaborative environments, where disconnected work is tolerable. Since various existing approaches suffer limitations, such as machine replication, which enforces high latency and does not provide support for disconnected operations, Telex opts to base itself on an approach that separates application logic from system logic, combining flexibility and correctness.

Furthermore, application logic is required to disseminate actions and concurrency invariants to Telex, which in turn takes care of replication, consistency and access control by detecting conflicts and computing conflict-free schedules. In order to eliminate contention and promote locality, Telex designed and efficient multilog data structure to store the Action-Constraint Graph (ACG). The ACG is the concept used by Telex to detect and correct conflicts, by summarizing concurrency semantics of applications. Thus conflict resolution is application independent, enabling users to operate on their local replica of shared documents, whilst able to work under disconnection circumstances and suffer no network latency.

Contrarily to the previously present works, Telex doesn't rely on a primary site for commitment, additionally it also takes advantage of commutativity when it is available and supports any mix of commutative and non-commutative operations. Unfortunately, Telex suffers from excessive memory consumption, since the ACG is accessed concurrently by many threads, and quickly reaches sizes of several tens of thousands of nodes.

2.4.4 Rover

Rover [36]: provides a toolkit for mobile application designers to optimize the utilization of network bandwidth, and allow for dynamic division of work between client and server. The Rover set of tools, supplies mobile communications based on two ideas, relocatable dynamic objects (RDOs) and queued remote procedure call (QRPC). In an effort to reduce client-server communication requirements, Rover presents relocatable dynamic objects, which are objects with a well-defined interface that can be dynamically loaded into a client from a server. In addition, queued remote procedure calls feature a communication system that concedes applications to proceed making non-blocking remote procedure calls even when a host is disconnected, having requests and responses exchanged upon network reconnection.

Applications that seek to support Rover's toolkit, must employ a check-in/check-out model of data sharing, were applications import RDOs into their address spaces, invoking methods provided by the RDOs, and export the RDOs back to servers. The authors exemplify RDOs as taking roles, such as simple calendar items with associated operations or even as complex modules that encapsulate part of an application, for instance the user interface of a calendar tool.

Moreover, Rover allows hosts to update shared objects under disconnection circumstances, guaranteeing object consistency by means of application-level locking or through application-specific algorithms to resolve uncoordinated updates to a single object. As such, objects are required to have a home server, since mobile hosts import objects into their local cache and then export updated objects back to their respective home servers, with the intent of detecting conflicts and reconciling them at the server, allowing automatic conflict resolution by employing type-specific concurrency control.

Rover has shown a good set of tools for deployment of applications that are less dependent on high-performance communication connectivity. However, Rover displays a drawback by demanding application designers to create relocatable dynamic objects whilst making sure they are loaded into the client's cache, and if the design must incorporate client updates under disconnection, then consistency constraints must be implemented.

2.4.5 SwiftCloud

SwiftCloud [44]: presents an adequate strategy to integrate client and server-side storage. To solve the issue of ad-hoc implementations that inadequately integrate with server-side storage, leading to the degradation of data consistency guarantees, SwiftCloud offers the first system to deliver geo-replication to the client machine, introducing a principled approach to access data replicas at client machines and cloud servers.

The system aims to tackle two big challenges, the first one being how to assure programming guarantees for applications running on client machines at an inexpensive price and the second one being, how to maintain these guarantees in scenarios of client disconnection. To surpass these challenges SwiftCloud supports server-side execution, whilst assuring the programming guarantees of client-side in-cache execution. In addition, a client-assisted failover protocol to preserve causality is employed by SwiftCloud, ensuring that clients always see a casually consistent sequence of updates from other users. By doing so, a client always observes its previous updates and is able to reconnect to other data centers, by replaying its own updates and the observed stable updates from other users, that are already present in other data centers.

At its core, SwiftCloud consists of a set of data centers that replicate every object (CRDTs), whilst in the periphery, client nodes access the system by communicating with a local module (*scout*). The scout module caches a subset of the replicated objects, and assuming that appropriate objects are in cache, a client node supports disconnected operation and obeserves an overall responsiveness improvement. Furthermore, SwiftCloud provides a transactional key-object API, where applications execute interactions by triggering read and update operations.

To address the issue where different clients observe the same set of concurrent updates applied in different orders, the system prohibits non-commutative concurrent updates. Thus, two types of transactions co-exist, the mergeable and non-mergeable transactions. Mergeable transaction commute with each other and with non-mergeable transactions, allowing these mergeable transactions to execute on client-side cache. In addition, mergeable transactions are read-only transactions, or update transactions that apply modifications to CRDTs. On the other hand, non-mergeable transactions provide the traditional strongly consistent transaction model where non-commuting concurrent updates conflict, thus these transactions are exclusively executed in data centers.

Henceforth, SwiftCloud has presented a reliable approach for bringing geo-replication to the client machine by defining a principled strategy for using client and data center replicas. Making applications run transactions in the client-side for common operations affecting a limited number of objects (mergeable transactions) and data center execution for transactions requiring strong consistency or accessing a large number of objects (non-mergeable transactions). Moreover, according to SwiftCloud's evaluation, the employed client-assisted failover mechanism made possible a trade between a small increase of stale read operations for better latency values.

2.4.6 Discussion

In order to provide eventual consistency to system users and resolve conflicts caused by concurrent operations being made by said users within a collaborative environment, many systems resort to outdated and not well suited approaches. Such techniques as rollback mechanisms, requiring the system to keep a long log of operations and the ability to analyze the commutative properties of conflicting operations in order to create a deterministic order for operations to be applied (Bayou).

Furthermore, many systems use specific replicas for conflict resolution, thus creating a congestion point. Also, very often when resolving conflicts, conflicting replicas must be frozen before they may continue their work, such as the case of IceCube. Lastly, one other common issue created by conflict resolution mechanism, is often an excessive memory consumption, due to storage.

Since one of the main subjects of this dissertation is mobile environments, when resolving conflicts, a system must do so in low latency and scalable approach, giving care to the disseminated data quantity and size. Thus, when comparing with the conflict resolution provided by CRDTs, it becomes obvious that CRDTs is the better approach, due to the versatility and lower latency values, hence making a good solution to guarantee eventual consistency to users in a mobile network.

However, many systems employing CRDTs, still require the knowledge of the replicas composing the network, as means of knowing the end targets for update dissemination. Hence by lifting the need to know which replica to send an update, we aim to further adapt to mobile environments, where maintaining this knowledge is inconceivable and not salable due to churn.

2.5 Publish/Subscribe Scheme

Over the years distributed systems have become increasingly complex and larger in scale, such that these systems now have to deal with thousands of entities distributed all over the world and still take into consideration that these entities location and behavior may change throughout the life span of said systems. As a result, many forms of interaction for such large scale systems arose. One of these forms stands out, the publish/subscribe interaction scheme [32], due to the capability of providing a loosely coupled form of interaction.

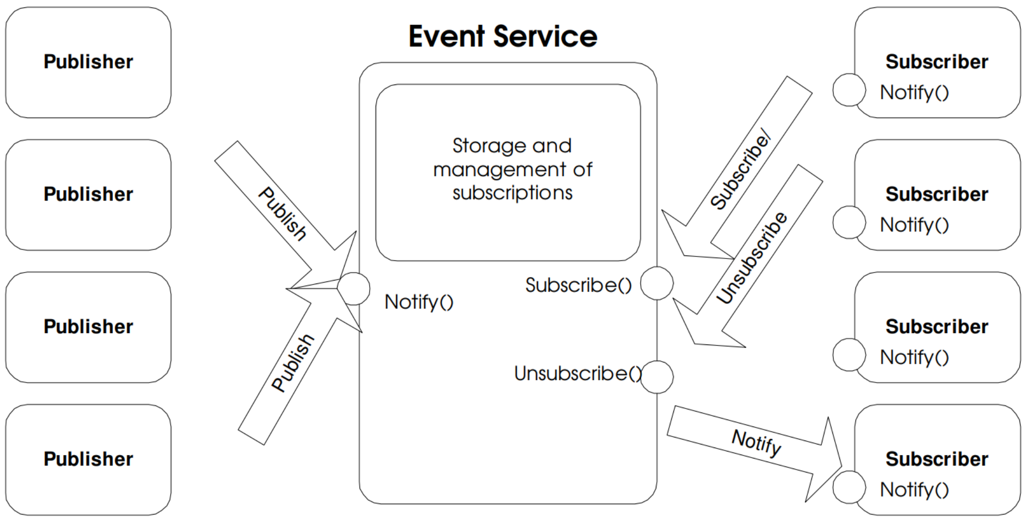


Figure 2.5: Basic publish/subscribe system interaction. Taken from [32].

For a better understating of the publish/subscribe paradigm, its best we differentiate the unit of work and user roles that interact in systems employing this scheme. As the basic units of the interaction, we have *events*, these units generally represent an area of interest. Such areas are commonly specified as *topics* in *topic-based* publish/subscribe systems, and frequently represented as *keywords*, for instance a football game might be characterized e.g."Benfica-Porto" and many events can be coo-related with this topic, e.g.goals or fouls.

As for the types of users intervening in the interaction, there are the ones denominated as *consumers* or *subscribers* that have the capability of conveying their interest in one or multiple *events* and the ones denominated as *producers* or *publishers* that generate content over certain events. Lastly, the system must contain an event notification service that takes the role of middle man between *publishers* and *subscribers*, providing subscription management, event distribution and storage.

Therefore, if a subscriber wants to receive content related to an event, he must then subscribe to the area of interest associated with the event by triggering the *"subscribe()"* action specified by the system, thus when a publisher generates content related to events in the system it must then trigger the corresponding *"publish()"* action specified by the system. Ultimately, it's the system's event notification service responsibility to *"notify()"* the subscribers whose interest match each individual publication, as seen in Figure 2.5.

Since publishers produce events by handing them over to the event notification service and subscribers consume the events indirectly through the service, as long as they are subscribed to the corresponding topic, it then is possible to observe the decoupling of these two distinct roles, considering that publishers produce independently of subscribers and that subscribers consume produced events independently of the publishers.

This loosely coupled form of interaction, makes the publish/subscribe scheme a great approach when dealing with complex and large scale distributed systems, such as mobile device networks, which is the focus of the study conducted throughout this thesis. Said mobile device networks require high scalability, flexibility and extensibility, thus and so mobile networks may hugely benefit from the characteristics provided by the publish/-subscribe scheme, which decouples three dimensions, *space*, *time* and *synchronization* [32]:

- ***Space decoupling***: Interaction between publishers and subscribers is made independently of each other, hence publishers do not maintain a reference to the subscribers, nor have the knowledge of how many subscribers are taking part in the activity. Correspondingly, subscribers do not maintain any reference to the publishers nor know how many publishers are taking part in the activity.

- ***Time decoupling***: Publishers and subscribers are not enforced to be actively participating in the interaction at the same time. For instance, when a subscriber is

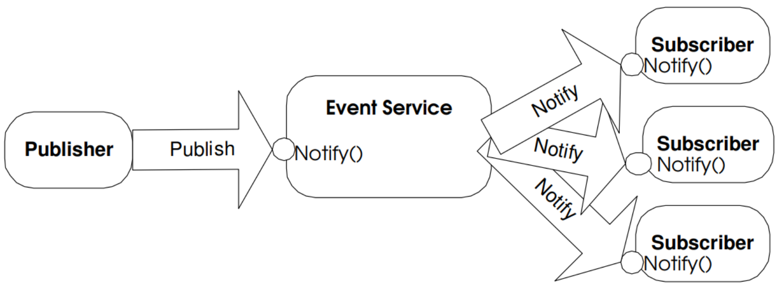


Figure 2.6: Space decoupling. Taken from [32].

disconnected, publications can still be made by publishers on events previously subscribed by the subscriber, and interchangeably, when a publisher is disconnected, the subscribers are still able to get notified of produced events by the event service.

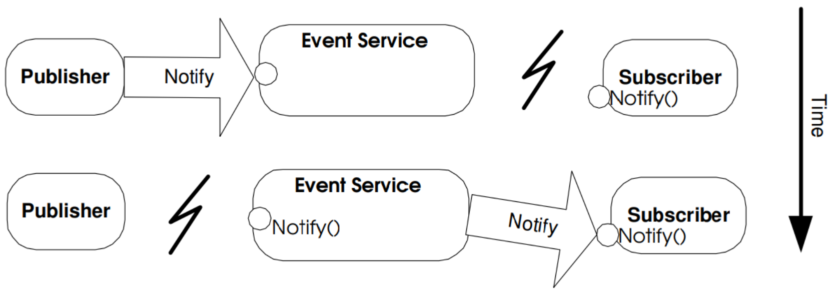


Figure 2.7: Time decoupling. Taken from [32].

- ***Synchronization decoupling***: Publishers production and subscribers consumption happen in separate control flows, thus the interaction happen in an asynchronous manner. Therefore, publishers may never be blocked from producing events and subscribers blocked from being notified of new events whilst fulfilling concurrent actions.

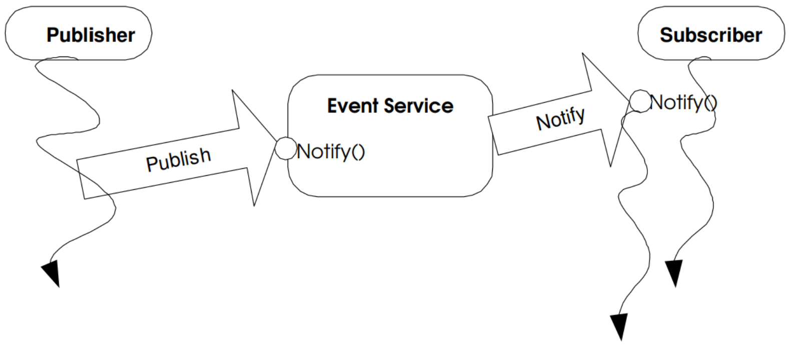


Figure 2.8: Synchronization decoupling. Taken from [32].

Considering, the loosely coupled form of interaction granted by the publish/subscribe scheme, it then is possible to give support for mobile environment data dissemination, where the mobility of nodes is a constant to be taken into consideration, leading to scenarios where nodes may leave involuntarily.

Fortunately, since the publish/subscribe interaction, describes two distinct roles with two different control flows, it allows for node disconnection, considering publishers may produce without the need of subscribers to be active and conversely, subscribers may consume without the need for publishers to be active at consumption. Thus, the publish/subscribe interaction scheme reveals an appropriate approach seeing that by nature the scheme already grants decoupling properties that support dynamic environments behaviors.

2.5.1 Discussion

As one of the objectives of this dissertation is to define an efficient update dissemination pattern, allowing users to propagate updates whilst not depending on the knowledge of whom to send the updates to, it was compulsory that we found a suitable interaction scheme.

The publish/subscribe scheme, providing a loosely coupled form of interaction, by decoupling time, space and synchronization, allows for multiple interesting and beneficial characteristics. Such characteristics as enabling disconnected work since publishers and subscribers workflow are done separately. Besides, it provides us with the characteristic of being able to produce and share updates without depending on the knowledge of whom to send them to, since in the publish/subscribe scheme, publishers and subscribers do not hold any reference from each other, as they may produce and consume whenever they like. Thus, the publish/subscribe scheme reveals itself as the chosen medium for update dissemination for collaborative environments using CRDTs.

Chapter 3

P/S CRDTs Model

In this chapter we introduce, the Publish/Subscribe CRDTs (P/S-CRDTs) model, which is the fundamental basis of the proposed solution hereby presented. The P/S-CRDTs model aims to define the update dissemination pattern for shared CRDT objects in a collaborative setting, using the publish/subscribe interaction model as the medium for sharing updates. By employing the proposed model, we intend on studying and demonstrating a suitable approach to circumnavigate some of the problems that originate from dynamic environments such as mobile networks. In these networks, mobile device users may struggle with regular periods of disconnection and overall poor connectivity due to the weak nature of mobile network communications leading to a multitude of issues, chief among these are state divergences. Such state divergences may induce mobile users to experience data inconsistencies, where some users may see a version of the state and others observe another state, making impossible collaborative activities such as cooperative text editing to function properly.

Within this chapter, we commence by giving a technical overview of the proposed model, its goals and a broad explanation of its elements in Section 3.1, followed by a description of the design of CRDTs and their updates to be disseminated by the publish/subscribe system in Section 3.2. To finalize the chapter in Section 3.3, we specify and describe update dissemination by clarifying how nodes should behave to propagate modifications, by taking a look at the publisher, subscriber, and broker roles. In addition, we formulate state and operation based objects, using the P/S-CRDTs model in Section 3.3.3, superseding with a succinct discussion in Section 3.4.

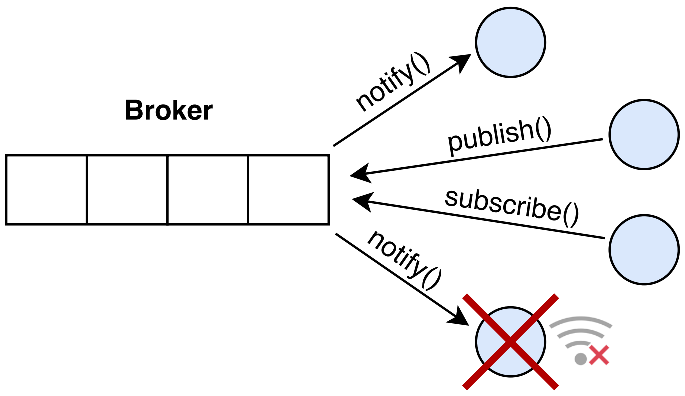


Figure 3.1: Node communication and disconnection.

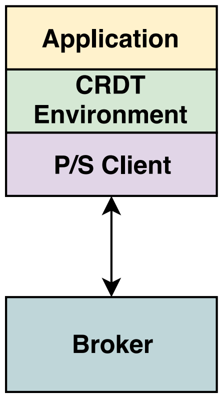


Figure 3.2: Base node interaction.

3.1 Overview

The proposed solution, the P/S-CRDTs model, defines a model in which the generic CRDT object design is adapted to dynamic environments, by coupling it with the publish/subscribe interaction scheme, as the medium for data dissemination. P/S-CRDTs are designed to be able to guarantee the eventual consistency model while dealing with volatile scenarios resulting from the poor properties of mobile networks, where users of collaborative applications in such networks may enter, leave voluntarily and involuntarily by disconnection and give re-entry into the activity at any given time (Figure 3.1).

Given the possible erratic behavior of mobile networks, it is inconceivable to maintain the active knowledge of the participants in the activity, therefore the P/S-CRDTs intend to decouple this need by making it not necessary to recognize the nodes present in the network for the propagation of updates between nodes. This is achieved by using the publish/subscribe interaction model, which delivers a loosely coupled form of interaction where the publishers may produce updates independently of subscribers and subscribers may consume updates independently of publishers, allowing for node disconnections since the publishers and subscribers interactions happen in separate control flows.

P/S-CRDTs model has been designed to specify the manner of how object updates must be disseminated, stored and retrieved given the publish/subscribe interaction model. Additionally, the following model has been conceived to be adaptable to the common topic-based publish/subscribe architecture, hence allowing developers to accommodate P/S-CRDTs to existing publish/subscribe systems with minimal implementation given the desired data type's characteristics (synchronization model and concurrency semantic).

Thus, in order to handle the generic CRDT design properties and respective synchronization models, some additional requirements and mechanisms were designed to certify that the shared state amongst nodes in the network is eventually consistent, enabling mobile users to interact in collaborative applications without experiencing state divergences. Furthermore, it should be noted that the design of P/S-CRDTs is highly focused on reducing the cost of communication, considering that the environments in focus are mobile networks, which require special attention to the size of the data disseminated throughout the network to avoid any memory overheads.

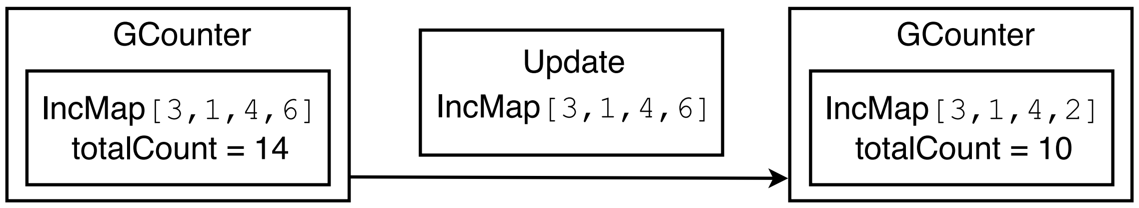
The model's intended usage encapsulates scenarios where mobile users take part in a collaborative activity, armed with CRDT objects representing the shared state used by the activity and a publish/subscribe system client for the dissemination of updates (Figure 3.2). For a better picture, one can conceive the use case of a collaborative text editing application, where a CRDT represents the shared text to be modified by multiple users and subsequently, all realized modifications must then be disseminated to other users through the medium of publish/subscribe interactions adapted with the P/S-CRDTs model specification. Hence, the model depicts nodes as taking the roles of publishers and subscribers, since they both are capable to produce updates and consume other nodes updates by communicating with the broker, who subsequently notifies matching updates to the ones subscribed by the respective nodes. Given the model specification on the dissemination of updates and respective requirements, it then must be possible for mobile users to interact and maintain a consistent sate during dynamic scenarios.

3.2 CRDTs and Updates

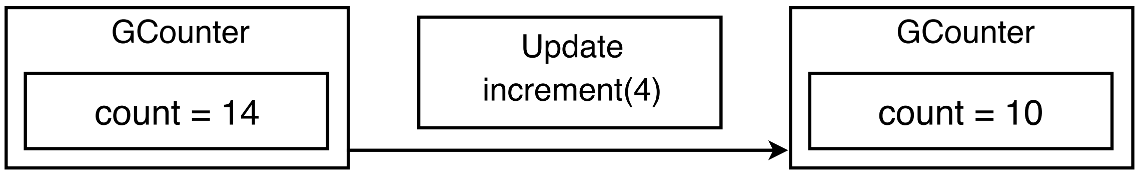
The CRDT object, within a collaborative environment, acts as the shared data between the users that compose the network. CRDTs can represent a multitude of data types e.g.counters, sets, registers and employ various synchronization models such as state and operation based synchronization. Thus, given the variety of configurations, modifications made to shared objects may be shared in different manners. For instance, the commonly found state-based object in literature requires an update to be the full global state of the data type, carrying all local and external modifications seen (causal context). On the other hand, operation-based objects commonly depict updates as unitary local operations made at source, to then be delivered and applied to target nodes (Figure 3.3).

At heart, CRDT objects contain a **state**, representing the structure where users will realize **updates** over, such as increments over an integer defining a counter or an element insertion for a set of elements. Such updates are firstly carried locally **at source** and in certain cases a precondition may be applied to the update operation to capture safety, for instance an element may be removed from a set if it is verified that the element is contained in the set at the source. Reciprocally, in some scenarios the precondition may be omitted, such as a *Grows-only* counter that only allows increments, thus not requiring any verification before the update is conducted locally. Afterwards, in a second phase described as **downstream**, updates are to be disseminated to target nodes whose CRDT object they share in the collaborative network.

Considering that after locally applying an update, the state suffers a mutation, making it non-convergent in relation to other nodes who have not yet received the updates, a convergence operation is required. To that effect, shared objects posses a **merge** operation, with the goal of guaranteeing that as soon as the updates from other nodes are received,



(a) State-based



(b) Operation-based

Figure 3.3: Update objects for state and operation based GCounter CRDTs shared between two nodes.

the convergence function takes place, adding the updates to the current local state of the CRDT. Through the process of convergence, the concurrency semantic is applied, solving any conflicts between any updates in a deterministic manner, thus guaranteeing a consistent state across all nodes, once all updates are merged by each individual replica (eventual consistency).

