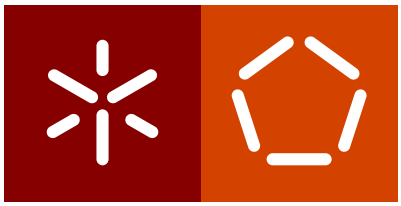
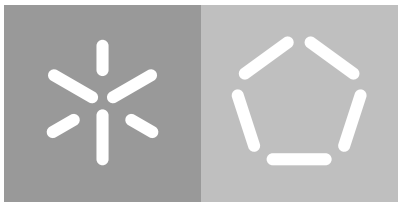


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Henrique Moreira Gomes Fernandes

Probabilistic Data Types

July 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Henrique Moreira Gomes Fernandes

Probabilistic Data Types

Master dissertation
Integrated Masters in Informatics Engineering

Dissertation supervised by
Carlos Miguel Ferraz Baquero-Moreno

July 2021

ACKNOWLEDGEMENTS

I would like to thank my thesis' supervisor, Carlos Baquero, for introducing me to the interesting topics this document revolves around, as well as for playing an integral part in the research behind them.

Additionally, I would like to acknowledge my parents for supporting me financially and in particular my mother, for allowing me to learn English from a young age, which made the creation of this document a much different experience.

ABSTRACT

Conflict-Free Replicated Data Types (CRDTs) provide deterministic outcomes from concurrent executions. The conflict resolution mechanism uses information on the ordering of the last operations performed, which indicates if a given operation is known by a replica, typically using some variant of version vectors. This thesis will explore the construction of CRDTs that use a novel stochastic mechanism that can track with high accuracy knowledge of the occurrence of recently performed operations and with less accuracy for older operations. The aim is to obtain better scaling properties and avoid the use of metadata that is linear on the number of replicas.

Keywords: Conflict-free Replicated Data Types, probabilistic representation of sets, Bloom filters, eventual consistency

RESUMO

Conflict-Free Replicated Data Types (CRDTs) oferecem resultados determinísticos de execuções concorrentes. O mecanismo de resolução de conflitos usa informação sobre a ordenação das últimas operações realizadas, que indica se uma dada operação é conhecida por uma réplica, geralmente usando alguma variante de *version vectors*. Esta tese explorará a construção de CRDTs que utilizam um novo mecanismo estocástico que pode identificar com alta precisão o conhecimento sobre a ocorrência de operações realizadas recentemente e com menor precisão para operações mais antigas. O objetivo é a obtenção de melhores propriedades de escalabilidade e evitar o uso de metadados em quantidade linear em relação ao número de réplicas.

Palavras-chave: Conflict-free Replicated Data Types, representação probabilística de conjuntos, Bloom filters, consistência eventual

COPYRIGHT AND CONDITIONS OF USE OF WORK BY THIRD PARTIES

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração. Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Pedro Henrique Moreira Gomes Fernandes

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

1

INTRODUCTION

1.1 CONTEXT

The CAP theorem [17, 27] states that only two of the three properties of shared-data systems can be achieved simultaneously - data consistency, system availability, and tolerance to network partitions. In view of the inevitability of the the occurrence of network partitions and the need to counter them, particularly in large-scale, geographically distributed systems, only one other property remains - either availability or consistency.

In order to achieve high availability, the model of consistency in a system must be relaxed. One of such models is eventual consistency [40, 41]. Under the conditions defined by this model, replicas are able to serve requests immediately, whether they mutate the state at the receiving replica or not. Replies are based on local state at the time of service and happen with no need for synchronizations with remote replicas. The effects of mutating operations can be shared asynchronously, at some point in the future, allowing replicas to converge to the same state. As such, synchronizations allow clients to operate on the same data objects by issuing requests to different replicas. The frequency of synchronization has direct impact on how stale local states can get, thus influencing the quality expected from replies. In spite of the possibility of replying with out of date information, many use cases allow for some flexibility, making eventual consistency a viable option.

Conflict-free Replicated Data Types (CRDTs) are data types intended for distributed use that employ the eventual consistency model, offering deterministic convergence of state, even in environments with concurrency. Due to the fact that some operations are not commutative, such as the addition and removal of the same element from a set, these data types may include a conflict resolution mechanism, to deterministically decide on the resulting state of a replica. CRDTs can take the shape of registers, counters, sets, maps, sequences, among others - each containing an appropriate conflict resolution strategy [36]. Among the different categories of CRDTs there are Causal CRDTs, which depend on knowledge of known events to provide adequate semantics to some data types.

1.2 Problem Statement and Main Objectives

Causal contexts, the structures at the heart of causal CRDTs, record events known by their respective replicas. In spite of being widely applicable and providing excellent support for compaction of information on known events, existing causal contexts hold an amount of data that grows linearly in relation to the number of replicas in the system. Given the fact that this property limits scalability, the present dissertation intends to explore the use of probabilistic structures as replacements of the classic, deterministic and error-free causal contexts.

Furthermore, due to the characteristics of the probabilistic structures considered, event identifiers may no longer need to be formed in the same way as they did previously, potentially reducing their overall size by doing away with the need to identify their origin replica. This anonymity, in addition to smaller states for causal CRDTs, provides the additional benefit of removing the requirement for management of identities and membership in the system, which may be especially useful in network with some degree of dynamicity. All of these are objectives of the newly proposed design.

Lastly, the evaluation of the conceived solution aims to inform the reader of the potential of such an approach regarding memory consumption and frequency of errors, so as to guide potential further implementations or evolutions of this work.

1.3 Main Contributions

The main contributions of this dissertation were the following:

- Recognition of the scalability limiting factors in existing solutions and assessment of alternative designs based on probabilistic structures.
- The identification of hindrances accompanying the use of Age-Partitioned Bloom Filters (APBFs), that could apply to similar probabilistic structures, and conception of filter union strategies to provide additional robustness to the existing design in the context of the present thesis.
- The reimplementation in Rust of Age-Partitioned Bloom Filters with their adequate adaptations [5]. This intends to offer the benefit of added memory safety and possible improvements in efficiency and memory management when compared with the original implementation.
- The analysis of the memory consumption and correctness of a concrete probabilistic CRDT, with the aim of determining potential use cases and informing future implementations under realistic conditions.

1.4 OUTLINE

The essential concepts of this thesis are presented by dissecting the title "Probabilistic Data Types" into components, in order to explain their reason to be and introduce the current state of the art in the topics that are the basis of this dissertation.

In this spirit, Chapter 2 presents Conflict-free Replicated Data Types, their different main categories, the context in which they are relevant and the inner structure designed to hold the causal history known by an instance of a data type.

Subsequently, Chapter 3 tackles the aforementioned causal history, presents the potential limitations of existing designs and introduces a series of probabilistic structures and related considerations that lead to the use of Age-Partitioned Bloom Filters as a substitute for the classic causal context.

Chapter 4 starts by presenting the inner structures of CRDT designs and the history and implementation details of sets, which are the focus of the present thesis. In addition, it explores how APBFs could be used in place of the classic and deterministic causal contexts, by inspecting the synchronization algorithm, analyzing the errors expected from the probabilistic approach and looking at adaptations of the designs of both APBFs and CRDTs, in an attempt to maximize the potential of the conceived solution.

Consequently, Chapter 5 presents the simulation environment used for evaluation, the parameters involved, and the results of their influence on the memory consumption, taking into consideration the classic variants, and the amount of inconsistencies observed when using a particular probabilistic data type.

Finally, Chapter 6 contains concluding remarks and suggestions of future work that might improve the results or expand on the work of this dissertation.

2

WHY "DATA TYPES"?

2.1 CONFLICT-FREE REPLICATED DATA TYPES

The words "Data Types" in the title refer to Conflict-free Replicated Data Types (CRDTs), presented and formally defined in [37, 36]. CRDTs are data types that provide deterministic results, even in the presence of concurrent executions. This determinism allows the data types to be conflict-free.

As their name suggests, CRDTs allow for replication of data. In particular, they employ optimistic replication, also known as *eventual consistency* (presented in section 2.3).

Each node that holds an instance of a CRDT accepts updates without synchronization with remote instances. The data replicas are able to store is kept consistent with other replicas through asynchronous synchronization and incorporation of remote message data into the local state.

Although solutions guided by the principles of eventual consistency existed previously, their implementations were ad hoc and error-prone. In the presence of conflicting concurrent updates, these solutions could require a consensus and roll-back for arbitration.

The formalization of CRDTs came to define sufficient conditions for convergence, which is the mechanism that enables eventual consistency. These data types are capable of ensuring absence of conflicts by leveraging mathematical properties, such as inflation in a semi-lattice and commutativity. Due to this, CRDTs are guaranteed to converge in a self-stabilising manner, even in spite of failures.

An initial portfolio of CRDTs [36] includes specifications and details on the implementation of various data types. These include counters, registers, sets, graphs and sequences. Since then there have been many improvements in correction evaluation techniques and performance, in addition to new data types, resulting from years of research.

Some projects that make use of CRDTs include Yjs [8], Automerge [1], Soundcloud's Roshi [4], Riak's state-based CRDTs [3] and also FlightTracker at Facebook [38], among others.

Conflict-free Replicated Data Types are divided into two major distinct design classes - state-based and operation-based. State-based designs include one other major class - delta-state CRDTs.

To summarize, in systems where the messaging layer guarantees reliable causal broadcast, operation-based CRDTs (in subsection 2.1.1) allow for simpler implementations, concise replica states and smaller messages, as stated in [12]. In contrast, state-based CRDTs (in subsection 2.1.2) have the advantages of supporting ad hoc dissemination of replica states and being able to handle duplicate messages and out-of-order delivery without breaking causal consistency; disadvantages include the consequent complex state designs and additional metadata stored. Delta-state CRDTs (in subsection 2.1.3) promise to bring the benefit of more compact messages from operation-based designs, by preferring to send deltas generated by update operations, instead of the complete state.

This chapter also introduces causal contexts, the structures that track knowledge of events observed by instances of causal CRDTs, in section 2.2. Eventual consistency, the consistency model used by CRDTs, makes an appearance in section 2.3, offering some perspectives regarding the type of environment where CRDTs may be useful.

The work described in this dissertation makes use of state-based designs.

2.1.1 *Operation-based*

A replicated object implementing an operation-based design (alternatively known as op-based) makes update requests received locally known to other replicas by disseminating data representing the operations themselves.

Op-based CRDTs have no *merge* method. Instead, each update is divided into two different methods: *prepare-update* and *effect-update*. The *prepare-update* is only executed at the replica that first received the update request (the source) and is immediately followed by an *effect-update*. Each *effect-update* is executed at all replicas, reaching replicas other than the source by propagation through a reliable causal broadcast communication protocol. The usage of such a protocol ensures that every message is delivered to every recipient exactly once and in an order consistent with the happened-before relation. This guarantees that two updates related by the happened-before relation execute at all replicas in the same sequential order. One update is considered to have happened before another if the former is already delivered at the moment when the latter executes. An update is considered delivered at some replica when it is included in its causal history. One other aspect in this approach is that concurrent updates must commute.

Queries and *prepare-update* method calls are side-effect-free, meaning that the state does not change as a result of their execution. As mentioned before, a *prepare-update* is performed only at the local replica. It uses the received operation request and its own local state to produce a message that aims to represent the operation, to be applied at all replicas. This application happens through the action of the *effect-update* method.

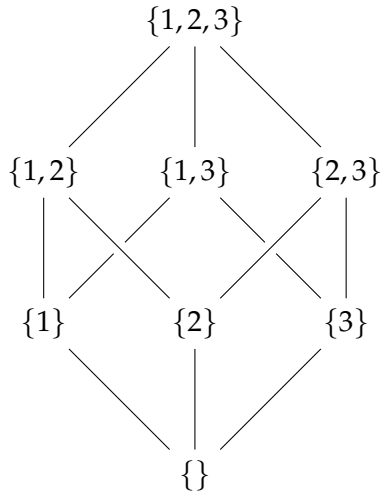


Figure 2.2: Semilattice for a set of integers

As already stated, op-based CRDTs have the benefit of enabling small sizes for both the object's own state size and message sizes, by only encoding the operation being disseminated, as seen in [12].

2.1.2 *State-based*

The state-based approach for replicated objects has every replica executing each update request, initially only modifying its own local state. Through synchronization, occasionally, each replica sends its local state to a set of other replicas, which merge the received state with their own, by using a *merge* function that deterministically reconciles the states.

The *merge* function, that aims to maintain convergence, is defined as a *join*: a least upper bound over a join-semillatice. Joins have some characteristics that make the state-based designs capable of handling scenarios such as the integration of duplicate messages or differently ordered deliveries at different replicas. Figure 2.1 presents these.

$$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c \qquad \text{(associativity)}$$
$$a \sqcup b = b \sqcup a \qquad \text{(commutativity)}$$
$$a \sqcup a = a \qquad \text{(idempotency)}$$

Figure 2.1: Properties of join

One example of such a join, for a grow-only Set, can be seen in Figure 2.2.

Eventually, every update request and its effects will be registered at every other replica, which by the convergence guarantees, will make every replica have equal state. This claim assumes that the set of known updates is the same among all of the replicas.

CRDTs following this approach have the benefits of idempotency and lack of need for message ordering over the op-based approach. These benefits will be carried to the delta-state versions. Due to the fact that state can grow unbounded for some data types, synchronizing by sending the full state to known replicas in the system is not an ideal solution. Delta-state CRDTs aim to solve this issue.

2.1.3 *Delta-state*

Delta-state conflict-free replicated data types [10] intend to take one of the benefits of operation-based CRDTs, by disseminating small messages with an incremental nature over unreliable communication channels. This is achieved with the use of delta-mutators, that return a delta-state. Delta-states are representations of the effect of updates and are typically much smaller than a full state. After being created, which happens every update, delta-states can be joined with local and remote states, as well as other delta-states, forming delta groups. They also preserve the idempotent nature of join, which ensures convergence over unreliable communication channels. Usually, they must be causally merged to maintain the desired semantics of the data object.

Op-based CRDTs have the aforementioned advantages of allowing for simpler implementations, concise replica state, and smaller messages. However, they are subject to limitations. One of them is the assumption of a message dissemination layer that guarantees reliable exactly-once causal broadcast. These guarantees are hard to maintain since large logs must be retained to prevent duplication. One other limitation, especially relevant in the context of this dissertation, is the fact that membership management is a hard task, not exclusive to op-based systems. Even so, in that particular context, the issue is exacerbated once the number of nodes gets larger or due to increased churn rate, since all nodes must be coordinated by the messaging middleware. One final constraint of the op-based approach is the need for operations to be executed individually, even when batched, on all nodes.

Although CRDTs in the state-based style avoid most of these hindrances, there is in them one major drawback - the communication overhead of shipping the entire state. Examples of such, given in [10], mention that the state size of a counter CRDT (a vector of integer counters, one per replica) increases with the number of replicas, whereas in a grow-only set, the state size depends on the size of the set itself. This condition limits the use of state-based CRDTs to data types with small state size (counters might be reasonable, though larger sets might not).

2.2 CAUSAL CONTEXT

Causal CRDTs, presented in [10], are a class of CRDT designs whose states are made up of a dot store and a causal context. The function of a dot store is to hold information specific to the data type, such as values and relevant metadata. The causal context, also known as the causal history or dot context, is used to remember what events are known by an instance of a distributed object. Further information regarding dot stores and causal CRDTs is presented in a later chapter of this document, in subsection 4.2.1 and subsection 4.2.2, respectively.

More specifically, a causal context is an ever-growing set of event identifiers. As [10] states, causal CRDTs make use of globally unique identifiers to tag locally occurring events. These identifiers are assigned only to specific update events, such as, for instance, $\mathsf{add}(e)$ operations to insert elements into a set. This is done in order to track knowledge of the updates and their effects across the replicas of the distributed object.

The construction of these identifiers is usually done in a manner that aims to facilitate the compaction of a set of them. With this in mind, the strategy that is normally followed is appending the value of a locally unique, monotonically increasing counter to a globally unique replica identifier. This way, by incrementing the local counter after every assignment, a given replica i in the system is able to generate unique values, starting with the pair $(i, 1)$, followed by $(i, 2)$, successively repeating the process for any operation the design deems worthy of an identifier. Each one of these globally unique identifiers is also known as a dot.

2.2.1 *Representation and Compaction*

By choosing an appropriate design for the dots, like the one that was previously mentioned, the lossless compaction of a causal context becomes possible. If that was not the case, the memory consumption of the set would grow linearly for every added dot. For a system that intends to run indefinitely, the size of the causal context could grow unbounded, which would eventually compromise the usefulness of causal CRDT designs.

Through synchronization with others, one instance is able to construct sequences of produced dots for each of the replicas in the system. When using an algorithm that guarantees causal consistency of the underlying CRDT, there is a single contiguous initial interval for every replica; one such algorithm is algorithm 2, introduced in [10].

For each replica $i \in \mathbb{I}$ ($\mathbb{I}$ being the set of replica identifiers), the contiguous initial interval of known dots from any replica $j \in \mathbb{I}$ can be represented by a single value, given by $\mathsf{max}_j(c_i)$. This is possible because the causal consistency guarantee given by the aforementioned algorithm ensures that, from replica j, every value from the initial one up to the one given by

$\mathsf{max}_j(c_i)$ is already known. To test the inclusion of a dot from j, with local sequence number n, in the causal context of i, the following condition is sufficient:

$$1 \leq n \leq \mathsf{max}_j(c_i) \implies (j, n) \in c_i \tag{1}$$

The $\mathsf{max}_i(c)$ function, defined in Figure 2.3, returns the greatest value in the causal context c from the sequence generated by replica i. Along with it there is a definition for function $\mathsf{next}_i(c)$, which is useful for producing dots in the previously described fashion. The dots in the following definition of causal context are pairs of replica IDs and integers ($\mathbb{I} \times \mathbb{N}$) that serve as locally unique identifiers, as presented in the introduction to this Section, 2.2.

$$\begin{aligned} \mathsf{CausalContext} &= \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\ \mathsf{max}_i(c) &= \mathsf{max}(\{n \mid (i, n) \in c\} \cup \{0\}) \\ \mathsf{next}_i(c) &= (i, \mathsf{max}_i(c) + 1) \end{aligned}$$

Figure 2.3: Causal Context

The conclusions here stated mean that the causal context could be represented as a compact vector, similar to a version vector [35]. If replica IDs coincide with indices in the vector, with each replica being assigned a single index, the causal context becomes especially compact. Otherwise, a simple map $\mathbb{I} \hookrightarrow \mathbb{N}$ suffices.

If the algorithm that supports the causal CRDT design is one that does not guarantee causal consistency (such as algorithm 1, from [10]), the same compaction principle can be applied, albeit with a caveat. Without the guarantee of causal consistency, there is no assurance that all known dots from a given replica form a contiguous sequence. However, the initial sequence of known dots from a given replica can be encoded in a structure similar to the ones already described; the remaining straggler dots can be stored in a set. The size of the set of stragglers is not expected to be problematic because as synchronizations occur, straggler dots eventually find their way into the compact representation, which means that the set is expected to be kept small.

An additional consideration is that compaction is not as likely for deltas, due to the fact that the mutators that generated them can produce or affect any dot, and not just an initial contiguous interval of dots. In spite of this, since deltas, even in their combined form, are transient and typically smaller than replica states, the set of affected dots is expected to remain limited in size.

Other methods exist for encoding known dots. These include Interval Version Vectors [33], that are capable of encoding non-contiguous dots as multiple intervals of sequential dots. One other approach is brought by the concise version vectors from [31]. For each replica,

these can be used to store the latest known sequence number produced locally and the set of dots yet to be seen, called exceptions.

An observation that can be made about this compaction scheme is that the size of the causal context grows linearly with the number of replicas. Furthermore, there is a reliance on replicas identifying themselves in the underlying algorithm. The solution described in this dissertation does away with the need for these, while introducing a probabilistic dimension. The probabilistic dimension inevitably adds a number of errors, that ideally, should be kept small. The context for this probabilistic alternative is given in chapter 3. Further comments on limitations in current designs appear in section 3.2.

2.3 Eventual Consistency

By the famous CAP theorem [17, 27], it is known that only two of the three properties of shared-data systems - data consistency, system availability, and tolerance to network partitions - can be achieved simultaneously.