Hence, due to the multiple existing definitions of what constitutes an update, P/S-CRDTs characterize updates impartially of their content, only defining in which manner updates are disseminated. Thus, an update is simply identified as the unit to be propagated and merged with the global state. Moreover, through the use of the generic CRDT specification containing the mechanism previously described, it then is possible with little implementation to apply any given CRDT object to the P/S-CRDTs model, independently of the selected data type and respective concurrency semantic assigned.

3.3 Data Dissemination

Taking in mind mobile users in a collaborative environment, user modifications are to be disseminated as update objects and retrieved by intervenients in the activity, so that that they may converge by merging updates into their state. Once nodes participating in the collaborative activity acquire the shared CRDT object necessary to interact, it then is possible for nodes to start modifying the state of the CRDT and subsequently share respective modifications through the dissemination of updates. In order to propagate updates through all participants in the interaction, P/S-CRDTs use the publish/subscribe interaction model.

Hence, to accomplish the dissemination of updates through a publish/subscribe system, some additional requirements are compulsory to guarantee that data is correctly delivered and stored, thus enabling the nodes to merge with the shared updates. Thus, to better understand all requirements and mechanisms, it is best to specify the behavior of each system role (publisher, subscriber, and broker) when dealing with the dissemination of CRDT updates.

3.3.1 Roles

Given the various publish/subscribe interaction components, the publisher, subscriber and broker each contain certain individual responsibilities when producing, delivering and retrieving data within the activity.

Thus when designing the P/S-CRDTs model, special attention was given to the behavior of every piece of the communication, guaranteeing that the data size of updates disseminated in the network only carry the precise information for CRDTs to converge state and giving assurance that updates are sent and delivered by the nodes.

3.3.1.1 Publisher

Taking into consideration the collaborative environment, the main objective of the publisher agent is the production of content and consequent delivery to the broker agent. The content in this collaborative environment, is referred to as update objects that must be delivered to all nodes interested in receiving modifications made for the shared CRDT.

Logically, before nodes are able to interact by publishing updates, first the CRDT must be made available at the beginning of the collaborative activity, to all nodes that compose (or will later join) the collaborative network, either by using the publish/subscribe system or any other available method that enables nodes to acquire the shared CRDT. The CRDT object carries needed information about the data type and the usual interface with a query method to access the value, update methods related to the datatype, such as *increment* for a counter or *insert* for sets and finally the convergence method employing the chosen concurrency semantic.

Thus, once nodes have acquired the shared object, making them ready to share updates, the interaction is done through the following generic topic-based publication method:

`publish(dataItem, topic)`

This method enables the publication of objects in the system.

`dataItem` – Object to be published in the system.

`topic` – Expression representing the object to be published.

When publishing updates, each object must be attached to a *topic* expression. Such *topic* when publishing an update must be uniquely descriptive of the CRDT, making sure that for each shared CRDT object in a collaborative network, there is exclusively one *topic* associated. By keeping a unique update *topic* per shared object, it then is possible for

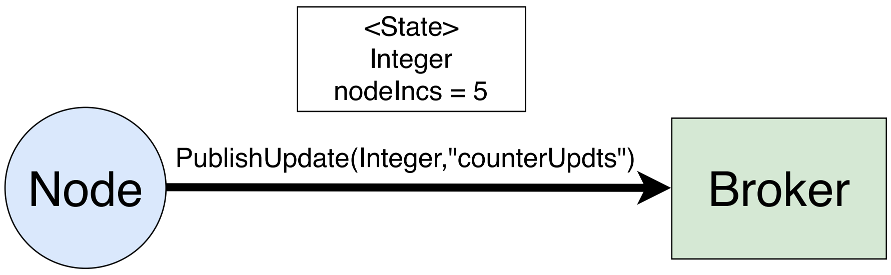


Figure 3.4: Update publication for a State-based Gcounter CRDT.

an application to simultaneously employ multiple CRTDs and ensure that each object receives their respective updates under the specific *topic*. For instance, an application that makes use of both an ORSet and a GCounter CRDT may specify that in order for users to publish updates for the ORSet object a publication must be made with the *topic* ”task_list” and updates directed to the GCounter be published under the ”task_quantity” *topic*.

When publishing content in mobile networks, one must be cautious not to send unnecessary or otherwise irrelevant data within updates. Thus, it is essential to make an emphasis on the data size of the items published in the network, with the intent of reducing any unnecessary memory overheads.

Given the caution over data size, P/S-CRDTs initially shared must be designed to carry a global state, representing the node’s global shared state, that must be convergent with all other nodes in the collaborative network. In addition, a local state variable must be attached to P/S-CRDTs, which only contains the modifications made at the source (Figure 3.4).

This local state object is to be used by state and operation based objects alike. Whilst state-based objects only carry in the local state object modifications made locally by the node, operation-based object instead carry the *effector* function and corresponding arguments. This local state object is to be later shipped to other nodes through a publication operation, hence the local state object effectively representing the update. By shipping only local modifications, P/S-CRDTs aim to carry only the precise information for nodes to converge. For instance, considering a state-based *Grows-only* counter CRDT, when publishing an update, only an integer representing the local increments made by the publishing node is contained in the update item, instead of the whole CRDT which includes more information than what required to merge.

It should be noted that the action of publishing updates, can be tailored by the likes of the developer. Either by manually publishing immediately after a modification or trough an automated task, such that nodes do not directly trigger it, but instead be a part of a scheduling task that checks for modifications done since last update publication step.

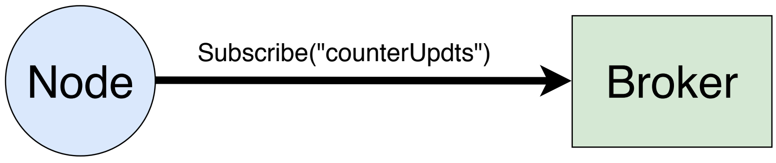


Figure 3.5: CRDT and update subscription.

3.3.1.2 Subscriber

In a contrariwise manner to the publisher agent, a node who desires to receive update content in a collaborative activity must subscribe to updates, by using the predetermined *topic* for CRDT updates (Fig. 3.5). Once a subscription is made under the update *topic* by a node in the activity, the node is then ready to receive other nodes updates via broker notifications. To be notified of new updates, nodes must trigger the following generic topic-based subscription with the update *topic*:

`subscribe(topic)`
This method enables the publication of objects in the system.

`topic` – Expression representing the object to be subscribed and afterwards notified by the broker.

Update subscription may be done whenever a node joins the collaborative activity, subsequently receiving all future updates made under the specified *topic*. In a collaborative activity, nodes may begin the interaction at a later time. Thus, to join later a node must be able to obtain content previously produced to maintain a consistent state in relation to the nodes already present in the activity (Figure 3.6). Hence, bringing a requirement of assured data retrieval for the subscriber role in the system, enabling users who join at later time to converge state.

Requirement - Assured data retrieval. *A collaborative network node must be able to retrieve enough information to keep a consistent state, either by retrieving missing updates from the broker or doing a full copy of another node's state.*

Furthermore, nodes in the context of mobile networks may experience disconnection, which implies that the node no longer is apart of the activity, thus losing update publications whilst out. Since P/S-CRDTs updates carry the local state object, only containing local modifications, in the case of state-based objects it then is required to receive at least the last local state published from each node in the activity. On the other hand, operation-based objects require not only the last published update from each node but all recent previously made updates or in extreme cases a full copy of another nodes state. To ensure once again that users keep a consistent state once they rejoin the activity, it is compulsory that when one or more notifications are lost, users are able to determine in cases of need if a past notification was lost.

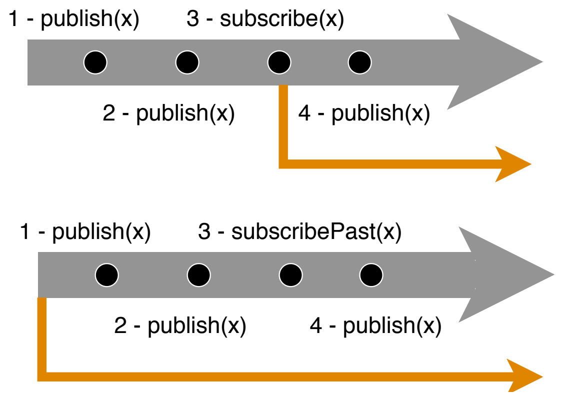


Figure 3.6: Subscription encompassing past publications.

To be able to retrieve lost notifications, each notification must then include some form of unique identification in the metadata, thus enabling subscribers to request the lost notification. This retrieval operation may be achieved by several mechanisms, for instance one may employ the blockchain logic, by forcing notifications to hold the hash of the previous notification, the subscriber will then be capable of knowing whether a notification was lost or not, and subsequently request the lost notification given the hash of the previously lost notification received in the latest notification.

Only when assuring both the previous requirements, the interaction guarantees consistency to nodes in all scenarios commonly found in dynamic environments, where users may disconnect and reconnect (*churn*).

3.3.1.3 Broker

As the middle man between publishers and subscribers, we have the broker. With the function of receiving messages from publisher s informing content produced over a topic and delivering notifications to the subscribers matching the content produced. From the broker's viewpoint (Fig. 3.7), throughout the lifespan of the collaborative application activity, multiple update publications and subscriptions messages will be received for the shared objects defined by the application. Subsequently, the broker must retain metadata associated with all publication and subscription requests, with the intent of notifying all nodes that have shown interested in receiving updates trough a notification of update arrival.

To accommodate the needs of dynamic environments, where previous published content may be accessed by nodes who disconnect and require a specific lost notification or nodes who join the activity later and require all content previously published, it then is required for the broker to provide publication persistency to maintain recently published content.

Requirement - Publication Persistency. *Let m be a publication message, if m was received by the broker and is contained in the broker's storage at request time, then a subscriber must be*

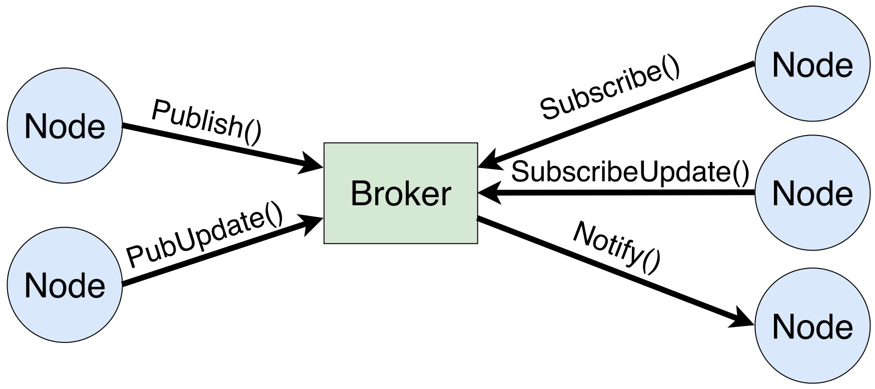


Figure 3.7: Broker interactions with system nodes.

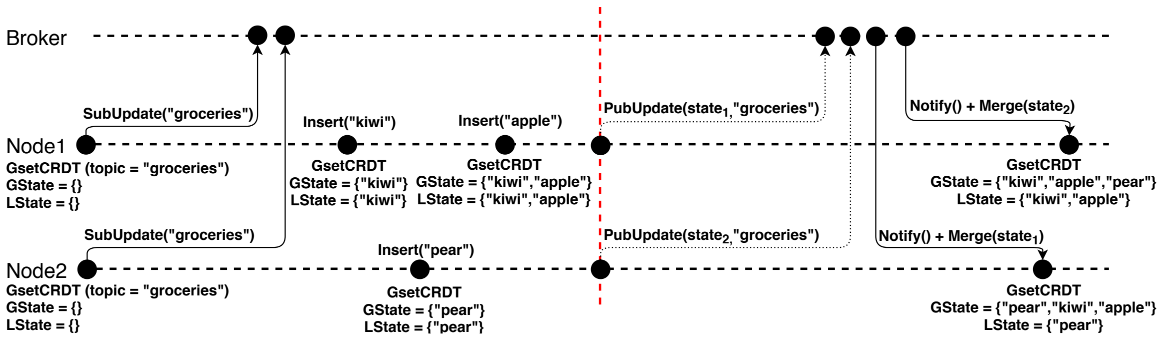

Broker
SubUpdate("groceries")
Insert("kiwi")
Insert("apple")
PubUpdate(state$_1$,"groceries")
Notify() + Merge(state$_2$)
Node1
GsetCRDT (topic = "groceries")
GState = {}
LState = {}
GsetCRDT
GState = {"kiwi"}
LState = {"kiwi"}
GsetCRDT
GState = {"kiwi","apple"}
LState = {"kiwi","apple"}
GsetCRDT
GState = {"kiwi","apple","pear"}
LState = {"kiwi","apple"}
SubUpdate("groceries")
Insert("pear")
PubUpdate(state$_2$,"groceries")
Notify() + Merge(state$_1$)
Node2
GsetCRDT (topic = "groceries")
GState = {}
LState = {}
GsetCRDT
GState = {"pear"}
LState = {"pear"}
GsetCRDT
GState = {"pear","kiwi","apple"}
LState = {"pear"}

Figure 3.8: Delivery and retrieval of updates for a Grows-only set shared between two nodes.

$GState$: Global State $LState$: Local State

able to request the retrieval of m.

By providing storage, the broker is able to support subscribers that require missing updates, thus enabling lost notification retrieval and late entry. However, storing every update since the start of the activity, is not a scalable and efficient approach to support lost notification retrieval.

Thus, to optimize the storage, the developer may select a quantity of notifications to be retained in the broker. Hence if a node requires a lost notification already processed by the garbage collection, it needs to copy the global state of the CRDT from one of its peers. If prior requirements were not assured, then consistency would be compromised, due to loss of updates and no option of recovery.

3.3.2 Disseminating updates

To achieve an interaction where replicas disseminate updates, through the publish/subscribe interaction scheme, it then is required to couple the notion of CRDTs with the generic operations available by the publish/subscribe interface. For a fully operational interaction between replicas participating in collaborative application using CRDTs, multiple specific interaction phases must be detailed (Figure 3.8).

Once a user acquires the shared CRDT object (*CRDT⟨Update⟩*), it then is possible to start interacting with other nodes by disseminating and retrieving updates. By taking the role of a publisher agent, users are able to propagate local modifications to the CRDT state to other participants in a the activity. Such modifications are made available for other participants by publishing update object in the form of *Update⟨localState⟩*. The *update* object carries a local state, only containing the local modifications made by the replica if employing a state-based synchronization, otherwise if employing operation-based synchronization the local state's content contains the operations made since last update publication step.

The update publication is accompanied by a *topic*, with the objective of holding a reference to the CRDT, to which the update is made for. The *Update⟨newState⟩* objects, do not carry the same information as the CRDT objects, such as concurrency semantics or synchronization model, instead they only carry the precise information (local state) for other replicas to converge. By not continuously disseminating the whole CRDT object but instead the update, we aim to reduce the data in traffic, considering that the environments in question are mobile networks, which require special attention to the data size disseminated.

Finally, in a last step before beginning the interaction with the remainder of the collaborative network, a node must subscribe to updates. Interviening replicas show interest in being notified when CRDT updates are made by subscribing to the predefined *update topic*. It is through the use of this operation that replicas may consume updates and subsequently merge the retrieved updates to converge to a new state, mutating to a consistent state with all other interviening replicas once all updates are eventually applied at said replicas.

3.3.3 State-based Synchronization

One of the basic models of CRDTs available in the literature is the state-based CRDTs (CvRDTs) which propagate the hole entirety of the state to other nodes to be merged. By applying the P/S-CRDTs model it is possible to couple CvRDTs with the publish/-subscribe scheme to surpass the inconsistencies introduced by dynamic environments. Although some details have to be specifically applied to support such a synchronization model.

As previously discussed, in the P/S-CRDTs model, the data size of the objects disseminated in the network must be reduced to the minimum. Since the environments in question are mobile networks and said environments don't scale well whenever data size is not carefully taken into consideration. Hence, to adapt state-based synchronization with P/S-CRDTs for mobile environments, some changes where made. Whilst the usual state object design employs a synchronization that disseminates the full global state to other nodes, the P/S-CRDTs adaptation of state-based objects (LState-CRDTs), only sends the node's local modifications to reduce overall communication volume.

Algorithm 1 State-based GCounter CRDT (taken from [46]).

```
payload int[] valP                              ▷ For any id, the initial value is 0
query value () : int
    return $\sum_r valP[r]$
update inc (int n)
    let id := repId()                           ▷ repId: generates the local replica id
        valP[id] := valP[id] + n
merge (X, Y) : payload Z
    for $r \in X.valP.keys \cup Y.valP.keys$
        $Z.valP[r] = max(X.valP[r], Y.valP[r])$
```

Algorithm 2 LState-based GCounter P/S-CRDT (adapted from [46]).

```
gState int[] valP                               ▷ For any id, the initial value is 0
lState int count                                ▷ Initial value: 0
query value () : int
    return $\sum_r valP[r] + count$
update inc (int n)
    count := count + n
merge (X, Y) : payload Z
    let repId := Y.repId()                      ▷ repId: retrieves the replica id
    $Z.valP[repId] = max(X.valP[repId], Y.count)$
```

Thus, when comparing the original design (Algorithm 1) of state objects with the one employed here (Algorithm 2), it is possible to differentiate them by the use of a local state object (*LState*) used by the LState-CRDTs. The local state object represents only the portion of the state which corresponds to the modifications made at source. For instance, a counter CRDT shared between two nodes with a total local value of 10, which is the result of the first node incrementing 7 times and second node incrementing 3 times. With this design, then the object *LState* that is used to represent the portion of the state made at source, is an integer count with the value of 7, and for the global state a count of 10, resultant of the sum of increments made by all nodes. Thus, for both nodes to be consistent the global state count must be the same in both nodes, such that $GState\langle node_i\rangle = LState\langle node_i\rangle \cup LState\langle node_j\rangle$.

Given that the local state object S only carries updates done at source and that to guarantee consistency after applying all updates, every node must reach the same causal history C, for any node x_i of x (adapted from [51]):

- *Initially,* $C(x_i) = \emptyset \wedge S(x_i) = \emptyset$
- *After carrying an update f,* $C(f(x_i)) = C(x_i) \cup \{f\} \wedge S(f(x_i)) = S(x_i) \cup \{f\}$
- *After carrying the merge against* $S(x_j)$ *from node* x_j, $C(merge(x_i, x_j)) = C(x_i) \cup S(x_j)$

Moreover, since updates within this design only carry a node's local modifications, to guarantee that users state are consistent with each other, each last update from each node must be shared and have assured delivery. Hence, by using the previously discussed requirements of lost notification detection and publication persistency, a disconnected node may rejoin the activity and seamlessly regain convergence. Thus, for nodes who disconnect and rejoin, it must be possible to detect the missing updates and retrieve such updates from the broker if still available or if otherwise unavailable retrieve a full copy of another node's global state.

As the broker must ensure publication persistency, it is important to realize that simply saving every update made be each node since the beginning of the interaction in a log, is not a scalable approach. Thus, instead of storing all updates made by a node since the beginning of the interaction as a list, the broker should now employ a storage system where only the latest local state shared by each node is stored. Hence the broker's log should now employ a storage system similar to a multi-value register where now for each node only the latest update is stored, discarding all older versions.

3.3.4 Operation-based Synchronization

Another model of CRDTs available in the literature is the operation-based CRDTs (CmRDTs), that disseminate each local update operation to the remainder of the collaborative network. When applying CmRDTs with the P/S-CRDTs model, additional requirements are compulsory to keep consistency ensured.

As specified by the P/S-CRDTs model, CRDTs carry a local state object and in the case of operation-based synchronization, the object now represents all update operations made locally since the last publication step. If updates are published periodically, these local update operations are inserted in the local state object representing a queue of operations maintaining first in first out order, to ensure that updates are merged with the state by the same order they were made at source (Algorithm 3). For instance, an operation-based GCounter CRDT shared between two nodes, where a node executes two *increment(1)* operations, increasing the counter value to 2, may then share with the other node it's operations by publishing the local state object containing the operations. Once the update is delivered to the other node, the operations contained within it are applied to its own counter value.

Since the state object S carries updates done at source, the causal history $C(x_i)$ of a node is defined as follows (adapted from [51]):

- *Initially,* $C(x_i) = \emptyset \land S(x_i) = \emptyset$
- *After executing the downstream phase of operation* f *at replica* x_i, $C(f(x_i)) = C(x_i) \cup \{f\} \land S(f(x_i)) = \{f_1..f_n\} \cup f$

Contrarily to the state-based model, receiving the last update is not enough to converge, as now updates carry only operations and to converge the CRDT needs the full log of operations or a copy of the global state. Furthermore, given that updates only contain

Algorithm 3 Operation-based GCounter P/S-CRDT (adapted from [46]).

```
1: gState int val                                              ▷ Initial value: 0
2: lState queue ops                                            ▷ Initial value: ∅
3: query value () : int
4:    return val
5: update inc
6:    generator (int n)
7:       ops.enqueue(inc, [n])       ▷ Operation name and parameters for effector
8:    effector (int n)
9:       val := val + n
```

the operations made since last publication step, it means that once a publication is acknowledged by the broker, operations inserted in the local state object are cleared and only available at the broker's storage. Thus, in order to guarantee consistency when using the operation-based synchronization, it is required for subscribers to have the ability to identify the loss of notifications and subsequently be able to retrieve update notifications, if contained in the broker's storage, otherwise a node requires a full copy of another node's state.

Additionally, since the broker has the requirement of publication persistency and no updates can be lost, then a given number of updates are to be preserved so disconnected nodes and late entry nodes can converge to the correct state, by retrieving missing updates from the broker if available at request time. Thus to model the broker storage, a list structure is attributed to each node, where only the last specified number of updates are inserted and maintained throughout the collaborative activity. By specifying the number of updates to be stored by the broker, we aim to limit the size of the broker's log, that if otherwise stored every update throughout the entirety of a collaborative interaction, it would achieve a non scalable size.

3.4 Discussion

In this chapter we presented the P/S-CRDTs model, describing how CRDTs are modeled and coupled with publish/subscribe operations. The selection of the publish/subscribe interaction scheme was due to the loosely coupled form of interaction permitting the decoupling of time, space and synchronization between publishers and subscribers.

Although the publish/subscribe scheme offers great properties for dynamic environments from the get-go, additional requirements must be enforced upon each interaction role to ensure that consistency is kept through the lifespan of the activity. Additionally, the interaction steps were demarcated, where updates are published, stored by the broker and subsequently all subscribers are notified of new updates, retrieving updates and merging them with the global shared state.

Furthermore, although the model is specified to be compatible with the majority of topic-based publish/subscribe systems, a relatively small user implementation is required when applying specific synchronization models. Such as the state and operation based synchronization models specified in this chapter, that have different abstractions of what the update object to be disseminated is, as well as having different requirements for each model. Thus, as long as requirements are followed and subsequent mechanisms to ensure them are implemented, then any other types of CRDTs can be adapted to work with publish/subscribe systems, using the P/S-CRDTs model. Thus, in the next chapter, we demonstrate the implementation, where multiple CRDTs were implemented using the P/S-CRDTs model.

Chapter 4

Implementation

In this chapter we present the implementation of the proposed solution, the P/S-CRDTs model, by assembling it with a publisher/subscriber system, detailing each component and mechanism necessary to complement the needs of the system to guarantee consistency among mobile nodes in a collaborative environment.

Starting with Thyme, a topic-based publish/subscribe system for mobile edge networks, where an in-depth description is given about its procedures and features, such as data publication, subscription, retrieval and replication, in Section 4.1. Followed by the systems architecture used for the implementation, exposing how each component interacts, in Section 4.2.

Subsequently, every detail about the implementation of P/S-CRDTs in Thyme, such as how CRDTs were designed to work with thyme, operations used to disseminate objects, mechanisms added to maintain consistency and the respective data types integrated are presented in Section 4.3.