In large-scale distributed systems, that may be geo-replicated, the occurrence of network partitions is expected and must be taken into account. Similarly, at a smaller scale, errors may regularly affect the network, due to the presence of low-quality hardware or difficult environmental conditions, such as extreme temperatures, rain, dust and electromagnetic noise from other connected devices.

Due to the inevitable presence of network partitions and the need to minimize their effect, according to the observations of the theorem, consistency and availability cannot be achieved at the same time. Consequently, a decision must be made - either relax consistency requirements or allow for more limited availability. To enable the system to remain highly available, a need for a weaker form of consistency came to be, known as eventual consistency [40, 41].

Under eventual consistency, a replica is capable of immediate serving of requests, including updates. The replies are based on local state and can be done without the need for synchronization with other replicas after each update. The updates, or their effects, are propagated later to other replicas and are applied eventually, asynchronously, and in possibly different orders. Therefore, the data a replica can serve, as a reply to a read, may be stale to some degree, not reflecting a write that has already occurred in the system. This might not be a problem for some use cases.

As suggested before, performant and highly available large-scale distributed systems may replicate data over multiple regions, following a model of eventual consistency. By relaxing consistency requirements, in order to achieve partition tolerance and high availability, system performance can be positively affected too. This fact is justified by the unbeatable lower bounds on round-trip times in communications between replicas in different regions,

imposed by the speed of the propagation of light. Thus, the speed of light hampers the use of algorithms designed to achieve stronger consistency - like [28] - since multiple round-trips are needed. Furthermore, in the presence of errors and failures, the number of messages needed increases, which is also an issue for both availability and performance perceived by clients.

3

WHY "PROBABILISTIC"?

3.1 INTRODUCTION

As stated in section 2.2, the causal context, when in use in a CRDT design, is a set of dots at its core, even if it can be encoded in a structure that differs from a set. The choice of which structure to use can depend on the underlying algorithm and whether it guarantees causal consistency to the distributed object, regardless of which data type it may be. Some of these different representations are discussed in subsection 2.2.1.

The structures used in current designs represent the set of dots in a compact, deterministic, and therefore, error-free fashion. The property of determinism in current causal contexts can be observed in two ways:

- If a dot is indeed in the set, the structure will always report it as present, which means that there are no false negatives.
- Similarly and expectedly, if a dot is not in the set, the structure will report it as absent. Thus, there is assurance of no false positives.

The compact, error-free solutions perfectly fit many of the common use cases for CRDTs. In spite of the frequent adequacy of existing methods and structures, some limiting factors can be found in their usage for specific use cases.

The following sections will present the limitations of existing implementations of causal contexts (section 3.2) and successively introduce concepts and various structures related to probabilistic set representation, while identifying constraints that invalidate the use of some of them in the context of this thesis. Finally, the Age-Partitioned Bloom Filter is presented in section 3.8 and remarks on event identifiers appear in section 3.9.

3.2 POSSIBLE LIMITATIONS OF EXISTING STRUCTURES

The major catalyst for this thesis was the fact that causal contexts in current designs make use of structures similar to version vectors [35]. One attribute of these is the storage of an amount of data that is linear with the number of replicas present in the system.

There is also a dependency on identifying replicas, which may not always be desirable. Usage of identification can be seen in the existing implementations of causal contexts, as well as in algorithm 2 (Algorithm 2 in [10]), which is used to achieve a causally consistent delivery when synchronizing with delta-states. However, anonymity throws causally consistent delivery out of the picture in algorithm 2. This happens due to the fact that it becomes no longer possible to track which deltas have been delivered to which remote replicas, rendering impossible their error-free garbage collection from the deltas' buffer. Thus, by striving to achieve an anonymous design, some possibilities are lost, including the use of such an algorithm. Losses of this nature limit some anonymous designs to more basic algorithms, such as algorithm 1 (also Algorithm 1 from [10]).

Dynamic environments are another potential generator of limitations. When overlays are dynamic and often changing, replicas may have to frequently refresh their set of neighbours and manage the entries of their causal contexts. The need for entry management stems not only from failures and intentional departures, but also from the appearance of previously unknown replicas. When it happens, it is essential to create an entry for the newly observed remote replica, in order to track known dots that originated there. Entry management can also be useful for garbage collection, in order to remove stale entries when faced with scenarios where a replica crashes or goes away for indefinite time. Depending on the underlying structure used to encode the causal context and the frequency of changes in the overlay, entry management may be more or less of a performance concern, as a result of resizing, for instance.

Dynamic environments are a known challenge to δ-CRDTs in particular, as briefly mentioned in [25]. Due to the fact that there may be no information about the last deltas acknowledged by a given new neighbour, emission of the full state may now be required more frequently. This need is observable in the above-mentioned Algorithm 2, on line 24.

3.3 PROBABILISTIC REPRESENTATION OF SETS - THE PROPOSED SOLUTION

The solution described in this dissertation aims to improve the existing solutions in scenarios where at least some of the limitations stated above are felt. The attempt makes use of a probabilistic structure to represent the set of dots, in place of deterministic ones. The use of a probabilistic structure spawns a number of errors, due to the inherent nature of stochastic methods. Nevertheless, if their occurrence can remain limited, they can allow the use of

inputs:
  $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbours
  $t_i \in \mathbb{B}$, transitive mode
  $\mathsf{choose}_i \in S \times S \to S$, state/delta
durable state:
  $X_i \in S$, CRDT state, $X_i^0 = \bot$
volatile state:
  $D_i \in S$, delta-group, $D_i^0 = \bot$
on $\text{operation}_i(m^\delta)$
  $d = m^\delta(X_i)$
  $X_i' = X_i \sqcup d$
  $D_i' = D_i \sqcup d$
on $\text{receive}_{j,i}(d)$
  $X_i' = X_i \sqcup d$
  if t_i
    $D_i' = D_i \sqcup d$
  else
    $D_i' = D_i$
periodically
  $m = \mathsf{choose}_i(X_i, D_i)$
  for $j \in n_i$ **do**
    $\mathsf{send}_{i,j}(m)$
  $D_i' = \bot$

Algorithm 1: Basic anti-entropy algorithm for δ-CRDTs

inputs:
  $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbours
durable state:
  $X_i \in S$, CRDT state, $X_i^0 = \bot$
  $c_i \in \mathbb{N}$, sequence number, $c_i^0 = 0$
volatile state:
  $D_i \in \mathbb{N} \hookrightarrow S$, deltas, $D_i^0 = \{\}$
  $A_i \in \mathbb{I} \hookrightarrow \mathbb{N}$, ack map, $A_i^0 = \{\}$
on $\mathsf{receive}_{j,i}(\mathsf{delta}, d, n)$
  if $d \not\sqsubseteq X_i$
    $X_i' = X_i \sqcup d$
    $D_i' = D_i\{c_i \mapsto d\}$
    $c_i' = c_i + 1$
  $\mathsf{send}_{i,j}(\mathsf{ack}, n)$
on $\mathsf{receive}_{j,i}(\mathsf{ack}, n)$
  $A_i' = A_i\{j \mapsto \mathsf{max}(A_i(j), n)\}$
on $\mathsf{operation}_i(m^\delta)$
  $d = m^\delta(X_i)$
  $X_i' = X_i \sqcup d$
  $D_i' = D_i\{c_i \mapsto d\}$
  $c_i' = c_i + 1$
periodically // ship interval or state
  $j = \mathsf{random}(n_i)$
  if $D_i = \{\} \vee \mathsf{min}(\mathsf{dom}(D_i)) > A_i(j)$
    $d = X_i$
  else
    $d = \bigsqcup\{D_i(l) \mid A_i(j) \leq l < c_i\}$
  if $A_i(j) < c_i$
    $\mathsf{send}_{i,j}(\mathsf{delta}, d, c_i)$
periodically // garbage collect deltas
  $l = \mathsf{min}\{n \mid (\_, n) \in A_i\}$
  $D_i' = \{(n, d) \in D_i \mid n \geq l\}$

Algorithm 2: Anti-entropy algorithm ensuring causal consistency of δ-CRDTs

CRDT designs, namely ones that synchronize by sending state to neighbours, in scenarios where they would otherwise be discouraged or even impossible to use.

The following sections of the chapter aims to provide context to the probabilistic representation of sets. Included is one of the two first structures of this kind, which came to be known as the Bloom Filter [14], and the variant of it that is used in the final design, the Age-Partitioned Bloom Filter [39].

3.4 BLOOM FILTERS

In 1970, Burton H. Bloom [14] introduced two novel hash-coding methods as an alternative to the existing error-free alternatives, such as hash tables, to represent sets and ascertain presence of elements in them. The new probabilistic methods are intended to reduce the amount of space required to represent sets, allowing the applications that did not rely on error-free performance to keep them in core memory, avoiding storing this data in slower memory, such as disks. Two methods were suggested - the first shared many similarities with the error-free approach, while the second, while still sharing some similarities, followed a more radical approach.

The first method, derived from existing error-free methods, had a hash address being calculated for every element being inserted or queried for presence in the set, which depended on the element itself. The hash address referred to one of the cells in the hash area. The cells in this new method were able to be smaller than those in conventional structures, since here they no longer needed to hold all of the bits of inserted elements. Instead, they would store c bits - called the element's *code*, dependent on the inserted element itself - with the first bit indicating presence in the set (the value 0 of this flag indicates absence and 1 otherwise).

For insertions, the addressed cell of the hash space would be checked for content. If empty, the cell would store the element's code, which would not be guaranteed to be unique. This is what allowed space savings in the first method, while also introducing errors, due to the fact that two distinct elements being inserted may now be mapped to the same hash code. If the addressed cell was already filled, another hash address would be calculated for the element until an empty cell was found.

For membership testing, the calculation of the element's code and its hash addresses would be repeated. Then, the content of the addresses would be checked. If the element's code was found in one of them, the element was said to be part of the set; otherwise, if it was not found - by observing codes of other elements or empty cells - the element was considered absent from the set.

The hash codes presented in this method are an alternative designation for fingerprints, more commonly found in dictionary-based approaches created ever since. Thus, this first method can be seen as a precursor to those.

The second method is what became known as the Bloom Filter. It utilizes a hash area consisting of N individual addressable bits, that can be seen as 1-bit cells. The initial state of the bits has them all set to 0.

On insertion, each element is mapped by each of a set of hash functions to an address in the hash area. From the resulting set of addresses, every corresponding bit is then set to 1.

To evaluate the membership of an input element, an address is calculated by each of the hash functions. The element is said to be in the set if all of the bits addressed are set to 1.

As is the case with the deterministic, error-free methods, inserted elements will always be identified as present by queries, meaning that there are no false negatives. In contrast, if an element has not been inserted in these structures, it can still be reported as present, generating a false positive. False positives can occur in both methods in the following form:

- In the first method, if an absent element has the same hash code (or fingerprint) as an inserted element, it will be reported as present.
- In the second method, if all the bits in the addresses calculated for an absent element are set to 1, it will be identified as previously inserted by queries. This can happen if the addressed bits were previously set by insertions of other elements.

Focusing on Bloom Filters, as it may be possible to infer, given the fact that the size of the hash area (or bit space) is limited and since queries rely on the existence of bits set to 0 to identify absent elements, there is a point of saturation beyond which too many false positives occur. Burton H. Bloom identified this point of saturation, considering it to be reached when ”half the bits in the hash field are 1 and half are 0”. This maximum fill ratio of $1/2$ is in accordance with the conclusions in [18].

Two main points influence saturation:

- The smaller the bit space, the faster it gets saturated.
- The more hash functions used, the more possible distinct addresses are generated to represent an element, which improves distinction between elements and therefore decreases the occurrence of false positives. However, too many hash functions saturate the bit space at a higher rate and increase the likelihood of a false positive occurring, suggesting there is a balance to be found regarding the number of hash functions in used, and subsequently, the number of bits representing an element.

One key idea common to both approaches is that a greater allowable error frequency enables the usage of a smaller area of memory, with the exact opposite being also true, indicating the existence of tunable parameters in implementations. These enable the creation of filters according to the needs of the applications that make use of them.

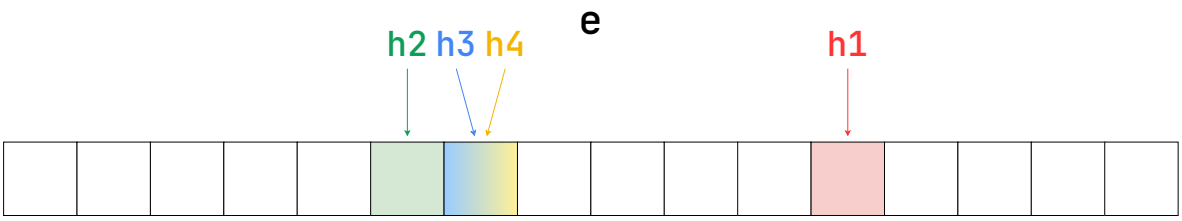


Figure 3.1: Insertion of element e in a Bloom Filter

3.5 THE NEED TO SCALE

The original Bloom Filter, as well as several subsequent variants that appeared during the following decades, requires the estimation of the cardinality of the set in advance. This prerequisite is a direct consequence of the Bloom Filter's maximum fill ratio of $1/2$ for optimal performance, mentioned at the end of subsection 3.4. Expectedly, the insertion of a number of elements greater than the value estimated at construction time can break the point of saturation, which increases the chance for false positives up to undesirable levels.

The set's estimated cardinality, in addition to the desired maximum false positive probability, is used for the calculation of the filter's dimension. The estimation of the size of the set may not be an easy feat, being dependent on the underlying application, which in turn generally leads to an overestimation of the number of elements in the set, in an effort to preserve the chosen false positive probability. Inevitably, this brings a consequent waste of memory space.

Given that CRDTs should operate indefinitely, it is expected that an endless stream of dots is generated throughout their lifetime. Being only capable of holding a predefined number of elements, static probabilistic structures for set representation can only represent a portion of the input stream of dots. Therefore, they are not a good fit for use in the context of CRDTs.

3.6 SCALABLE BLOOM FILTERS

Scalable Bloom Filters [9] aim to solve the issue of not knowing the size of the set in advance by dynamically adapting to the number of stored elements, while ensuring that the false positive probability is kept under the desired upper bound.

As described before, classic Bloom Filters use k hash functions to set and query each index in its hash area, an array of M bits. The inputs of the hash functions are elements to be inserted or tested for presence in the set and their outputs are indices in the array. Since these hash functions are independent, their outputs for a given input have a probability of collision greater than 0, increased for lower values of M. It is possible to conclude that

the more collisions there are in the bit space locations calculated for a given element, the more likely a query for the same element is to generate a false positive if it is not in the filter. A possible extreme case appears when $h_1(x) = h_2(x) = \cdots = h_k(x)$, where element x is represented by the minimum of only 1 bit. Less extreme scenarios are more likely, and yet, still potentially troublesome. Therefore, in the classic design, given that input elements can be represented by n bits, with $1 \leq n \leq k$, they have varying degrees of susceptibility to false positives. This can be seen in Figure 3.1, where element e is represented by only 3 bits, even when the Bloom Filter uses 4 hash functions.

Scalable Bloom Filters, building upon the design in [20], have improved robustness over the original Bloom Filter by dividing the M bits in k slices of $m = M/k$ bits, with k being the aforementioned number of hash functions mapping elements to the positions in the filter. With each of the k hash functions mapping each input element to an index in its assigned slice of m bits, there is a guarantee that every element is represented by k bits. Hence, the division of the space of M bits into k slices allows the possibility of making every element equally susceptible to the false positive probability.

The false positive probability decreases if there is an increase in:

- the number of slices k, with increased opportunity for differentiation between elements;
- their size m, with decreased chance for collisions within a slice.

Taking the previously mentioned extreme case where an element was only represented by 1 bit, as all of the k hash functions mapped it to the same index of the filter, if that 1 index was set by any other element already inserted, the reply to a membership query would indicate presence, without regard to the actual presence of the queried element in the filter. By slicing the space of M bits, a membership query for that element would only consider it as part of the filter if $h_i(x) = 1$ for $1 \leq i \leq k$, meaning that each of its indices in the k slices is set to 1, which would be comparatively less likely to trigger a false positive result.

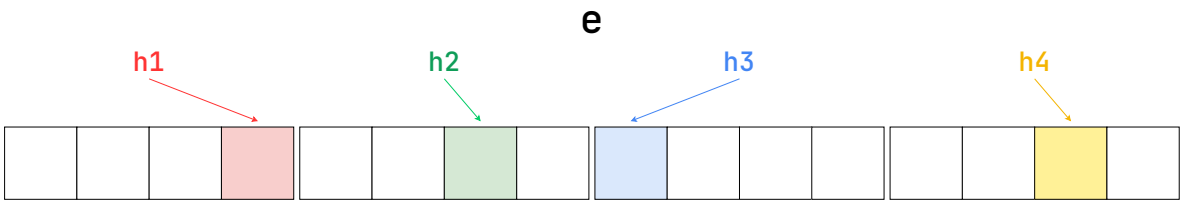


Figure 3.2: Insertion of element e in a Partitioned Bloom Filter

Scalable Bloom Filters are supported by two key ideas:

1. A Scalable Bloom Filter is made up of a series of one or more regular partitioned Bloom Filters. As each gets saturated by reaching its predefined maximum number of added elements, a new Bloom Filter is added. By reaching the predefined number of elements

added, the fill ratio of the filter is expected to be around the optimum maximum value, 1/2. Consequently, a new filter is added to receive subsequent insertions, thus avoiding disrupting the desired false positive probability of previous filters in the sequence. Membership queries test for presence of elements in each filter.

2. Each new Bloom Filter in the sequence is created with a smaller maximum false positive probability, decreasing on a geometric progression. Otherwise, if the initial probability was kept for each successive filter, as the number of filters increased, so would the overall chance for false positives being reported, since every added query would be an additional opportunity for such a result. The false positive probability for newly added filters is calculated in such a way that the compounded probability over the series of Bloom Filters converges to some wanted value, even accounting for an infinite series.

Thus, a Scalable Bloom Filter starts with just one subfilter, divided into k_0 slices and an error probability P_0. Once the initial subfilter in the sequence becomes full, a new one is added with k_1 slices (defined in the following paragraph) and a false positive probability of $P_1 = P_0 r$, where r is a tightening ratio such that $0 < r < 1$. The same is repeated for following Bloom Filters. As such, at any given moment, the sequence of l subfilters will have false positive probabilities $P_0, P_0 r, P_0 r^2, \ldots, P_0 r^{l-1}$ respectively.

The number of slices for a partitioned filter is derived from the equation $P = p^k$, where P is the false positive probability of the filter, p is the fill ratio of a slice, and k is the number of slices of the filter. The fill ratio of a slice p is equivalent to the probability that a bit will be set at a random index of the slice. Accordingly, the false positive probability P is the probability that a set bit is found in every slice when querying an absent element, which is given by p^k. Since the maximum value for p should be 1/2, referring again to the conclusions at the end of a section 3.4, the number of slices k is given by $k = \log_2 P^{-1}$. As a result, and considering that the error probability of filter i is given by $P_0 r^i$, the general formula for the number of slices of a particular filter is:

$$k_i = \log_2 P_i^{-1} = k_0 + i \log_2 r^{-1} \tag{2}$$

Bearing in mind the range of the tightening ratio r, the number of slices per filter in the sequence increases.

As the problem they are trying to solve implies that the cardinality of the set is not known in advance, Scalable Bloom Filters are designed to adapt to sets of different orders of magnitude in size. The suggested way to deal with this has new filters being added with the size of its slices varying exponentially. Thereby, the subfilters in the series have slices with respective sizes of $m_0, m_0 s, m_0 s^2, \ldots, m_0 s^{l-1}$, where s is the slice growth factor and l is the number of stages of the Scalable Bloom Filter. It is stated that a practical choice for the slice

growth factor is $s = 2$, which preserves m_i as a power of 2, if m_0 already starts as such. This becomes particularly useful because the range of a hash function is typically a set with a power of 2 size, avoiding imbalances in the distribution of indices.

In spite of using several important features in Bloom Filter design and making them possible to use with an increasing number of elements being inserted, Scalable Bloom Filters are not adequate replacements for existing causal contexts. An intended feature of Scalable Bloom Filters, as described before, is the growth in size as elements are inserted. Thus, memory consumption grows unconstrained in this design, as no added input is forgotten, which is the major pitfall for the desired use case of this dissertation.

3.7 THE NEED TO FORGET

3.7.1 *Introduction*

It was established at the end of the previous subsection, 3.6, that memory consumption can be a problem for probabilistic, compact representations of ever-growing sets. Sets of this nature, including the set of known dots at a replica, can be seen as infinite streams of elements that, over time, become known at a particular location.

Two possible ways of avoiding unbounded growth are:

- Have one or only a few values represent a bigger set of them.
- Do not represent all of the elements in the stream/set, only keeping in memory representations of some of them.

The first scenario is akin to what happens in vector clocks, as stated in the proposition at 1, where the value at an index represents all smaller or equal values. That specific solution benefits from an underlying causally consistent delivery. In the absence of such a delivery guarantee, an additional set for non-contiguous dots can be used, as mentioned before in subsection 2.2.1. Here, the set of known dots is represented by each value in the vector clock, that, again, represents itself and all the smaller preceding values, and by values in the auxiliary set, which is expected to be kept small, as the mechanism to achieve consistency continuously operates. In spite of the compactness of these representations, they present memory requirements that vary linearly with the number of replicas in the system, which is one of the possible limitations the solution proposed in this document is trying to circumvent. Among others, this limitation is presented in section 3.2.

The second scenario takes into account that keeping either exact or summarised information for all observed elements is too much of a burden regarding space consumption. In response to this difficulty, there is the option to represent only a subset of the elements and

eventually deleting them definitively, keeping memory usage limited. This is made possible by models designed to process infinite streams.

3.7.2 *Stream Management Models*

The management of a stream can be done following two major models, as seen in [32] - the sliding window model and the landmark window model.

The sliding window model, introduced in [21], aims to make queries relevant to only the last w observed elements of the stream. In an exact implementation of this model, if the window is full, meaning that at least w elements have already been seen, every new insertion implies the deletion of the oldest element in the window. Succinctly, any solution to the sliding window problem must store representations of the last w inserted elements, allow querying over the window and continuously forget the oldest. Its dimension in memory is related to the size of the window w. In the case of solutions that make use of approximate representation of inserted elements, the dimension is also tied to the chosen maximum false positive probability.

The landmark window model divides the stream into disjoint segments, where the points of division are landmarks. Landmarks can be arbitrary events, including time-based ones, such as the conclusion of a day or week, or quantity-based, namely, observing N elements since the initial moment or the previous landmark. Due to the dependence on the occurrence of a landmark, the size of a landmark window is not fixed. Keeping this in mind, the dimension of a landmark window solution is reliant on the fact that it must provide enough space to represent elements until the next landmark takes place. When a landmark event arises, memory of elements in the current window is lost as a consequence of the re-initialization of the underlying structures, in preparation for a new generation of elements.

3.7.3 *Evaluation*

For the purposes of the objectives of this thesis, the preferred solution should integrate a structure that supports the sliding window model or similar.

The reason there might be a problem with handling the stream using the landmark window model is the way elements are forgotten. By having rigid breaking points in memory, if by chance a replica tries to ascertain the membership of a dot shortly after a landmark, not much, if anything at all will be remembered. Additionally, this scenario is problematic even if a dot was registered fairly recently. In case it was seen in the final moments of the lifetime of the previous landmark window, the landmark will wipe the records of its existence and others like it. Breaks in memory of this kind lead to the somewhat frequent oblivion of

elements that might still be worth remembering. These are the cause of inconsistencies, as will be discussed later in subsection 4.5.1.

The use of a sliding window model or others alike may avoid the previous issues because the sliding of the window allows for a more continuous representation of the last observed elements. The continuity of this representation may be on a spectrum. Some solutions, like [11], when at full capacity of their window, forget their elements one by one, as each new entry arrives. Others, like [34, 39], might prefer a more coarse approach, by dividing the incoming stream into disjoint groups of elements, said to be of the same generation, and forgetting them in blocks. Sliding windows have the characteristic of considering more relevant elements inserted more recently rather than older ones. This is beneficial for causal contexts because if a dot is more recent, it may still not be known at others replicas, thus needing to be remembered so that it can be shipped to remote locations. Conversely, older dots and their associated effects may already be known at remote replicas. As such, it may be possible to forget them with no consequence with regards to consistency. Nonetheless, older elements that eventually fall outside the window may still be the cause of inconsistencies. To counteract this, the window must be appropriately sized, in order to minimize the occurrence of scenarios where an element that needed to be remembered was irreversibly forgotten.

3.7.4 *Slack*

First described in [34], the slackness parameter is an extension to the sliding window model. It refers to the amount of previously inserted elements that may still be identified as present in the structure with heightened probability, even if they recently fell outside the window. The probability of it happening is said heightened because it is higher than the chosen false positive probability, applicable to elements that were definitively forgotten or never observed in the first place. In practical terms, in a probabilistic setting, for a window of size w and with a slack of size s, the w most recently observed elements must always be reported as present, considering they are within the window. The preceding s elements, that fall just outside the window, have heightened probability of being reported as present. The remaining elements, including unobserved elements and those that precede the last $w + s$, must always be reported as absent if no false positive is generated.

The degree of slack that certain sliding window solutions employ affects the exactitude of the size of the window. Hence, it relates to how fast a previously known element that falls outside the window can be forgotten. For some use cases, it may be useful to have the memory of known elements linger for longer. For example, the existence of slack is not an issue to the solution here proposed, as it is not necessary for the window to be of an exact size. Dots that are considered to be part of the slack are previously inserted dots. Therefore, reporting them as present with increased probability, even if they fall outside the

sliding window, is not a problem, since they would be previously known dots, even if using a representation of the complete infinite stream of dots.

3.7.5 *Existing Sliding Window Solutions*

Solutions implementing the sliding window or similar models can be divided into those based on Bloom Filters and those based on dictionaries. The related work review in [39] considers solutions to be Bloom Filter variants if they use k hash functions to choose k cells to update, with each cell being 1 or more bits wide, so it can be capable of holding counters or timestamps. One additional characteristic is the fact that results of insertions of different elements can end up being mixed in the same cell, by OR or ADD operations. Cited designs supporting the description include Counting Bloom Filters [15] and Generalized Bloom Filters [29]. On the other hand, dictionary-based approaches are described as using one or only a few hash functions to choose one cell. Within a cell there may be alternative storage locations, like the slots in a bucket, where content is placed, such as fingerprints or timestamps. Furthermore, the contents related to different elements are not combined and cells are kept separate. These solutions, revealing some resemblance to the first method Bloom described in [14], presented in section 3.4, are said to be essentially variants of hash tables that hold hashes representing elements, instead of the elements themselves. The cited examples are Cuckoo filters [26] and Morton filters [16].

These designs have uses in duplicate detection in streams [23], of which click fraud detection is a specific use case [32], representation of approximate caches [20] and others. The best performing solutions in various metrics are typically dictionary-based. Among them is [34], which utilizes a Backyard Cuckoo Hashing dictionary, and SWAMP [11], which improves over [30], combining a TinyTable [24] with a circular buffer. The TinyTable is used for representation and management of the sliding window, by means of queries, insertions and deletions. The circular buffer, its collaborator, is used to keep the window with an appropriate size and sliding, informing the TinyTable of which old element must be evicted when the window moves. Age-Partitioned Bloom Filters [39] are a competing design based on a Bloom Filter variant and are presented in the following section.

3.8 AGE-PARTITIONED BLOOM FILTERS

3.8.1 *Overview*

Age-Partitioned Bloom Filters (APBFs) [39] are a solution to the (w, s, p, e)-Sliding filter problem. A solution to this problem is applicable to an infinite stream of incoming elements $x1, x2, \ldots$ of a domain and aims to approximately represent those that were most recently

observed, while efficiently forgetting older ones. It is characterized by four parameters, which are:

1. w - the sliding window size;
2. s - the slack;
3. p - the probability-weighted slack;
4. e - the false positive probability.

As stated in section 3.7, the sliding window size w refers to how many of the most recently observed elements must be accurately remembered. The slack s is the number of elements that precede the last w that may still be reported as present, with a probability higher than the specified false positive probability e, chosen for the window. The false positive probability e is the probability by which an element absent from the window (and slack) is reported as present. It is therefore also known as the error probability. The apparent absence of the element may be due to either it never having been previously seen or having been forgotten in the meanwhile. The probability-weighted slack p is a parameter newly introduced in [39] and refers to the average number of elements that just left the window that may still be reported as present with probability higher than e. The regular slack s is different because it refers to how many elements that fall just outside the window are affected by the heightened probability of being reported as present, giving no information about how many of those may actually be reported as present on average. The example given presents a scenario where two different filters have a slack of $s = 100$ elements. Despite sharing this characteristic, one filter always reports those 100 as present, while another reports only 10 of them on average. Consequently, the representation of the window of the second filter is tighter, even if the slack made both solutions appear identical in that regard. Benefiting from its lower probability-weighted slack, the second alternative may use an even higher slack value, while still reporting fewer elements outside the window as present.

3.8.2 *Structure and Design*

The design of the bit space of APBFs follows a partitioned approach similar to that used in section 3.6, where the bit space is divided into sub-vectors, here known as slices. Over time, slices move within the filter, promoting aging of elements. Slices hold batches of elements named generations and, upon the beginning of a new batch (a generation change), the oldest slice is removed and a new one is added. Consequently, this fact allows making room for new elements and forgetting the oldest ones, maintaining the size of the filter and keeping the window sliding.

A filter of this kind is characterized by three structural parameters - k, l and m. An APBF is made up of a sequence of $k + l$ equally sized slices of m bits each, from s_0 to s_{k+l-1}. The first k slices, from s_0 to s_{k-1}, are the ones active for insertions. The remaining l slices, from s_k to s_{k+l-1}, are relevant for storage capacity, allowing the storage of older batches.

The input stream gets divided into different generations, with a new generation beginning after a certain number of insertions. When enough insertions were made, leading to a generation change, slices are said to age. In this process, the slices shift by taking the position of the subsequent slice in the sequence. Accordingly, slice s_0 becomes s_1, and the same happens for every slice but s_{k+l-1}, which as the last in the filter, is discarded. In addition, a new slice is placed in the first index, with all of its m bits set to 0. In practice, to avoid new allocations and moving the contents of the slices at every shift, a circular buffer is used and slices only move logically, by updating the index of the base slice and clearing the contents of the oldest.

Elements inserted in a slice should be available for accurate querying for as long as the slice lives in the filter. Given the fact that slices shift, the independent hash functions used in the filter's operations should be tied to each physical slice of bits, rather than a position in the sequence. The independent hash function that operates on slice s_i is identified by h'_i; in the original literature h_i denotes the hash function of bit array b_i. The designation b_i differs from that of s_i because it is the slice at physical index i in the circular buffer, whereas s_i is the slice at logical index i. As an example, the slice considered to be the first in a circular buffer of 5 entries may be the one at physical index 4. In this case, s_0 is b_4. When a physical slice goes from s_0 to s_1 and further, so does its fixed hash function, going from h'_0 to h'_1 and further.

3.8.3 *Operations*

In contrast with designs like [23], expressly dedicated to duplicate detection, APBFs allow for more flexibility by providing two separate basic operations:

- $\mathsf{add}(e)$, for insertions of elements in the filter;
- $\mathsf{query}(e)$, for querying their presence in the approximate window.

The insertion operation $\mathsf{add}(e)$ sets bits at locations $s_i[h'_i(e)]$ to 1, for every i such that $0 \leq i < k$. Due to the possible shifting of slices, queries must look for sequences of k slices containing the queried element, from s_0 up to s_{k+l-1}. Hence, a query returns true iff there is an initial logical index j, where $0 \leq j \leq l$, such that for all i, where $j \leq i < j + k$, $s_i[h'_i(e)] = 1$.

3.8.4 *Metrics*

Taking into account that there are k slices that are actively used for insertions, the first slice of the circular buffer and every subsequent slice to take its place will be used for the insertion of k generations. Alluding one more time to the conclusions in section 3.4, the fill ratio of a slice for optimum performance should be $1/2$ at most. Considering all this, its follows that the capacity of a slice should be divided equally between all the k generations that fill a slice, with each taking about $1/k$ of the slice's capacity. The number of insertions per generation is then defined by:

$$g = \lfloor \frac{m \times \ln 2}{k} \rfloor \tag{3}$$

Figure 3.3 can aid in the visualization of the design. The first k slices, as mentioned previously, are the ones used for insertions and, as such, are the ones identified as *filling*. Their perfectly staggered fill ratios, with values approximate to those shown in Equation 4, also allude to the fact that each generation takes a roughly equal portion of the capacity of each slice. When they attain the *full* status, slices become exclusively used for queries, since any further insertions have the potential to invalidate their optimum performance, by increasing the false positive probability above the desired value.

$$[r_0, r_1, \ldots, r_{k-1}, r_k, \ldots, r_{k+l-1}] = [\frac{1}{2k}, \frac{2}{2k} \cdots, \frac{k}{2k}, \frac{1}{2}, \ldots, \frac{1}{2}] \tag{4}$$

One other characteristic is shown in Figure 3.4 - the set of slices that hold the data of the sliding window. Up until and including phase (c), every generation previously inserted is remembered. However, in phase (d), one of the slices that held information relative to generation A was retired. By no longer being wholly represented, generation A is said to be *expiring*. In this status, generation A is no longer a part of the window. Despite this, as there are some lingering memories of it, supported by 2 slices in phase (d), there is still some chance that elements from generation A are reported as present, materialized if their queries generate a false positive in slice 2. As further insertions and slice shifts occur, memory of generation A weakens. This can be seen in phase (e), where reporting an element from A as present would require triggering a false positive from one more slice than it was in (d). Elements from generations not fully represented in the filter are part of what constitutes the slack. Their heightened probability of being reported as present depends on the proportion of the slices used in their insertion that still live. All of this is observable in Figure 3.3 as well, where the diagonal line separates expiring generations from those still completely remembered.

To identify the size of the sliding window, the authors in [39] ask to consider the state of the filter immediately after l shifts. In Figure 3.4, this is equivalent to what is depicted

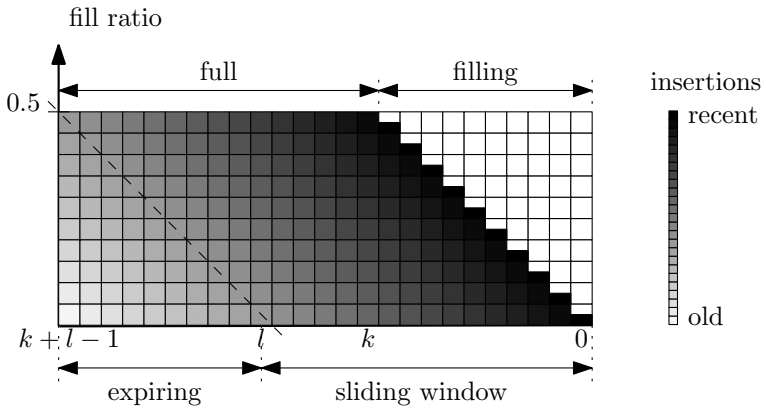


Figure 3.3: Maximum fill ratio of each slice, for an APBF with $k = 15$ and $l = 10$

in phase (c), prior to the insertion of generation C. In this state, the filter holds two whole generations, A and B, and is capable of holding an additional one, C. Since there have been l shifts, the initial slice of generation A is now at logical index l. Given that insertions span k slices, the last slice representing A coincides with the last slice of the filter, at index $k + l - 1$. Any further shifting, truncates the representation of A, bringing the filter from a state where it held 3 whole generations, A, B and C, to a state where it holds only 2 again, B and C, equivalent to the depiction of phase (d) without generation D. From this we can conclude that the amount of elements able to be completely represented in the structure fluctuates between l and $l + 1$ generations. As there is only the guarantee that l generations will be held in their entirety, the size of the window is said to be:

$$w = l \times g$$

Additional elements beyond those w that may still be completely remembered are considered part of the slack. The slack of a filter in a steady state is at its peak size just before a shift occurs, when the number of elements it holds is:

$$s = k \times g$$

From these values of w and s it is possible to infer that the normalized slack, relative to the size of the window w, is given by the ratio between k and l. The filter is said to be in a steady state after $k + l$ shifts have been performed, as is seen in phase (f), storing between l and $l + 1$ generations.

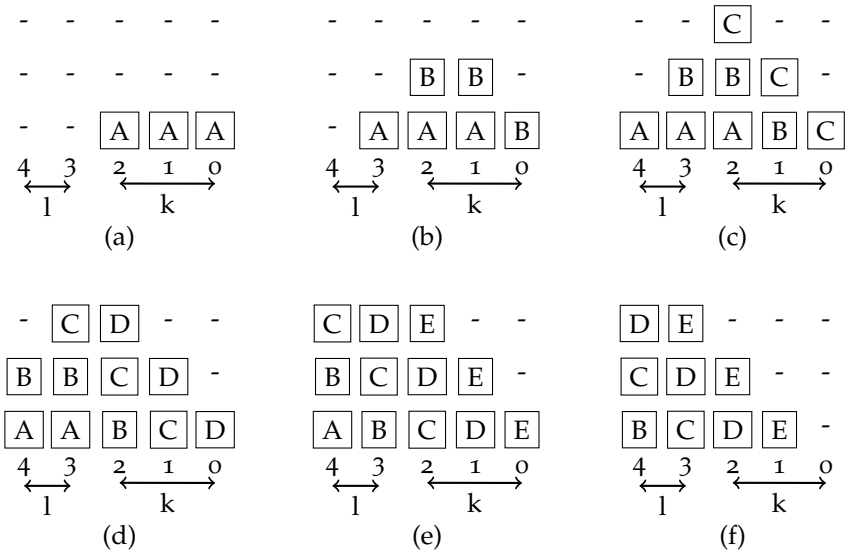


Figure 3.4: Filter evolution when adding elements and shifting.
Each letter (A,B,...) represents a generation of g elements. With $k = 3$ and $l = 2$.

3.9 DOTS

Elements to be shared in the system - either update operations or their effects, through states and delta-states - should be uniquely identified in the system. This identification allows testing knowledge of their existence at remote replicas, in order to determine if and how they should be delivered and potentially integrated into the local state. The construction of the identifiers, also known as dots, as presented in section 2.2, can be based on the source replica's own identifier and extended by appending a locally unique part to it, like the value of a counter incremented for every update. Alternatively, the IDs can contribute to the probabilistic dimension of the solution. As a means to achieve enough uniqueness in the system, IDs could be obtained by generating a fixed length sequence of bytes at random.

This latter option has the benefit of potentiating anonymity in the system, which is one of the main drivers of the work here proposed, in accordance with 3.2. Anonymity, for instance, can be a desired characteristic if the underlying distributed system has significant churn, which is not dealt with easily in existing solutions.

It may seem that the addition of another probabilistic facet to a system where stochastic mechanisms are already used can increase the error rate. Here, the error probability added by the random dot solution can be reduced by increasing the length of the randomly generated ID, or alternatively, the range of the random number generator. However, these random dots are intended for use in probabilistic data structures, as is the case in this dissertation. Considering the fact that probabilistic data structures represent their input elements probabilistically, whatever uniqueness they had upon insertion will be lost, even if

they were perfectly unique to begin with. Furthermore, this is true whether input elements are transformed into fingerprints, as they are in dictionary-like structures, or by separate and dispersed bits, as they are in Bloom Filter variants. The loss of uniqueness is manifested in collisions of outputs. It is assumed and expected that the loss of uniqueness generated by the probabilistic structures will be bigger than that caused by the random generation of identifiers, potentially by some orders of magnitude. Due to this, the increase in the error rate by using random dots is not expected to be noticeable.

One further point of consideration lies in memory consumption. Memory consumption should be comparatively limited in a probabilistic structure, regardless of it being a Bloom Filter variant or a dictionary-based alternative, since the amount of memory used to represent an inserted element does not depend on its size. In current solutions, particularly in the compaction scheme presented in subsection 2.2.1, the amount of memory consumed by the structure representing the causal context depends on two factors: the size of the identifiers in use and the number of replicas represented. The impact of the size of dots in the space complexity of a causal context is often not an issue, because the compaction scheme offsets any added burden, usually rendering the use of probabilistic structures unattractive. Nevertheless, for a sufficiently large number of replicas represented in the classic causal context, as suggested in section 3.2, there could be an advantage for stochastic methods, including random dots.