4.1 Thyme

The publish/subscribe system selected as the base to apply the P/S-CRDTs as proof of concept, is the Thyme [53], a topic-based time-aware publish/subscribe system, precisely designed for mobile edge networks, that forms a persistent collaborative storage system by the medium of peer-to-peer communication between mobile devices.

Thyme distinguishes itself from other publish/subscribe systems by being the first to provide an interface allowing users to specify a time interval in their subscriptions, thus enabling the retrieval of items that were published in the past, presently produced or be notified of future items matching the respective topic subscribed.

In Thyme, nodes do not have differentiated roles, instead nodes share the same responsibilities, being able to be a publisher, subscriber or both, hence there are no centralized components in Thyme. In addition, the system creates a geographical space divided into multiple cells, where nodes belonging to a given cell cooperate with each other to make a virtual node, forming a cluster. Such clusters use a cluster hash table (CHT), with the intent of storing and managing the metadata of published items.

Given the very peculiar characteristics of being designed for mobile edge environments and considering time to be a first order dimension [30], it makes Thyme a very suitable system to apply the P/S-CRDTs model, since not only takes into account node mobility in mobile edge networks but also provides support by allowing the retrieval of items in the past, which is crucial for nodes who have a late entry in collaborative applications and require all updates made.

4.1.1 Thyme's Interface

Thyme is a publish/subscribe system and thus it offers the regular interface methods of publication, subscription and subscription cancellation. Additionally, since Thyme makes an emphasis on the time variable, some operations were tailored to manipulate such variable to its advantage, like the operation of unpublish where a node can remove previously shared data, deleting the data from storage, hence making future nodes not able to retrieve data from the past.

Furthermore, some of the commonly found publish/subscribe operations, such as subscribe have had changes made to take into account the time variable where now a time interval can be specified, making possible the retrieval of data published in the past, also whilst the usual topic-based subscribe interface allows to specify a single topic keyword, in Thyme a more set of features is supported by allowing the formulation of a set of topics/keywords as literals. Moreover, since throughout the implementation multiple Thyme operations were used to accomplish the realization of the P/S-CRDTs model and for better visualization of the specification provided by Thyme, a list of operations is presented as follows:

`publish(dataItem, tags, description, opHandler)`

This method allows the publication of objects in system.

`dataItem` – Representation of the object to be published to the system.

`tags` – A topic *tag* or set of *tags* related to the object to be published.

`description` – Optional object description.

`opHandler` – Callback function characterizing the behaviour to be executed when the operation finishes successfully or fails.

`subscribe(tags, startTime, endTime, notHandler, opHandler)`

This method allows the subscription of objects in system.

`tags` – A topic *tag* or set of *tags* related to the object to be subscribed.

`startTime` – Starting time for the subscription's existence.

`endTime` – Ending time for the subscription's existence.

`notHandler` – Callback function characterizing the behaviour to be executed when a notification matching the subscription topic is received.

`opHandler` – Callback function characterizing the behaviour to be executed when the operation finishes successfully or fails.

`unPublish(objectId, opHandler)`

This method allows the removal of previously published objects in system.

`objectId` – Identification of the object to be removed.

`opHandler` – Callback function characterizing the behaviour to be executed when the operation finishes successfully or fails.

`unSubscribe(tags, opHandler)`

This method allows to deactivate subscriptions made by the node in the system.

`tags` – A topic *tag* or set of *tags* related to the subscription to be deactivated.

`opHandler` – Callback function characterizing the behaviour to be executed when the operation finishes successfully or fails.

`download(metaData, dowHandler)`

This method allows users to download items published and stored in the system.

`metaData` – Metada info belonging to the object to be downloaded.

`dowHandler` – Callback function characterizing the behaviour to be executed when the object is received successfully or fails.

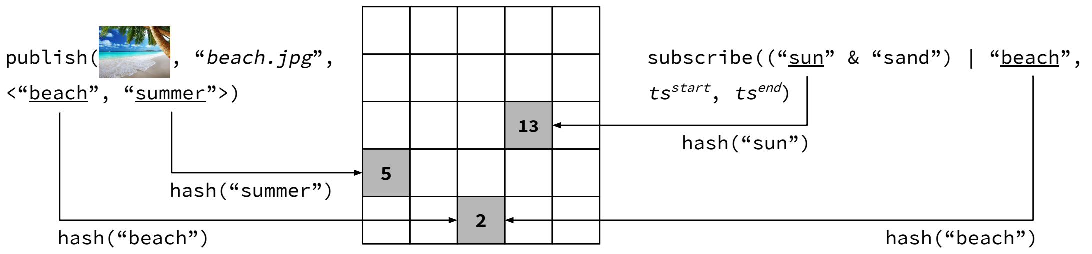

publish(, "beach.jpg", <"beach", "summer">)
hash("summer")
hash("beach")
subscribe(("sun" & "sand") | "beach", ts^start, ts^end)
hash("sun")
hash("beach")
13
5
2

Figure 4.1: Thyme's publication and subscription mechanism, where the hashes computed from the *tags* ("beach" and "summer") related to the publication of the image object ("beach.jpg"), determines the cells accountable for the object metadata (cells 2 and 5) and the subscription (cells 2 and 13). Adapted from [29].

4.1.2 Publishing Data

When a node accomplishes a data object publication, said object is attached to one or multiple *tag* expressions representing the associated topic related to the content produced. Additionally, for each object published a metadata item is created, containing information about the content, ownership and time of publication, given by the tuple $\langle id_{\mathrm{obj}}, T, s, ts^{\mathrm{pub}}, id_{\mathrm{owner}} \rangle$, where each field represents the following specification:

- id_{obj} is the object's unique identifier;
- T is the set of topic *tags* associated to the object;
- s is a brief description of the shared object, such as a thumbnail for an image;
- ts^{pub} is the timestamp of the object's publication;
- id_{owner} is the identifier given to the publishing node.

Upon a node's object publication, the provided Cluster-Based Hash Table (CHT) manages the metadata by disseminating such information to the cells assigned to the tags held in T, given the hashes computed from T, managed by the CHT. Thus, every cluster's inner node receives and permanently stores the given metadata, as illustrated in Figure 4.1, where the cells assigned with the hash corresponding to the published *tags*, "beach" and "summer", receive the object's metadata.

Whilst the cell simply stores the metadata item, the produced data object remains in the publisher's node. Hence, the unit transmitted through the network is only the metadata, with the intent of saving on bandwidth cost and network resources, as metadata tends to have a lesser data size than the actually produced object, resulting in a more efficient approach than continuously propagating the whole object through the network.

A previous shortcoming from the employed approach by Thyme is that only read operations were allowed on produced objects, thus not being possible to edit an object after its original publication. Now with the implementation of the P/S-CRDTs model in Thyme, it is possible to share CRDT objects and continuously edit it trough the dissemination and convergence of updates.

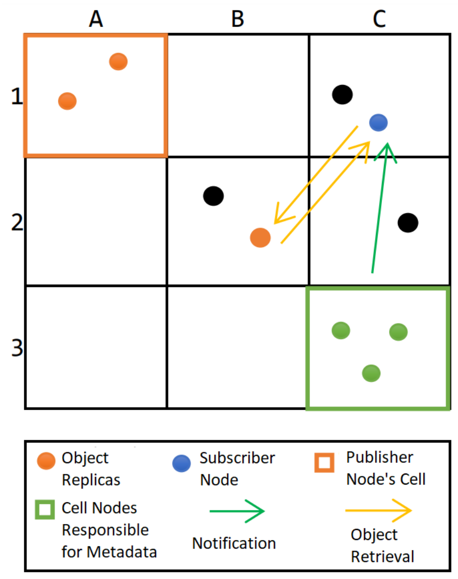


Figure 4.2: Thyme's subscription notification & data retrieval mechanism. Adapted from [29].

4.1.3 Subscription and Data Retrieval

In order to carry out a subscription, system users must send a message to the network withholding metadata information organized as $\langle id_{\text{sub}}, q, ts^{\text{s}}, ts^{\text{e}}, id_{\text{owner}}, cell_{\text{owner}} \rangle$, where each field represents the following specification:

- id_{sub} is the subscription identifier;
- q is a query's logic formula that allows the formation of keyword conjunctions and disjunctions;
- ts^{s} is the starting timestamp of the time interval of published content to be retrieved. To retrieve items published since the beginning of the system matching the specified query, the parameter must have value of 0;
- ts^{e} is the ending timestamp of the time interval of published content to be retrieved. To retrieve all future content matching the specified query, the parameter must have value of ∞;
- id_{owner} is the subscribing node's identifier;
- $cell_{\text{owner}}$ is the identifier of the cell where the subscribing node is located.

Once a subscription is made, the specified query q is initially parsed by the client's Thyme instance, going through the CHT to confer which cells shall be the recipient of the subscription metadata, instead of targeting individual nodes inside the cell's cluster. This is done with the intent of reducing the total number of packets needed to be sent throughout the network.

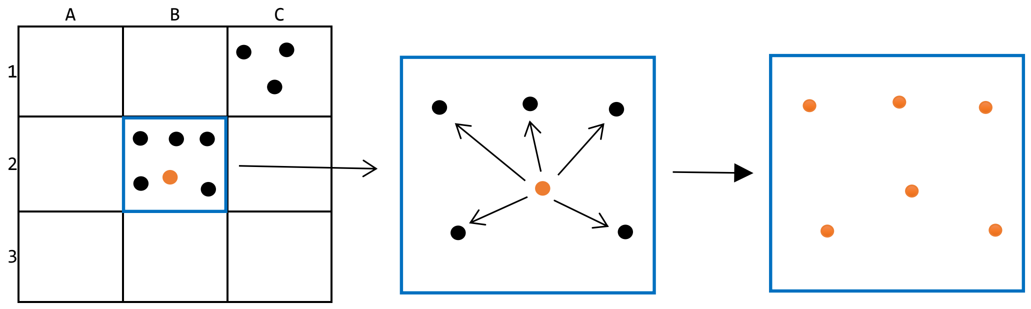


Figure 4.3: Thyme's active replication mechanism. Adapted from [29].

As the messages are delivered to the nodes contained in the selected cells, those nodes are now responsible for maintaining and storing the list of active subscriptions and subsequently notifying nodes of published content matching the topic and time interval (ts^{s}-ts^{e}) specified. Since only the metadata is received once a subscribing node is notified, if the node wishes to retrieve the published object corresponding to the metadata received, then the node must send a data retrieval message to a replica. Such a replica must be selected according to a replica selection algorithm, preferably an algorithm that takes into account the distance between replicas. Thus, if preference is given to the closest available replica in the L_{rep} field (Figure 4.2), it is avoided unnecessary latency input due to data travel time.

In the case where the closest replica doesn't reply within a predetermined threshold, by chance of being offline or left the network, the client's Thyme instance will iterate through the remainder of the L_{rep} list, picking the next closest replica and reiterating the whole process until the desired data is successfully retrieved.

4.1.4 Replication

Given that Thyme is designed for mobile networks, that are dynamic environments with highly volatile behaviors, it is important to ensure that published data doesn't disappear or become inaccessible. Thus, to support data persistence, node entry and disconnection (churn) tolerance and content availability, Thyme offers two replication strategies:

- **Active Replication** consists of the replication of published data inside the publisher's cell. Once an object publication is considered successful, the author of the produced object then shares the item with the nodes composing the publisher's cell cluster, such as depicted in Figure 4.3 where the orange node (in cell B2) upon publishing, replicates the object with its peers, effectively storing the object on all the nodes in the cell.

  Hence, each node that obtained the object after the publication, is capable to reply to data retrieval requests, removing the need to rely on the single publishing node. Thus alleviating the load imposed on the author and effectively boosting the

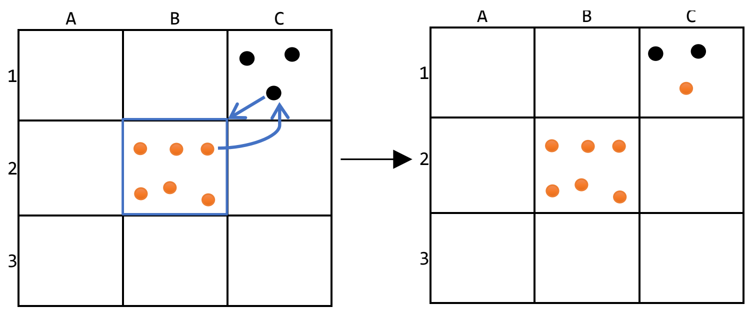


Figure 4.4: Thyme's passive replication mechanism. Adapted from [29].

network's tolerance to node disconnections, since now the object can be accessed without the need for the author node to be active;

- **Passive Replication** takes advantage of object copies replicated across all nodes in all cells of the system, that were obtained through the download operation. Hence, as a node's download request is successful and the object is obtained, the node then becomes the object's passive replica, being capable to reply to future download requests from other nodes, as depicted in Figure 4.4 where the node in cell C1 becomes a passive replica after obtaining the object from cell B2.

  Since objects are now scattered across the network, this approach brings an increase over data availability and performance, considering that nodes interested in obtaining the object can now opt to retrieve such item from the closest replica, thus reducing latency inputs due to high distance communication between replicas.

Additionally, to support the previously described replication mechanisms, Thyme's metadata items now include information about the replication list (L_{rep}), where all nodes retaining the object corresponding to the metadata are listed with their respective location. Hence the metadata tuple is now formed by $\langle id_{\text{obj}}, T, s, ts^{\text{pub}}, id_{\text{owner}}, L_{\text{rep}} \rangle$, where L_{rep} is a list of pairs $\langle id_{\text{node}}, cell_{\text{node}} \rangle$.

4.1.5 Namespaces

The authors of Thyme define as *namespaces* or *worlds*, the realm space where mobile nodes coexist, effectively composing the network. Furthermore, Thyme enables multiple *namespaces* to exist together in the same area, even if overlapped, without creating any issues related to shared resources and synchronization, restricting nodes to belong be part of only one *namespace* at a time.

When beginning the interaction, Thyme provides nodes with the choice of joining an existent *namespace* or to create a new one. In order to join an existent *namespace*, a node must check the heartbeat by listening to packets transmitted via the network (e.g. "hello" messages), for a predetermined interval of time, ensuring that the *namespace* to be joined is live.

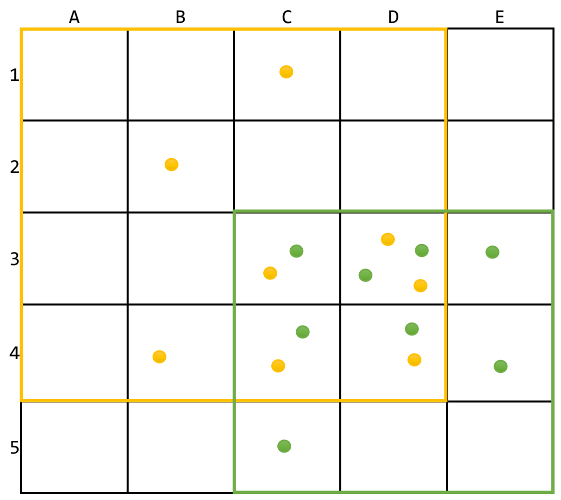


Figure 4.5: Overlapping Thyme *namespaces*. Adapted from [29].

On the other hand, if the node chooses to create its own *namespace*, the specification of multiple parameters is required, such as the time value of how long the realm will be active and the name used by other nodes to search and join. As the *namespace* is created, a unique identification is generated as well as the establishment of a cell to accommodate future *namespace* nodes.

Overlapping realms may have nodes interacting to provide better performance and availability, enabling nodes from different realms to store as well as share objects and metadata, as depicted in Figure 4.5. Although nodes are only able to reply to object requests from nodes forming the cluster where the data item belongs.

4.1.6 Node Mobility

Taking everything in mind about mobile environments, where nodes may leave and enter the network at their very own free will, Thyme considers then node mobility to be of the uttermost importance, by specifying mechanisms to embrace mobility whilst diminishing any complications caused by node's movement. Therefore, when a node's Thyme instance detects an abnormal amount of movement, a warning message is broadcasted to composing cell's cluster nodes, informing that the node is no longer viable, therefore not capable of replying to requests and only apt to receive local update messages.

Contrarily, when a node has small movement or is stationary, it becomes stable and thus the node's Thyme instance communicates with its peers, informing that the node is now available to regain full responsibilities, being able to reply to requests. Although the node may viable again, two scenarios may have happened, such that the node may have moved but not covered enough distance to justify the transition to another cell, making it immediately qualified to cooperate with the network.

On the other hand, if it transitioned to another cell, there are multiple node information fields that require an update, such as its own state that needs to be updated with the state of the cell, by asking one of its peers for the full state, obtaining knowledge of objects responsible by the cell so that it may help with future requests. Furthermore, if a transitioning node, was passively replicating data objects, it is required to instruct cells to update the metadata of the replicated objects, updating the L_{rep} list about the new location ($\langle id_{\text{node}}, cell_{\text{node}} \rangle$).

As for all previously made subscriptions, a notification must be sent to all cells responsible for managing subscriptions, informing the new location, so that when matching content is published, the notifications are sent to the new location instead of getting lost by ending up on the previous location. Conversely, if the transitioning node made any prior publications, a message must be sent to all cells retaining the delivered object's metadata, to update once again the location field with the new one. As a consequence, the node now will be considered a passive replica to the subscribed object, since now it individually holds the object data in the new cell.

For nodes who took the liberty of leaving the defined network's geographical space, it will now cease to be apart of the system and no longer cooperate.

4.1.7 Discussion

In the context of the P/S-CRDTs model, the aim is to adapt the CRDT design with dynamic environments, lifting the demand of knowledge to whom state updates must be delivered to. Since maintaining the knowledge of active nodes throughout the lifespan of a collaborative activity is not a sustainable approach, due to the weak nature of mobile communications, where mobile nodes may periodically enter and leave the activity unwillingly due to network disconnections, an interaction scheme that decouples the knowledge is then a must.

Thus, publish/subscribe systems presented themselves by being an adept choice, since publishing and subscribing roles happen in two separate control flows, enabling the dissemination of updates without the need to hold the recipient's reference, as well as making possible the retrieval of updates even if the author is no longer active.

Therefore, by providing the topic-based publish/subscribe interaction scheme, being designed for mobile edge networks, allowing the retrieval of past updates through the specification of time on the subscription operation, as well as having mechanism to cope with node mobility and offering data persistence through replication techniques, Thyme showed itself as the best candidate for the implementation of the P/S-CRDTs model.

Although providing great properties from the get-go, additional mechanism have to be added to Thyme, such as the detection of lost notifications and subsequent retrieval, as well as storage optimizations concerning the selected CRDT synchronization models implemented, that are state and operation synchronization, each having different requisites.

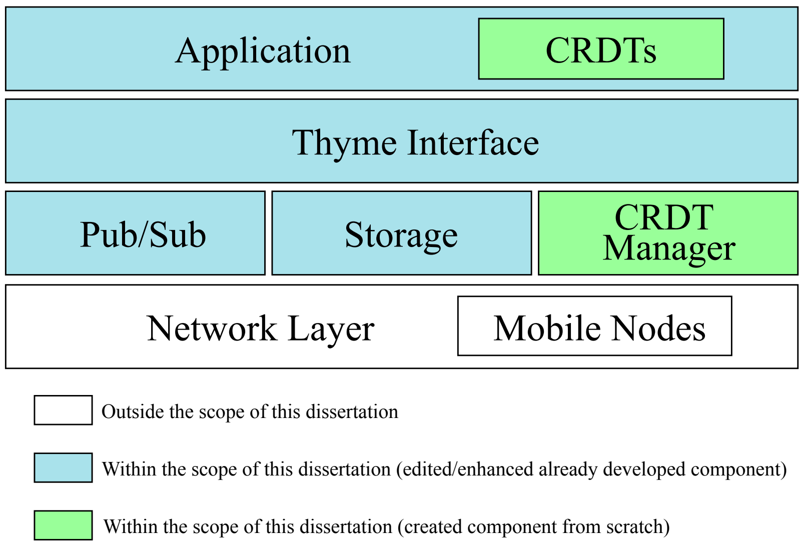

Application
CRDTs
Thyme Interface
Pub/Sub
Storage
CRDT Manager
Network Layer
Mobile Nodes
Outside the scope of this dissertation
Within the scope of this dissertation (edited/enhanced already developed component)
Within the scope of this dissertation (created component from scratch)

Figure 4.6: System architecture.

4.2 System Architecture

As the implementation of the model encompasses the junction of P/S-CRDTs and Thyme, the structure of the software stack used by mobile nodes in the system to accomplish a collaborative activity, is then composed by several layers establishing a connection between the user application level to Thyme's interface and provided services, as depicted in Figure 4.6. Thus the system presents the following architecture:

Application. As the top-level layer available to the mobile user, the application, comprises the logic unit of the application interacting with the author's defined CRDTs, by locally modifying their internal state with designated update operations, e.g. element insertion and removal over a set CRDT.

Later in Section 5.6, a demonstration of a use case scenario is described, showcasing a simple java collaborative *Todo list* application, employing the specified P/S-CRDTs as proof of concept.

CRDT Manager. This component was developed with the intent of supervising the produced and obtained application CRDTs. Whilst developers may wish to implement an approach where updates are propagated immediately after an update, here a periodic dissemination was chosen. Thus, the manager contains a periodic task scheduled for a defined time interval, where it is checked if modifications were made.

As an additional implementation detail and in order to check whether or not a modification was made, CRDT items have employed a *save* operation where the node can specify in what moment he wishes to make modifications since previous publication available to other subscribing nodes in the next publication. Hence the manager checks to see if the *save* operation was triggered since last update dissemination round.

Thyme ***Interface.*** Thyme provides an interface for instance initialization and access to respective interaction operations, such publish, unpublish, subscribe, unsubscribe and download, as previously shown in Section 4.1.1.

The interface may be directly accessed by an application or other external components, in order to trigger the publication of objects, in this context CRDTs and subsequently execute subscriptions in the past, present or future, given the time interval arguments exposed by the subscribe operation in the Thyme's provided interface.

Publish/Subscribe. The publish/subscribe module, incorporates all the logic for the dissemination and retrieval of objects from mobile nodes in the network. This module is branched in two distinct roles, the client and the server, where each one has its functionalities. Whilst the client role manages the publication and subscription requests as well as all notifications delivered to the node, the server, in turn, manages the storage and link between subscriptions and publications.

Through the implementation of the proposed solution, this module saw several changes to accommodate the dissemination and retrieval of updates as well as the additional mechanisms to prevent CRDT sate divergences due to the chance of lost notifications.

Storage. Storage is provided in Thyme as a service, responsible by managing stored data in the system. Storing data is accomplished by using the abstraction of cells, where the active replication of data inside the cell, allows containing nodes to keep the content and subscription metadata thus granting data persistence, while at the same time users act virtual brokers, by replying to incoming requests.

This service, saw small changes added throughout the development to help with the storage of updates, differentiating the manner of how storage is conducted for state-based and operation-based types since state synchronization only requires the last full local set of modifications made to converge, whilst operation synchronization requires the reception of all individual set of updates.

Network Layer. The mobile network formed by Thyme is done in this layer, where *namespaces* (realms where nodes are part of) are generated and managed. It provides asynchronous interaction between multiple Thyme instances and allows nodes to join or create new *namespaces*.

Additionally, the network layer accounts for node mobility by applying mechanisms to update the localization of the node, making sure data requests are always sent to the updated location, ensuring requests are delivered (Section 4.1.6).

4.3 P/S CRDTs in Thyme

To demonstrate the concept of P/S-CRDTs, in this section a demonstration will take place, defining the implementation details behind the chosen design of CRDTs to cooperate with

THYME's system details and provided operations. Furthermore, for the management of CRDTs with the intent of automation of update dissemination, a simple component was developed.