4

DESIGN AND IMPLEMENTATION

4.1 INTRODUCTION

Having presented the related work that covers the essential concepts of this thesis throughout chapters 2 and 3, the current chapter will offer an overview of the designs used and some implementation details of the proposed solution.

The following sections will open with an introduction to some data types whose designs depend on the existence of a dot context (section 4.2). The structures that hold dots in those designs will be presented as well in subsection 4.2.1 and additional remarks on the design of causal CRDTs appear in subsection 4.2.2. To make matters more concrete, a specific data type, the set, is chosen, and is accompanied by a description of operations and alternative designs pertaining to it (section 4.3). The description of the data type is followed by an analysis of the consequences of using probabilistic methods, touching on expected errors and limitations caused by the use of an Age-Partitioned Bloom Filter (APBF) as a dot context (in section 4.4 and 4.5.1). In relation to this analysis, the chapter includes implementation details of the stochastic structure in section 4.5, presenting some strategies that aim to mitigate the anticipated issues. The chapter ends with an additional design intended to produce improved results (subsection 4.5.3).

4.2 CAUSAL DATA TYPES

The original literature for δ-CRDTs [10] includes an introduction to a class of data types named causal CRDTs. The building blocks of the states of these data types, the way they work to produce the desired functionality, along with some relevant designs, are described in the following sections and subsections.

Among the initial designs described in [37], there were observed-remove sets and multi-value registers, in addition to the eventually consistent shopping cart presented in [22], in the context of Amazon's Dynamo. In spite of being functional, the analysis in [19] found them to be sub-optimal with regards to their scalability properties. One of these, a δ-state

based design of a set with tombstones, is shown in subsection 4.3.1. A further iteration in [13] removed the need for tombstones, allowing the use of a more efficient and compact representation of metadata state. Two δ-based designs of these optimized alternatives are defined in subsection 4.3.2. These designs continue the practice of tagging events or their effects with unique identifiers, which has been a feature of distributed data types for decades (as seen in [42]). These identifiers, also known as dots, are essential metadata and their management was a focal point of improvement for subsequent suggested designs.

4.2.1 *Dot Stores*

Dot stores, as the name suggests, are where dots are stored, along with other information specific to the data type. This concept was introduced in [10]. Together with causal contexts, described in section 2.2, they form the state of a causal CRDT. Event identifiers corresponding to relevant operations can be obtained from a dot store of a given node by the use of a function, dots, which given a dot store, returns the set of dots it holds. Figure 4.1 presents the definition of three instances of dot stores - a DotSet is a simple set of dots, a $\mathsf{DotFun}\langle V\rangle$ is a map from dots to some lattice V, and a $\mathsf{DotMap}\langle K, V\rangle$ is a map whose keys are elements from a set K and whose values are some dot store V. These can be used to build several data types, including causal counters, observed-remove sets and flags with both conflict resolution policies (add-wins/remove-wins for sets and enable-wins/disable-wins for flags), multi-value registers, and bags.

$$
\begin{aligned}
&\mathsf{DotStore} \\
\mathsf{dots}\langle S : \mathsf{DotStore}\rangle \quad &: \quad S \to \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\\
\mathsf{DotSet} : \mathsf{DotStore} \quad &= \quad \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\mathsf{dots}(s) \quad &= \quad s \\
\\
\mathsf{DotFun}\langle V : \mathsf{Lattice}\rangle : \mathsf{DotStore} \quad &= \quad \mathbb{I} \times \mathbb{N} \hookrightarrow V \\
\mathsf{dots}(s) \quad &= \quad \mathsf{dom}\, s \\
\\
\mathsf{DotMap}\langle K,\ V : \mathsf{DotStore}\rangle : \mathsf{DotStore} \quad &= \quad K \hookrightarrow V \\
\mathsf{dots}(m) \quad &= \quad \bigcup \{\mathsf{dots}(v) \mid (\_, v) \in m\}
\end{aligned}
$$

Figure 4.1: Dot Stores

4.2.2 *Causal CRDTs*

The state of causal CRDTs is a pair consisting of a dot store and a causal context, forming a join-semilattice. Different lattices and join operations for each of the dot stores presented in subsection 4.2.1 are defined in Figure 4.2.

In these pairs, if a dot is present in the causal context, but absent from the dot store, it means that at some point, at some replica, it was part of the dot store and removed in the meanwhile. This is the technique used in [13] in order to represent removals while avoiding the need for individual tombstones, generalized for different types of dot store. The definitions of the $\sqcup$ (join) operations are transparent in that regard. Used to merge state from two replicas, besides simply joining the causal contexts (with $c \cup c'$), each operation joins the different dot stores in specific ways, presented in the following.

When the dot store is a DotSet, the dots kept by a join in the dot store are the ones that are common to the stores of both replicas or those present in the store of a replica and absent from the context of the other. In this latter scenario, the absence from the causal context means that the dot is unknown, as neither its addition nor removal has been acknowledged. The fact that one of the replicas has that dot in its store (and context) indicates further knowledge of it in comparison with the other replica, thus justifying the presence of the dot in the resulting dot store.

The same principle is applied to a $\mathsf{DotFun}\langle V \rangle$, with the appropriate adjustments, given that a $\mathsf{DotFun}\langle V \rangle$ is a map. As such, mappings of dots exclusive to the store of one replica and unknown to the other are kept in the dot store resulting from the join, for the same reason the dots were kept in the DotSet scenario. For this kind of dot store, the value for a given dot can evolve independently over time at each replica. Taking into account that the value set was determined to be a join-semilattice, when joining the states from two nodes, whenever there is a dot common to both dot stores, the value associated with it in the resulting dot store is the join of its values in the two original stores - $m(k) \sqcup m'(k)$.

In the case of the $\mathsf{DotMap}\langle K, V \rangle$ there is a map from elements of some K to a dot store V that supports their presence in the data type. The join for a $\mathsf{DotMap}\langle K, V \rangle$ executes by performing, for each key in either replica, a join in the lattice $\mathsf{Causal}\langle V \rangle$, by pairing the value of each key, a dot store, with the causal context of the replica. From the result of the per-key join, only the first component - the dot store - is extracted, with it being the dot store that supports the key's permanence in the store. Consequently, the key-value pairing is only kept if the calculated value is not the bottom value $\bot_V$, which can be used to represent the value of an unmapped key. This method allows the disassociation of a composite embedded value from its respective key, avoiding the need for a per-key tombstone, by representing all dots from the composite value in the causal context of the node.

$$
\begin{aligned}
\mathsf{Causal}\langle T : \mathsf{DotStore}\rangle &= T \times \mathsf{CausalContext} \\
\\
\sqcup &: \mathsf{Causal}\langle T\rangle \times \mathsf{Causal}\langle T\rangle \to \mathsf{Causal}\langle T\rangle \\
\\
\textbf{when} \quad & T : \mathsf{DotSet} \\
(s,c) \sqcup (s',c') &= ((s \cap s') \cup (s \setminus c') \cup (s' \setminus c), c \cup c') \\
\\
\textbf{when} \quad & T : \mathsf{DotFun}\langle \_\rangle \\
(m,c) \sqcup (m',c') &= (\{k \mapsto m(k) \sqcup m'(k) \mid k \in \mathsf{dom}\, m \cap \mathsf{dom}\, m'\} \cup \\
& \quad \{(d,v) \in m \mid d \notin c'\} \cup \{(d,v) \in m' \mid d \notin c\}, c \cup c') \\
\\
\textbf{when} \quad & T : \mathsf{DotMap}\langle \_, \_\rangle \\
(m,c) \sqcup (m',c') &= (\{k \mapsto \mathsf{v}(k) \mid k \in \mathsf{dom}\, m \cup \mathsf{dom}\, m' \wedge \mathsf{v}(k) \neq \bot\}, c \cup c') \\
& \quad \textbf{where } \mathsf{v}(k) = \mathsf{fst}\,((m(k),c) \sqcup (m'(k),c'))
\end{aligned}
$$

Figure 4.2: Lattices for Causal δ-CRDTs

4.3 SETS

Sets are naturally suitable to be conflict-free replicated data types because they are ordered by inclusion ($\subseteq$) and have a well known operation to determine the least upper bound in union ($\cup$), enabling the states of such a data type to form a monotonic join-semilattice. Due to their simplicity, they are useful in the construction of other more complex data types, such as maps and graphs. This fact, as well as the diverse management of metadata in existing designs, makes sets an adequate data type to subject to analysis in this dissertation.

The basic operations capable of mutation considered for a set are addition, by using function add, and removal, through function remove. Given that these operations do not commute, a set cannot both be a CRDT and conform to the sequential specification of a set, even if there is the guarantee of a causally ordered delivery at each node. As a result, different designs may use different semantics, that determine the policy used for arbitration when a conflict arises. When an add and a remove for the same element happen concurrently, it may be desirable to suppress the effect of the removal, letting the addition prevail. This decision creates an add-wins set, where the element affected by the concurrent operations is seen in the set after the conflict's resolution. On the other hand, the opposite decision may be desirable, creating a remove-wins set, where an element is made absent when affected by conflicting operations.

Among a collection of other data types, an initial comprehensive study of CRDTs [36] presents some variants of sets. Included set designs are the Grow-Only Set (G-Set), Two-Phase Set (2P-Set), U-Set, Last-Writer-Wins Set (LWW-Set) and PN-Set, with some variants being mentioned. One additional design, for the Observed-Remove Set (OR-Set), is the highlight, due to its features. OR-Sets avoid some of the pitfalls of the alternatives by, for example:

- supporting additions and removals, unlike G-Sets;
- allowing the insertion of an element after its deletion, contrasting with the permanent deletion in 2P-Sets;
- avoiding issues with timestamp allocation in Last-Writer-Wins-based designs, that can be related with the unavoidable problem of clock drift between nodes.

4.3.1 *Tombstones and the Observed-Remove Set*

Some of the initial literature touching on CRDT sets [37, 36, 13] mentions designs derived from work by Wuu and Bernstein from 1984 [42], where a replicated log is used to support a replicated dictionary. One of the adaptations described is essentially a 2P-Set that assumes that elements are unique, a characteristic easily achievable by attaching globally unique identifiers to elements. This variant, known as U-Set, supports additions and removals. As such, its state-based version makes use of two G-Sets, A and R, which store, respectively, elements added and removed. The set of removed elements R, commonly known as the tombstone set, is needed to ensure that if a remove arrives earlier than its respective add to a downstream replica, its effect is still registered. The query to determine the value of the set returns the set difference $A \setminus R$.

The Observed-Remove Set (OR-Set), introduced in [36], is an identical construction. The essential property of OR-Sets is that each added element is tagged with a dot. Through this unique identification of added elements, a remove of an element may only affect its respective entries in the dot store with a dot observed at the node of origin of the operation when the removal was executed. It is this characteristic that gives the set the "observed-remove" name.

Figure 4.3 presents a δ-state CRDT for an OR-Set with tombstones. The conflict resolution gives preference to additions over removals, making it an add-wins set. The state of this OR-Set is a pair of a dot store and a set of tombstones T. In this particular specification, the chosen dot store is a DotMap. However, in an analogous fashion to what can be seen in subsection 4.3.2, a DotFun would be a suitable substitute.

The function used to query the value of the set, by obtaining the set of elements present in the data type, is $\mathsf{elements}((m,t))$. Given that the elements in the set are the keys of DotMap m, they are obtained by $\mathsf{dom}\, m$.

The insertion of elements is made through $\mathsf{add}_i^\delta(e,(m,t))$. As previously mentioned, each element added is tagged with a dot. Taking into consideration that elements are mapped to a DotSet in this design, a newly added element e is paired with the singleton set containing the unique identifier generated for it, given by $d = \{\mathsf{unique}_i()\}$. The dot, as suggested in the previous chapters in section 2.2 and section 3.9, can be made up of the replica identifier i and the value of a monotonically increasing counter or a random number from a sufficiently large set of possible values, thus ensuring uniqueness with a high probability. The entry associating element e with its respective dot set d gets added to the DotMap, which is reflected in the produced delta, that includes a dot store with a single entry: $\{e \mapsto d\}$. Assuming that the new insertion produces an overwrite, every entry for the same element, associated with previously existing dots, is to be marked as removed. Hence, these obsolete dots, that supported the presence of the element prior to the new insertion, are added to the tombstones' set, which can also be seen in the delta state as $m(e)$. Otherwise, if element e did not have a previous entry in the map, its associated value $m(e)$ will be the bottom value $\bot_V$, producing an empty set, as expected.

The removal of an element e, $\mathsf{remove}_i^\delta(e,(m,t))$, clears its respective entry from the dot store of the source replica. As a result, the dot store of the delta state produced by the operation is simply the empty DotMap ($\{\}$). As the name of the design implies, all the observed dots that previously supported the element's presence in the set, and only those, will be considered removed from the set. This is observable in the placement of those dots ($m(e)$) in the tombstones' set of the resulting delta. The procedure for the $\mathsf{clear}_i^\delta((m,t))$ operations is analogous, but applicable to every element in the set and not just one in specific. Hence, the set of tombstones of the delta is the set of dots in the DotMap, which is the union of every DotSet in it.

$$
\begin{aligned}
\mathsf{TOR\text{-}Set}\langle E\rangle &= \mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle \times T \\
\mathsf{add}_i^\delta(e,(m,t)) &= (\{e \mapsto d\}, m(e)) \quad \textbf{where } d = \{\mathsf{unique}_i()\} \\
\mathsf{remove}_i^\delta(e,(m,t)) &= (\{\}, m(e)) \\
\mathsf{clear}_i^\delta((m,t)) &= (\{\}, \mathsf{dots}(m)) \quad \textbf{where } \mathsf{dots}(m) = \bigcup\{v \mid (k,v) \in m\} \\
\mathsf{elements}((m,t)) &= \mathsf{dom}\, m
\end{aligned}
$$

Figure 4.3: δ-CRDT add-wins set with tombstones for replica i

Given that the set of tombstones is needed in an environment without guarantees of causal delivery, it is possible for it to accumulate deletions over time, growing unconstrained.

Wuu and Bernstein [42] acknowledged the issue and proposed a mechanism of garbage collection in order to avoid unbounded growth. In the proposed algorithm, replicas record the clock value for the latest acknowledged mutating event by each replica, for every source in the system. This way, every replica knows what is the lower bound of knowledge for every other replica in the system, given a source of updates. Whenever an entry's clock value falls behind this moving frontier, it is confirmed to have been delivered at all nodes in the system. Since every replica is expected to persist updates in durable storage, globally acknowledged entries are pruned, as they no longer serve any purpose.

However, as alluded to in a different context, in section 3.2, garbage collection is hampered by some factors. In case anonymity is a desired feature, as it is in this thesis, accurately tracking the knowledge of each replica becomes naturally impossible. Furthermore, the acknowledgement method makes garbage collection dependent on network delays and brings communication and processing overhead. Holding in memory the knowledge information for every pair of replicas is also a problem for large enough systems, as well as systems that exhibit significant churn. In this latter scenario, in case a node crashes or gracefully leaves, every subsequent update it does not see cannot be garbage collected in the system, degrading efficient storage of replica state. Taking into account that a node that becomes absent might not return, the matter is made worse. An additional drawback is the fact that disperse dots in the tombstone set are not well suited for compaction schemes.

4.3.2 *Optimized Observed-Remove Sets*

With the aim of improving the main pitfall of the original OR-Set - the management and resource utilization of its set of tombstones - [13] presents an optimized Observed-Remove Set design, later deemed to fall in the category of causal CRDTs by [10]. Being a causal CRDT, the optimization of this variant is based on the principle presented previously in subsection 4.2.2: a dot registered in the causal context and simultaneously absent from the data type's dot store is considered removed. Similarly to the tombstones' set in the original OR-Set design, the causal context holds knowledge of an increasing amount of dots over time. Nonetheless, due to the contiguity of the represented dots, a causal context can be appropriately compacted, as was stated previously in subsection 2.2.1. The set of tombstones does not benefit from the same optimization because deleted dots may be dispersed over the space of the unique identifiers.

Figure 4.4 presents the specification of an optimized Observed-Remove Set with an add-wins policy for conflict resolution and a $\mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle$ as the designated dot store. Even though it concerns a δ-CRDT, it can be trivially used as a simpler state-based design.

Querying to obtain the elements in the set uses the exact same method presented previously in subsection 4.3.1 for the specification of the non-optimized OR-Set in Figure 4.3. This is expected, given the use of the same type of DotStore.

The insertion of elements, made through function $\mathsf{add}_i^\delta(e,(m,c))$, is also similar to the non-optimized version. Nevertheless, there is one difference. Considering that the causal context c contains every dot known by replica i (instead of just removed ones), the new dot must also be added to it, in addition to the possible pre-existing ones. This can be seen in the causal context of the delta state generated by the update: $d \cup m(e)$. By making use of the optimization, the removed dots can be identified as such in the delta due to their absence from the DotStore, which expectedly only contains a single entry: $\{e \mapsto d\}$.

The function for removal of an element e - $\mathsf{remove}_i^\delta(e,(m,c))$ - is identical to its non-optimized counterpart. As such, the element's entry is removed from the dot store of the source replica, producing an empty DotMap in the delta. Any existing dots supporting the presence of e in the set at the instant of its removal at the source replica, represented by $m(e)$, are placed in the delta's causal context. Due to the fact that the causal context already knows of the dots removed, either as a result of a previous local insertion or a merge with state from remote replicas, there are no changes to the state of the source replica. Once again, by being absent from the dot store and present in the dot context, removed dots can be seen as such by remote replicas who receive the full state or delta states from the source. The same applies to the set clearance function $\mathsf{clear}_i^\delta((m,c))$.

$$
\begin{aligned}
\mathsf{AWSet}\langle E\rangle &= \mathsf{Causal}\langle\mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle\rangle \\
\mathsf{add}_i^\delta(e,(m,c)) &= (\{e \mapsto d\}, d \cup m(e)) \quad \textbf{where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{remove}_i^\delta(e,(m,c)) &= (\{\}, m(e)) \\
\mathsf{clear}_i^\delta((m,c)) &= (\{\}, \mathsf{dots}(m)) \quad \textbf{where } \mathsf{dots}(m) = \bigcup\{v \mid (k,v) \in m\} \\
\mathsf{elements}((m,c)) &= \mathsf{dom}\, m
\end{aligned}
$$

Figure 4.4: δ-CRDT add-wins OR-Set for replica i based on a DotMap

The optimized OR-Set has an alternative specification based on a $\mathsf{DotFun}\langle E\rangle$ dot store, presented in Figure 4.5. The differences in the specification are solely related to the different DotStore.

A $\mathsf{DotFun}\langle E\rangle$ is a map whose keys are dots and whose values are elements of a set of elements E, a join-semilattice. Accordingly, the single entry in the dot store of the delta produced by the add operation maps the most recently created dot with the added element.

The removal functions also show different ways of obtaining dots. Starting with the $\mathsf{remove}_i^\delta(e,(m,c))$ function, the set of dots tied to e are not simply its value in m, as it was in the DotMap version. Instead, the observed dots are the keys of all entries whose values

are equal to e: $\{v \mid (k, v) \in m \land v = e\}$. The procedure to determine the dot context of the delta for a set clearance is simple. Taking into consideration that the operation applies to all elements, the observed dots are simply the set of keys of map m, or, in a different notation, $\mathsf{dom}\, m$.

Finally, one last observable difference lies in the $\mathsf{elements}((m, c))$ function. Once again, keeping in mind that the elements of the DotFun are the values of map m, they can be obtained by $\mathsf{range}\ m$. Due to it being possible that different dots may support the same element's presence in the set, as would be the case if there were concurrent insertions of it, the range of m may contain repeated elements. To correct this, when taking the value of the set, only unique elements are taken into consideration: $\mathsf{unique}(\mathsf{range}\ m)$.