Additionally since, through the development of this thesis, two CRDT synchronization models were chosen. Those being the vanilla models of operation and state-based CRDTs, that given the models differences, require additional implementation details related to the design of the update unit to be disseminated. Thus, for each model, a step by step description of the objects to be used in said synchronization models is given.

4.3.1 Mutable Objects

In THYME the unit characterized as the object to be disseminated in the system is detailed as a *DataItem*. In the previous THYME implementation, said units could only be non-mutable objects, thus not being able to be edited after being published, hence THYME only supported immutable items.

With the introduction of CRDTs onto the THYME's system, the extension of the original *DataItem* was required to integrate the CRDTs common methods that allow users to edit the item and converge with updates made by other user nodes in the network. Thus, the *MutableDataItem* was created, to represent the generic interface the CRDTs (mutable objects), where usually convergence, query and update methods are provided.

CRDTs, when produced by the author, must be instantiated with the global state value and a local *state* object, representing only the updates made by the node. To better understand, let's take as an example of a state-based Observed-Remove Set (OR-Set) CRDT. The OR-Set uses for each element contained in the set, a list with addition tags (*addTag*) and another list with removal tags (*rmvTag)*. To update the OR-Set CRDT with an element insertion, a new unique *addTag* must be generated and attached to the element in the observed list. Conversely, to remove an element from the set, all *addTags* must be put onto the removed list, becoming *rmvTags*.

Therefore, taking the example of the OR-Set, the global state of the object is the two lists (addition and removal lists) where all seen updates from other nodes are merged into, whilst the local state object is merely the subset of the global state, being also two lists but with only the locally inserted and removed operations. The usage of the local state object is made with the intent of reducing at maximum, the data size of updates transmitted in the network.

Subsequently, to converge the state between two nodes, *a* and *b*, when *a* triggers the merge method, the local state object from *b* is received and merged with the global state value of *a* by making the union of both addition lists and union of removal lists. Whilst, the common implementation found in literature uses the global node state objects to merge, combining all the updates seen by the other node, in this context of mobile networks, it puts a much bigger load on the network than needed, since propagating all seen updates constitutes a *dataItem* with potential bigger than average data size. Besides, a query method is always available to retrieve the global value of the CRDT. Such query for the OR-Set, consists on returning the resulting set of the addition list minus the removal list. Therefore, an element is contained in the set if the addition list less the removal list is not empty.

Moreover, as an implementation choice, *MutableDataItems* contain a *save* operation. By saving, the node shows that it's interested in sharing the updates with the network. To check whether or not a CRDT has a new state to be published, two version values are present, given the tuple $\langle innerVer, outerVer \rangle$, where *outerVer* is the last version of the state that has left the node to be shared and *innerVer* the current local version of the state. Hence, if *innerVer* > *outerVer*, the node is ready to share the new updates (Listing 4.1).

It should be noted, that the *MutableDataItem* is made as an abstract object, enabling users to incorporate any desired data type with respective convergence, query and update methods.

Listing 4.1: Mutable Data Object Specification

```
// Payload
protected State<S> state;
// Latest version published
private int outerVersion = 0;

// Merging Function
public abstract void merge(final State<S> otherState);

public final State<S> shareState() {
 if (state.getVersion() > outerVersion)
  return this.state;
}

public void save() {
 state.update(outerVersion);
}

public void updateVersion() {
 outerVersion = state.getVersion();
}
```

4.3.2 Managing CRDTs

When designing collaborative applications with the use of CRDTs, one may incorporate one or more CRDT objects of multiple data types and synchronization models, thus having a component that manages, the application CRDTs is beneficial. With that in mind, the CRDT manager was developed to alleviate the mobile user's responsibilities by automatically checking each node CRDT for a new state version to publish.

CRDTs are registered in the manager once a publication acknowledge is delivered

Algorithm 4 Outline of a the CRDT manager specification.

```
1:  storage CRDTMap<id,CRDT> mutableItems                    ▷ Initial value: $\emptyset$
2:  query items () : CRDTMap
3:      return this.mutableItems
4:  insert addMutable (objId, item)
5:      if !mutableItems.contains(objId)
6:          mutableItems.put(objID, item)
7:  periodically do (update dissemination)
8:      for $r \in$ mutableItems.keys
9:          if $r$.innerVersion > $r$.outerVersion
10:             publish(mutableItems.get($r$), topic)
```

to the publisher indicating that the publication was successfully made, or when a subscriber receives a notification matching the desired *tags* and follows up with the download operation to retrieve the object from the responsible cell.

Once registered in the manager, the CRDTs are periodically accessed by a *timerTask* scheduled to trigger every time interval the developer wishes, for testing purposes the chose interval was one second. The task consists of checking if CRDTs have a new state containing the newly made updates ready. Since the manager has access to the $\langle innerVer, outerVer \rangle$ tuple, it can know whether there is a new local state version available or not. If there is a new version, then the manager spontaneously publishes the update, containing the new state (Algorithm 4). Once and the specified callback handler responds with the successful acknowledgment from Thyme's publish/subscribe module, the manager updates the CRDTs *outerVersion* value to the *innerVer* value, meaning there are no new modifications until the next save operation is triggered by the node.

4.3.3 LState-based CRDTs

As state-based synchronization is one of the chosen CRDT synchronization models chosen to be implemented using the P/S-CRDTs model specification, some additional changes to the original model were compulsory to accommodate a scalable approach for mobile networks (Algorithm 5). Thus as previously defined by the P/S-CRDTs model definition in Section.3.3.3, a new iteration was made over the state-based model, called local state-based synchronization (LState). Hence, when comparing both the original and new iteration model of state-based synchronization, changes we made to the construction of updates objects and one should be merged.

For implementation demonstration purposes, we will now describe how an LState GCounter data type is constructed using the P/S-CRDTs model and Thyme's publish/-subscribe features. Any CRDT object specified must begin by instantiating its state object, as the state object is the item to be sent within an update item to other nodes in the collaborative activity. Thus for a GCounter object, the local state is of type $\langle replicaID, Integer \rangle$, where a replica identifier is given along with its local counter value. In addition, as per

Algorithm 5 LState-based GCounter P/S-CRDT

```
1: gState int[] valP                                  ▷ For any id, the initial value is 0
2: lState int count                                   ▷ Initial value: 0
3: onInit():
4:     CRDTManager.register(this)
5: save():
6:     CRDTManager.updateVersion()
7: query value () : int
8:     return $\sum_r valP[r] + count$
9: update inc (int $n$)
10:    $count := count + n$
11: merge ($X$, $Y$) : payload $Z$
12:    let repId := $Y.repId()$                       ▷ repId: retrieves the replica id
13:    $Z.valP$[repId] = $max(X.valP$[repId]$,Y.count)$
```

original in the state-based definition of GCounter a structure is used to store all other node's counter values, thus a map *incs* is used to formulate the global shared state. The global shared state is then used to calculate the global value of the GCounter by doing the sum of stored value per node known, thus for all nodes to be consistent in a collaborative activity, the sum of all values stored by this structure must be the same across all other nodes. In Thyme the replica identification is the address which is included in the notification's metadata produced, hence whenever an update is received a node may use the address corresponding to a local state of the publishing node, to store its counter value.

Furthermore, two object constructors are given, one for CRDT creation by the author node and another when nodes instantiate their own counter object with a copy from another node. As GCounter objects stand for the *Grows-only* semantic used, where only increments are allowed, the interface only exposes the increment operation, that must first be conducted locally, applying its value to the local and global state alike.

Once a user wishes to expose its modifications with the network, it may trigger the save action, accessing the *CRDT Manager* component, increasing its inner version, triggering the publication of the local state as the *CRDT Manager* verifies that the user inner version is superior than the outer version meaning an update is available for sharing.

Finally, as nodes retrieve updates, the *merge* function is used to apply the semantic concurrency employed by the data type. Hence, in this instance, the GCounter data type verifies that the received update has a bigger count value compared to the one stored for the respective update author, if so then the new count value is applied to the global state.

4.3.4 Operation-based CRDTs

The other chosen synchronization model to be adapted by using the P/S-CRDTs model specification is the operation-based synchronization model. To do so, some additional changes were made to facilitate update dissemination. Comparatively to the originally

Algorithm 6 Operation-based GCounter P/S-CRDT

```
1:  gState int val                                        ▷ Initial value: 0
2:  lState queue ops                                      ▷ Initial value: ∅
3:  onInit():
4:      CRDTManager.register(this)
5:  save():
6:      CRDTManager.updateVersion()
7:      ops.clear()
8:  query value () : int
9:      return val
10: update inc
11:     generator (int n)
12:         ops.enqueue(inc, [n])        ▷ Operation name and parameters for effector
13:     effector (int n)
14:         val := val + n
```

defined model, this implementation chooses to accumulate operations in a queue structure instead of immediately disseminating the operation to other nodes. Thus in this adaptation, the update object contains all local operations since the last publication step.

For implementation demonstration purposes, we will once again describe how a GCounter data type is implemented by using the P/S-CRDTs model. To begin, the CRDT object must be instantiated with the local state in the form of $\langle replicaID, Queue < Ops > \rangle$, where a replica identifier is given along with a queue structure to hold local made updates. In addition, as per the original in the operation-based definition of GCounter a structure is used to store the global state, in this case an integer representing the counter (Algorithm 6).

Furthermore, as every CRDT implemented in Thyme using the P/S-CRDTs model, two object constructors are given, one for CRDT creation by the author node and another when nodes instantiate their own counter object with a copy from another node. Once more, for this data type, the only available update operation is the increment operation. Whenever an update is conducted, the *generator* function is called, creating an operation object specifying the operation to be applied once delivered to other nodes and specific arguments attached to the operation.

As a user wants to expose its updates with the network, it may do so by triggering the save action, accessing the *CRDT Manager* component, increasing its inner version, consequently triggering the publication of the local state as the *CRDT Manager* verifies that the user inner version is superior than the outer version meaning an update is available for sharing. Finally, as nodes retrieve updates, the *effector* function is called, retrieving operation from the queue structure contained in the update, applying each operation locally.

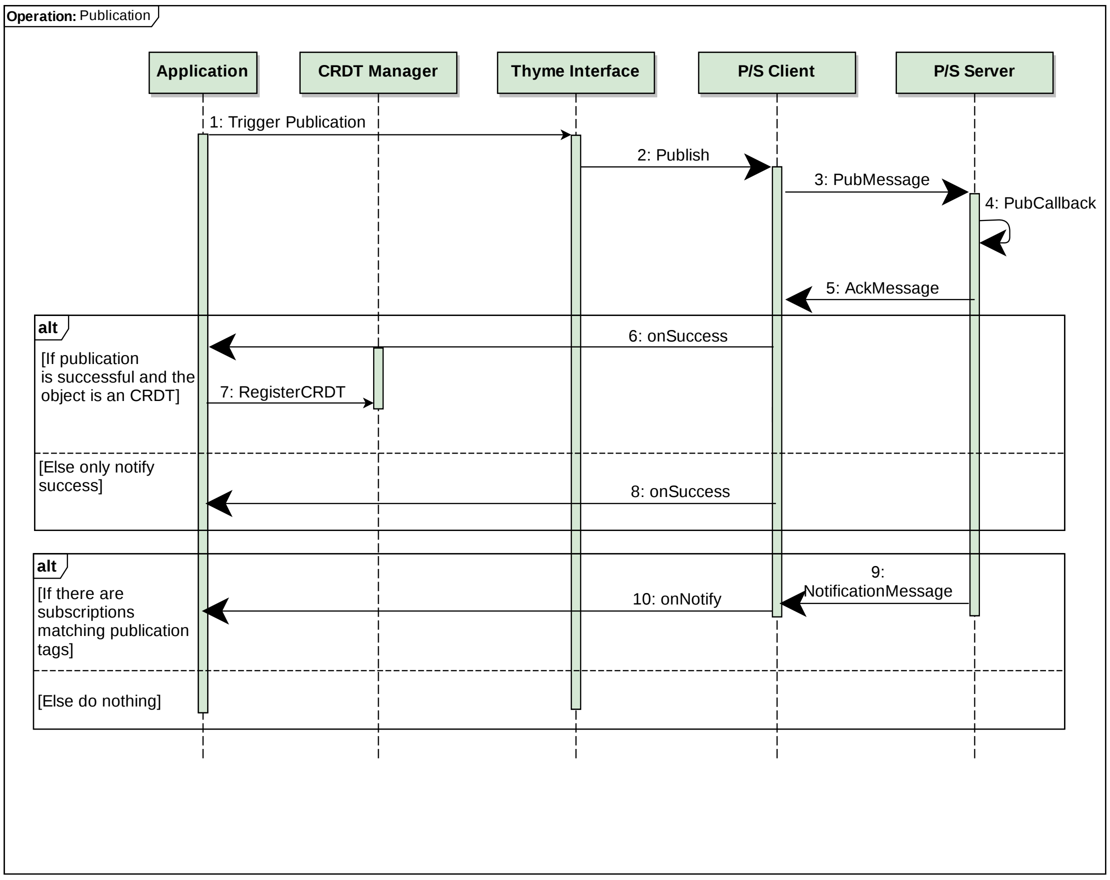


Figure 4.7: Object publication sequence diagram.

4.4 Collaborative Node Interaction

As the end game goal is to demonstrate the application of P/S-CRDTs in dynamic environments, one of the main usages of today for CRDTs in the mobile network context, are mobile collaborative applications. Such collaborative activities, consist of one or more CRDT objects replicated amongst the intervenients of the activity. Throughout the lifespan of the activity, users may edit the shared CRDT by applying their own local updates and eventually later disseminating the changes made to other user nodes, for merging. Once updates in the form of state or operations are merged, the CRDTs guarantee data consistency, more precisely the eventual consistency model.

Thus, in this section we aim to explain the connection made between components developed and services provided by Thyme, showcasing the interaction and shared data among components to achieve the dissemination and retrieval of updates in a collaborative activity.

4.4.1 Acquiring CRDT Objects

As one of the main steps to be accomplished by the nodes in the collaborative activity, is to attain the CRDT object, in order to edit the state and share updates, effectively commencing the cooperation between nodes. Seeing that the select means of data dissemination is via a publish/subscribe system, in this case Thyme, then to attain the CRDT object, first one must initially produce the object. Much like in collaborative text editing, first the author node creates the text document to subsequently share it later.

Once produced, the author must use Thyme's interface to publish the CRDT in the system under any desired topic. If nodes who desire to be part of the activity already know the topic in which the CRDT will be attached to prior the publication and used it to subscribe, these will immediately receive the notification (Figure 4.7), else if only after the authors publication they obtain the knowledge of the topic used, they must use Thyme's subscribe operation, specifying in the starting value of the time interval where published objects will trigger a notification, the value of 0, corresponding to the beginning of time for Thyme.

Upon notification arrival, the node will receive the metadata corresponding to the object, informing the location cell to be targeted to retrieve the copy of the object. Thus, using Thyme's download operation, the node may attain the CRDT. To be noted that as CRDTs are obtained by the nodes, these must be registered to their respective management component, to begin the dissemination of updates.

4.4.2 Sharing state modifications

As CRDTs are attained by the intervening nodes, these may start to update the object with the methods provided by the respective data type's interface. But before, updates may be received by all intervening nodes, each must get subscribed to receive updates by a specific topic where only update objects to the respective CRDT are published to. For the implementation, the topic used for updates is the *objID* field held in the metadata information of the object, generated at publication time as a universal unique identifier. Hence, nodes how already have a copy of the CRDT, already have the *objID* since when a subscription notification is received, only the metadata is contained in the notification and not the published object.

As nodes subsequently update their CRDT's state, they may chose to share the modifications with the network by executing the save operation, resulting in the increment of the *innerVer* value, indicating a new available state to be shared. In order to share the new state, the CRDT manager checks that a new state is ready and publishes it under the previously received *objID* as the topic.

Once nodes receive notification of updates from other intervenients, the download operation is automatically called and as the update is retrieved, the nodes merge the state object contained in the update, effectively converging the state with all other nodes once all updates are retrieved. For nodes who experience disconnection, updates may be lost

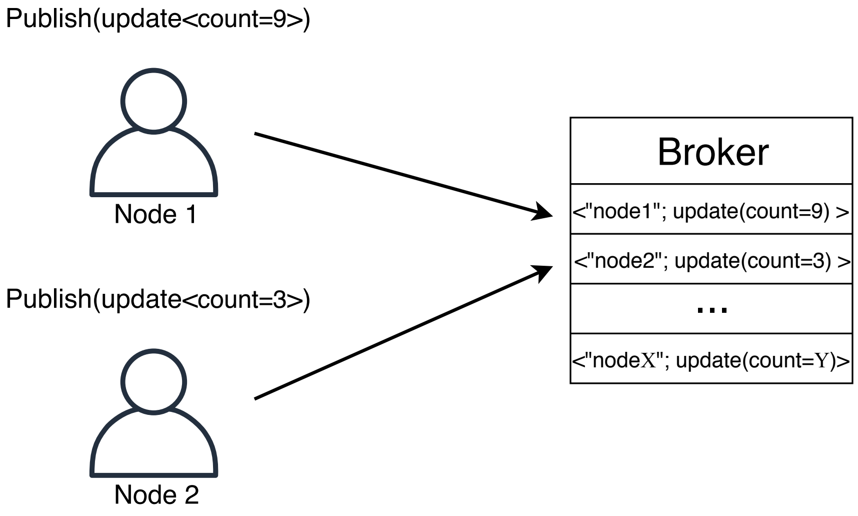

Publish(update<count=9>)
Node 1
Publish(update<count=3>)
Node 2
Broker
<"node1"; update(count=9) >
<"node2"; update(count=3) >
...
<"nodeX"; update(count=Y)>

Figure 4.8: Broker storage for lState-based objects.

requiring aid of a mechanism to detect which updates were lost, so further retrieval of said lost updates may be possible.

4.4.3 Storing updates

Given that Thyme grants the storage service where recent data persistence is assured, data is always expected to be available. Thus, as updates are published and stored at the broker, care must be given to not store unnecessary data, to avoid any memory overheads.

For instance, state-based P/S-CRDTs share their local state which means that a newer update publication from the same node, carries a new local state containing the union of the previously published state with the new update modification, hence for this synchronization type, only storage is required for the newer versions. Thus, at the storage layer changes were made to identify when updates for state-based objects are stored, changing the storage approach to behave such a multi-value register, by saving only the last update published for a specific CRDT by a node, thus as the key is the update topic for a CRDT and the multiple values are tuples of $\langle nodeAddress, lastUpdt \rangle$ (Figure 4.8).

For operation-based objects, the story is quite different, since now updates carry the operations carried by publishing node since last publication, thus a new update from the same node does not contain the operations contained in older updates. Hence now no updates may be lost and for that reason, the storage now acts as list were all updates are ordered by their time of publication (Figure 4.9).

4.4.4 Lost notification detection and retrieval

As mobile users suffer from the weak nature of mobile networks, being prone to periodic involuntary disconnections, they may leave the collaborative active, losing updates.

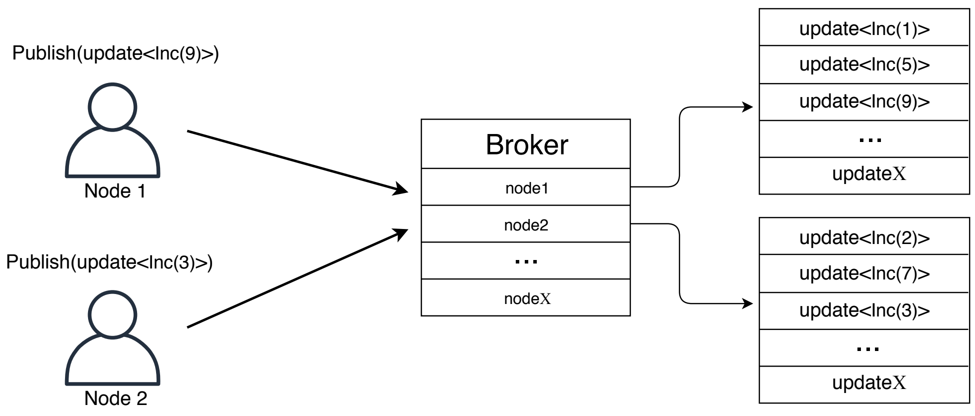


Figure 4.9: Broker storage for operation-based objects.

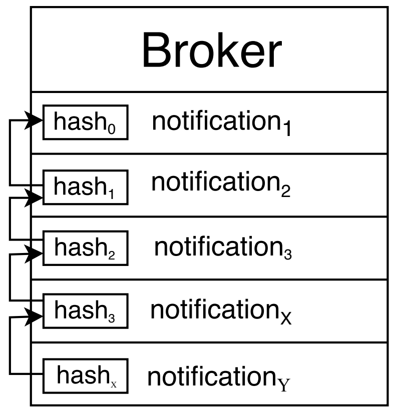

Broker
hash$_0$ notification$_1$
hash$_1$ notification$_2$
hash$_2$ notification$_3$
hash$_3$ notification$_X$
hash$_x$ notification$_Y$

Figure 4.10: Notification hash mechanism used to identify and retrieve missing update notifications.

For cases such as operation-based objects where update notification cannot be lost to guarantee consistency, it is important to be able to detect and retrieve lost updates.

For this reason a mechanism was developed, using the popular block chain logic, in the sense that it uses the next notification to carry the hash of the last notification created (Figure 4.10). When a node receives a notification, it is checked if the contained hash was previously seen, if not it means that a notification was lost. To retrieve the lost update or updates, an operation is provided indicating the last seen hash and the newly received hash, downloading all updates with hashes between the ones previously specified in the arguments. To be noted that in state-based objects, if the last notification is not retrievable and no more updates are made, there is no feasible way for a node to know that it is missing an update and thus cant converge because it cant retrieve the missing update. Hence, one of the requirements is that the last notification delivery, must always be assured.

4.5 Summary

In this chapter we presented the implementation of the proposed solution, by coupling the P/S-CRDTs with Thyme, detailing each component and mechanism necessary to complement the needs of the system to guarantee consistency among mobile nodes in a collaborative environment. We described Thyme's system services and interaction mechanisms, followed by the stack architecture used by our mobile users, describing each layer and its contents.

Furthermore, a description of how P/S-CRDTs were designed to cope with Thyme was given, showing how objects are managed and the manner how they are specified, from the content of the updates to the way CRDTs are checked to see if a new update is available to share. Moreover, the sequence of operations made to achieve a fully fledged collaborative activity was detailed, going through how to acquire CRDTs, share updates as well as how to store updates and retrieve lost update notifications.

Chapter 5

Evaluation

In this chapter, we showcase the obtained experimental results, with the intent to examine the behavior and performance of P/S-CRDTs in cases of volatile scenarios. For testing purposes, various simulations of collaborative activities were developed where multiple mobile nodes interact with each other, whilst experiencing instances of disconnection and late entry in a controlled environment.

To better understand how P/S-CRDTs compare to existing approaches two study cases were created. Firstly, the Δ-CRDT model is compared against the current P/S-CRDTs designs implemented on top of Thyme as a competitor solution. Additionally, an adaptation of an open-source collaborative Java todo list application from AntidoteDB was used to compare results against the CRDT update dissemination mechanism used by the AntidoteDB application versus the P/S-CRDTs model implemented on Thyme.

We begin by presenting the evaluation methodologies, detailing how simulations were built and what questions we intent on answering with the simulations in Section 5.1. We start by answering the question of how much network volume is generated with P/S-CRDTs in Section 5.2. Following with the analysis of churn and late entry events in Sections 5.3 and 5.4. As a competitor model, the delta-based CRDTs are presented, by describing how the implementation with Thyme was done, along with its respective mechanisms in Section 5.5. As a use case, a simple Java collaborative todo list application was made, based on an existent Java AntidoteDB todo list application, with the objective of comparing against the update dissemination model of AntidoteDB, in Section 5.6.

5.1 Evaluation Methodologies

In order to test and gather results from our solution, a simulated environment was made. To create various simulated scenarios where mobile nodes experience volatile behaviors, it was used a Java trace-based simulation framework, previously developed in the context of Thyme's development. The simulation framework offers an emulation of Thyme's network layer, allowing the logical dissemination of messages between any given number of virtual mobile nodes within a single machine. Simulation scenarios comprise multiple virtual nodes, each represented by a single thread, that obey to user defined behavior detailed in a trace file.

Furthermore, trace files are composed of multiple unitary actions, each action always being instantiated with a time unit representing the moment in which an action must be executed, and with a node's numeral identification in order to attach a specific action to the specified node (Listing 5.1). Along with the common arguments, actions additionally carry all the necessary custom fields belonging to a specific action e.g. subscription action includes a topic *tag* and Thyme's optional description field. Since the traces developed have the aim of emulating a collaborative activity using P/S-CRDTs, new custom actions were added to the trace framework, such as specific CRDT update operations e.g. increment and decrement for counter objects, as well as an action for the implemented save mechanism that allows nodes to share their local modifications and an assert action used at the end of all traces to attest eventual consistency. By constructing custom traces, it then is possible to subdivide test scenarios into multiple categories where we gather results. The studied scenarios depict the possible common behaviors characteristic of dynamic environments, thus the three main scenarios studied were:

- A perfect activity interaction where all participants join at the beginning to interact, never leaving and never arriving late, hence never losing updates.
- An activity where some participants experience occasional disconnection and subsequent reconnection events (*churn*), making them lose other participant updates.
- The introduction of late entry participants in the activity, meaning that to start interacting the participants must retrieve an up to date copy of the shared state.

Traces created to express the considered scenarios, revolve around the developed implementations of P/S-CRDTs. For instance, a scenario may be created to test an ORSet implementation, using either the adapted LState or Operation synchronization models, mirroring a supermarket purchase list where nodes insert and remove elements in the set. Usually, test traces include a varying number of nodes, raging from low quantities to high quantities. In this evaluation, the lowest number of nodes used in a trace was 3 nodes and the highest being 100 nodes. In scenarios where users disconnect, the intended behavior is for a node to lose updates and attest that after reconnecting a node can still converge to a consistent state, for that reason nodes who disconnect in this evaluation may lose between 5 and 20 updates. If a node loses updates that are no longer stored by the broker agent, it is required for a node to acquire a copy of the CRDT from one of its peers.

Listing 5.1: Example of a trace file.

```
NODE$|$0.00000$|$0
NODE$|$0.00000$|$1
NODE$|$0.00000$|$2

#Share PNCounter object identified by "0550-0001".
PUB_MUTABLE$|$10.00000$|$0$|$State$|$PNCounter$|$0$|$0550-0001$|$counterApp1
SUB_MUTABLE$|$20.00000$|$1$|$0550-0001$|$counterApp1

#Subscribe to updates for PNCounter identified by "0550-0001".
SUB_UPDT$|$40.00000$|$0$|$0550-0001
SUB_UPDT$|$50.00000$|$1$|$0550-0001

#Nodes 0 and 1 modify their local PNCounter value (increment/decrement)
INC$|$80.00000$|$0$|$0550-0001$|$1
INC$|$80.00000$|$1$|$0550-0001$|$2
DEC$|$85.00000$|$1$|$0550-0001$|$1

#Disconnect node 0.
PAUSE$|$100.00000.00000$|$0

#Nodes 0 and 1 modify their local PNCounter value (increment/decrement)
DEC$|$110.00000.00000$|$0$|$0550-0001$|$2
INC$|$110.00000.00000$|$1$|$0550-0001$|$5

#Node 1 saves and shares its modifications.
SAVE$|$130.00000$|$1$|$0550-0001

#Reconnect node 0.
RESUME$|$150.00000$|$0

#Node 0 saves and shares its modifications.
SAVE$|$120.00000$|$0$|$0550-0001

#Late entry for node 2.
SUB_MUTABLE$|$160.00000$|$2$|$0550-0001$|$counterApp1
SUB_UPDT$|$170.00000$|$2$|$0550-0001

#Check if counter value is 5, if so consistency has been achieved.
ASSERT_STATE$|$250.00000$|$0$|$COUNTER_STATE$|$5$|$0550-0001
ASSERT_STATE$|$260.00000$|$1$|$COUNTER_STATE$|$5$|$0550-0001
ASSERT_STATE$|$270.00000$|$2$|$COUNTER_STATE$|$5$|$0550-0001
```

The results gathered, were obtained by running test simulations with Java 8 in a remote machine belonging to Grid5000 testbed [16], located at Lyon with the following specification: Intel Xeon E5-2620 v4 2.10GHz octa-core processor with 64GB of RAM and a 10 Gbps Ethernet connection. For consistency sake, when creating the traces of the same data type to later compare with other implemented benchmark components, the same number of nodes and actions are used, only changing the description of the item used by the trace. For instance, when comparing local state and operation based PNCounter P/S-CRDTs, the same number of nodes, operations and disconnection events are applied, at the same specified time. Additionally, two benchmark components were developed as a means of comparing different update dissemination models. An adaptation of the Δ-CRDT model and an implementation of an open-source Java collaborative todo list application using AntidoteDB CRDTs were used as competition benchmarking. These two components have been adapted to Thyme and simulation traces were created to gather results against P/S-CRDTs.

Moreover, by executing the stack of trace scenarios in a simulated environment, we are able to perceive in greater detail, the scalability of the proposed solution without suffering potential hardware limitations. Hence, as the main objective of the evaluation is to study the update dissemination model, the gathered metrics revolve around the number of messages and network traffic size used throughout the complete duration of a collaborative activity, distinguishing the type of messages sent (publications, subscriptions, and downloads) and respective memory size used for each message type. Ultimately, with this evaluation we look forward to answering the following questions:

1. What is the volume of communication placed on the network, concerning the number and size of each type of messages disseminated?

2. What trade-offs exist when fast-growing and slow-growing data types use local state and operation synchronization models?

3. What is the impact of multiple nodes missing update notifications, due to disconnection and reconnection events (*churn*)?.

4. What is the impact of multiple nodes having late entry in the activity?

5. How does the P/S-CRDTs model in Thyme compare to the dissemination of updates in AntidoteDB in terms of communication volume?

5.2 Network Communication Volume

The network communication volume is one of the most relevant aspects of mobile networks, given that for these types of environments, data propagation and data design must be carefully tailored to not incur any unnecessary memory overheads that may lead the system to not meet mobile user's quality standards. To this effect, in this section we aim to answer the two first questions previously declared in Section 5.1, studying the

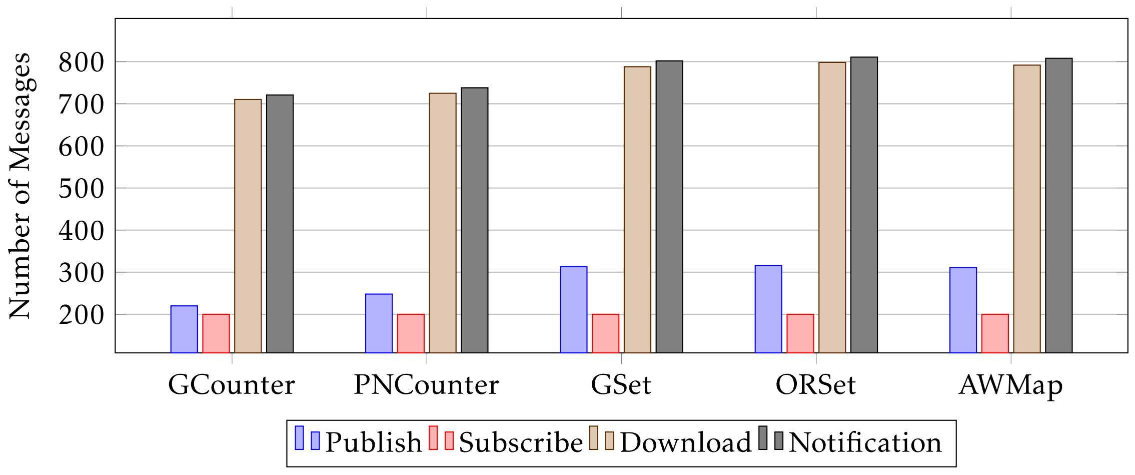


Figure 5.1: Number of messages transmitted throughout the entirety of collaborative activities with 100 nodes.

volume created in collaborative simulations using the P/S-CRDTs model and observing the behavior of the developed synchronization models for different data types.

The following experiments were conducted using simulations of 25, 50, 75 and 100 nodes with no disconnection and late entry events, as the aim is to inspect how the volume created by the collaborative interaction scales with an increasing number of concurrent users. Hence, to observe and compare the memory usage of the different CRDTs implemented with the P/S-CRDTs model in Thyme, each simulation shares the same number of local modifications per node. By keeping the number of updates per node and increasing the number of nodes per simulation, it is possible to gather how comparatively scalable each synchronization type is.

For a collaborative simulation, each node must begin by subscribing to the initial CRDT object. Once a notification is received for the CRDT publication, the node may download the object and immediately trigger the update subscription. Thus, every node realizes two subscription operations throughout the entirety of the interaction. As for publications, at the beginning of the interaction an *author* node is responsible for creating and sharing the shared object trough a publication operation. Subsequently, all other publication operations in the interaction are of update publication type. Hence, each publication apart from the initial CRDT sharing action, are all carrying updates. Since nodes use the developed *save* mechanism to indicate when they are ready to share their modifications, it means that each update publication is a direct consequence of a node triggering the save action.

As for download operations, similarly to the subscriptions, every download initially downloads the CRDT object and subsequently all other download operations are made to retrieve updates for said CRDT. Additionally, once updates arrive at the broker, its the brokers responsibility to notify its subscribers of new updates, enabling them to retrieve them as they are notified. Thus, notification messages are the most prominent type of messages seen by the network throughout the interaction, since for each update download

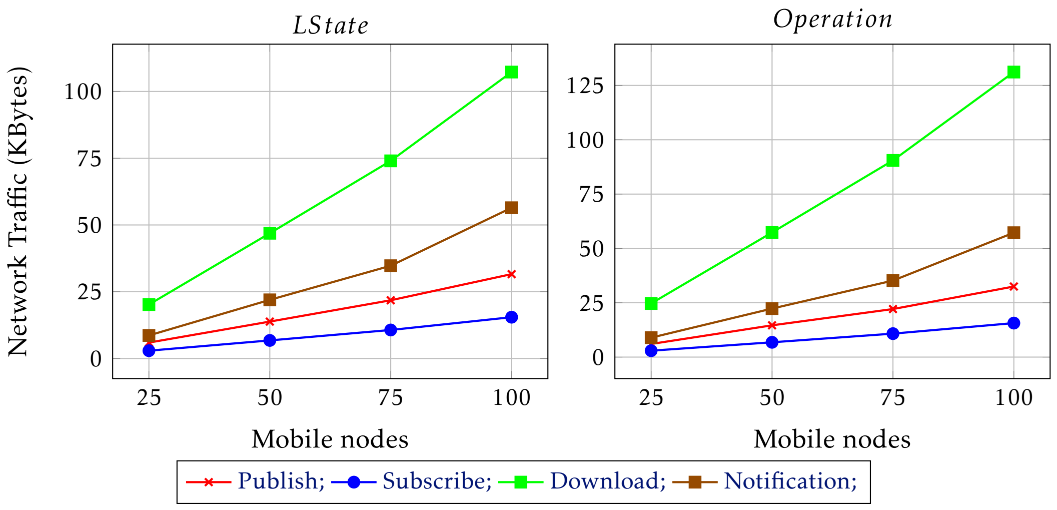


Figure 5.2: Network traffic usage per message type within a LState and Operation GCounter simulations.

by a node, first a notification must be delivered. As one may observe in Fig. 5.1, for simulations for each data type with 100 mobile nodes, the most seen transmitted message are of notification type, followed by downloads, publications, and lastly subscriptions, having always a fixed number of 200 subscription messages seen across all data type simulation with 100 nodes, since each node subscribes twice throughout the interaction. It should be noted that in Fig. 5.1, there is no synchronization model differentiation and only the data types are discriminated since for every simulation for a given data type and a number of nodes, the number of publications, subscriptions, notification, and download messages are the same. Thus the same number of messages is seen for the simulation of 100 nodes with LState and Operation GCounter simulations, however, the data size of the messages will vary with the synchronization model, as we shall see.

In Fig. 5.2 it is shown the total network traffic accomplished by each message type and in Fig. 5.3 the number of respective messages, during a collaborative simulation with 100 nodes using a GCounter object as the shared data, without churn or late entry events. According to the number of messages and content, one can assume that the download operations will represent the major percentage of traffic, as shown in Fig. 5.2. Additionally, downloads can either carry the update content or the full object, thus meaning that messages of this type are the biggest in size compared to all other message types, averaging 203 bytes per update for a GCounter update. On the other side of the spectrum, the subscriptions are shown as representing the lowest percentage of traffic not only because of their quantity but also because each subscription only carries a topic expression and two timestamps, the start and expiration time of a subscription. Hence, subscription messages carry lesser size when compared to publications that carry more information in the form of metadata, such as the address of item owner, timestamp of publication, topic, description, and several other optional information. In addition, notification messages

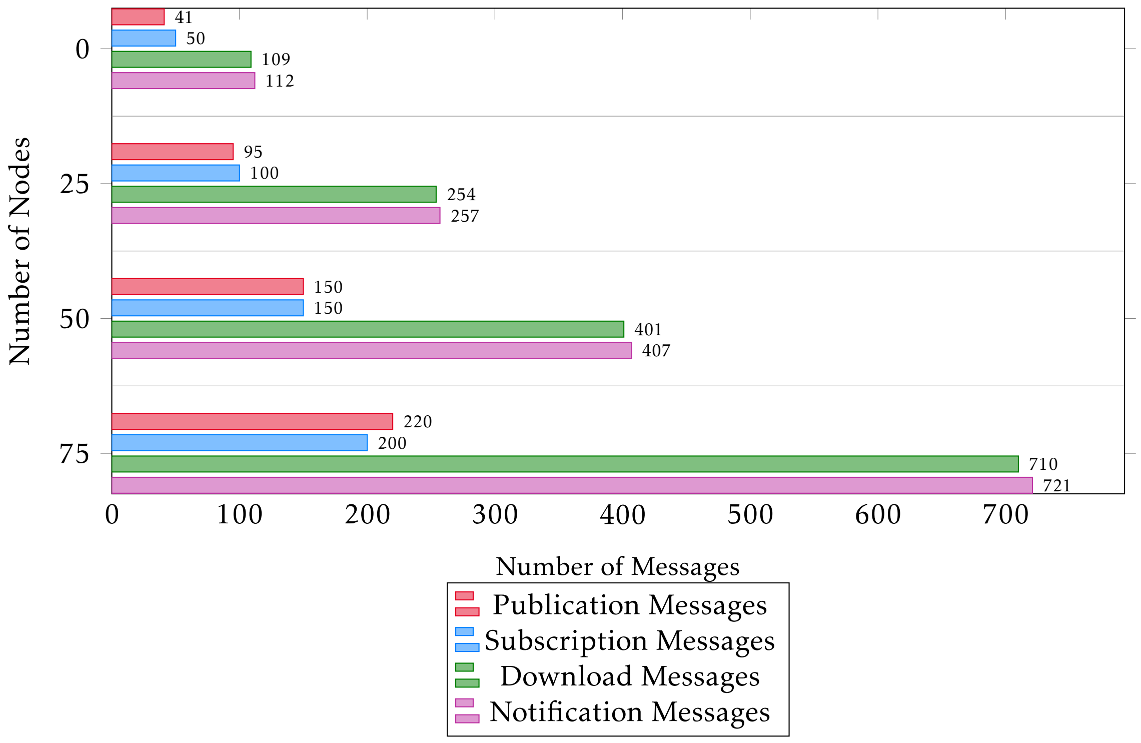


Figure 5.3: Number of messages per type within LState and Operation GCounter simulations using 25-50-75-100 nodes.

are the most prominent type of message in the system, as they are sent by the broker once updates and objects are published, however notification messages are not as heavy sized as downloads, thus not surpassing the total network traffic caused by the download requests. Furthermore, although notification messages are superior in quantity, the same cannot be said about their message size, as these messages only carry metadata indicating the object identification and address the of object's location (node), so that other nodes may retrieve updates by downloading from the address.

Moreover, when comparing the synchronization models in Fig. 5.2, although similar in respect to how network traffic is composed in both models, a slight increase is noticeable due to the total download's volume. This is a consequence of how the update content is designed for each of the models, bringing us to the second question of how each synchronization model behaves with slow and fast-growing data types. Thus, to compare P/S-CRDTs using LState and Operation synchronization we use the next two graphs where we isolate fast and slow-growing data types.

In Fig. 5.4 it is shown the comparison only between the developed counter data types since all share the fact that they all work over an integer value, meaning the data size of the state and respective updates won't be as significant as other data types that severely grow over time, such as sets. At first glance, one of the most notable similarities is that for both LState GCounter and PNCounter objects, the update data size is inferior when compared to the operation based counterparts, resulting in overall inferior memory usage

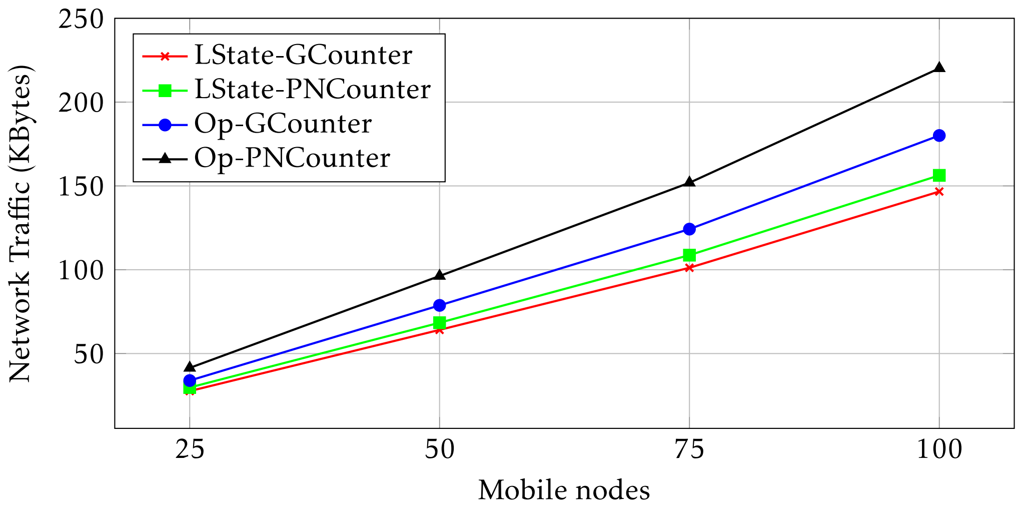


Figure 5.4: Network traffic for LState and Operation counter data types.

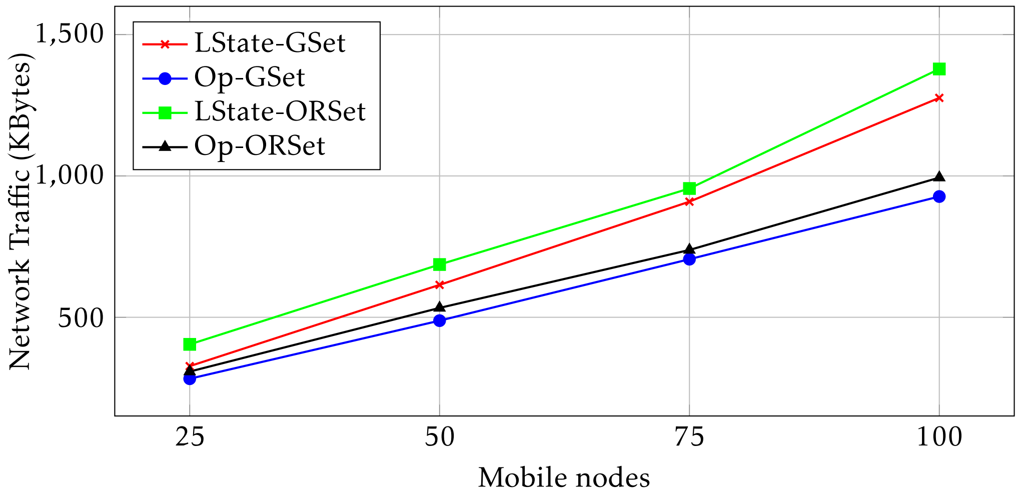


Figure 5.5: Network traffic for LState and Operation set data types.

for the whole activity. This is due to the fact that the design of these CRDTs, require the update's content to only include the local modifications made by the node, hence what is shared among GCounters is only the pair $\langle nIncs, ownerId \rangle$ and for PNCounters the tuple $\langle \langle nIncs, nDecs \rangle, ownerId \rangle$. Comparatively to the update objects of operation based CRDTs, where a queue of operations is contained since last save operation, each operation holds the arguments and operation identifier, as well as a timestamp of when the operation was locally executed, to preserve causal order. Thus it makes sense that operation based updates have more data size on average than state based updates for counter data types.

In Fig. 5.5, it is possible to observe that contrarily to the previous comparison among counter data types, here the operation based updates don't apply as much network load as the state based updates. This can be explained due to the fact that contrarily to counters, set data structure increase significantly as nodes share their updates throughout the interaction, inserting and removing elements. A closer look at the displayed graph, one can also realize the ORSet CRDTs have a bigger data size compared to GSet CRDTs, since the ORSet implementation makes use of two data structures, one for elements added and other for removed elements, whereas for the GSet only one set of elements is used to represent the state of the CRDT, since GSet's interface only allows the insertion of new elements.

Furthermore, operation based objects share updates that contain the local operations made since last save action within a queue object. Thus, each operation within the update comes with the identification of the operation to be executed on the receiving end and the arguments to be used for the operation. Hence, operation based objects have a lighter load compared to the LState based approach, that although carrying only the local updates, it still is considerably bigger in size, since for sets the local state is a set containing only the inserted elements at source for GSet CRDTs and an additional set for the removed elements at source for ORSet CRDTs. Thus, as more insertion and removal operations are made, the more elements the sets within the LState's update will hold, resulting in bigger updates for the LState based approach, whilst the operation based approach contains only the operations made, that as a whole make for a lighter update in terms of data size.

Ultimately, we can conclude that for slow-growing data objects, such as counters, the LState synchronization model outperforms the operation model resulting in overall less memory volume used to propagate updates. Whilst in the other hand, for fast-growing data objects, such as sets, the operation based approach is preferable.

Moreover, one possible optimization to reduce the communication volume could be added, by using one of Thyme's features, which is the possibility of sending the update directly into the description field of a notification, if the update content is under a certain size. Thus, when a node receives a notification of a newly published update, the update content is already within the notification and node is no longer required to request a download for the update. Preferably, this feature could be used to disseminate small sized updates, as including big updates in the notifications is not a scalable approach.

5.3 Churn communication impact

As one of the possible volatile behaviors in mobile networks is involuntary disconnection, which promptly causes users to lose updates whilst out of the interaction, it is relevant to study the impact of said events. The P/S-CRDTs model specifies as one of its requirements that nodes must be able to converge, even in the occurrence of update notification loss, by either retrieving the lost updates if still in broker's storage or by downloading an integral copy of the CRDT object from a node within the interaction upon reconnection. Hence to study the developed mechanisms to overcome update loss, the conducted experiments vary the percentage of nodes that lose updates within a simulation.