Beyond the specification, the differences between the DotMap and DotFun designs are found in other details of the implementation and design. For example, when removing an element, to obtain the set of dots associated with it, the DotFun alternative forces an iteration through every entry in the dot store, whereas the equivalent in the DotMap case is a simple lookup. This matter is worthy of further consideration because addition operations for an element perform the deletion of its possible previous entries, in order to coalesce repeated adds and avoid polluting the dot store with redundant dots supporting the same element. Therefore, the choice of a DotFun as a dot store may become problematic, depending on the volume and distribution of operations the instances of a data type see; consequently, the size of the dot store may be a reason for concern.

Even so, it can be advantageous to use the DotFun version. Leveraging the fact that the values of a $\mathsf{DotFun}\langle E \rangle$ are part of a join-semillatice, the value for a each dot can vary over time, independently at each replica. Whenever states merge, if the local and remote values for a given dot are different, they can simply be joined by calculating their least upper bound.

$$
\begin{aligned}
\mathsf{AWSet}\langle E \rangle &= \mathsf{Causal}\langle \mathsf{DotFun}\langle E \rangle \rangle \\
\mathsf{add}_i^\delta(e, (m, c)) &= (\{d \mapsto e\}, \{d\} \cup c) \quad \textbf{where } d = \mathsf{next}_i(c) \\
\mathsf{remove}_i^\delta(e, (m, c)) &= (\{\}, \{v \mid (k, v) \in m \land v = e\}) \\
\mathsf{clear}_i^\delta((m, c)) &= (\{\}, \mathsf{dots}(m)) \quad \textbf{where } \mathsf{dots}(m) = \mathsf{dom}\, m \\
\mathsf{elements}((m, c)) &= \mathsf{unique}(\mathsf{range}\ m)
\end{aligned}
$$

Figure 4.5: δ-CRDT add-wins OR-Set for replica i based on a DotFun

4.4 GOING PROBABILISTIC

The designs presented previously are typically fully deterministic. One possible exception to this would be the use of random dots, which is contemplated in the specification for the TOR-Set in Figure 4.3. However, due to the lack of ordering and contiguity these exhibit, they are not adequate for compact representations of the dot context, thus severely compromising their usefulness in established designs.

By contrast and as discussed in section 3.9, random dots are not a problem for probabilistic dot contexts, since their size does not influence memory consumption, and potential collisions are not expected to have any significant effect on the rate of errors the implementation produces. As such, they are the preferred option in a probabilistic setting for their ability of providing anonymity to the distributed instances of data types.

Following the intentions expressed in Chapter 3, this thesis proposes to study the potential of probabilistic causal CRDTs. The state of such data types is given by the following definition, meaning that the CausalContext in its traditional form is replaced by an Age-Partitioned Bloom Filter.

$$\mathsf{Causal}\langle T : \mathsf{DotStore}\rangle = T \times \mathrm{APBF} \tag{5}$$

4.4.1 *Consequences*

The major consequence of having an APBF serving as a causal context is forgetting dots, since only a predefined number of the most recently registered dots are remembered. With no loss of generality, in the case of the set, a dot in the dot context can be in one of two different categories: it is either supporting the presence of an inserted element, or has been the target of a removal or equivalently, an overwrite. Dots in the first category are still remembered in the dot store, so their absence from the context can be overcome. As a result of no longer being in the dot store, dots in the second category do not benefit from the same leniency. Whenever a dot is locally removed, whether it results from an actual removal or an overwrite, memory of its removal must last long enough to allow its propagation to remote distributed objects. In other words, the data of a removed dot must stay within the region of the APBF that is used for synchronization at least until a moment of synchronization, so that remote states can incorporate the change. The possible designated synchronization regions of the APBF are to be discussed in the following, in subsection 4.5.2. Depending on the characteristics of the synchronization and assuming the graph of the overlay network is connected, a replica might be able to forget about the removed dot after synchronizing, without affecting overall consistency. By doing this, the source replica forces the responsibility of disseminating knowledge of the removed dot on the remote replicas.

One other less impactful consequence is the possibility of seeing false positives. When querying the presence of an absent element in a stochastic structure like an APBF, there is a small, tunable chance of it being falsely reported as present. These infrequent events have negative consequences for the correction of the mechanisms used to achieve consistency, to be presented in subsection 4.5.1.

4.4.2 *Synchronization*

Synchronization is the mechanism by which nodes communicate to attain consistency between their distributed objects and is powered by the join operation specified in Figure 4.2. Through a join ($\sqcup$), being elements of a join-semilattice, the states of a causal CRDT can be merged by calculating their least upper bound.

In order to clarify the operations needed from a causal context, algorithm 3 distances itself from the mathematical notation of the aforementioned specification to follow a more imperative approach, presenting the join algorithm for an optimized OR-Set with a $\mathsf{DotFun}\langle E\rangle$ as a dot store. The choice of that particular dot store is not limiting, considering the operations required from the dot context are the same for the join algorithm of a $\mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle$ design. Both appear in subsection 4.3.2.

```
1  on join((m, c), (m′, c′))
2      foreach (d, e) ∈ m do
3          if d ∉ m′
4              if d ∈ c′
5                  remove(d, m)
6          else
7              m(d) = m(d) ⊔ m′(d)
8      foreach (d′, e′) ∈ m′ do
9          if d′ ∉ c
10             insert((d′, e′), m)
11     c = c ∪ c′
```

Algorithm 3: Join algorithm for an OR-Set with a DotFun as the dot store

The join algorithm starts by iterating through each and every entry in the local dot store, a map m with dots as keys and elements as values (line 2). If the locally stored dot d is not present in the remote dot store m' (line 3), it may be in one of two phases in the remote state: either it is unknown, or is known, having been removed at some point. To allow a decision to be made about the entry of the current iteration, the remote causal context c' is queried at line 4. If d is present in c', there is now enough information to conclude that the remote replica has further knowledge about the dot, since it knows of its removal. Consequently, the entry is removed from the local state (line 5). Otherwise, if d is unknown to the remote

state, it is possible to infer that the local knowledge pertaining to that dot is the most current. In that case, nothing is done about the current entry locally. Had the dot been reported as part of the remote dot store at line 3, it could happen that the remote value tied to the dot was different than the local one. In this scenario, leveraging the fact that the values in a $\mathsf{DotFun}\langle E\rangle$ are elements of a lattice E, the local entry for the dot may update its value by taking the result of the least upper bound between the local and remote values, as seen in line 7. This happens by means of another join, defined for lattice E.

Having dealt with the local entries, it is then necessary to iterate over the remote entries (line 8), in an effort to determine if the remote state contains any new entry to integrate locally. With that in mind, each remote dot d' has its presence checked in the local dot context c (line 9). Regardless of its presence in the local dot store m, if d' is in the causal context, it is possible to infer that the local state has at least as much knowledge of that entry as the remote; it may even be more up to date, by knowing of the entry's subsequent removal. Nevertheless, if d' is locally unknown, (d', e') must become a new entry in m, which happens due to the insertion at line 10.

Finally, the local and remote contexts are merged (line 11), with the local context subsuming the remote. The main reason for merging contexts is recording dots of deleted entries unknown to the local state, thus ensuring the propagation of removals, which was not possible by iterating the local and remote dot stores. In doing so, if the local replica receives another remote state including such a dot in its dot store, it will know of its removal already. Additionally, if the dots of possible new entries, inserted at line 10, were not introduced to the context by the insert function, the merger of contexts will aid by locally registering new additions.

From algorithm 3 and its subsequent description, it can be seen that no less than two operations are needed from a causal context: one to query the presence of dots, supported by lines 4 and 9, and another to merge it with another, as observed at line 11. There is the additional need for an individual insertion operation. That is what happens in a set when a single element is added and what could happen around 10, to insert dot d' of the new entry into the local context c.

4.5 The Age-Partitioned Bloom Filter

As the substitute of the traditional deterministic causal context, an APBF or any other probabilistic structure that aims to provide such functionality must take into account the necessary operations, identified at the end of subsection 4.4.2. Considering the inexact nature of stochastic methods, the use of an APBF as a dot context generates an expected volume of errors (shown in subsection 4.5.1), which must be minimized in order to negatively affect the consistency between replicas as little as possible.

The essential operations of insertion and querying are provided by an APBF, as well as any other similar approximate membership alternative. Although the union functionality is indispensable, a statement justified by line 11 of algorithm 3, probabilistic structures do not usually allow such a feature. To counter this inadequacy, subsection 4.5.2 presents some union strategies devised for the APBF.

4.5.1 *Expected Errors*

By knowing the properties of APBFs and from the uses of causal contexts in algorithm 3, it is possible to present some expected errors. Some events and their consequences are considered to be errors because they diverge from the behaviour of CRDTs with deterministic causal contexts. This divergence affects the synchronization process and causes differences in consistency.

The following definitions are used to allow distinction:

- a divergence of state between the probabilistic and deterministic versions of a replica is said to be an intra-replica inconsistency;
- a divergence of state between replicas is called an inter-replica inconsistency.

Because there is no reliable way to distinguish entries resulting from intra-replica inconsistencies from regular and correct entries, as synchronization progresses, they propagate to remote replicas. As a consequence, replica states can be consistent between themselves (therefore, inter-consistent) while diverging from the state they would have were they using deterministic dot contexts (hence, intra-inconsistent).

Querying the Remote Context

Consider the query of the remote context c' to determine the presence of local dot d at line 4.

If d is reported as absent from c', it may be in one of two situations: either it is legitimately unknown, or has been forgotten since its insertion, due to falling outside of the sliding window. In the former scenario, no problem arises. In the latter, given that the dot is not in the store (line 3), it is possible to infer that the remote replica has acknowledged the removal of the dot at some point. As a consequence of its reported absence, dot d will not be removed from the local dot store as it should (line 5), since the local replica cannot conclude that the remotes state holds more current knowledge of the dot. Whereas the traditional deterministic structures would have deleted the element's entry at this point, in the probabilistic scenario, the local replica will hold an additional entry, which is an intra-inconsistency. To suppress this kind of issue, replicas can either synchronize more often or have their APBF-based causal contexts hold dots for longer, by increasing their capacity, for instance.

On the other hand, if d is reported as present in c', it may be due to one of two different reasons: either the dot is legitimately present in the filter's window, having been the target of a previous insertion, or its query produces a false positive. In the first case, the dot's entry is rightfully removed from the local dot store m. In the second, if dot d produces a false positive and is completely unknown by the remote state, the dot's entry is wrongly removed. To reduce the frequency of this kind of inconsistency the false positive probability must be adjusted at the time of construction of the APBF. However, if d generates a false positive but was at some point in the filter's window, the false positive helps, by removing an entry that should be removed, supported by its absence from the remote store (line 3).

Querying the Local Context

The error analysis continues by shifting the focus to the query of the local context at line 9.

If the query of a remote dot d' in the local causal context c determines that it is absent, once again, it may be completely unknown to the remote state or a forgotten dot. If the former is the true scenario, the insertion at line 10 is justifiable and correct. Otherwise, if reality aligns with the latter, the dot's entry is set to either be overwritten or reappear erroneously in the local state. To avoid overwriting existing entries for dots forgotten from the context while still supporting an element in the store, the local dot store m should also be queried when using a probabilistic structure like an APBF. With this adjustment the condition in line 9 becomes $d' \notin c \wedge d' \notin \mathsf{dots}(m)$. The limitation of occurrences of this inconsistency relies one more time on replicas sharing their knowledge more often or implementing measures in an effort to hold dots for longer, such as increasing the capacities of their APBFs.

Another possibility is the causal context reporting dots as present, which has the consequence of no insertion happening. Once more, there is a chance for a false positive. As such, dots that were previously inserted in the dot context are correctly not reinstated. This is the case regardless of the dot still being part of the window or slack of the APBF or having been forgotten and triggered a false positive. This last scenario is another instance of a false positive correcting the normal loss of memory of the filter. However, intra-inconsistencies can still happen for completely unknown dots. By setting off a false positive, a previously unobserved dot d' mistakenly evades insertion at line 10, depriving the local state of an entry. The way to minimize this issue is to set a lower false positive probability for the filter.

4.5.2 *Union Strategies*

A union operation that allows the local causal context to subsume knowledge of a remote one is crucial in the continuous process of striving to achieve inter-replica consistency, as seen in line 11 of the join algorithm. In spite of being widely used to represent sets, approximate membership structures similar to the APBF do not implement a fundamental set feature:

a union operation to combine instances. Among the reasons for this omission is the lack of flexibility of the underlying structures combined with the loss of information about the contained elements, a consequence of the probabilistic approach.

In view of the choice of a Bloom Filter variant to serve as the causal context, one may decide to keep matters simple and do the obvious. Assuming the hash functions and structural parameters (size, number of slices, etc.) are exactly the same between the filters to be combined, a simple bitwise OR operation combines the contents of filter cells with matching indices. In an original Bloom Filter (presented in section 3.4), with just one hash area shared by a number of hash functions, the zone to merge corresponds to the whole filter. Alternatively, partitioned Bloom Filters, equivalent to a non-dynamic version of the Scalable Bloom Filter from 3.6, are divided into slices. In these constructions, slices are individual zones that have to be matched according to their dimensions and fixed hash functions. The number of slices of both filters should also be the same, so that each slice has a match, allowing the fill ratios of slices of the same filter to grow roughly evenly.

Matching Zones

For partitioned Bloom Filters, the combination of contents of two slices must be done within some boundaries. To illustrate what is needed from a working solution, consider two partitioned Bloom Filters PBF_1 and PBF_2, one slice from each $\mathsf{PBF}_1[i]$ and $\mathsf{PBF}_2[j]$, at indices i and j respectively, and their corresponding hash functions h_1 and h_2.

The reason to combine union zones of matching hash function is to allow previously inserted elements to be accessed, while not polluting the bit space. If slice $\mathsf{PBF}_1[i]$ sets and queries bits with a hash function h_1 that is different from hash function h_2 of slice $\mathsf{PBF}_2[j]$, whatever bits it deterministically associates with a given input element will be very likely different than those of h_2. Accordingly, when merging both slices with a bitwise OR, the resulting slice will have a subset of bits set by h_1 and another by h_2. Queries in such a setting become problematic, because with just one hash function, the resulting slice will not be able to determine how to access the set bits of every element contained in each of the merged slices.

Take, for example, the setting in Figure 4.6; two slices of the same size hash their input elements with different hash functions - one with BLAKE2s and the other with SHA-256. Although the bit representing element D is set at different locations by the different hash functions, the bit for T is coincidentally mapped to index 6 by both; element P is exclusive to the first slice. Therefore, a slice resulting from the bitwise OR between the two depicted would luckily have no trouble retrieving the bit set for T, whether it used BLAKE2s or SHA-256 to hash elements. Nevertheless, the mismatch between set bit positions is much more common, occurring more frequently the bigger slices are. In this case, the resulting slice would have bits 0, 2 and 4 set with the contents of P and D, but would only be able

to query the bit tied to P if BLAKE2s was its hash function. If this was in fact the chosen function for the slice, the bit set by SHA-256 at index 0 would not be tied to element D, simply polluting the bit space, increasing its fill ratio and thus contributing to a higher false positive rate.

Had the hash function of both initial slices been the same, the slice containing the combined results could simply use an equal hash function $h = h_1 = h_2$ and capably query set bits from both its ancestors.

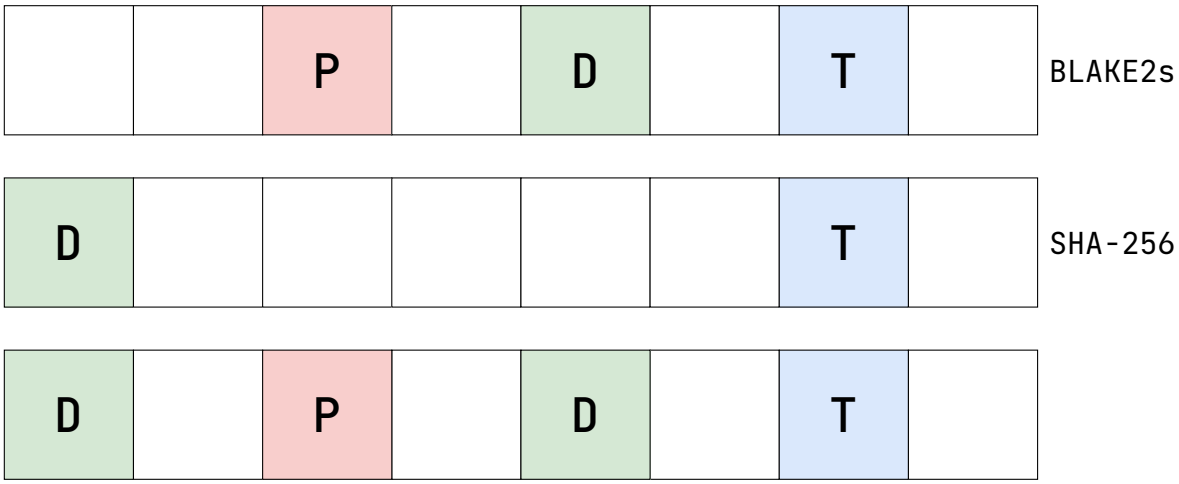


Figure 4.6: Merger of slices of the same size with different hash functions

Another factor to consider is the size of slices. If $\mathsf{PBF}_1[i]$ and $\mathsf{PBF}_2[j]$ have the respective sizes of n and m, h_1 will set bits at indices $[0 \dots n)$ and h_2 in $[0 \dots m)$. If $n \neq m$, even if h_1 and h_2 produce hashes in an identical way, whenever they map the hash of an input element to an index of their own slices, the index selected by one of them may be different or out of range for the other.

For instance, Figure 4.7 presents the states of two slices of sizes $n = 6$ and $m = 8$ (unaligned relative to the size of the hash), after the insertion of three strings: P, D, and T. In spite of using the same function to hash incoming elements, SHA-256, the hashes are mapped to ranges of different sizes. Aside from the possibility of being mapped to different indices, which happens with all of the elements in the example, an index set by h_2 may be out of reach by h_1. This is observable for string T, which is mapped by h_2 to index 6, outside the interval of indices $[0 \dots 5]$ reachable by h_1.

Hence, it is imperative to match slices of equal size.

Static Saturation

Even with perfectly matching union zones, there is a monotonic increase in the number of filled cells, provoked by continuous individual insertions and unions with the bitwise OR

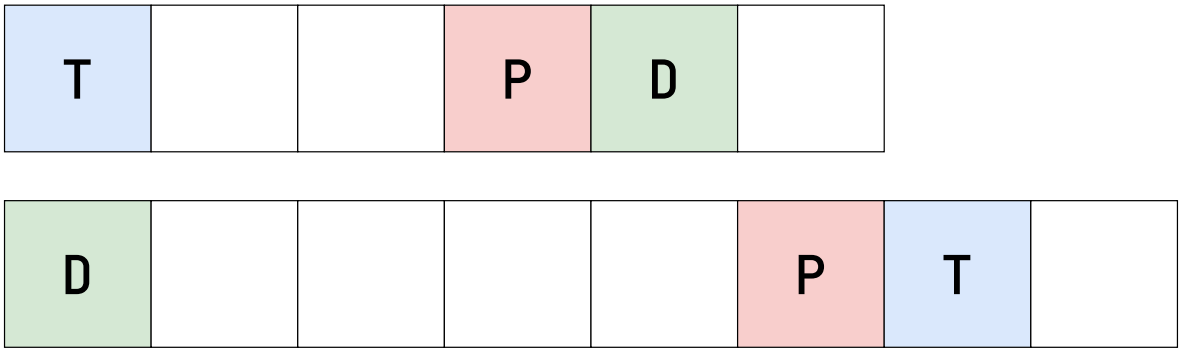


Figure 4.7: Merger of slices of different sizes with SHA-256 hashing

between filters. Consequently, static Bloom Filter variants eventually reach their point of saturation, beyond which the probability of false positives can increase to values that render the filter unusable. The fill ratio of 1/2 is the maximum for a bit space before performance becomes degraded past the level promised for a given set of parameters chosen upon the filter's formation, as stated towards the end of the discussion in section 3.4.

In the face of this problem, the Bloom Filter implementation in Guava [2], a popular Java library by Google, simply requires the caller of the merging method to guarantee that the input does not cause the absorbing filter to be overfilled. The method's documentation states:

> Callers must ensure the Bloom Filters are appropriately sized to avoid saturating them.

In such an environment, it would be impossible to use static filters for an unlimited duration. Conversely, owing to their dynamic nature, Age-Partitioned Bloom Filters can be adapted to continuously operate while enabling partial union of contents.