The simulations presented to study the impact of churn, have 100 nodes and vary the number of nodes who lose updates, starting with 0 nodes losing updates as a baseline result and subsequently increasing to 25, 50 and 75 nodes that suffer disconnection.

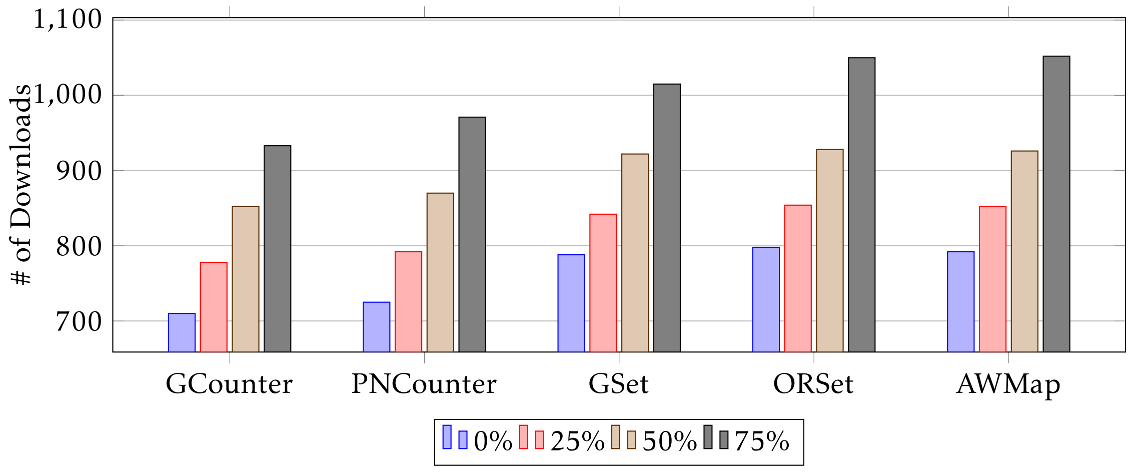


Figure 5.6: Number of download messages transmitted in operation-based synchronization simulations with 100 nodes, where the percentage number of nodes who lose updates vary.

Churn Rate	Total Downloads	Download Object	Download Update	Download Lost Update
0%	788	99(13%)	689(87%)	0(0%)
25%	842	129(15%)	665(79%)	48(6%)
50%	922	183(20%)	646(70%)	93(10%)
75%	1015	214(21%)	644(63%)	157(16%)

Table 5.1: Number of download operations made for operation GSet simulations with 100 nodes with introduced churn events, showcasing the number of downloads by type, between object, update, and lost notification downloads.

Churn Rate	Total Downloads	Download Object	Download Update	Download Lost Update
0%	784	99(13%)	685(87%)	0(0%)
25%	839	99(12%)	677(80%)	63(8%)
50%	916	99(11%)	711(77%)	106(12%)
75%	1008	99(10%)	730(72%)	179(18%)

Table 5.2: Number of download operations made for lState GSet simulations with 100 nodes with introduced churn events, showcasing the number of downloads by type, between object, update, and lost notification downloads.

Whenever a node disconnects in these simulations he always later reconnects, and he may lose between 5 to 20 updates. By varying the number of updates a node loses, we aim to represent a more realistic scenario where nodes may be able to recover only the lost updates if still present at the broker's storage or requires a full object copy if otherwise.

In Fig. B.2 it is possible to analyze the number of download messages generated for collaborative activities using each data type. Naturally, as disconnections happen at larger scales, the number of download requests increases. This is due to nodes that receive updates and promptly verify that the hash of the last notification within the

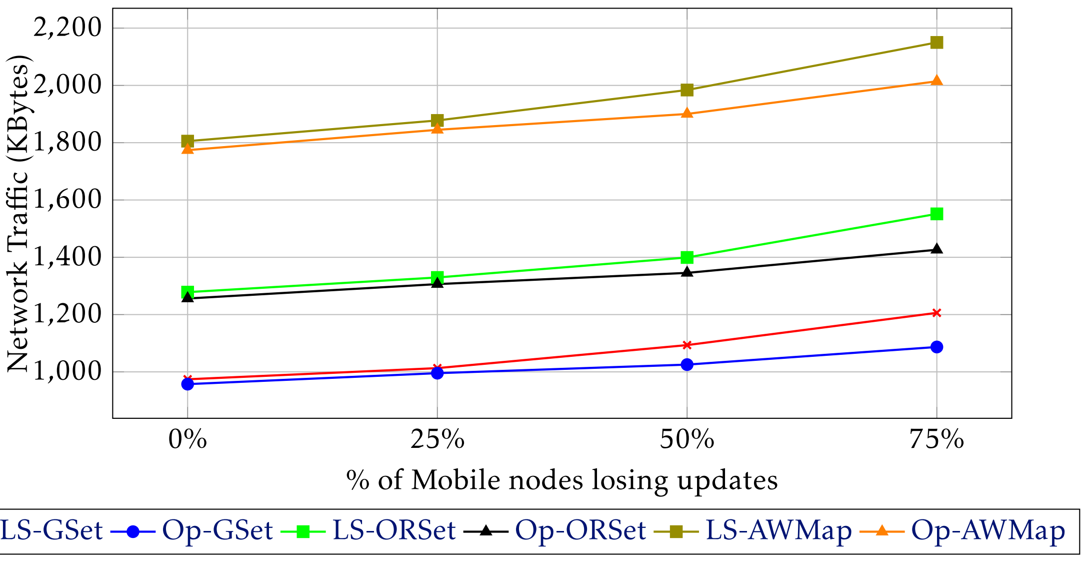


Figure 5.7: Network traffic for LState and Operation data types with the increase of nodes who lose updates.

newly received notification's metadata, does not match the last seen notification hash by the node. Meaning that one or more updates were lost whilst out of the interaction. By using the newly received hash and the last seen hash, it is possible to request lost notifications within the chain link created between hashes, however if one of the hashes is no longer associated with any of the stored notifications at the broker, then the full copy of the global object's state is required. For implementation purposes, when working with operation-based objects, the broker's storage can keep up to ten updates, these being order by publication timestamp. If contrarily, the broker was designed to store all update notifications made throughout the interaction in a log, the system would not scale, hence limiting the number of notifications stored at the broker. Hence, in Tab. 5.1 we observe the ratio between the download requests for complete object copies and updates. It should be noted that for a simulation with 100 nodes, there will always be 99 download object requests, as each node except the CRDT author are required to subscribe to the CRDT object and subsequently download it.

Thus, as the number of nodes losing updates increases and more download operations are issued to retrieve lost updates, the more network traffic is generated. In Table 5.1, we can observe in greater detail the multiple cases where download messages are issued. Downloads can be issued for full object copies, regular updates disseminated and for detected missing updates. Thus, as the number of users experiencing churn events increases, the number downloads for missing updates increases. However, by comparing with Table 5.2, that uses the same GSet data type but with the lState synchronization model, a peculiar case happens. As lState objects only require the last update containing the another node's entire local state to converge, lState then don't need to retrieve the full copy of the object, as the broker will always have the last received update. On the other hand, as operation-based objects require all updates made throughout the interaction, be applied to their local state to converge, then if updates are lost and not retrievable, a full copy of the object is need. Although, operation-based objects make use of both lost update retrieval mechanisms, as operation objects and respective updates are considerably smaller in size, the network traffic of for operation-based data types still outperforms state-based data types, in the cases where said data types grow considerable size as updates are applied throughout the time of a collaborative activity.

In Fig. 5.7 for simulations of 100 nodes, one can see a gradual increase in traffic for each of the presented data types, noticing once again a slight increase of traffic when the local state synchronization model is applied to fast-growing data types. Hence, as a consequence of being able to recuperate from disconnections by retrieving lost notifications, P/S-CRDTs add more volume to the network traffic, proportionally to the downloading of updates and full copies of the global shared state.

5.4 Node late entry communication impact

Late entry is one more possible scenario of dynamic environments, where a user may join a collaborative activity at a later time, meaning that the user must either apply all updates previously done before arrival or get a full copy of the global shared state. Much like when updates are lost, when a node has a late entry an increase of network traffic is to be expected due to the increase of download requests.

To observe the impact of late entry on network traffic, simulations were created with 100 nodes, starting with a baseline of 0 nodes having a late entry, increasing the number of nodes who have a late entry by 25, 50 and 75. Nodes that join the interaction at the beginning, must go through the process of acquiring the shared object and only then they may start to modify and share the modifications. On the other hand, nodes that join at a later time, must acquire a copy of the CRDT object and realize a subscription that encapsulates the time of entry and indefinite future time. In doing so a node requests only updates shared after his arrival time.

In Table 5.3, we observe very similarly as to when updates are lost by churn events, in cases of late entry, the number of download still increases as users who join may converge

Late Entry Rate	Total Downloads	Download Object	Download Update	Download Lost Update
0%	788	99(13%)	689(87%)	0(0%)
25%	857	142(17%)	676(79%)	39(4%)
50%	937	201(22%)	644(69%)	92(9%)
75%	1058	227(22%)	686(64%)	145(14%)

Table 5.3: Number of download operations made for operation GSet simulations with 100 nodes with introduced late entry events, showcasing the ratio between object and update downloads.

Late Entry Rate	Total Downloads	Download Object	Download Update	Download Lost Update
0%	784	99(13%)	685(87%)	0(0%)
25%	849	99(12%)	705(83%)	45(5%)
50%	930	99(11%)	723(78%)	108(11%)
75%	1038	99(10%)	775(74%)	164(16%)

Table 5.4: Number of download operations made for lState GSet simulations with 100 nodes with introduced late entry events, showcasing the ratio between object and update downloads.

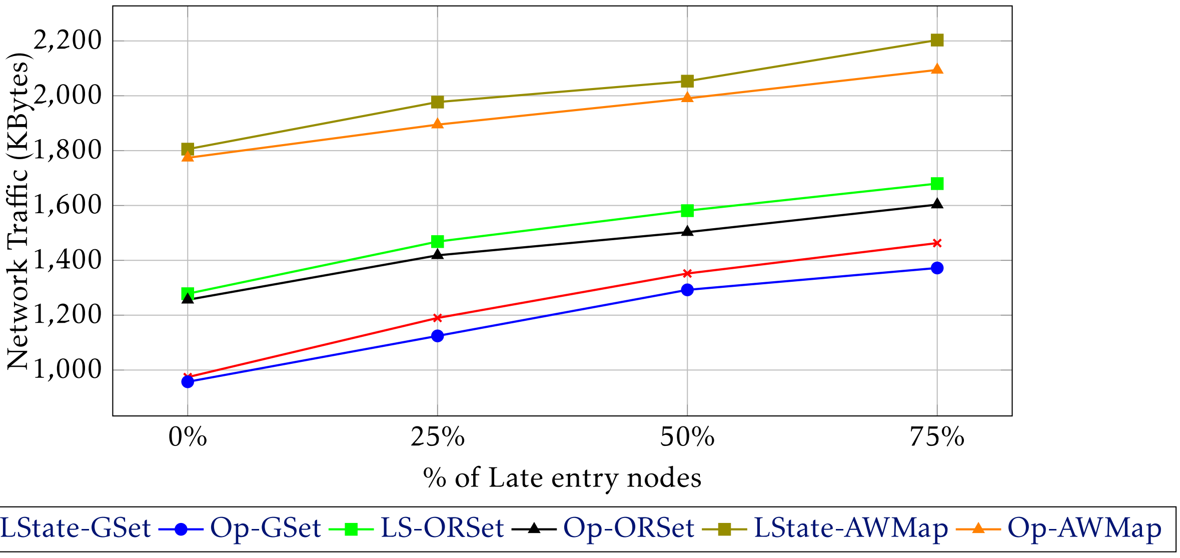


Figure 5.8: Network traffic for LState and Operation data types with the increase of late entry nodes.

by downloading the required updates or obtain a full copy of the state. By comparison, in Table 5.4, where the same GSet data type is used in simulations of 100 nodes, it is observable the previously explained behavior of downloading only missing updates, since lState objects only require the last update to converge, however for operation-based objects a mix of behaviors is possible as to take advantage of the broker's storage.

In Fig. 5.8 it is shown the network traffic as the number of late entry nodes increases. Once the number of late entry events increases it is observable an increase of memory volume input, this being a direct consequence of nodes requiring a copy of the global shared state and subsequent future update downloads. Although much like when nodes experience update loss in terms of network traffic results, in this scenario a slight increase is seen as a result of the increased number download operations for the full global state instead of single updates.

Algorithm 7 Δ-CRDT replication, taken from [38].

```
upon $onVersionVector$(VV, REPLICA) do
    $\Delta \leftarrow$ getDelta(vv)
        if $\Delta$.size $< 0$
            REPLICA.send($\Delta$)
        optionally do ($push\ model$)
            if vv after self.versionVector
                REPLICA.send(self.versionVector)

upon $delta(\Delta)$ do
    self.state.applyDelta($\Delta$)
    self.versionVector.update($\Delta$)

periodically do ($pull\ model$)
    $r \leftarrow$ randomReplica()
    r.send(self.versionVector)

on local operation do ($push\ model$)
    $r \leftarrow$ randomReplica()
    r.send(self.versionVector)
```

5.5 Big delta CRDTs (Δ-CRDTs)

As a competitor to the developed solution, we chose the Δ-CRDTs model, as they represent an admissible approach for dynamic environments. Thus, for evaluation purposes, the Δ-CRDTs concept was adapted to THYME, where multiple traces were used to compare results against identical data types using the P/S-CRDTs model.

Δ-CRDTs belong to the family of delta-based CRDTs, being a new better iteration of the original concept. Whilst δ-CRDTs propagate a delta containing all modifications made since last communication step, Δ-CRDTs are able to compute the minimal delta that needs to be propagated to another node, only containing the missing portion of the state that target nodes require to converge. To compute the minimal delta, internal CRDT metadata is used, characterizing the causal context between nodes, most commonly implemented by using a vector clock [38].

To replicate, the minimal delta is first computed by sending a *getDelta* request containing the causal context (version vector) of the replica which initiated the communication. Once a replica receives a *getDelta* request with the causal context, the minimal delta is computed by comparing the received version vector with its own, checking for missing updates. Once computed the delta is to be shipped back to the replica who initiated the communication (Algorithm .7).

Δ-CRDTs use a garbage collection mechanism to remove old metadata associated with all operations that happened before a given point in time. Thus, in cases where the local replica's garbage collection point is further ahead in time than the sender's causal context, Δ-CRDTs fall back to a state-based CRDT behavior, requiring the whole state and causal context shipped and applied at the replica [38].

In order to adapt Δ-CRDTs to Thyme, a meeting with the author of the model was arranged, where an exchange of implementation advices was made. As Δ-CRDTs were initially implemented by the author using Legion, a peer-to-peer JavaScript framework, some concerns arose when contemplating the adaptation of Δ-CRDTs to a publish/subscribe system (Thyme). To this extent, additional implementation details were advised by the author, as to make possible the Δ-CRDT model comply with the communication pattern provided by publish/subscribe systems. As Thyme does not expose the network topology, sending *getDelta* requests to a random node as specified in Algorithm 7, was not initially possible.

To circumnavigate the shortcoming, the developed adaptation uses an additional publication and subscription round to a specific topic, designated to store and share the addresses of all nodes participating in the collaborative network. Hence each user who joins the activity, must first publish his address and subscribe to receive other nodes addresses who join the activity. Once the addresses are made available, nodes may start to request the minimal delta.

For evaluation purposes, the Δ-CRDT model was applied to two data types, the Grows-only Counter and the Add-wins Map. The chosen data types represent different sides of the spectrum when considering size growth in time. Whilst a counter object maintains a considerable small size throughout the duration of a collaborative activity, a map object, on the other hand, will most likely grow at a faster rate, as users issue updates. Hence, by having these two distinct data types implemented in both Δ-CRDT and P/S-CRDTs we aim to compare the volume of communication placed on the network created by the different update dissemination techniques.

5.5.1 P/S-CRDTs versus Δ-CRDTs

To evaluate Δ-CRDTs against P/S-CRDTs, two data types were implemented, the GCounter and AWMap. As these data types belong to opposite sides of the spectrum in terms of memory size, we used them to study if the Δ-CRDTs obtain better communication volume against the developed LState and Operation P/S-CRDTs.

In Fig. 5.9 we compare the total memory volume, including publications, subscriptions and download traffic. Similarly to the operation synchronization model, the delta obtains better memory values for fast-growing data types. This is due the fact that for Δ-CRDTs, the update content corresponds only to the missing modifications (delta mutator) missing in target nodes. Thus, for slow-growing data types such as the GCounter, the LState wins over since it always only carries the local state of the counter, which is an integer, whilst the delta carries all the calculated missing modifications for every node. The same logic cannot be applied to the AWMap since the LState model contains a full local state of the map which is considerably bigger in size and grows faster, hence delta

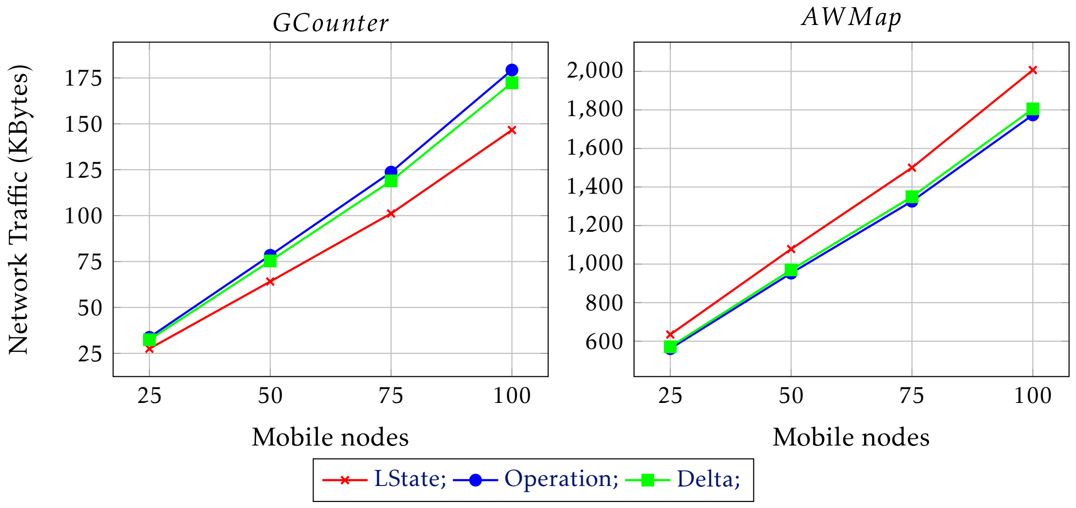


Figure 5.9: Memory volume within LState, Operation, and Delta GCounter and AWMap simulations.

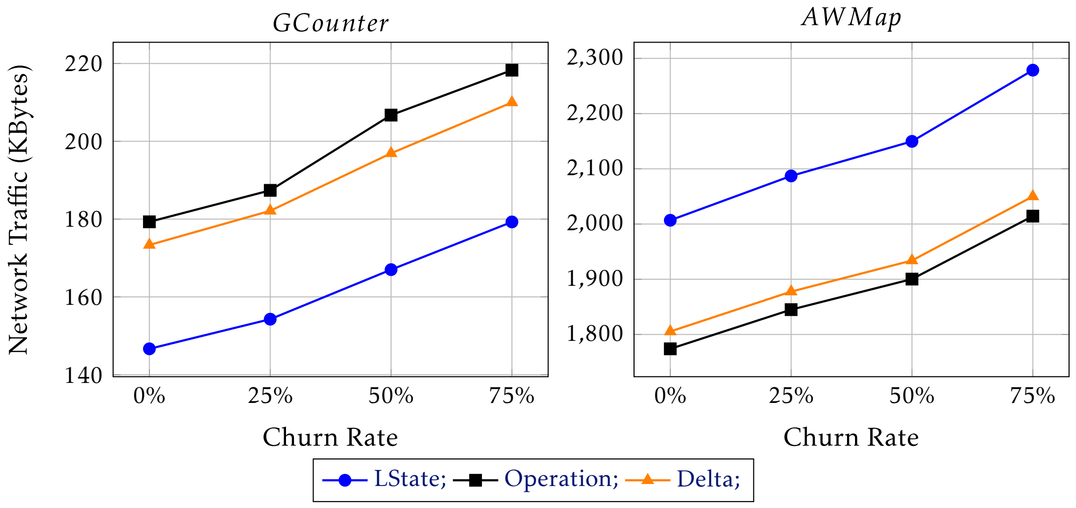


Figure 5.10: Traffic volume within LState, Operation, and Delta GCounter and AWMap simulations, with the introduction of churn events.

Δ-CRDTs win by sending the missing portion.

For the experiment conducted in Fig. 5.10, both Δ-CRDTs and P/S-CRDTs have the same threshold of having the last ten notifications stored in the broker, thus whenever the update is not recoverable the full copy of the global state is made. Hence, once again an increase is noticeable due to the download operations to recover the full global state, as the number of users losing updates increases.

Moreover, although a very small discrepancy between the results obtained between Δ-CRDTs and P/S-CRDTs is visible, it can be attributed to the Δ-CRDTs implementation restrictions when using a publish/subscribe system requiring an additional subscription message per node and additional communication step to compute deltas whenever a node

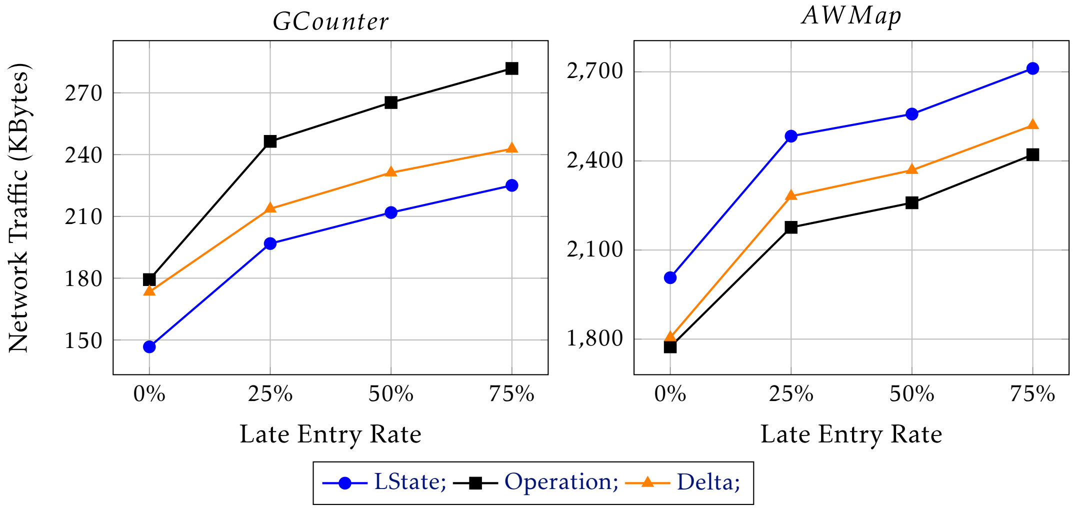


Figure 5.11: Traffic volume within LState, Operation, and Delta GCounter and AWMap simulations, with the introduction of churn events.

is contacted to retrieve the delta. Furthermore, in Fig. 5.11 where late entry events were introduced, a similar volume of network traffic is added, as nodes request downloads for the full copy of the state, noticing an abrupt increase of network traffic as soon as nodes start joining later.

5.6 Todo List Application

As a use case and benchmark tool, a simple Java collaborative todo list application was made using P/S-CRDTs on Thyme. The application was based on an existent open-source Java AntidoteDB todo list implementation, using the AntidoteDB Java Client. The selection of this application was due to the fact that it functions by sharing a Map CRDT among its users and that its done based on a distributed database, therefore giving us the opportunity to compare the P/S-CRDTs model to a read/write transaction model.