Strategies

Three possible strategies devised for the APBF combine the following subsets of its contents:

- All slices;
- Active slices;
- Current generation.

General Requirements

For the purpose of having a one-to-one match between the slices of the local and remote APBFs, all of these solutions require that the zones to be combined are composed of slices

with distinct hash functions. Additionally, in agreement with the conclusions of the previous discussion at 4.5.2, both the local and remote zones must use the same set of hash functions and slices of the same size, so as to eliminate the possibility of polluting the bit space with unreachable set bits. In other words, if the merge region of a local APBF is made up of s slices, each of them must use a distinct hash function in $\{h_1, \ldots, h_s\}$. If the size of each of the slices is m, the merge region of the remote APBF must equally have s slices of size m; moreover, the set of hash functions these use should also be $\{h_1, \ldots, h_s\}$.

Merge Zone Description

The merge zone in the strategy that proposes the combination of all slices corresponds to the complete filter. As such, it requires the APBF not to reuse hash functions internally, meaning that each of its slices must be tied to a different hash function.

In contrast, the local merge zone in the strategies that merge only the active slices or only the current generation is formed by the set of active slices. Therefore, within the sequence of k active slices, there are k distinct hash functions. For the remaining l, APBFs merging with any one of these strategies can employ the reuse of hash functions, which allows reusing computed hashes in membership queries.

As their descriptions suggest, the difference between the active-slices and current-generation strategies is that in the former a local active slice subsumes the content of a matching remote active slice, while in the latter a local active slice merges only the content of the current generation of its remote match. Knowing which of the set bits of an active slice correspond to the current generation is made possible by storing a duplicate bit space for every active slice. With that purpose in mind, insertions affect both the slice itself and the respective current generation slice. Upon generation change, the current generation slice is cleared, guaranteeing that older set bits are not included in future content mergers.

Union Process

During the union process, for each slice in the merge zone of the local APBF, a matching slice is searched in the merge zone of the remote APBF. In practice, the search looks for slices with matching hash function seeds. After a match is found, the local slice absorbs the contents of the remote. Expectedly, the bitwise OR between the slices increases the ratio of 1s of the absorbing slice (assuming the absorbed one is not empty), thus increasing its false positive probability and, ergo, of the overall filter.

In conformity with Figure 3.3, each active slice, of logical index 0 to $k-1$, has a maximum value for the fill ratio it must not exceed. The remaining l, in both the active-slices and current-generation strategies, are only used for queries and keep the fill ratio they had upon retirement from the set of active slices. In the remaining all-slices union strategy, since each

local slice is combined with a remote match, the fill ratio of an inactive slice can increase for as long as it stays in the filter.

Reiterating Equation 4, given that an active slice might have to store up to k generations of elements without going over the optimal maximum fill ratio of $1/2$, each generation must fill at most $1/k$ of that available $1/2$. Accordingly, the maximum fill ratios for the slices affected by insertions are the following:

$$[r_0, r_1, \ldots, r_{k-1}] = [\frac{1}{2k}, \frac{2}{2k}, \ldots, \frac{k}{2k}]$$

Whenever it is possible for fill ratios to change, whether by the actions of individual insertions or combinations of slices, the first k slices are checked. If the fill ratio of a given slice exceeds the maximum for its logical index, the APBF must shift its slices an appropriate amount of times, so that the invariant maximum fill ratios are preserved.

Fill Ratios

Since singular insertions can change at most 1 bit in each slice, they are more likely not to cause any shifting. The combination of slices, contrastingly, is capable of setting many new bits in the local slice, which makes this operation more susceptible to cause shifting. Slices shift independently at each APBF, depending on the volume of operations and synchronizations the corresponding replica sees and performs. Due to this fact, matching slices may be in different logical indices, and therefore, may have different fill ratios.

Let us consider different fill ratios for the remote slice. If the remote has a low fill ratio, the fuller the local match is, the least affected its fill ratio is by the union of contents. In any case, if shifting occurs at all, the number of shifts the APBF must perform should not be too high. Alternatively, if the remote has a high fill ratio, it is very likely to cause a shift, regardless of the fill ratio of the local absorbing slice. Even so, if the local match has a low fill ratio, its combination with the remote will set more new bits than it would on a fuller local slice, where more 1s would already be set. The bigger the difference between the fill ratios, the greater is the number of needed shifts that move the local slice to a logical index where it respects the maximum fill ratio invariant.

A big number of shifts is an undesirable consequence, because shifting is the process that allows forgetting elements. In turn, by losing memory of multiple generations at once, it becomes more likely that inconsistencies occur such as the ones described in subsection 4.5.1. Beyond the amplitude of the shifting, another property that must be minimized is shifting frequency. Lowering the shifting rate contributes to forgetting elements less often. By holding them for longer, there is more time to synchronize, which may lead to fewer inconsistencies. Consequently, two policies are desirable from a union strategy, so as to avoid shifting as much as possible:

- try to make each local slice subsume as few new bits as possible;
- attempt to provide small differences between fill ratios of matching slices.

Strategy Analysis - Whole Filter

The whole-filter union strategy involves all of the slices of an APBF. Subsequently, their fill ratios can increase for as long as they are in the filter, therefore allowing the possibility of them significantly exceeding the optimal maximum of $1/2$. As a result, the false positive probability can grow higher than the value specified for the filter. One other factor is that with the independent shifting at each APBF instance, the match of the emptiest local slice, with maximum fill ratio of $1/2k$, may be an inactive remote one, possibly overfilled. The combination of slices in such a scenario would lead to the shift of the initial slice at 0 to logical index k, outside the sequence of active slices. Similar frequent scenarios with large amounts of shifting, would lead to the loss of many generations of elements at once. For these reasons, this strategy is expected to yield very poor results.

Strategy Analysis - Active Slices

In order to limit the pitfalls of the previous strategy, the active-slices union strategy involves only its active slices. Since only the active slices are in use, if there is any content merger that makes a local slice exceed its maximum fill ratio, the shifting mechanism will simply move it out of the first k indices, where it will see no further changes. As such, the false positive probability of the APBF should not be meaningfully affected.

With no loss of generality, take into consideration Figure 4.8. The example presents the merge zone of two APBFs where $k = 4$. Here, due to independent shifting at each filter, matching slices are 2 logical indices apart in the circular buffer. For instance, in the top filter, the slice with hash function h_1 is at index 0 and has a maximum fill ratio of $1/8$, represented by the bars below it. In the bottom filter, its match is at index 3 and has a maximum fill ratio of $3/8$. Whenever they merge, the amount of shifting may move the h_1 slice from logical index 0 to index 3, where the h_4 slice is. Even with the smaller differences in the ratio of set bits, this approach, while better than the previous, may still cause too many shifts and the consequent loss of many elements.

Strategy Analysis - Current Generation

In an effort to affect fill ratios even less, the current-generation union strategy combines only the bits of the current generation stored in the companion slice of the remote, which may increase the fill ratio by at most $1/k$. Given its characteristics, mergers share even less membership information than the previous strategy; still, with enough synchronization, it is expected to yield comparatively decent results among all the approaches here presented.

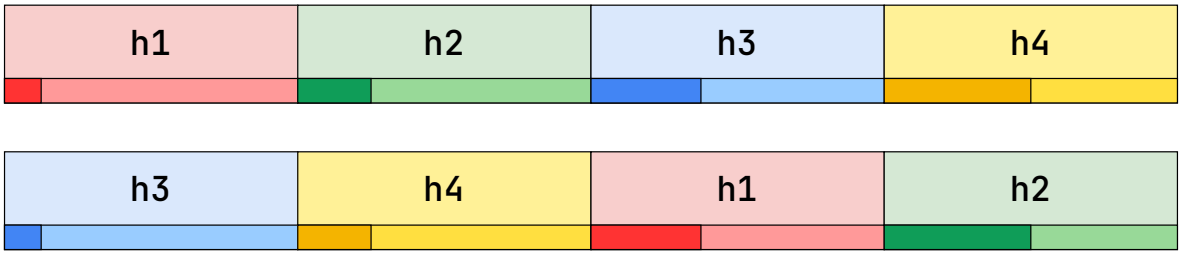


Figure 4.8: Matching slices of different fill ratios in a merge zone

4.5.3 *Resurrection of the Tombstones*

From the conclusions of the previous subsection, we now know that affecting the fill ratio of the involved slices as little as possible is a desirable feature from a union strategy, in order to minimize shifting. One way to achieve it would be to represent a smaller set of elements in the probabilistic structure and spread their insertions over time.

In section 4.4, a probabilistic alternative to the optimized OR-Set designs from subsection 4.3.2 is presented. In this alternative, the APBF takes the place of a traditional dot context, which stores every dot known to a replica. The optimized designs improve over the original OR-Set design with tombstones shown in subsection 4.3.1.

Revisiting the original unoptimized design, there are two of its characteristics that stand out, making it worthy of a revival. The first is the fact that the set of tombstones is a subset of the causal context, which stores dots that have yet to see a removal, in addition to the removed dots that would have a tombstone. The second is related to the problem of scalability. Having an ever-growing set not as compactable as the full dot context, the tombstones design became obsolete. Nevertheless, if the set of tombstones became APBF-based, it would be able to forget the oldest removed dots. These are exactly those we wish to prune because they are expected to become irrelevant over time, as knowledge of the removal of their corresponding entries propagates and eventually reaches all replicas in the system.

The shifting of slices that enables the APBF to forget elements can be seen as a kind of blind, automatic garbage collection of tombstones, which contrasts with the garbage collection supported by the original design, mentioned at the end of subsection 4.3.1. Therefore, the probabilistic variant here proposed is prone to inconsistencies, primarily caused by the premature oblivion of some dots. Nonetheless, it is possible to set up the APBF and the synchronization mechanism in a way that hopes to minimize mistakes, trying to maximize the probability that every replica knows of a dot before it is cleared and no longer obtainable. The implementation of the AWOR-Set with the aforementioned adaptations appears in [6].

5

EVALUATION

5.1 INTRODUCTION

In spite of the assumptions and expected behaviour from the designs and implementations described in chapter 4, the actual performance of the devised solution remains to be evaluated. The current chapter will present the simulation environment used to test the implementation in section 5.2, as well as performance metrics collected to compare with classic designs and aid in identifying the most appropriate approaches and possible use case scenarios in section 5.3.

5.2 EVALUATION ENVIRONMENT

The simulator constructed for the purpose of evaluating performance [7] operates in an environment defined by a number of settings. In general, these settings determine:

- the number of CRDT replicas active at the start of the simulation;
- how the network of participating nodes may change (as a result of a specified type of network dynamicity or churn rate);
- the number of iterations the simulation runs for;
- how the origin replica of an operation is selected;
- the frequency of synchronizations.

5.2.1 *Network Setup*

A round of execution of the simulator starts by constructing a given initial number of CRDT instances, containing both the classic structures and the equivalent new probabilistic variants [6]. While the classic segment pairs a complete causal context with its dot store, for the purposes of compaction seen in subsection 2.2.1, the probabilistic part pairs its own dot store

with a tombstone set, as described in subsection 4.5.3. Depending on the type of network dynamicity selected, the set of participating nodes may be affected in different ways.

Static

If the network is defined to be static, every iteration of the simulation preserves the starting set of nodes.

Intermediate

In case the network dynamicity is set to an intermediate level, the number of participating nodes is kept equal to the initial value. However, every iteration, there is a tunable probability that an active instance leaves and new one takes its place. This probability can be considered to be the churn rate. In this simulator, every active replica is equally likely to leave.

Dynamic

When the network is set to be fully dynamic, at every iteration, there are 3 different possibilities, with tunable probabilities:

1. a node leaves the network;
2. the set of participating nodes remains the same;
3. a node joins the network, being either completely new or a rejoining node, that previously left temporarily.

5.2.2 *Operations*

Following the possible changes to the network at the beginning of the iteration, an operation may take place in the system if the necessary requirements for its execution are met. Otherwise, the current iteration is interrupted and a new one initiates.

Origin Selection

The operation's process begins by selecting an origin replica where the operation may be executed. The probability of selection of the origin can be uniform among the CRDT instances active in the system or follow a Zipfian distribution, making some replicas more likely to be selected. This alternative origin distribution can help to simulate a system where the load of operations is asymmetrically distributed.

Operations

Following the selection of the origin replica, a particular operation must be chosen. This choice assigns either equal or custom weights to each of the supported operations, with any mutating operation affecting both the classic and probabilistic structures of the CRDT instance. Given the focus of this dissertation on the Add-wins Observed-Remove Set (AWOR-Set), 3 operations can occur at the origin CRDT:

*Operations - **Add***

A value from a domain is randomly chosen and added to the set of values of the origin. This operation never fails.

In more detail, an addition of a value is divided into 2 major steps:

1. Removal of previous entries of the same value from both variants of dot store, in order to avoiding polluting the state with similar entries. For the classic segment, nothing else is done with the removed dots, whereas on the probabilistic side the dots are added to the set of tombstones, as each of their entries are retired from this point forward;

2. Generation of dots for the new addition - deterministically, on the classic side, and randomly, on the probabilistic. The new pairs of $(\text{dot}, \text{value})$ are inserted in their respective dot stores. Additionally, there is the registration of the new classic dot in the causal context, that holds knowledge of every known dot to the replica up until the current moment; expectedly, the tombstones' set on the probabilistic segment stays untouched.

*Operations - **Remove***

A value in the set at the origin replica, chosen in a random, uniformly distributed manner, is removed. The operation fails if the set is empty, as there are no elements to remove.

Equally to what happened in the step for the `add` operation, existing entries for the value targeted by the `remove` are cleared from both dot store variants. Since the value is chosen from the probabilistic dot store, it may be tied to an intra-inconsistent entry and therefore, be absent from the classic dot store. In this scenario there is a (possibly accidental) recovery from an inconsistency. Yet again, the newly retired dots are registered in the set of tombstones on the probabilistic side; on the classic side, the causal context is unaltered, as the dots were already known.

*Operations - **Synchronization***

Described in subsection 4.4.2, this is the process through which the origin replica receives state information of a remote and hopefully attains some degree of inter-replica consistency. Synchronizations here share the full state and are unidirectional, which means that only the local replica is expected to reflect the state at the remote. The network topology takes the form of a complete graph. As such, the target remote replica of the synchronization is chosen at random, with equal probability, from the set of active replicas, excluding the origin. This fact means that the operation fails if there are less than 2 active nodes in the system.

5.2.3 *Error Determination*

After the local execution of an operation, there is a need to determine if it caused any intra-replica inconsistency (defined in subsection 4.5.1) between the classic/deterministic and the probabilistic underlying dot stores of the CRDT. Since the classic variants are error-free, the set of values read from them are considered to be the reference of correctness.

Taking this into account, at this stage of the iteration of the simulation loop, two sets are calculated. These contain the differences between the classic and probabilistic sets. Any values that are part of the classic set and erroneously absent from the probabilistic are said to be part of the *classic exclusives* set; conversely, values erroneously present in the probabilistic set and absent from the classic are designated as members of the *probabilistic exclusives* set.

The occurrence of these inconsistent values stems from the scenarios identified in subsection 4.5.1, which, in summary, are the following:

- classic exclusives happen as a result of false positives, created by a phantom presence in the set of tombstones when querying the APBF supporting the causal CRDT;
- probabilistic exclusives are a direct consequence of the fact that as the APBF becomes overfilled over time, it shifts slices to make space for new generations and eventually forgets older generations of elements that maintain some usefulness.

Due to the fact that inconsistencies may carry over from previous iterations, the sets of exclusives of the current iteration, for both classic and probabilistic sets, are saved. This way, in a future iteration for the same origin replica, it is possible to determine if the sets of classic or probabilistic exclusives remain what they were in the past or if any new inconsistent value has emerged.

5.3 RESULTS

5.3.1 *Memory Consumption*

In order to evaluate the memory consumption of the proposed solution, one has to take into consideration the characteristics of the deterministic and probabilistic structures, their contents, and the network where the CRDTs operate.

In a completely static network, where the set of nodes remains unchanged over time, a deterministic solution could employ the use of a simple vector of counters as the causal context, such as the one described in subsection 2.2.1, where every index corresponds globally to the same replica. Furthermore, if there is no guarantee of a causally consistent delivery in the system, a supporting set of straggler dots must be represented, contributing to a variable and expectedly minor increase in the amount of memory used.

Assuming a size of 64 bits per counter, allowing the continuous operation of the system without concerns regarding overflows, Figure 5.1 depicts the slowly increasing use of memory as more replicas are included in the system. When compared with the APBF with capacity for 2048 items, they only match in number of bytes used at around 750 replicas in the network, while the one with capacity for 4096 elements only matches at above 1400 replicas.

In a static network, there is a clear disadvantage when comparing the probabilistic approach to the error-free deterministic one. However, when the network becomes dynamic, the set of known replicas progressively increases. Over sufficient time, the number of entries in the context may increase to values that justify the use of a probabilistic structure.

In addition, in a dynamic network, a vector of counters may not be a flexible enough structure, due to the management of entries and the mapping of indices to replicas. As an alternative, trading the compact size for more flexibility, a map could be used, with replica IDs as keys and counters as values. Given the fact that 128-bit sized UUIDs are commonly used for identification, due to the very low probability of collision of generated values, Figure 5.1 assumes their use for replica naming.

One other aspect to take into account is the fact that map implementations may allocate more memory than that used by the total of its entries. This aspect has practical relevance and thus must not be ignored. The plot line that considers the memory overhead of the map in Figure 5.1 relates to the `HashMap` implementation in the standard library of the Rust language at the time of writing.

Following these changes, the match in memory usage between the classic deterministic structure and the probabilistic one happens at around 250 replicas for the filter with capacity for 4096 dots and at around 125 for the one with space for 2048. In a dynamic network

with sufficient churn rate, these figures do not seem unreasonable and support the potential validity of the use of probabilistic structures within causal CRDTs.

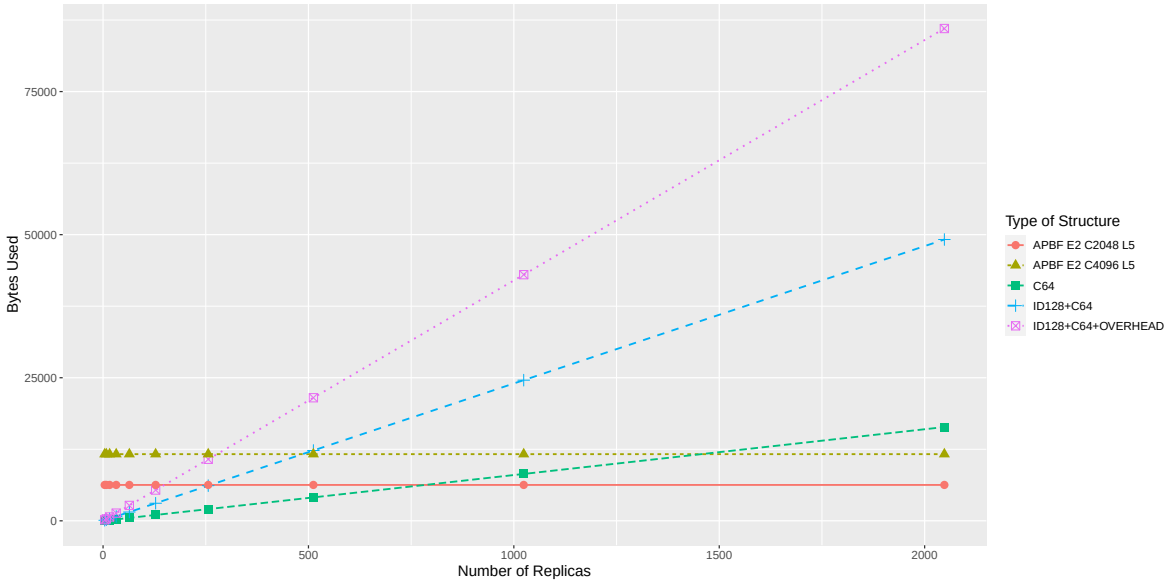


Figure 5.1: Memory Consumption

Absent from the analysis is the memory usage in dot stores (from subsection 4.2.1), given that it depends on the amount of entries they contain. As observed in Figure 5.1, the memory use of APBFs supporting the probabilistic CRDT design is not influenced by the number of nodes in the system. Moreover, the memory used by an APBF does not depend on the size of the input, since every element is represented with the same number of bits upon insertion. Consequently, the size of dots is only relevant to avoid the risk of collision between contemporary dots and memory consumption in the dot store.

Considering that classic dots are pairs of replica IDs and counter values and their sizes in the previous analysis, each classic dot is $192 = 128 + 64$ bits in size. In contrast, probabilistic dots are randomly generated. For reasonable robustness against equal contemporary dots, the size of 64 bits could be an option. Correspondingly, for every dot in equivalent dot stores, the classic version would consume 128 more bits. In case the number of dots in the dot stores were to be sufficiently big, this difference could become meaningful.

One other point of consideration is the fact that the values of bytes used in figures 5.1 and 5.2 assumes the choice of the *Current Generation* union strategy, which holds duplicate bits spaces for every active slice, for storage of the current generation. In spite of this, as the following results will present, the use of other union strategies aiming for an adequate volume of errors may not lead to a more compact APBF.

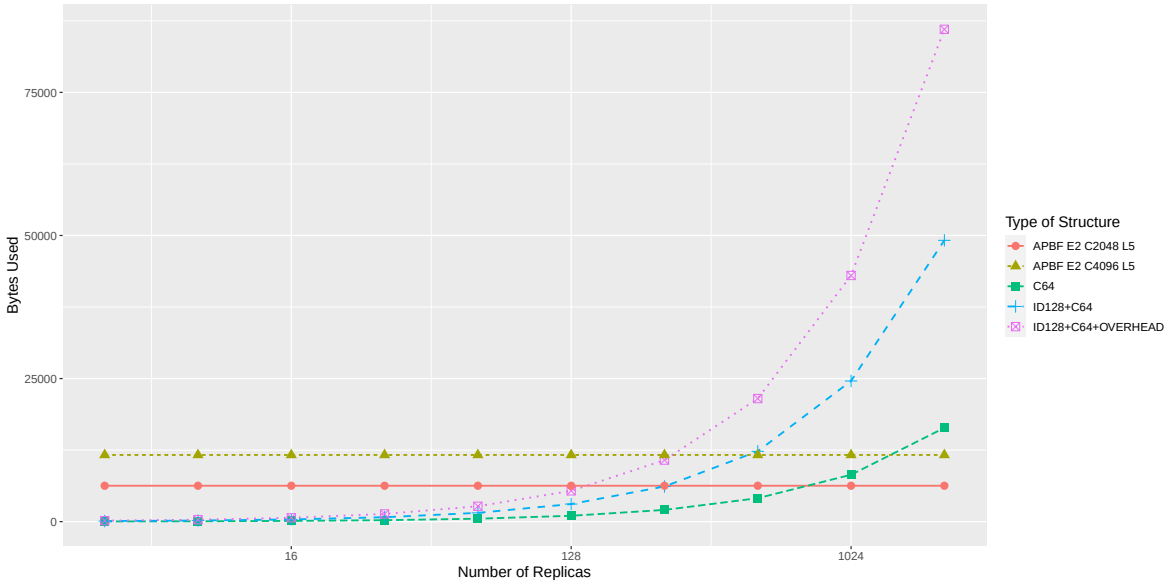


Figure 5.2: Memory Consumption - $\log_2$ x-axis

5.3.2 *Age-Partitioned Bloom Filter Metrics*

In the implementation used for this dissertation [5], an APBF is parameterized by 3 variables:

- *error*, tied to the false positive probability of $10^{-\mathbf{error}}$. For instance, an error of 2 corresponds to 0.01 or 1%;
- *capacity*, indicating how many elements should fit in its sliding window;
- *level*, which influences the overall number of slices and the proportion between the number of active slices and of those used to store older generations - the higher the level, the greater the total number of slices and the bigger the portion of slices in the filter dedicated to store older generations.

An increase in either *error* or *capacity* increases the size of slices. When a lower false positive probability is required, there is a need for more differentiation between elements in the filter, which is achieved by also increasing the number of slices. The larger the number of active slices, used at the time of insertion, the more likely elements are of being represented by bits set at differing locations. An increase in the value of the *level* parameter, despite increasing the total number of slices in the filter, also lowers the size of each slice, which is inversely proportional to the number of slices used for older generations. As a consequence, a higher *level* generates a more compact filter. In spite of the greater compactness of APBFs with higher *level*, considering they have more slices, the performance of their queries and

insertions is hindered, as they have to access more slices. This statement also applies to APBFs with a higher *error* value (corresponding to a lower false positive probability).

Union Strategy Comparison

In subsection 4.5.2, 3 union strategies for the Age-Partitioned Bloom Filter were presented - *Whole Filter*, *Active Slices* and *Current Generation*. Through each successive strategy in order of presentation, there is an attempt to affect the fill ratio of slices progressively less during unions, in order to avoid false positive related errors. Although it synchronizes a smaller amount of information, affecting the fill ratios of the receiving filter the least, the *Current Generation* strategy may hold information for less time than the remaining strategies. Consequently, it should be used in an environment with enough synchronizations, so that the corresponding APBF shares its knowledge before forgetting it.

Due to the nature of the operations previously described, only synchronizations generate new inconsistencies. Accordingly, Figure 5.3 presents the percentage of synchronizations that generate new inconsistencies between the classic and the probabilistic dot stores, for each of the strategies, in a network with 16 replicas, some churn (1% chance of a node leaving, with new one taking its place) and where 80% of the operations are synchronizations (the remaining 20% are evenly distributed between adds and removes).

Let us first consider the scenario depicted where the filter has *error* 2. In runs of 20000 iterations, about 80% (or 16000) of those will execute a synchronization. Of these synchronizations, around 2.1%, 2.4% and 1.9% will generate new inconsistencies for the *Whole Filter*, *Active Slices* and *Current Generation* strategies, respectively.

Having such frequent synchronizations allows CRDT instances using any of these strategies to reduce errors, due to the fact that only a few bits are expected to be newly set at each synchronization. By avoiding the increase of fill ratios of matching slices in significantly sized chunks, the frequency of errors related to false positives reduces, as detection of saturation of slices can occur before their fill ratios go exaggeratedly over the limit. This is especially useful for the strategies most prone to increase fill ratios over the limit of saturation, such as *Whole Filter* and *Active Slices*. Errors related to forgotten dots may also decrease, due to the fact that smaller increments in fill ratios during filter operations lead to fewer slice shifts, and consequently, fewer generations of dots forgotten at once. These factors contribute to the similar results among the strategies.

By lowering the false positive probability to 0.001% using an *error* of 5, we arrive at the other scenario in Figure 5.3, where the percentages of error-inducing synchronizations are of about 2%, 3% and 2.3% for each of the strategies.

Once again, the results are very similar between the strategies, with the same factors still in play. For this reason, the variations in results for the different *error* values are

small. Considering that the proportion of false positive related inconsistencies was already insignificant, the reduction of the false positive probability had little to no effect.

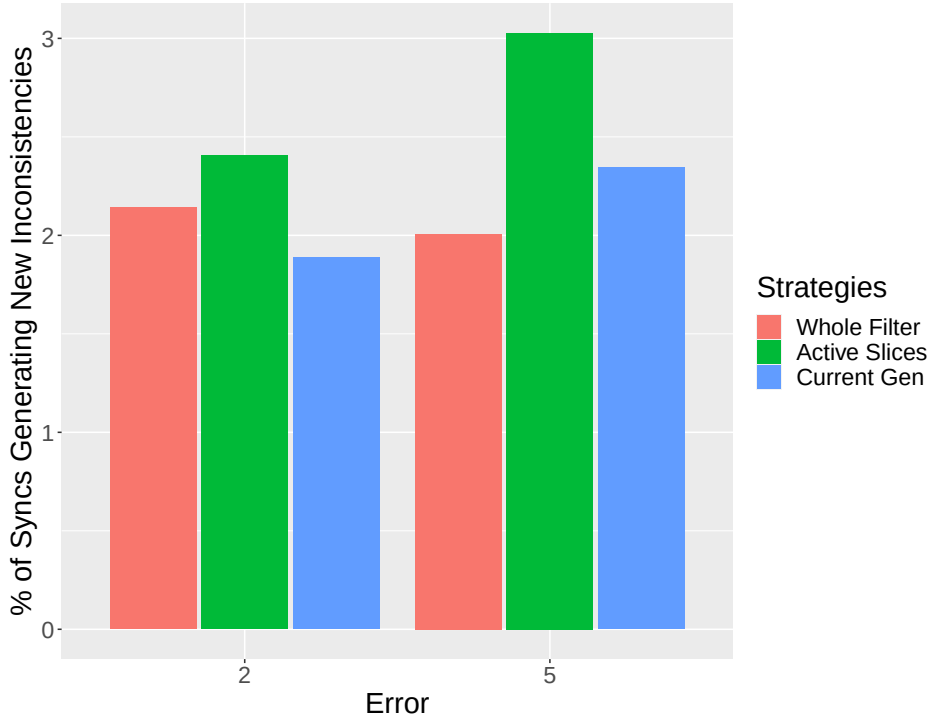


Figure 5.3: Percentage of error-inducing syncs by union strategy - *error* 2 | 5 and 80% syncs

What happens when synchronizations are less frequent? With fewer synchronizations, each of them is expected to set a greater number of bits to 1, increasing the fill ratio in bigger chunks, in comparison with the previous case of Figure 5.3. By increasing the fill ratio in bigger chunks at once, the detection of saturation of a slice might arrive too late, producing a number of slices with fill ratio values over the designated limit. Furthermore, with more elements inserted at once into the window by the union of slices, the more are expected to move out of the window by slice shifts, which can also contribute to an increased number of errors.

Observing Figure 5.4, it is possible to see an overall increase in the proportion of synchronizations that generate new inconsistencies. The strategy expected to suffer the most from overfilled slices, *Whole Filter*, has now around 58% of its synchronizations generating new inconsistencies, with nearly 92% of them coming from false positives when *error* is set to 2. Decreasing the false positive probability, with an *error* of 5, brings the proportion of error-inducing synchronizations down to about 37.5%, bringing the proportion of false positive related inconsistencies down to around 61%.

The *Active Slices* and *Current Generation* strategies produce very similar proportions of erroneous synchronizations, with 18.2% and 19.3% when *error* is set to 2 and 18.4% and 18.7% when *error* is set to 5, respectively. Both strategies seem largely unaffected by false positives, which justifies the negligible change when *error* increases to 5. Additionally, both are able to keep the probabilistic side reasonably consistent with the classic, with generally under 1% of entries exclusive to the probabilistic dot store for all replicas in the system throughout the simulation runs.

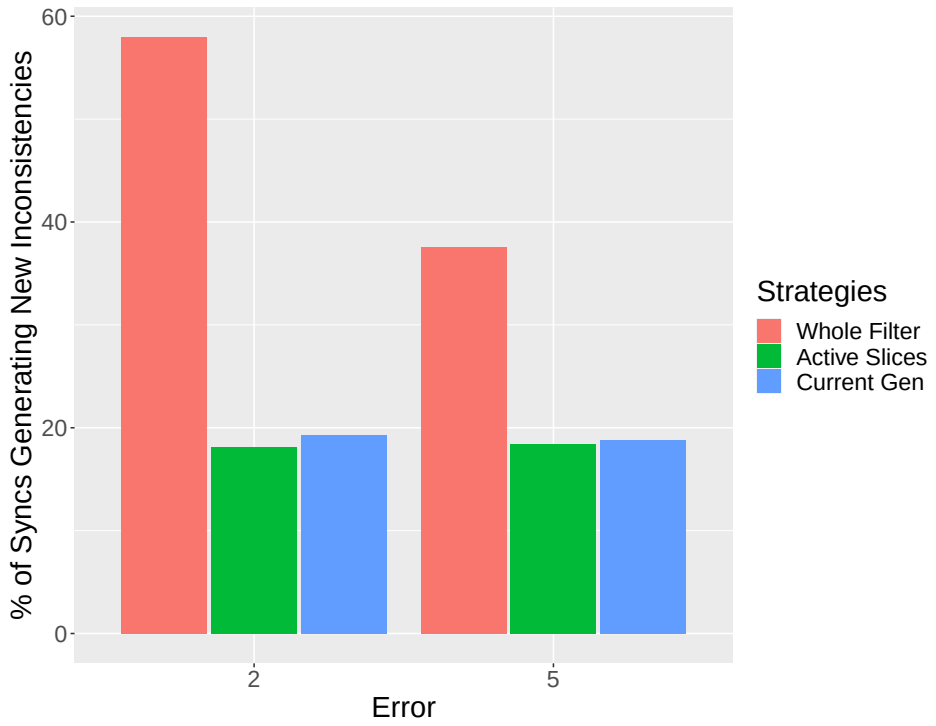


Figure 5.4: Percentage of error-inducing syncs by union strategy - *error* 2 | 5 and 40% syncs

One factor that produces different results between the *Active Slices* and *Current Generation* strategies is the distribution of load. When load is uneven, as a result of selecting an origin replica by using a Zipfian distribution with an exponent of 1.03, the *Active Slices* strategy shows more sensitivity to the conditions in the network, as show in figures 5.5 and 5.6. While practically unaffected by false positives, *Active Slices* produces more errors related to forgotten dots than the alternatives. These inconsistencies might occur more frequently as a result of very different fill ratios between matching slices during synchronization, which cause a greater number of shifts and push out several generations of dots that are still relevant. This is contemplated in subsection 4.5.2.

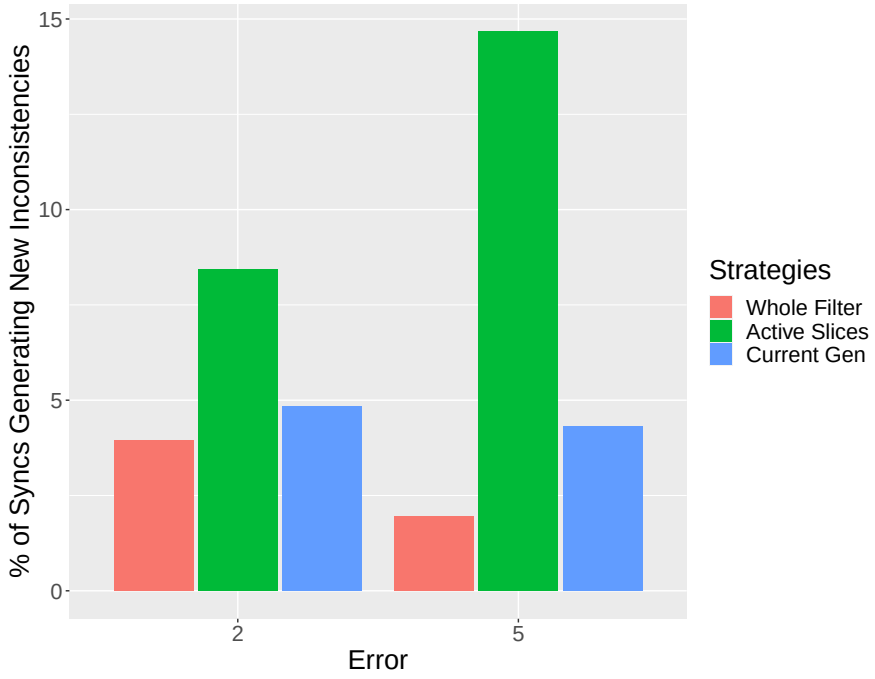


Figure 5.5: Percentage of error-inducing syncs by union strategy - *error* 2 | 5, 80% syncs and asymmetrical load

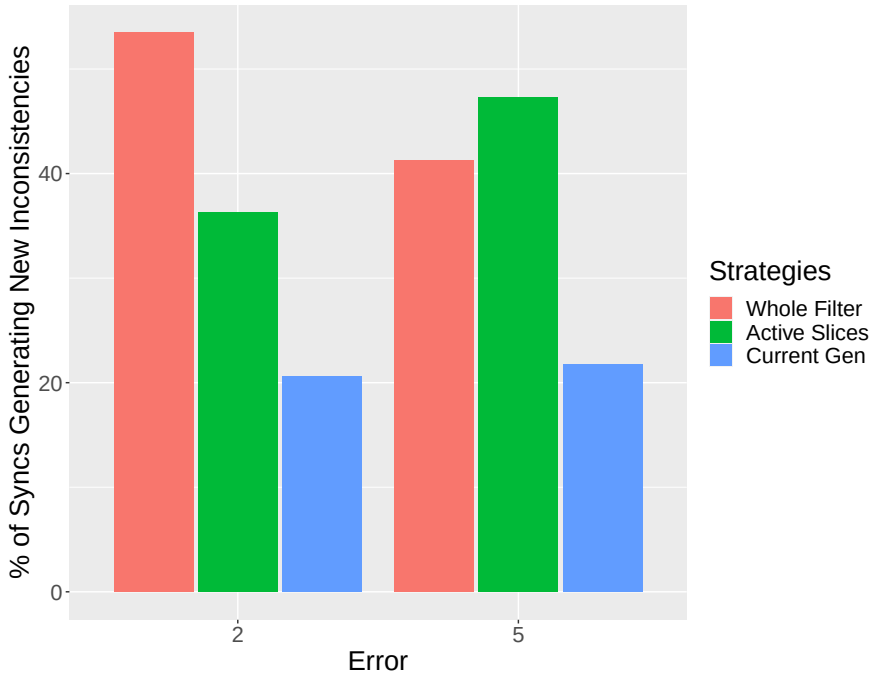


Figure 5.6: Percentage of error-inducing syncs by union strategy - *error* 2 | 5, 40% syncs and asymmetrical load

Since it generates fewer inconsistencies in these scenarios, *Current Generation* can be more useful at lower *error* values than its competitors for different load distributions. Being more useful at lower *error* values allows it to produces a smaller memory footprint. For instance, even with the duplicate active slices for storing the current generation, an APBF of *error* 2, *capacity* of 4096 and *level* 5 occupies 11648 bytes. A similar APBF with *error* 5, without the duplicate active slices, uses 20592 bytes and still 13528 bytes with *error* 3.

From subsection 4.5.1, it is known that errors related to forgetting dots lead to values wrongly exclusive to the probabilistic dot store. Since these can be resolved by synchronizing with replicas which still hold memory of the deletion of the value or through explicit removals of the value, the *Current Generation* strategy, almost exclusively affected by errors of this type, seems to be adequate still. The results of simulation runs seem to confirm these claims, with an average of under 1% of the state of the probabilistic dot stores of all replicas being these intra-inconsistent additional values.

Comparatively, errors stemming from false positives are much more difficult to recover from. Given the fact that these errors generate values exclusive to the classic side, the probabilistic variant of algorithm 3 will not be able to iterate through the entries of those inconsistent values and potentially act on them. Furthermore, in the same algorithm, if a false positive is triggered at line 4, there is a removal at line 5. Considering that a removal shows further knowledge of a given dot, that action may propagate correctly to other nodes in the system. In this scenario, through the normal process of synchronization, the states of the replicas converge to be consistent with each other but inconsistent with the correct state of their classic/deterministic dot stores. For the same probabilistic algorithm, a false positive at line 9 leads to no insertion at line 10, which would happen on the classic side, generating a classic exclusive. Recovery from this type of error is only possible if there are any generation changes in the underlying APBF that remove the triggering of the false positive.

In conclusion, when synchronizations are very frequent, strategies other than *Current Generation* can produce comparable results. However, due to its possible use with smaller, less accurate filters, its generation of more recoverable errors and the lower percentages of inconsistent state it exhibits in suboptimal conditions, *Current Generation* seems to be the more versatile strategy. As such, *Current Generation* is the strategy of choice for the following evaluation.

5.3.3 *Network Metrics*

Dynamicity, Churn Rate and Synchronization Frequency

Several characteristics of the network influence the error rate of probabilistic CRDTs. One aspect that affects how different the states of replicas are when they synchronize is the level

of dynamicity. In a network, as synchronizations occur, nodes share information among themselves and converge to the same state. When some churn is introduced, new nodes begin to appear. Bigger differences between fill ratios of matching slices of different replicas, as shown in subsection 4.5.2, lead to more shifts, which cause generations of older dots to be forgotten. If these dots were forgotten while still useful, new inconsistencies appear.