Thus, by evaluating an implementation using P/S-CRDTs on Thyme versus an implementation based on AntidoteDB, we aim to compare the communication cost of both approaches. To do so, when the adaptation took place, special care was given to maintain the implementation almost true to the original. Hence, one of the crucial parts of this adaptation, was the use of the same map CRDT, that gives preference to update operations over delete operations (add wins semantics). To better understand and compare both approaches, its best to first comprehend the AntidoteDB architecture (Section.5.6.1) and subsequently the application's implementation details (Section.5.6.2).

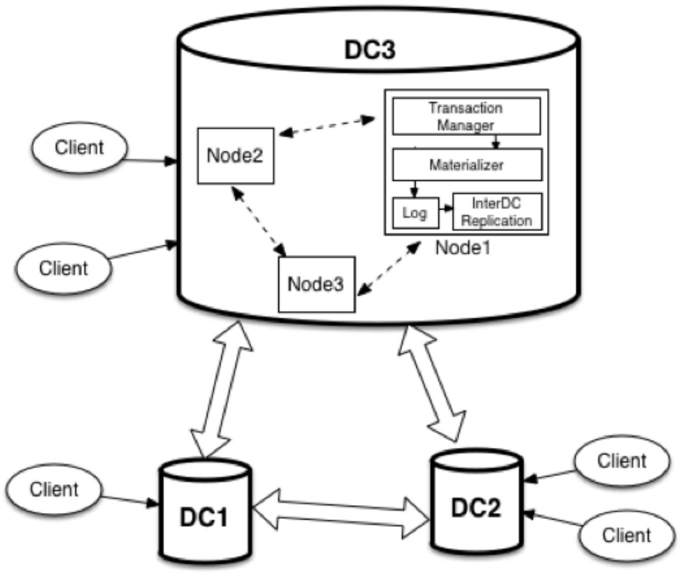


Figure 5.12: AntidoteDB Architecture. Taken from [2].

5.6.1 AntidoteDB Architecture

AntidoteDB is a distributed database natively developed in Erlang and employing Cure, a highly scalable protocol for update replication from a cluster to another. Updates operations replicate asynchronously, as means to provide guarantees of high availability under network partitions [2].

To guarantee scalability, data is sharded among a cluster's replicas using consistent hashing techniques and a ring organization. For transactions that read/write to multiple objects, contact is made only to servers that have the objects accessed by the transaction (master-less design). Thus enabling the serving of requests even when some servers fail. By using Cure, AntidoteDB is able to provide causal consistency, hence guaranteeing that related events are made accessible corresponding to their order of occurrence, whilst unrelated events (events done concurrently) may be in a different order in different replicas.

The AntidoteDB architecture (Figure 5.12) assures the correct propagation of operations to different replicas, whilst at the same time assuring the asynchronous propagation of said operations to remote data centers. To this effect, each AntidoteDB node consists of the following components [2]:

- **Transaction Manager:** This component implements the transaction protocol, being responsible for receiving client's requests, subsequent request execution, transaction coordination and replying to said client requests. Operations contact the materializer component to generate the snapshot of the objects present in other nodes.
- **Materializer:** This component generates and caches the object versions requested

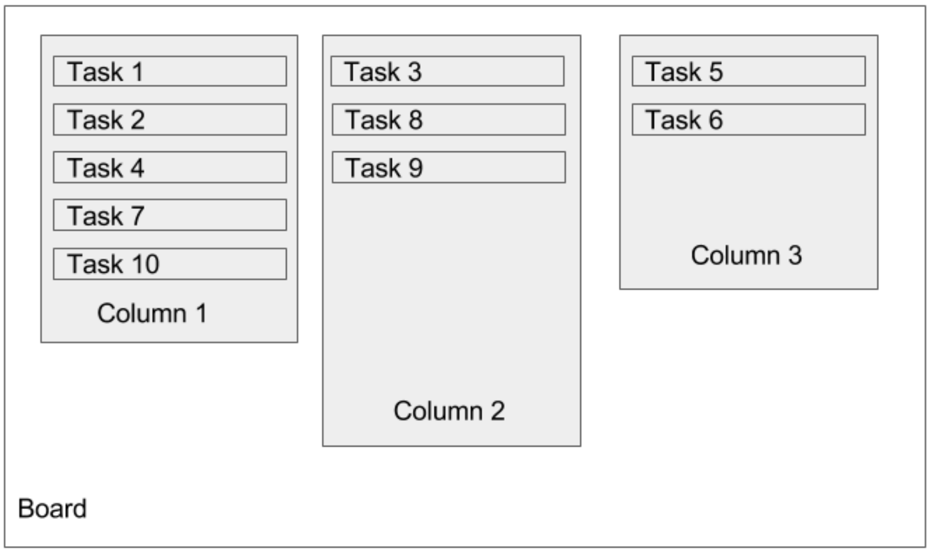


Figure 5.13: Representation of the application's utility. Taken from [17].

by clients (snapshots). It connects the log and the transaction manager components. Additionally, to avoid system degradation over time, the materializer incorporates pruning mechanisms.

- **Log:** This component employs a log-based persistent layer, where updates are stored in a log, this one being persisted to disk for durability. Additionally, this component maintains a cache layer to provide faster accesses to the log.

- **InterDC Replication:** This component is responsible for retrieving updates within the log and subsequently propagation to other data centers. Furthermore, communication between data centers is done partition-wise.

5.6.2 Collaborative Activity

The collaborative activity to be conducted consists of a collaborative editing application, where a shared todo list is concurrently updated by multiple nodes[17]. Within the application, it is possible to identify three main editable objects, boards, columns and tasks (Figure. 5.13). Boards retain a set of columns and are identified by a given string expression, likewise columns are identified by a given string expression and retain a set of task. Conversely, tasks are the basic unit of the application, being only identified by their description.

A usage scenario where the todo list is well suited is for a supermarket list of must-buy items. For instance, a user may begin by creating a board identified as "*supermarket item list*". Once created, the board is made available for all users to edit, being able to rename the board, query the content, add and remove columns. The next logical step is to create a column to contain the tasks, in this case the items to buy, hence one may create a column identified as "*dairy products*"and subsequently add dairy products as tasks e.g. "*milk*",

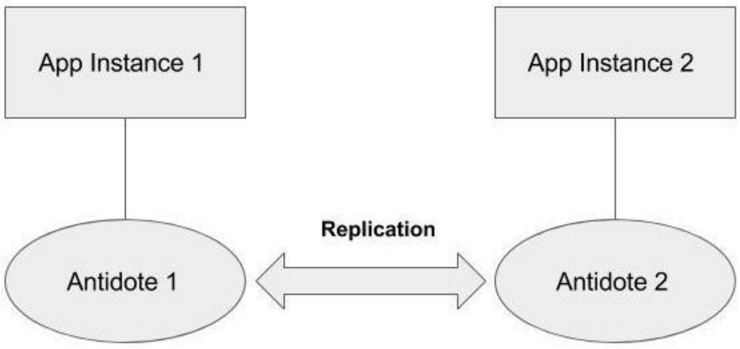


Figure 5.14: AntidoteDB cluster with two replicas. Taken from [17].

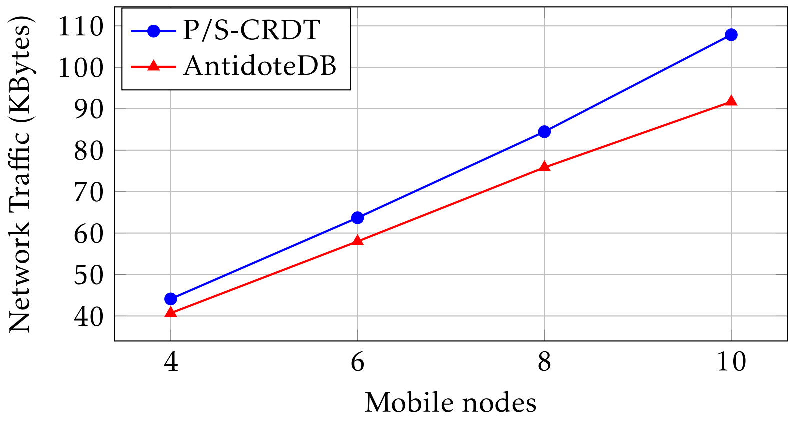


Figure 5.15: Network traffic for P/S-CRDTs and AntidoteDB simulations with no disconnection and late entry events.

"*yogurt*", "*cheese*". On a similar note, users may rename the column, query the content, and move, add and remove tasks.

In order to test the AntidoteDB implementation versus the P/S-CRDTs on Thyme, the same principle of using trace executed scenarios were used. In Antidote's implementation, to simulate users, it was used an AntidoteDB node instance per user. Each instance is launched in the form of a Docker container, where application instances run on top of each AntidoteDB client instance (Figure. 5.14). In addition to the core application functionalities, it is possible to simulate a real world network partition between replicas by disconnecting the docker network between instances and conversely reconnect. Hence, being able to compare the impact of *churn* in the communication volume of both implementations.

5.6.3 P/S-CRDTs versus AntidoteDB

In order to fairly compare P/S-CRDTs against AntidoteDB, the experiments have been equally applied to both P/S-CRDTs on Thyme and AntidoteDB, each simulation having

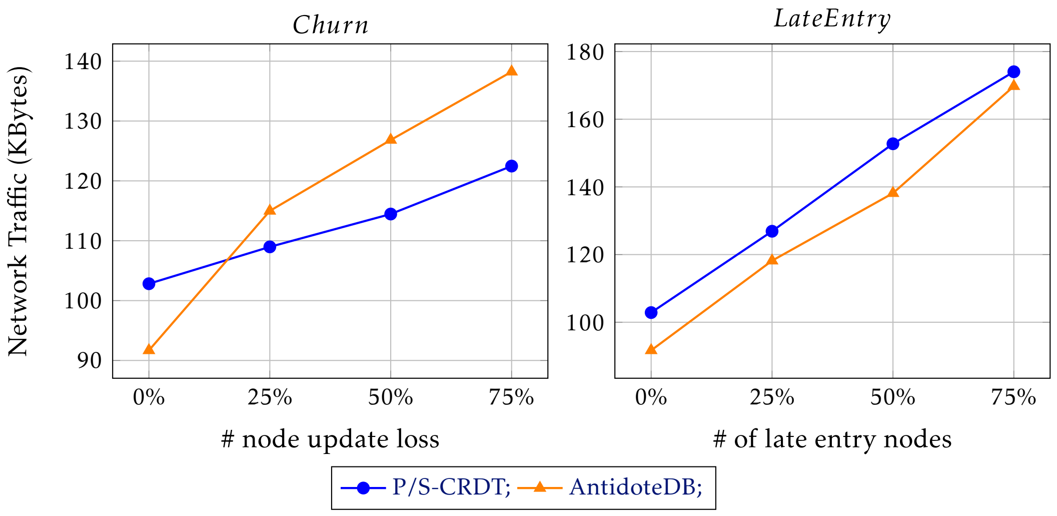


Figure 5.16: Memory volume for the AntidoteDB and P/S-CRDTs application, for simulations with loss of updates and late entry.

the same number of nodes and operations executed per node. However, one of the shortcomings faced during developed reduced the number of maximum concurrent nodes to 10. Said shortcoming comes from the deployment of AntidoteDB client instances since the provided application implementation provided by AntidoteDB uses docker containers to launch and create the network between the multiple instances of AntidoteDB Client. Hence a recurring bug happened whenever more than ten container instances were active, launching an error when an update transaction was made for the todo list, making impossible transactions reach other AntidoteDB instances, causing nodes to never receive updates. Thus, for the following results, we used 4, 6, 8 and 10 nodes.

To gather the total network traffic in the AntidoteDB application, each transaction containing updates was measured and accumulated to make the total sum size of data disseminated within a simulation. Comparatively, in Thyme we accumulate the total traffic of publications, subscriptions, and downloads. Across all simulations, each node realizes 10 operations, creating and deleting boards, columns, and tasks.

In Fig. 5.15 we present the cumulative network traffic results for both P/S-CRDTs and AntidoteDB application simulations. Both implementations of the application, make use of a Map CRDT with add wins semantics. For the P/S-CRDTs, the chosen synchronization model for the application is the operation-based model, similarly to the update dissemination realized in AntidoteDB, which classifies each update operation as a unit to be shared within a write transaction. A close observation of Fig. 5.15, shows that the P/S-CRDTs implementation adds more communication volume than Antidote's implementation. Since each operation for the P/S-CRDTs on Thyme holds multiple information such as timestamp, operation identifier ("addBoard", "addColumn", "addTask", etc...) and needed arguments, it results in bigger sized update operation compared to AntidoteDB.

In order to gather the behavior of both implementations in scenarios of disconnection and late entry, the following experiments in Fig. 5.16 were conducted with the maximum possible number of nodes, which is 10. Nodes that disconnect lose from 5 to 15, in order to test both scenarios where the update is retrieved or a full copy of the global state is mandatory, depending on what is available at the broker's storage at the time.

Initially, as seen before in the network traffic analysis, AntidoteDB presents itself with a smaller size throughout a simulation with no disconnections, however a soon as 25% of the users in the simulation start to lose updates AntidoteDB obtains a significant increase in communication volume. As users disconnect in Antidote's application, write transactions that are not able to be sent are queued to allow disconnected work, and once the connection is made available again a snapshot lookup is made to retrieve all transactions not received. Since on the other hand, the P/S-CRDTs recover by retrieving the immediate missing updates or the full global state, which in certain cases may be more efficient to make a full copy than outright retrieve every single update, results in a lesser memory volume for the P/S-CRDTs. Conversely, in a scenario where users have late entry AntidoteDB displays less memory volume, as nodes who join with a new AntidoteDB Client instance automatically execute a snapshot copy from another instance composing the collaborative network. In turn, P/S-CRDTs have a higher volume due to the full copy of the global local state and subsequent update subscription immediately made at the time of entry in the activity.

Chapter 6

Conclusion

6.1 Conclusion

Mobile users are prone to the volatile behaviors of mobile networks, where disconnections are a recurring incident that leads to data inconsistencies, due to the loss of updates. Users who are out of the network and later re-enter the interaction may have an outdated state, effectively creating a state divergence if not resolved. Furthermore, collaborative applications also incur with concurrent work, causing conflicts that require resolution. If left unresolved, inconsistencies are created since multiple nodes may apply updates in a non-deterministic way (different orders), making users permanently see different state values.

As one of the more widely used approaches to solve the problem previously described, CRDTs are presented as a great solution by providing automatic conflict resolution at low latency costs. However, a problem residing in the application of CRDTs is that usually when an update has to be shared, the targets to receive such update must be known to the sender. This requirement is inconceivable for dynamic environments, such as mobile networks, since the active knowledge for a system where nodes may leave and enter willingly and unwillingly, is very taxing and not a scalable approach.

In this thesis, we proposed a solution that provides collaborative mobile networks who want to employ CRDTs, a way of sharing and receiving updates between users, whilst being capable of removing the need for the knowledge of active users in the network. Thus, we introduced the P/S-CRDTs model, consisting on the adoption of the generic CRDT model coupled with the publish/subscribe interaction scheme, as the medium for update dissemination. The publish/subscribe scheme was chosen from the get-go as the updates dissemination medium, since publish/subscribe systems decouple the workflow between producers and consumers of such updates, not requiring the knowledge of each other to publish and subscribe to content and additionally allowing for disconnected work. Thus in the context of collaborative activities for mobile networks, the P/S-CRDTs model specifies all the interaction phases, where users begin to modify the shared state with local operations, to later share modifications through update objects using the publish/subscribe system for dissemination of updates. Hence, nodes in the specified model take the roles of publishers and subscribers, whereby publishing the updates to other users and subscribing to other user's updates, it is possible to create a fully collaborative network.

To ensure that all users within a collaborative activity keep a consistent state, some additional requirements are specified by the model, such as the need for ensuring that lost update notifications are detectable and retrievable. By employing such requirements, then it is possible to circumnavigate the possible volatile scenarios commonly found in mobile networks. Thus, by being able to retrieve missing updates or a full copy of the shared state, users are able to suffer disconnections and late entry events, whilst being able to keep a fully consistent state.

As proof of concept, we coupled the P/S-CRDTs model with Thyme a topic-based publish/subscribe system for mobile edge networks, adapting state-based and operation-based CRDTs with the P/S-CRDTs specification. Moreover, tests were conducted in a simulated environment, taking into consideration all the possible volatile scenarios, to observe the impact of disconnection/re-connection (churn) and late entry events. Furthermore, two benchmark components were implemented, the Δ-CRDTs model as a direct competitor and as a use case, a Java To-do list application originally employing AntidoteDB CRDTs and respective update dissemination mechanisms.

By evaluating the implemented P/S-CRDTs on Thyme and comparing against the benchmark components, it was possible to observe that the developed solution showed adequate network communication volume values for mobile environments, whilst being able to maintain user's consistency in volatile behaviors. When comparing the adapted P/S-CRDTs against the Δ-CRDTs, it was possible to determine that for slow-growing data types, the lState P/S-CRDTs create less communication volume than the Δ-CRDTs, however for fast-growing data types, the operation P/S-CRDTs and Δ-CRDTs results came very close. Thus for certain data types and synchronization models, P/S-CRDTs were able to outperform Δ-CRDTs, by being able to efficiently recover from disconnection events.

Furthermore, when comparing the use case application using P/S-CRDTs on Thyme against the same implementation using AntidoteDB, for scenarios of no disconnection and late entry AntidoteDB revealed lower memory traffic due to its lightweight update transaction and snapshot lookup mechanism. However, in scenarios of disconnection P/S-CRDTs outperformed AntidoteDB by recovering updates in a more efficient manner.

In conclusion, we think that the objectives of this dissertation were achieved with the development of the P/S-CRDTs model, by being capable to provide mobile users with a consistent collaborative network activity, resilient to volatile behaviors. Moreover, the P/S-CRDTs model can specify an update dissemination pattern that does not require the knowledge of whom to send updates to. When evaluating the solution, we observed that P/S-CRDTs assume acceptable memory traffic values for mobile networks, thus making it a suitable approach to dynamic environments to guarantee the eventual consistency model.

6.2 Future Work

Currently, the solution has been proven as a suitable approach to circumnavigate some of the problems that originate from dynamic environments and is currently deployed onto Thyme as proof of concept. Although functional, some refinements and new features to the system could only result in an overall better solution. Some of the possible improvements are as follows:

- To mitigate possible storage overheads, it is feasible the addition of log compaction algorithms to further improve the storage of updates, since in the context of mobile networks great care is given to the size of data transmitted and stored. In cases where CRDTs state may grow at a very fast rate, such as in set data types, sending even the local state of the node instead of the global state with all seen updates, is still a considerable data size object. Furthermore, update storage for operation based CRDTs can be greatly improved by studying the commutative properties of the operations and compacting such operations accordingly;

- The development of a more complex, practical and real use case application of the model, would permit a better overview of the P/S-CRDTs model behavior in a real context. Such as collaborative mobile games that require low latency values for its users, hence better exposing the full potential of CRDTs and their conflict resolution low latency values;

- More synchronization mechanisms could be applied, following the P/S-CRDTs specification, alongside with more data types. Some synchronization models may even showcase better performance values, such as a modified delta-based model where the additional delta computation step is removed from the communication and only the portion of the state changed since last publication is shared;

As continued work is possible out of the scope of what was already accomplished, the previous list depicts some possible remaining edges and new features that can be developed to provide and enrich the P/S-CRDTs model with better results and adaptability.

Bibliography

[1] Riak® KV. URL: http://basho.com/products/riak-kv.

[2] AntidoteDB. URL: https://antidotedb.gitbook.io/documentation.

[3] Amazon S3. URL: https://aws.amazon.com/s3.

[4] Google Docs. URL: https://www.google.com/docs.

[5] GitHub. URL: https://github.com.

[6] Bet365. URL: https://www.bet365.com.

[7] SoundCloud. URL: https://soundcloud.com.

[8] Redis. URL: https://redis.io.

[9] Dynamo. URL: https://aws.amazon.com/dynamodb.

[10] Eventuate. URL: https://github.com/soundcloud/roshi.

[11] P. Bourgon. Roshi. URL: http://rbmhtechnology.github.io/eventuate.

[12] Amazon. URL: https://www.amazon.com/.

[13] TomTom, NavCloud. URL: https://www.tomtom.com/automotive/products-services/navigation-software/.

[14] Basho Technologies, Inc. Riak for Gaming. URL: http://basho.com/posts/technical/riak-for-gaming/.

[15] Akka. URL: https://akka.io/.

[16] Grid5000. URL: https://www.grid5000.fr.

[17] AntidoteDB Todo List Application. URL: https://antidotedb.gitbook.io/documentation/tutorials/java-client.

[18] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. "Mobile Edge Computing: A Survey." In: *IEEE Internet of Things Journal* 5.1 (2018), pp. 450–465. DOI: 10.1109/JIOT.2017.2750180. URL: https://doi.org/10.1109/JIOT.2017.2750180.

[19] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient State-Based CRDTs by Delta-Mutation." In: *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*. Ed. by A. Bouajjani and H. Fauconnier. Vol. 9466. Lecture Notes in Computer Science. Springer, 2015, pp. 62–76. ISBN: 978-3-319-26849-1. DOI: 10.1007/978-3-319-26850-7_5. URL: https://doi.org/10.1007/978-3-319-26850-7_5.

[20] P. S. Almeida, A. Shoker, and C. Baquero. "Delta State Replicated Data Types." In: *CoRR* abs/1603.01529 (2016). arXiv: 1603.01529. URL: http://arxiv.org/abs/1603.01529.

[21] H. Attiya and R. Friedman. "Limitations of Fast Consistency Conditions for Distributed Shared Memories." In: *Inf. Process. Lett.* 57.5 (1996), pp. 243–248. DOI: 10.1016/0020-0190(96)00007-5. URL: https://doi.org/10.1016/0020-0190(96)00007-5.

[22] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting consistency back into eventual consistency." In: *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. Ed. by L. Réveillère, T. Harris, and M. Herlihy. ACM, 2015, 6:1–6:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972.

[23] C. Baquero, P. S. Almeida, and A. Shoker. "Pure Operation-Based Replicated Data Types." In: *CoRR* abs/1710.04469 (2017). arXiv: 1710.04469. URL: http://arxiv.org/abs/1710.04469.

[24] A. Barreto, J. A. Silva, H. Paulino, and N. Preguiça. *CRDTs em Ambientes Dinâmicos*. Communication in INForum 2019 - 11o Simpósio de Informática. URL: https://drive.google.com/open?id=1LpPwOhOudMSZOoDlNpJK2_6pvqL5JDNX.

[25] L. Benmouffok, J. Busca, J. M. Marquès, M. Shapiro, P. Sutra, and G. Tsoukalas. "Telex: Principled System Support for Write-Sharing in Collaborative Applications." In: *CoRR* abs/0805.4680 (2008). arXiv: 0805.4680. URL: http://arxiv.org/abs/0805.4680.

[26] E. A. Brewer. "A certain freedom: thoughts on the CAP theorem." In: *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*. Ed. by A. W. Richa and R. Guerraoui. ACM, 2010, p. 335. ISBN: 978-1-60558-888-9. DOI: 10.1145/1835698.1835701. URL: https://doi.org/10.1145/1835698.1835701.

[27] R. Brown, Z. Lakhani, and P. Place. "Big(ger) Sets: decomposed delta CRDT Sets in Riak." In: *CoRR* abs/1605.06424 (2016). arXiv: 1605.06424. URL: http://arxiv.org/abs/1605.06424.

[28] S. Burckhardt. "Principles of Eventual Consistency." In: *Foundations and Trends in Programming Languages* 1.1-2 (2014), pp. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011.

[29] F. Cerqueira. "Um Sistema Publicador/Subscritor com Persistência de Dados para Redes de Dispositivos Móveis." Master's thesis. FCT NOVA, Sept. 2017. URL: http://hdl.handle.net/10362/28553.