Figure 5.7 compares the frequency of occurrence of errors for different levels of dynamicity. As expected, when the network is static, nodes evolve more synchronously and their differences are less significant when compared to environments with higher levels of node movement.

Thus, when the size of the network is fixed but there is still some chance for a node to leave and be replaced by a new one (in the configuration shown in Figure 5.7 for the *Intermediate* dynamicity it is of 1%), the frequency of new inconsistencies increases. However, considering that most of the replicas in the system remain the same, the increase in errors is slight.

When the network is fully dynamic and able to grow and shrink in size as it pleases, with nodes departing and joining, the differences between fill ratios of matching slices increase and so does the percentage of incorrect synchronizations. The specific setup of Figure 5.7 uses a probability of 10% for both node departure and arrival, with the remaining 80% being dedicated to preserve the network. Also, arrivals are defined to reuse a node that left whenever possible; the network only grows when there is no node to reuse. This intends to simulate a network where the recovery of nodes is the most likely scenario, while still allowing for new inclusions. Allowing a greater percentage of new nodes exacerbates the differences between replica states and vastly increases the number of errors.

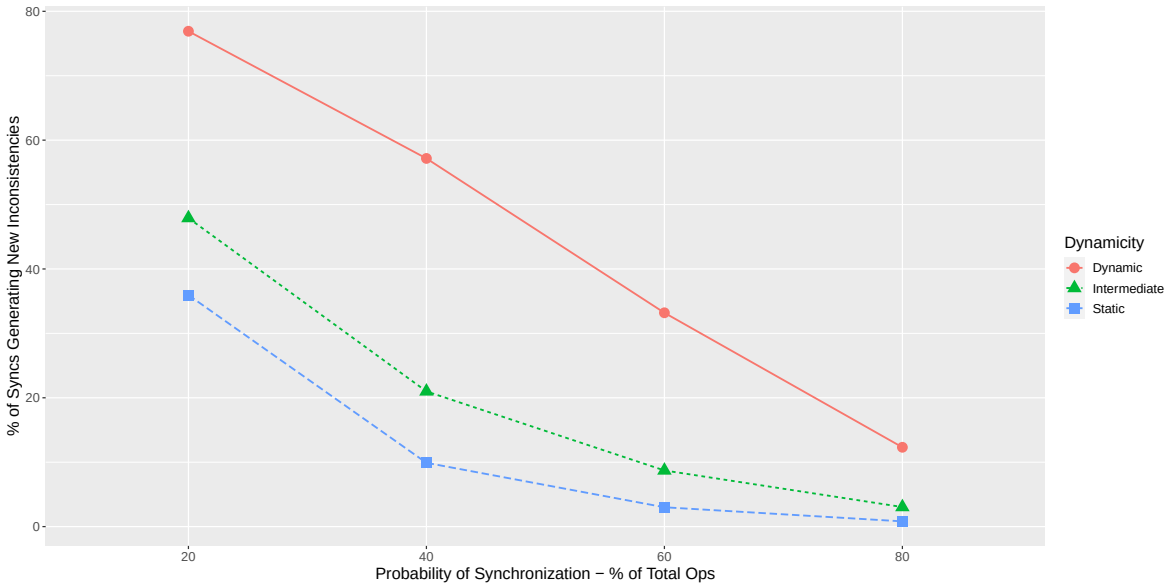


Figure 5.7: Percentage of syncs generating errors for each type of dynamicity

Figure 5.8 compares the performance of a network of 16 replicas that allows for nodes to leave and empty ones to substitute them, at different percentages, interpreted as churn rates. The results generally match those of the previous experiment, with higher churn rates leading to increased frequency of new errors. In some cases, the increase is not very significant - at 0.1% there is no relevant difference when compared with the environment with *No Churn*; similarly, at 1%, the increase over 0.1% may be considered to be reasonable.

For the 2 lowest synchronization rates (10% and 20%), the results for the churn rate of 10% appear to be an anomaly, contradicting the rest of the measurements. Taking into account that approximately once every 10 iterations a node leaves and is replaced by a new, empty one, combined with the infrequency of synchronizations, in these two scenarios, the size of dot stores in the system may not get the opportunity to grow significantly. For synchronization percentage of 10% in particular, even after 20000 iterations of the simulator, the dot stores seem to grow only up to single digits in size, with many of them empty; for synchronization percentage of 20%, some dot stores get to 3 digits in size, but many in the system, as a result of churn and insufficient synchronizations, are empty. The small states shared lead to fewer inconsistencies being generated each time, providing surprisingly low percentages of error-causing synchronizations. As synchronizations become more frequent, the results appear more in line with expectations.

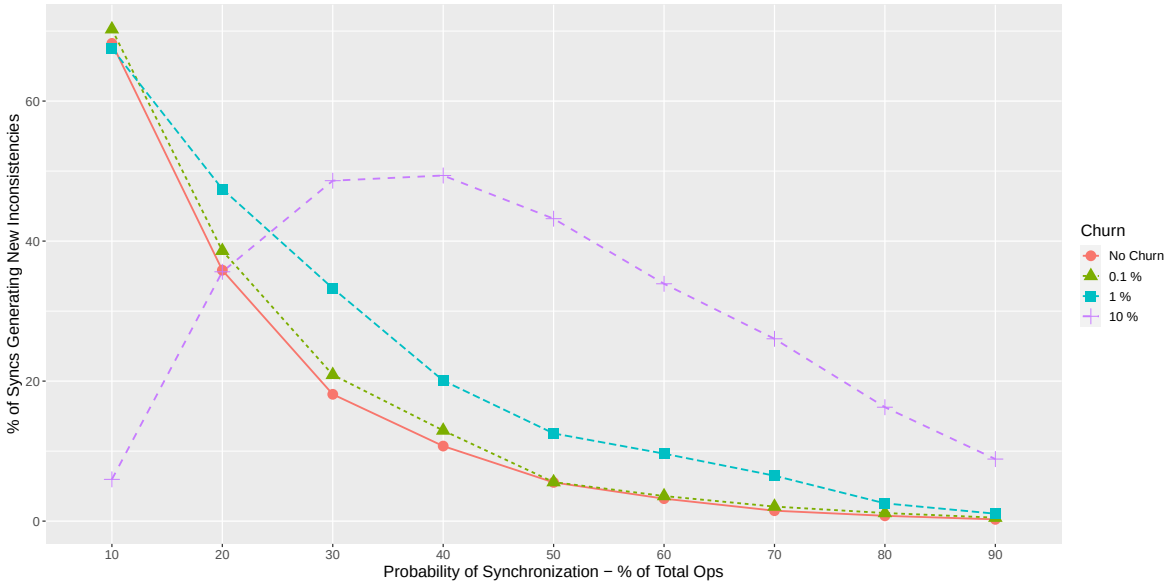


Figure 5.8: Percentage of syncs generating errors for different churn rates

In conformity with the conclusions presented throughout this evaluation chapter, the more frequent synchronizations are, the fewer new inconsistencies are generated. Figure 5.9 shows the relation between the frequency of synchronizations and the frequency of new inconsistencies, for a fixed number of synchronizations, suggesting that it is frequency and not total volume of synchronizations the defining factor of adequate results.

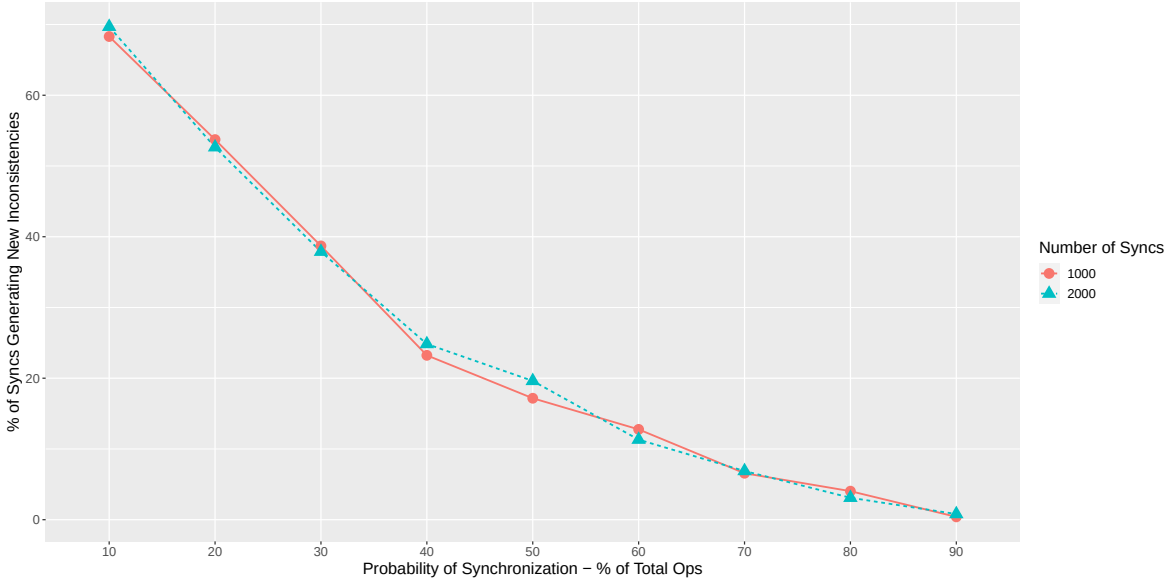


Figure 5.9: Percentage of syncs generating errors for a fixed number of syncs

Asymmetrical Load

Figure 5.10 intends to observe the effects of asymmetrical load on the proportion of inconsistency-inducing synchronizations. The simulation used a network of 16 nodes, with a churn rate of 1% (intermediate level of dynamicity), as shown originally in Figure 5.8. The asymmetrical load is achieved by selecting origin replicas for operations using a Zipfian distribution parameterized by an exponent taking the value of 1.03. For synchronization rates of 10% and 20%, the network with Zipfian distribution of load appears to perform better, while for 30% and above the results seem similar.

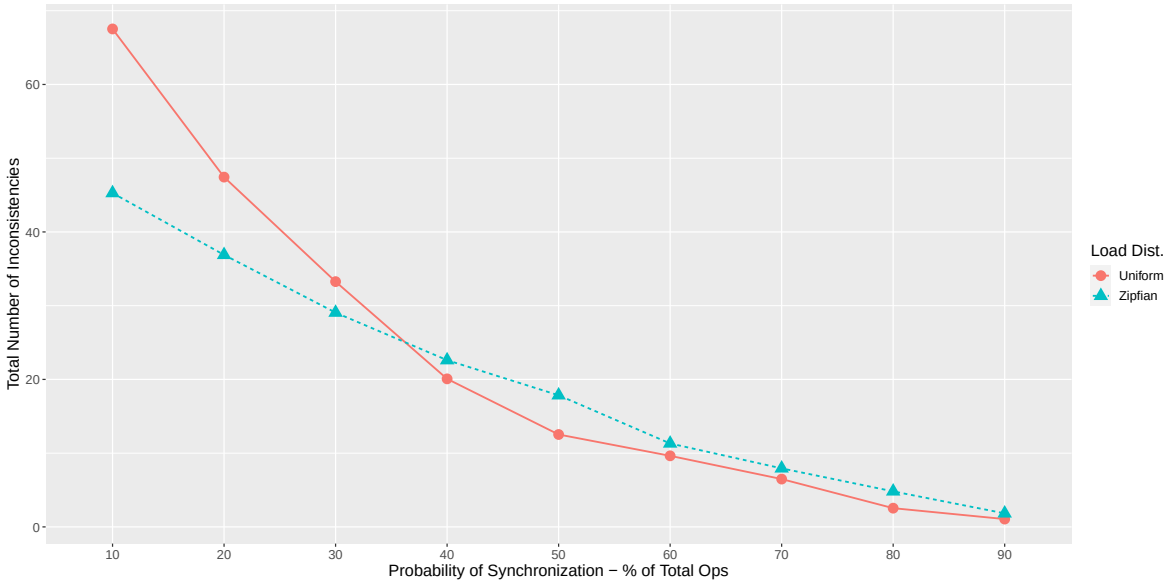


Figure 5.10: Percentage of syncs generating errors for different load distributions

In this comparison, however, the proportion of error-causing synchronizations does not reflect the complete truth. Figure 5.11 shows the total number of inconsistencies observed during the same simulation run, for each load distribution. Even if it produced fewer erroneous synchronizations for the lowest synchronization rates and similar numbers for 30% and above, the simulation setting with asymmetrical load always generates more inconsistencies. In spite of this, perhaps due to the ability to recover from inconsistent entries in the probabilistic dot stores, that constitute the vast majority of the total of inconsistencies, the states of the probabilistic dot stores for both load distributions do not differ too much from their respective classic structures with enough synchronizations.

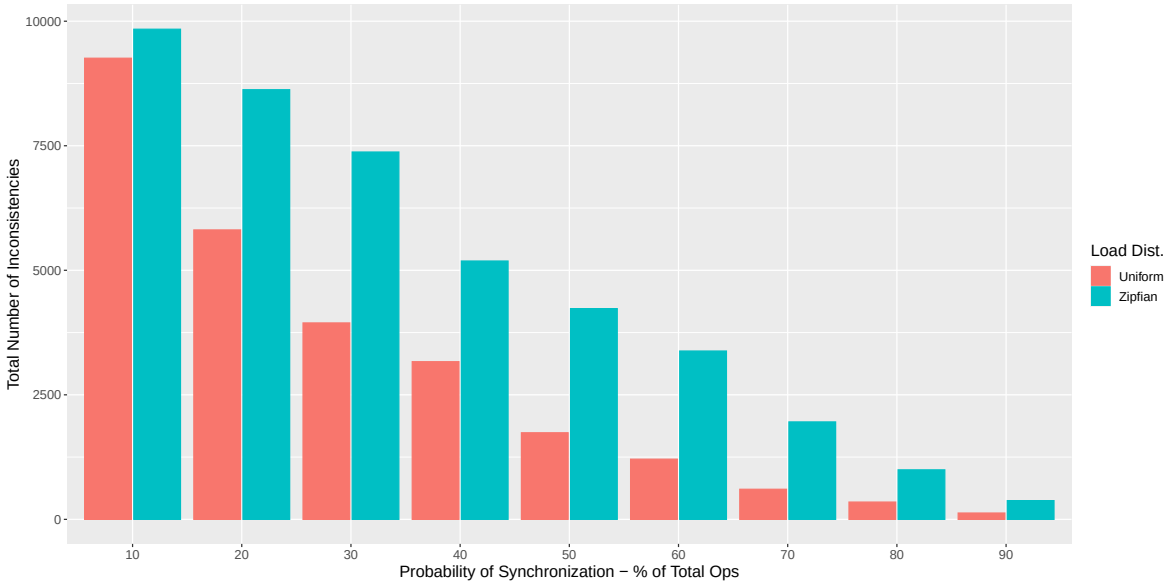


Figure 5.11: Total number of new inconsistencies for different load distributions

6

CONCLUSION

This thesis explored the viability of probabilistic structures at the core of causal CRDTs.

The probabilistic structure of choice was the Age-Partitioned Bloom Filter (APBF) [39] due to its ability to operate on a sliding window over a stream, keeping track of more recently observed elements and forgetting older ones, thus avoiding saturation or unbounded growth.

Owing to the APBF's lack of native support for union of contents, adaptations were needed so that it could support every operation demanded of a causal context, the particular structure of the classic causal CRDT design targeted to be replaced.

Even considering the expected errors, related to false positives (inherent to Bloom Filter variants) and oblivion of events that still held relevance in the system (inherent to sliding window solutions), the use of APBFs in causal CRDTS showed some promise. In networks with some degree of dynamic behaviour, the probabilistic version of causal CRDTs was capable of providing space savings when compared to the classic design, while keeping the effect of errors somewhat limited. These characteristics, in addition to the lack of need for identifying replicas in the system, may offer additional flexibility over the classic solution, which may be useful for some use cases.

6.1 FUTURE WORK

One expected source of improvements over the APBF-based solution would be the conception of a structure capable of representing a set probabilistically that not only allows insertions and membership queries, but also supports other set operations, with an emphasis on set unions. The same structure should follow the sliding window model, to enable improved scalability and continuous operation. Even though the errors resulting from the oblivion of still-useful elements seem inevitable, there is some expected margin for improvement, which could have very positive effects on the results.

In addition, the evaluation could benefit from further analysis, with the inclusion of other causal data types and different network topologies.

Finally, bearing in mind that state CRDTs are in use, the possibility of sharing delta-states during synchronizations, with support for transitive sharing of state, could align the work described in this document with the state of the art. However, the accuracy of synchronization with deltas might only be sufficient with a probabilistic structures with an improved union operation, like the one suggested previously. Moreover, due to the nature of the existing anti-entropy algorithms for delta-CRDTs, there could be some benefits in exploring what an algorithm that considers that replicas are anonymous would be like, due to the need for identification in the context of garbage collection of deltas.

BIBLIOGRAPHY

[1] Automerge. https://github.com/automerge/automerge. Last Accessed: July 2021.

[2] Bloom filter combination (method putAll) in Google's Guava. https://github.com/google/guava/blob/master/guava/src/com/google/common/hash/BloomFilter.java. Last Accessed: July 2021.

[3] Riak's CRDTs. https://github.com/basho/riak_dt. Last Accessed: July 2021.

[4] Roshi. https://github.com/soundcloud/roshi. Last Accessed: July 2021.

[5] Age-Partitioned Bloom Filter implementation. https://github.com/A77377/filte-rs. Last Accessed: July 2021.

[6] Hybrid (classic + probabilistic) delta-state causal CRDTs. https://github.com/A77377/delta-crdts. Last Accessed: July 2021.

[7] Simple simulator used for comparison between classic and probabilistic AWOR-Sets. https://github.com/A77377/probabilistic-aworset-sim. Last Accessed: July 2021.

[8] Yjs. https://github.com/yjs/yjs. Last Accessed: July 2021.

[9] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[10] P. S. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018.

[11] E. Assaf, R. B. Basat, G. Einziger, and R. Friedman. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2204–2212. IEEE, 2018.

[12] C. Baquero, P. S. Almeida, and A. Shoker. Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469*, 2017.

[13] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

[14] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *European Symposium on Algorithms*, pages 684–695. Springer, 2006.

[16] A. D. Breslow and N. S. Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.

[17] E. A. Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[18] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[19] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. *ACM Sigplan Notices*, 49(1):271–284, 2014.

[20] F. Chang, W.-c. Feng, and K. Li. Approximate caches for packet classification. In *IEEE INFOCOM 2004*, volume 4, pages 2196–2207. IEEE, 2004.

[21] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[23] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 25–36, 2006.

[24] G. Einziger and R. Friedman. Counting with tinytable: Every bit counts! In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, pages 1–10, 2016.

[25] V. Enes. Efficient synchronization of state-based CRDTs. Master's thesis, Universidade do Minho, Nov. 2017.

[26] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

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

[28] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[29] R. P. Laufer, P. B. Velloso, and O. C. M. Duarte. A generalized bloom filter to secure distributed network applications. *Computer Networks*, 55(8):1804–1819, 2011.

[30] Y. Liu, W. Chen, and Y. Guan. Near-optimal approximate membership query over time-decaying windows. In *2013 Proceedings IEEE INFOCOM*, pages 1447–1455. IEEE, 2013.

[31] D. Malkhi and D. Terry. Concise version vectors in winfs. In *International Symposium on Distributed Computing*, pages 339–353. Springer, 2005.

[32] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate detection in click streams. In *Proceedings of the 14th international conference on World Wide Web*, pages 12–21, 2005.

[33] M. Mukund, G. Shenoy, and S. Suresh. Optimized or-sets without ordering constraints. In *International Conference on Distributed Computing and Networking*, pages 227–241. Springer, 2014.

[34] M. Naor and E. Yogev. Tight bounds for sliding bloom filters. *Algorithmica*, 73(4):652–672, 2015.

[35] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, (3):240–247, 1983.

[36] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[37] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[38] X. Shi, S. Pruett, K. Doherty, J. Han, D. Petrov, J. Carrig, J. Hugg, and N. Bronson. Flighttracker: Consistency across read-optimized online stores at facebook. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 407–423, 2020.

[39] A. Shtul, C. Baquero, and P. S. Almeida. Age-partitioned bloom filters. *arXiv preprint arXiv:2001.03147*, 2020.

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

[41] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[42] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242, 1984.