[30] F. Cerqueira, J. A. Silva, J. M. Lourenço, and H. Paulino. "Towards a persistent publish/subscribe system for networks of mobile devices." In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2017, Las Vegas, NV, USA, December 11 - 5, 2017*. 2017, 2:1–2:6. DOI: 10.1145/3152360.3152362. URL: https://doi.org/10.1145/3152360.3152362.

[31] *Cisco Mobile Visual Networking Index (VNI) Forecast Projects 7-Fold Increase in Global Mobile Data Traffic from 2016-2021*. Cisco. URL: https://newsroom.cisco.com/press-release-content?articleId=1819296.

[32] P. T. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. "The many faces of publish/subscribe." In: *ACM Comput. Surv.* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078. URL: https://doi.org/10.1145/857076.857078.

[33] *Forecast number of mobile devices worldwide from 2019 to 2023 (in billions)*. Statista. URL: https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/.

[34] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. "Verifying strong eventual consistency in distributed systems." In: *PACMPL* 1.OOPSLA (2017), 109:1–109:28. DOI: 10.1145/3133933. URL: https://doi.org/10.1145/3133933.

[35] M. B. Jones, ed. *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. ACM, 1995. ISBN: 0-89791-715-4. DOI: 10.1145/224056. URL: https://doi.org/10.1145/224056.

[36] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. "Rover: A Toolkit for Mobile Information Access." In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Ed. by M. B. Jones. ACM, 1995, pp. 156–171. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224069. URL: https://doi.org/10.1145/224056.224069.

[37] R. Kraft, B. Erb, D. Mödinger, and F. Kargl. "Using conflict-free replicated data types for serverless mobile social applications." In: *Proceedings of the 8th ACM International Workshop on Hot Topics in Planet-scale mObile computing and online Social neTworking, HOTPOST@MobiHoc 2016, Paderborn, Germany, July 5, 2016*. Ed. by Y. Chen and M. Sirivianos. ACM, 2016, pp. 49–54. ISBN: 978-1-4503-4344-2. DOI: 10.1145/2944789. URL: http://dl.acm.org/citation.cfm?id=2944868.

[38] A. van der Linde, J. Leitão, and N. M. 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@EuroSys 2016, London, United Kingdom, April 18, 2016*. Ed. by P. Alvaro and A. Bessani. ACM, 2016, 12:1–12:4. ISBN: 978-1-4503-4296-4. DOI: 10.1145/2911151.2911163. URL: https://doi.org/10.1145/2911151.2911163.

[39] A. van der Linde, P. Fouto, J. Leitão, N. M. Preguiça, S. J. Castiñeira, and A. Bieniusa. "Legion: Enriching Internet Services with Peer-to-Peer Interactions." In: *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. Ed. by R. Barrett, R. Cummings, E. Agichtein, and E. Gabrilovich. ACM, 2017, pp. 283–292. ISBN: 978-1-4503-4913-0. DOI: 10.1145/3038912.3052673. URL: https://doi.org/10.1145/3038912.3052673.

[40] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, and J. Bernardino. "Choosing the right NoSQL database for the job: a quality attribute evaluation." In: *J. Big Data* 2 (2015), p. 18. DOI: 10.1186/s40537-015-0025-0. URL: https://doi.org/10.1186/s40537-015-0025-0.

[41] B. Nédelec, P. Molli, A. Mostéfaoui, and E. Desmontils. "LSEQ: an adaptive structure for sequences in distributed collaborative editing." In: *ACM Symposium on Document Engineering 2013, DocEng '13, Florence, Italy, September 10-13, 2013*. Ed. by S. Marinai and K. Marriott. ACM, 2013, pp. 37–46. ISBN: 978-1-4503-1789-4. DOI: 10.1145/2494266.2494278. URL: https://doi.org/10.1145/2494266.2494278.

[42] G. Oster, P. Urso, P. Molli, and A. Imine. "Data consistency for P2P collaborative editing." In: *Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006, Banff, Alberta, Canada, November 4-8, 2006*. Ed. by P. J. Hinds and D. Martin. ACM, 2006, pp. 259–268. ISBN: 1-59593-249-6. DOI: 10.1145/1180875.1180916. URL: https://doi.org/10.1145/1180875.1180916.

[43] K. Petersen, M. Spreitzer, D. B. Terry, M. Theimer, and A. J. Demers. "Flexible Update Propagation for Weakly Consistent Replication." In: *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*. Ed. by M. Banâtre, H. M. Levy, and W. M. Waite. ACM, 1997, pp. 288–301. ISBN: 0-89791-916-5. DOI: 10.1145/268998.266711. URL: https://doi.org/10.1145/268998.266711.

[44] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. 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*. 2014, pp. 30–33. DOI: 10.1109/SRDSW.2014.33.

[45] N. Preguiça, M. Shapiro, and C. Matheson. "Semantics-Based Reconciliation for Collaborative and Mobile Environments." In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. Ed. by R. Meersman, Z. Tari, and D. C. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 38–55. ISBN: 978-3-540-39964-3.

[46] N. M. Preguiça. "Conflict-free Replicated Data Types: An Overview." In: *CoRR* abs/1806.10254 (2018). arXiv: 1806.10254. URL: http://arxiv.org/abs/1806.10254.

[47] N. M. Preguiça, M. Shapiro, and C. Matheson. "Semantics-Based Reconciliation for Collaborative and Mobile Environments." In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003*. Ed. by R. Meersman, Z. Tari, and D. C. Schmidt. Vol. 2888. Lecture Notes in Computer Science. Springer, 2003, pp. 38–55. ISBN: 3-540-20498-9. DOI: 10.1007/978-3-540-39964-3_5. URL: https://doi.org/10.1007/978-3-540-39964-3_5.

[48] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. "A Commutative Replicated Data Type for Cooperative Editing." In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pp. 395–403. ISBN: 978-0-7695-3659-0. DOI: 10.1109/ICDCS.2009.20. URL: https://doi.org/10.1109/ICDCS.2009.20.

[49] H. Qi and A. Gani. "Research on mobile cloud computing: Review, trend and perspectives." In: *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP), Bangkok, Thailand, May 16-18, 2012*. IEEE, 2012, pp. 195–202. ISBN: 978-1-4673-0733-8. DOI: 10.1109/DICTAP.2012.6215350. URL: https://doi.org/10.1109/DICTAP.2012.6215350.

[50] J. Rodrigues, E. R. B. Marques, L. M. B. Lopes, and F. M. A. Silva. "Towards a middleware for mobile edge-cloud applications." In: *Proceedings of the 2nd Workshop on Middleware for Edge Clouds & NV, USA, December 11 - 15, 2017*. ACM, 2017, 1:1–1:6. DOI: 10.1145/3152360.3152361. URL: https://doi.org/10.1145/3152360.3152361.

[51] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[52] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types." In: vol. 6976. July 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[53] J. Silva, H. Paulino, J. M. Lourenço, J. Leitão, and N. Preguiça. "Time-Aware Reactive Storage in Wireless Edge Environments." In: *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Houston, United States, November 12-14, 2019*. ACM, 2019.

[54] J. A. Silva, J. Leitão, N. M. Preguiça, J. M. Lourenço, and H. Paulino. "Towards the Opportunistic Combination of Mobile Ad-hoc Networks with Infrastructure Access." In: *Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2016, Trento, Italy, December 12-16, 2016*. Ed. by R. Martins and H. Paulino. ACM, 2016, p. 3. URL: http://dl.acm.org/citation.cfm?id=3022873.

[55] A. Teófilo, D. Remédios, J. M. Lourenço, and H. Paulino. “GOCRGO and GOGO: Two Minimal Communication Topologies for WiFi-Direct Multi-group Networking.” In: *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Melbourne, Australia, November 7-10, 2017*. Ed. by T. Gu, R. Kotagiri, and H. Liu. ACM, 2017, pp. 232–241. DOI: 10.1145/3144457.3144481. URL: https://doi.org/10.1145/3144457.3144481.

[56] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System.” In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Ed. by M. B. Jones. ACM, 1995, pp. 172–183. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224070. URL: https://doi.org/10.1145/224056.224070.

[57] W. Vogels. “Eventually consistent.” In: *Commun. ACM* 52.1 (2009), pp. 40–44. DOI: 10.1145/1435417.1435432. URL: https://doi.org/10.1145/1435417.1435432.

[58] S. Weiss, P. Urso, and P. Molli. “Logoot-Undo: Distributed Collaborative Editing System on P2P Networks.” In: *IEEE Trans. Parallel Distrib. Syst.* 21.8 (2010), pp. 1162–1174. DOI: 10.1109/TPDS.2009.173. URL: https://doi.org/10.1109/TPDS.2009.173.

[59] G. Younes, A. Shoker, P. S. Almeida, and C. Baquero. “Integration Challenges of Pure Operation-based CRDTs in Redis.” In: *First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17, 2016*. ACM, 2016, p. 7. ISBN: 978-1-4503-4775-4. DOI: 10.1145/2957319.2957375. URL: https://doi.org/10.1145/2957319.2957375.

APPENDIX A

Network Communication Volume Results

A.1 PNCounter

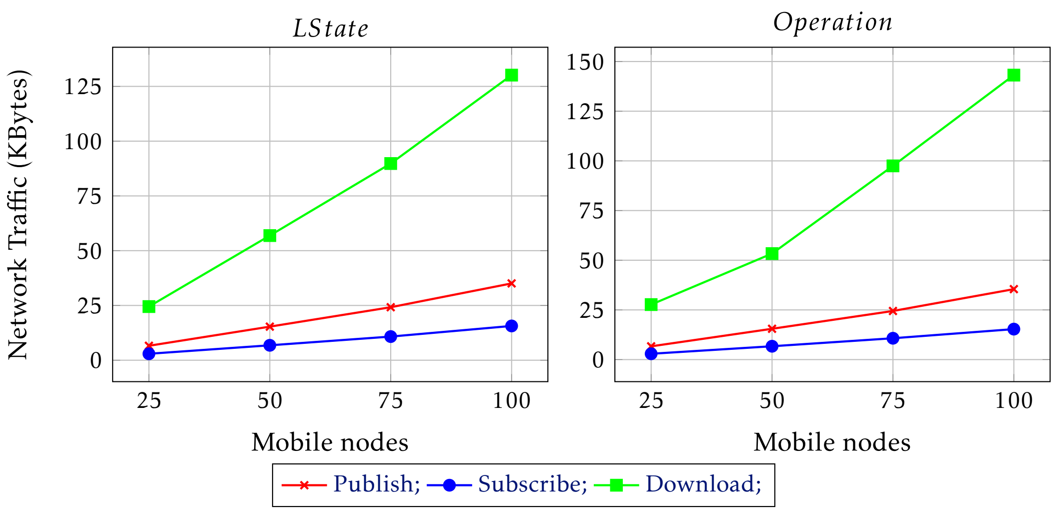


Figure A.1: Network traffic usage per message type within a LState and Operation PN-Counter simulations.

A.2 GSet

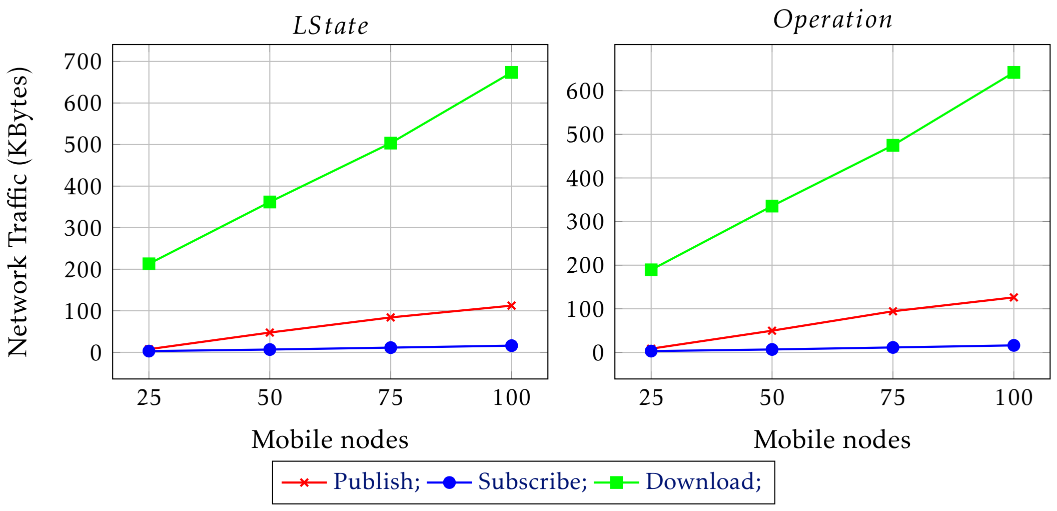


Figure A.2: Network traffic usage per message type within a LState and Operation GSet simulations.

A.3 ORSet

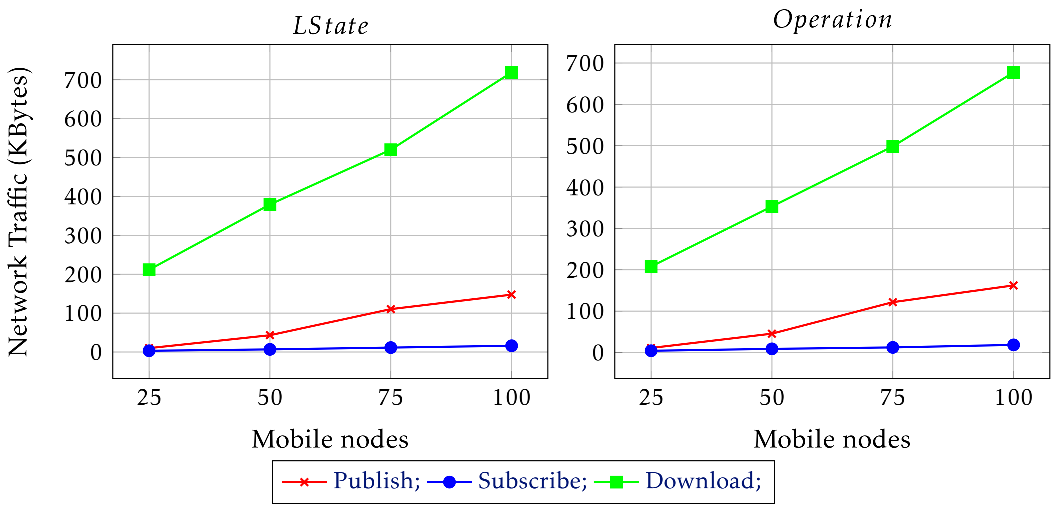


Figure A.3: Network traffic usage per message type within a LState and Operation ORSet simulations.

A.4 AWMap

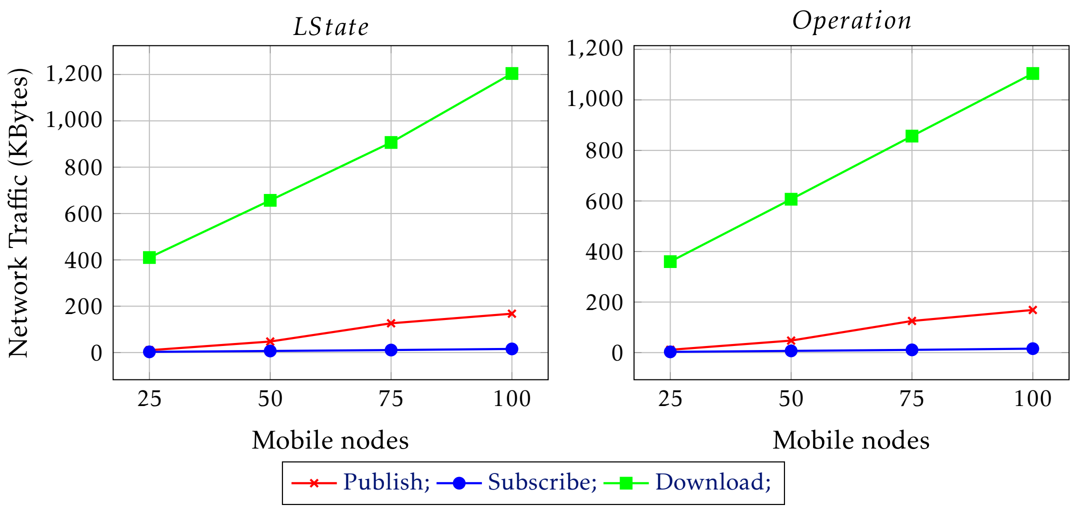


Figure A.4: Network traffic usage per message type within a LState and Operation AWMap simulations.

Appendix B

Churn & Late Entry Impact Results

B.1 Churn

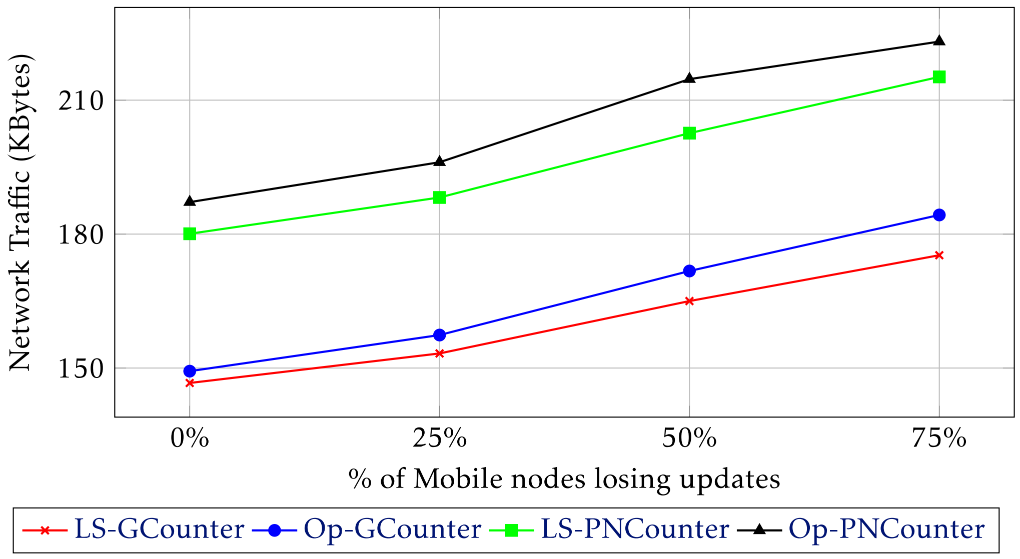


Figure B.1: Network traffic for LState and Operation counter data types with the increase of nodes who lose updates, for simulations with 100 nodes.

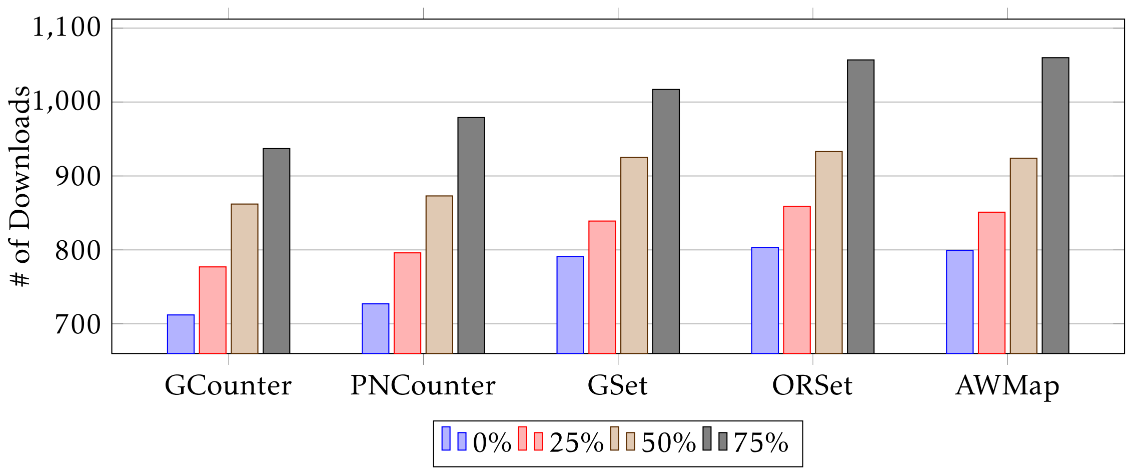


Figure B.2: Number of download messages transmitted in lState data type simulations with 100 nodes, where the percentage number of nodes who lose update vary.

Churn Rate	Total Downloads	Download Object	Download Update
0%	710	99	611
25%	778	113	665
50%	852	162	690
75%	933	196	737

Table B.1: Number of download operations made for operation GCounter simulations with 100 nodes with introduced churn events, showcasing the ratio between object and update downloads.

Churn Rate	Total Downloads	Download Object	Download Update
0%	725	99	626
25%	792	117	675
50%	870	170	700
75%	971	198	773

Table B.2: Number of download operations made for operation PNCounter simulations with 100 nodes with introduced churn events, showcasing the ratio between object and update downloads.

B.2 Late Entry

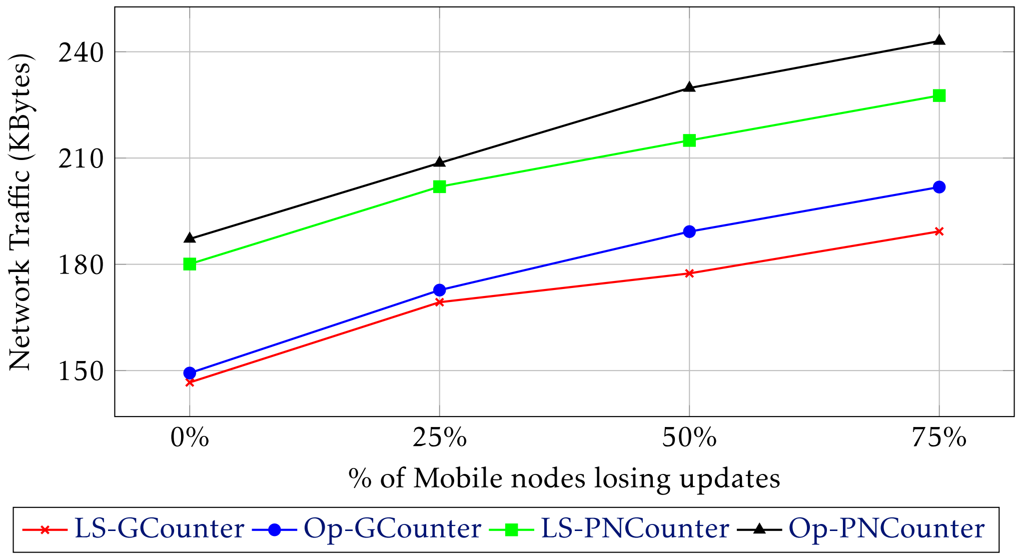


Figure B.3: Network traffic for LState and Operation counter data types with the increase of nodes who experience late entry, for simulations with 100 nodes.

Churn Rate	Total Downloads	Download Object	Download Update
0%	710	99	611
25%	795	121	674
50%	865	168	697
75%	965	203	762

Table B.3: Number of download operations made for operation GCounter simulations with 100 nodes with introduced late entry events, showcasing the ratio between object and update downloads.

Churn Rate	Total Downloads	Download Object	Download Update
0%	725	99	626
25%	803	125	678
50%	887	172	715
75%	978	209	769

Table B.4: Number of download operations made for operation PNCounter simulations with 100 nodes with introduced late entry events, showcasing the ratio between object and update downloads.

2019 **Conflict-Free Replicated Data Types in Dynamic Environments** António Barreto

António José Sá Barreto

BsC in Computer Science and Engineering

Conflict-Free Replicated Data Types in Dynamic Environments

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

December, 2019

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

António José Sá Barreto

BsC in Computer Science and Engineering

Conflict-Free Replicated Data Types in Dynamic Environments

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

December, 2019

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA