Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vitor Manuel Enes Duarte

Efficient Synchronization of State-based CRDTs

November 2017

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vitor Manuel Enes Duarte

Efficient Synchronization of State-based CRDTs

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Carlos Miguel Ferraz Baquero Moreno
João Carlos Antunes Leitão

November 2017

ACKNOWLEDGMENTS

I would like to thank my advisors, Carlos Baquero and João Leitão, for their support and great advice. A special thanks to Paulo Sérgio Almeida, Ali Shoker and Georges Younes for all the insightful discussions. At last, to my family and friends, thank you for all the support and motivation you gave, and all the patience you had.

ABSTRACT

Data consistency often needs to be sacrificed in order to ensure high-availability in large scale distributed systems. *Conflict-free Replicated Data Types* relax consistency by always allowing query and update operations at the local replica without remote synchronization. Consistency is then re-established by a background mechanism that synchronizes the replicas in the system.

In state-based CRDTs replicas synchronize by periodically sending their local state to other replicas and by merging the received remote states with the local state. This synchronization can become very costly and unacceptable as the local state grows.

Delta-state-based CRDTs solve this problem by producing smaller messages to be propagated. However, it requires each replica to store additional metadata with the messages not seen by its direct neighbors in the system. This metadata may not be available after a network partition, since a replica can be forced to garbage-collect it (due to storage/memory limitations), or when the set of direct neighbors of a replica changes (due to dynamic memberships).

In this dissertation we further improve the synchronization of state-based CRDTs, by introducing the concept of Join Decomposition of a state-based CRDT and explaining how it can be used to reduce the synchronization cost of this variant of CRDTs.

We validate our proposal experimentally on Google Cloud Platform by comparing the state-based synchronization algorithm against the classic and improved versions of the delta-state-based algorithm. The results of this comparison show that our proposed techniques can greatly reduce state transmission, even under normal operation when the network is stable.

RESUMO

Frequentemente a consistência dos dados é sacrificada para garantir alta-disponibilidade em sistemas distribuídos de grande escala. *Conflict-free Replicated Data Types* relaxam a consistência permitindo operaçõess de query e update na réplica local sem sincronização remota.

Nos state-based CRDTs as réplicas sincronizam periodicamente enviando o seu estado local para as outras réplicas e combinando os estados remotos recebidos com o estado local. Esta sincronização pode tornar-se muito custosa e inaceitável à medida que o estado local cresce.

Delta-state-based CRDTs resolvem este problema produzindo mensagens mais pequenas para serem propagadas. No entanto, requer guardar metadados adicionais com as mensagens que ainda não foram vistas pelos vizinhos diretos no sistema. Estes metadados podem não estar disponíveis depois de uma partição na rede, visto que a réplica pode ser forçada a apagá-los (devido a limitações de armazenamento/memória), ou quando o conjunto dos vizinhos diretos da réplica muda (devido a vistas dinâmicas).

Nesta dissertação melhoramos ainda mais a sincronização de state-based CRDTs, introduzindo o conceito de Join Decomposition de um state-based CRDT e explicando como é que pode ser usado para reduzir o custo de sincronização desta variante de CRDTs.

Validamos a nossa proposta experimentalmente na Google Cloud Platform comparando o algoritmo de sincronização de state-based CRDTs com a clássica e melhoradas versões do algoritmo dos delta-state-based. Os resultados desta comparação mostram que as técnicas propostas podem reduzir muito a transmissão de dados, mesmos em operação normal quando a rede está estável.

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

1

INTRODUCTION

1.1 CONTEXT

Large-scale distributed systems resorting to replication techniques often need to sacrifice the consistency of the system in order to attain high-availability. One common approach is to allow replicas of some data type to temporarily diverge, making sure these replicas will eventually converge to the same state in a deterministic way. *Conflict-free Replicated Data Types* (CRDTs) [26, 27] can be used to achieve this.

It's possible to design CRDTs that emulate the behavior of a sequential data type, but some effort has to be done in order to resolve conflicts that result from operations that are not commutative in their sequential form, such as adding and removing the same element from a Set, for example. Therefore, there are design options that need to be made when implementing a CRDT, in particular regarding its semantics for non-commutative operations (*add-wins*, *remove-wins*, . . .) to ensure these data types converge deterministically for the chosen semantic (which might be application specific). Hence, multiple CRDTs can exist to materialize a single sequential data type [3, 8].

CRDTs come mainly in two flavors: operation-based and state-based. In both, queries and updates are always possible at the local replica, the source of these operations, and this is why the system is available (as it never needs to coordinate with remote replicas to execute operations). Operation-based CRDTs perform update operations in two phases: *prepare* and *effect*. The *prepare* phase, at the source of the operation, produces a message (that represents that operation) to be sent to all replicas, using a reliable causal broadcast channel. Once delivered, this message is applied using *effect*. These messages have to be delivered exactly once since the operations they represent might not be idempotent. State-based CRDTs need fewer guaranties from the communication channel: messages can be dropped, duplicated and reordered, and the system state remains convergent. When an update operation occurs, the local state is updated through a mutator, and from time to time (since we can disseminate the state at a lower rate than the rate of the updates) the state is propagated to the other replicas. When a replica receives the remote state of another replica, it merges this remote state with its local state using a binary join operator that is designed to be idempotent, commutative and associative.

1.2 motivation

Although state-based CRDTs can be disseminated over unreliable communication channels, as the state grows, sending the full state can be very costly and become unacceptable. Delta-state-based CRDTs (δ-CRDTs) [2, 3] address this issue, by defining δ-mutators that return a delta (δ), typically much smaller than the full state of the replica, to be merged with the local state and propagated to remote replicas. This strategy requires keeping track of which updates have been effectively received by other replicas of the system with which the local replica exchanges information (*i.e.* synchronizes) directly, which leads to the maintenance of additional metadata that may have to be garbage collected (due to storage limitations) or not be available (due to dynamic memberships).

1.3 problem statement

Current solutions perform bidirectional full state transmission when a replica joins a system (either for the first time or after a network partition) in order to this replica receive the missed updates and to propagate the ones observed locally. This strategy can be unacceptable when the size of the CRDT state is not small. After this initial synchronization, replicas can synchronize with a neighbor replica by sending groups of δs (all the δs that haven't been received by that neighbor), avoiding full state transmission on each synchronization step. However, careful must be taken when computing these groups of δs: we have noticed some executions where this optimization is equivalent to state-based synchronization.

1.4 main contributions

In this thesis we revisit the delta-state-based synchronization algorithm and propose modifications that further reduce the amount of state transmitted. We also introduce the concept of Join Decomposition of a state-based CRDT, and present two novel algorithms, *state-driven* and *digest-driven*, used for efficient synchronization of state-based CRDTs when the metadata storage required for delta-state-based synchronization is not available.

A preliminary version of part of the work described in this thesis was published in the following workshop paper:

- **Join Decompositions for Efficient Synchronization of CRDTs after a Network Partition: Work in progress report** Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and

Ali Shoker. First Workshop on Programming Models and Languages for Distributed Computing. PMLDC@ECOOP 2016 [17]

1.5 DISSERTATION OUTLINE

The rest of this dissertation is organized as follows. Chapter 2 presents current strategies to synchronize state-based CRDTs. Chapter 3 introduces the concept of Join Decomposition and explains how it can be used to efficiently synchronize state-based CRDTs when no metadata is available, presenting two synchronization algorithms: *state-driven* and *digest-driven*. Chapter 4 revisits the classic delta-state-based synchronization algorithm and proposes some modifications that reduce state transmission. In Chapter 5, we evaluate the synchronization algorithms presented in Chapter 2, 3 and 4. Finally, in Chapter 6, we conclude this dissertation and present some ideas for future research.

2

SYNCHRONIZATION OF STATE-BASED CRDTS

Data replication is used to increase the performance and availability of distributed systems: geo-distributed replicas give us low-latency links increasing performance, and the system remains available even when some replicas are unreachable . Traditional techniques adopting strong consistency give the illusion of a single copy of data by synchronizing replicas on each update. Within the data center, these techniques have been proven very successful [23], but are clearly not suited for wide-area networks, where the synchronization required between replicas increases the latency of requests, and decreases the system's throughput.

There is a well known trade-off [18] in replicated distributed systems: if we want to tolerate network partitions (which effectively happen on wide-area networks [4]), a system can either be highly available or strongly consistent. Pessimistic replication give us the later, while in optimistic replication [25], data consistency is sacrificed in exchange for higher availability. Updates are processed locally and propagated to other replicas in the background, improving the availability of the system: each replica can keep operating even if it can't communicate with others.

In this chapter we start by presenting the system model assumed throughout this thesis. We then present recent data types designs for optimistic replication called *Conflict-free Replicated Data Types* (CRDTs) [26] and two different synchronization strategies for state-based CRDTs: state-based synchronization in Section 2.2 and delta-state-based synchronization in Section 2.3. A portfolio of CRDTs is presented in Section 2.4, and the chapter is concluded in Section 2.5.

2.1 SYSTEM MODEL

Consider a distributed system with nodes containing local memory, with no shared memory between them. Any node can send messages to any other node. The network is asynchronous; there is no global clock, no bound on the time a message takes to arrive, and no bounds on relative processing speeds. The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted). Some messages will, however, eventually get through: if a node sends infinitely many messages to another node, infinitely many of these will be delivered. In particular, this means that there can be arbitrarily long partitions, but these will eventually heal. Nodes have access to durable storage; they can crash but will eventually recover with the content of the durable storage just before the crash occurred. Durable state is written atomically at each state transition. Each node has access to its globally unique identifier in a set $\mathbb{I}$.

2.2 STATE-BASED

A state-based CRDT can be defined as a triple $(S, \sqsubseteq, \sqcup)$ where S is a join-semilattice (lattice from now on), $\sqsubseteq$ its partial order and $\sqcup$ is a binary join operator that derives the least upper bound for every two elements of S, such that $\forall s, t, u \in S$

$$
\begin{aligned}
s \sqcup s &= s && \text{(idempotence)} \\
s \sqcup t &= t \sqcup s && \text{(commutativity)} \\
s \sqcup (t \sqcup u) &= (s \sqcup t) \sqcup u && \text{(associativity)}
\end{aligned}
$$

These properties allow the use of unreliable communication channels when operating with state-based CRDTs as reordering and duplication of messages won't affect the convergence of the system [26]. Moreover, messages can be lost without compromising the correctness of the replicated system (*i.e.* convergence) since the local state is non-decreasing across updates (as we further discuss in Subsection 2.2.1): a message containing the updated state makes messages containing the previous states redundant.

Several state-based CRDTs can be found in the literature [3, 15, 16, 27]. One of the primitive lattices [6] is $\mathsf{MaxInt} = (\mathbb{N}, \leq, \mathsf{max})$ where $\mathbb{N}$ is the set of natural numbers, $\leq$ a total order over the set, and max the binary join operator that returns the maximum of two elements, accordingly to $\leq$. Another lattice can be constructed from any set of elements E, by taking the set of all subsets of E, $\mathcal{P}(E)$, and specifying $(\mathcal{P}(E), \subseteq, \cup)$ where $\subseteq$ is set inclusion and $\cup$ the operator that returns the union of two sets. This defines a known CRDT called *grow-only set*, $\mathsf{GSet}\langle E \rangle$, and Figure 2.1 shows its Hasse diagram with $E = \{a, b, c\}$.

Typically these lattices are bounded lattices, thus a bottom value $\bot$ is defined. For MaxInt, $\bot = 0$ and for $\mathsf{GSet}\langle E \rangle$, $\bot = \{\}$.

$$
\begin{tikzcd}[arrows=dash]
 & \{a,b,c\} \arrow[dl] \arrow[d] \arrow[dr] & \\
\{a,b\} \arrow[d] \arrow[dr] & \{a,c\} \arrow[dl] \arrow[dr] & \{b,c\} \arrow[dl] \arrow[d] \\
\{a\} \arrow[dr] & \{b\} \arrow[d] & \{c\} \arrow[dl] \\
 & \{\} &
\end{tikzcd}
$$

Figure 2.1.: Hasse diagram of $\mathsf{GSet}\langle\{a,b,c\}\rangle$

2.2.1 *Mutators*

State-based CRDTs are updated through a set of mutators M designed to be inflations. We say that $t \in S$ is an inflation of $s \in S$ if $s \sqsubseteq t$. Thus, for every mutator $m \in M$, the following holds:

$$\forall s \in S \cdot s \sqsubseteq \mathsf{m}(s)$$

Figure 2.2 shows a complete specification of a $\mathsf{GSet}\langle E\rangle$, including its mutator add.

$$
\begin{aligned}
\mathsf{GSet}\langle E\rangle &= \mathcal{P}(E) \\
\bot &= \{\} \\
\mathsf{add}(e,s) &= s \cup \{e\} \\
\mathsf{value}(s) &= s \\
s \sqcup s' &= s \cup s'
\end{aligned}
$$

Figure 2.2.: Specification of $\mathsf{GSet}\langle E\rangle$

Note that the specification in Figure 2.2 does not define the partial order since it can always be defined, for any state-based CRDT S, in terms of its binary join operator $\sqcup$:

$$\forall s,t \in S \cdot s \sqsubseteq t \Leftrightarrow s \sqcup t = t$$

2.2.2 *Synchronization algorithm*

Algorithm 1 presents a synchronization algorithm for state-based CRDTs. Each node $i \in \mathbb{I}$ (where $\mathbb{I}$ is the set of node identifiers) is connected with a set of neighbors $n_i \in \mathcal{P}(\mathbb{I})$ (**line** 2) and has in its local durable storage a state-based CRDT S (**line** 4). When update operations are performed (**line** 7) the local state X_i is updated with the result of the mutator.

Periodically, X_i is propagated to all neighbors (**line** 9), behaving as a flood protocol [22]. When a node receives some remote state s (**line** 5), it updates its local state with the result of the binary join of its local state X_i and the received remote state s.

1 **inputs:**
2 $\quad n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
3 **durable state:**
4 $\quad X_i \in S$, CRDT state, $X_i^0 = \bot$
5 **on** $\mathsf{receive}_{j,i}(\mathsf{state}, s)$
6 $\quad X_i' = X_i \sqcup s$
7 **on** $\mathsf{operation}_i(\mathsf{m})$
8 $\quad X_i' = \mathsf{m}(X_i)$
9 **periodically** // ship state
10 $\quad$ **for** $j \in n_i$
11 $\quad\quad \mathsf{send}_{i,j}(\mathsf{state}, X_i)$

Algorithm 1: State-based synchronization algorithm on replica i

This approach can be problematic since the local state has always to be propagated to the neighbors in the system. When the state grows significantly, this might affect the usage of system resources (such as network) with a negative impact on the system performance. However, if each node keeps track of the updates effectively received and processed by its neighbors, it will be able to send a smaller amount of information than the (complete) local state, that represents the state changes since the last synchronization with that neighbor (Subsection 2.3).

Figure 2.3 illustrates an execution with three nodes, **A**, **B** and **C** connected in a *line* topology s.t. **A** $\rightarrow$ **B** $\rightarrow$ **C** (Appendix A), synchronizing a state-based $\mathsf{GSet}\langle E\rangle$:

- all nodes start from bottom value $\bot = \{\}$
- synchronization with neighbors is represented by $\bullet$

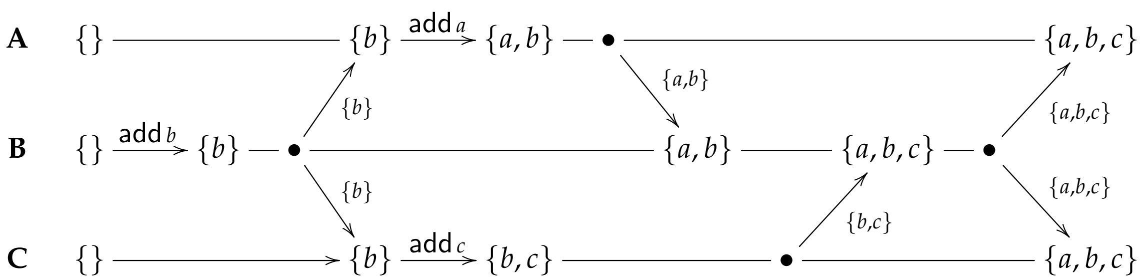

Figure 2.3.: Synchronization of a $\mathsf{GSet}\langle E\rangle$ with three nodes connected in a *line* topology

In this execution, node **B** starts by adding b to the set, and then synchronizes with its neighbors, nodes **A** and **C**, by sending its full state. When these nodes receive the state from **B**, they merge the received state $\{b\}$ with their local state. Then, **A** adds a to the set, and sends the resulting state $\{a, b\}$ to its neighbor **B**; node **C** adds c, and synchronizes with **B** by sending its local state $\{a, c\}$; node **B** merges these two states with its local state resulting in a new state $\{a, b, c\}$. Finally, **B** synchronizes again with its neighbors **A** and **C**, and all nodes converge to the same state.

2.3 DELTA-STATE-BASED

In Figure 2.3 we can notice that, as the state grows, sending the full state can become very expensive. Ideally, as shown in Figure 2.4, a node will only propagate the most recent modifications incurred in its local state.

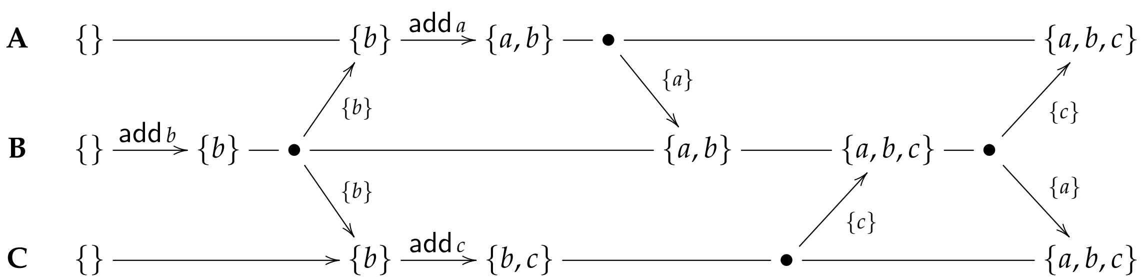

Figure 2.4.: Ideal synchronization of a $\mathsf{GSet}\langle E \rangle$ with three nodes connected in a *line* topology

Delta-state-based CRDTs [2, 3] can be used to achieve this, by modifying the current specification of mutators to return a different state, smaller than the full state, that when merged with the local state, produces the same inflation: the resulting state is the same as it would have been if the mutator was applied. These new mutators are called δ-mutators.

2.3.1 *δ-mutators*

In Section 2.2 we saw that state-based CRDTs are equipped with a set of mutators M. Each of these mutators $m \in M$, has in delta-state-based CRDTs a correspondent δ-mutator $m^\delta \in M^\delta$ such that respects the following property:

$$\forall s \in S \cdot \mathsf{m}(s) = s \sqcup \mathsf{m}^\delta(s) \tag{1}$$

Figure 2.5 shows a complete specification of a $\mathsf{GSet}^\delta\langle E \rangle$, including its δ-mutator add^δ.

$$
\begin{aligned}
\mathsf{GSet}^\delta\langle E\rangle &= \mathcal{P}(E)\\
\bot &= \{\}\\
\mathsf{add}^\delta(e,s) &= \{e\}\\
\mathsf{value}(s) &= s\\
s \sqcup s' &= s \cup s'
\end{aligned}
$$

Figure 2.5.: Specification of $\mathsf{GSet}^\delta\langle E\rangle$

The δ-mutator add^δ respects the aforementioned **Property** 1:

$$
\begin{aligned}
\forall e \in E, \forall s \in S \cdot \mathsf{add}(e,s) &= s \sqcup \mathsf{add}^\delta(e,s) && (\textbf{prop. } 1)\\
s \cup \{e\} &= s \sqcup \mathsf{add}^\delta(e,s) && (\textbf{def. } \mathsf{add})\\
s \cup \{e\} &= s \sqcup \{e\} && (\textbf{def. } \mathsf{add}^\delta)\\
s \cup \{e\} &= s \cup \{e\} && (\textbf{def. } \sqcup)
\end{aligned}
$$

Besides respecting **Property** 1, the δ-state resulting from the δ-mutators should be the smallest state in the lattice (in terms of the partial order) causing that inflation in the local state:

$$
\forall s,t \in S \cdot s \sqcup \mathsf{m}^\delta(s) = s \sqcup t \Rightarrow \mathsf{m}^\delta(s) \sqsubseteq t \tag{2}
$$

δ-mutators that respect **Property** 2 are called *minimum* δ-mutators. The δ-mutator add^δ presented in the $\mathsf{GSet}^\delta\langle E\rangle$ specification (Figure 2.5) is not *minimum* (proof by counterexample):

$$
\begin{aligned}
\forall s,t \in S \cdot s \sqcup \mathsf{add}^\delta(e,s) = s \sqcup t &\Rightarrow \mathsf{add}^\delta(e,s) \sqsubseteq t && (\textbf{prop. } 2)\\
\{a,b\} \sqcup \mathsf{add}^\delta(a,\{a,b\}) = \{a,b\} \sqcup \{\} &\Rightarrow \mathsf{add}^\delta(a,\{a,b\}) \sqsubseteq \{\} && (e=a, s=\{a,b\}, t=\{\})\\
\{a,b\} \sqcup \{a\} = \{a,b\} \sqcup \{\} &\Rightarrow \{a\} \sqsubseteq \{\} && (\textbf{def. } \mathsf{add}^\delta)\\
\{a,b\} = \{a,b\} &\Rightarrow \{a\} \subseteq \{\} && (\textbf{def. } \sqcup, \textbf{def. } \cup, \textbf{def. } \sqsubseteq)\\
\mathsf{True} &\Rightarrow \mathsf{False} && (\textbf{def. } \subseteq)
\end{aligned}
$$

By instantiating $e = a, s = \{a,b\}$ and $t = \{\}$, we reach a contradiction, proving that add^δ is not a *minimum* δ-mutator. In order to have a *minimum* δ-mutator add^δ, we need to inspect the local state (as is typically the case for state-based CRDTs) to decide the resulting δ-state. Figure 2.6 shows a modified specification of $\mathsf{GSet}^\delta\langle E\rangle$ with *minimum* δ-mutators.

$$\begin{aligned}
\mathsf{GSet}^{\delta}\langle E\rangle &= \mathcal{P}(E) \\
\bot &= \{\} \\
\mathsf{add}^{\delta}(e,s) &= \begin{cases} \{e\} & \textbf{if } e \notin s \\ \{\} & \textbf{otherwise} \end{cases} \\
\mathsf{value}(s) &= s \\
s \sqcup s' &= s \cup s'
\end{aligned}$$

Figure 2.6.: Specification of $\mathsf{GSet}^{\delta}\langle E\rangle$ with *minimum* δ-mutators

2.3.2 *Synchronization algorithm*

Algorithm 2 presents a synchronization algorithm for delta-state-based CRDTs [3]. Each node $i \in \mathbb{I}$, besides having in a durable storage the CRDTs state X_i, it stores a monotonic increasing sequence counter c_i (**line** 3). If $c_i = 5$, it means that the local state has suffered five inflations, either by local operations, or by merging some received remote state. As a volatile state (**line** 6), each node keeps an acknowledge (ack) map A_i from node identifier to a sequence counter $n \in \mathbb{N}$ and a δ-buffer B_i which maps sequence numbers $n \in \mathbb{N}$ to lattice states $s \in S$. When operations are performed (**line** 17), the result of the δ-mutator is merged with the local state X_i and added to the δ-buffer map B_i with the current sequence counter c_i as a key. Periodically, a δ-group is propagated to neighbors (**line** 22). This δ-group can either be the local state when the content of the δ-buffer is more advanced that the last received ack from that neighbor, or the join of all entries in the δ-buffer that have not been acknowledged by that neighbor. When a node receives some remote δ-group (**line** 9), it replies with an ack (**line** 14). If the received δ-group will inflate the local state, then it's merged with the local state and added to the δ-buffer with c_i as key. When an ack is received (**line** 15), the ack map A_i is updated with the max of the received sequence number and the current sequence number stored in the map. Garbage collection on the δ-buffer is periodically performed (**line** 30) by removing the entries that have been acknowledged by all neighbors.

Following this algorithm won't result in the desired execution scenario presented in Figure 2.4. In fact, for that example, this algorithm will transmit the same state as the state-based synchronization algorithm would (Figure 2.3). However, if each node keeps track of the origin of the entries in the δ-buffer, and filters them out when computing the δ-group that has to be sent to each neighbor, we'll have the ideal synchronization shown in Figure 2.4. This technique, *avoiding back-propagation of δ-groups*, is further explained in Chapter 4 when proposing modifications to the classic delta-state-based algorithm.

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

Algorithm 2: Delta-state-based synchronization algorithm on replica i

Figure 2.7 shows the previously shown example but with the three nodes connected in a *ring* topology. This execution avoids back-propagation of δ-groups, otherwise we would observe full state being sent in every synchronization among neighbors.

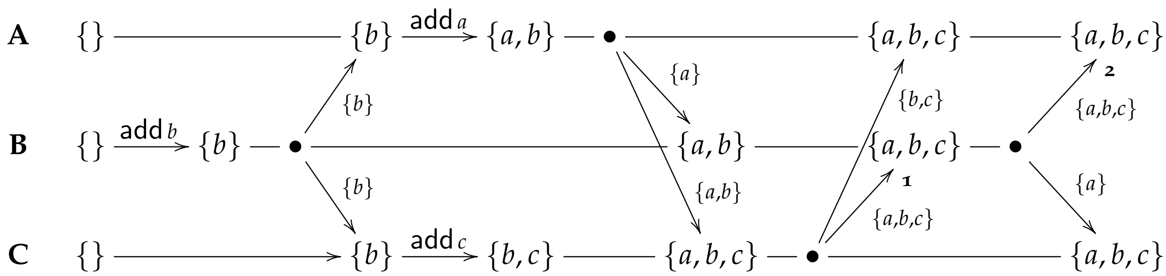

Figure 2.7.: Synchronization of a $\mathsf{GSet}^\delta\langle E\rangle$ with three nodes connected in a *ring* topology avoiding back-propagation of δ-groups

Arrows **1** and **2** can be improved by removing the redundant state present in the received δ-groups, before adding them to the δ-buffer. For example, when node **C** receives $\{a, b\}$, instead of adding $\{a, b\}$ to the δ-buffer, it should only add what causes the inflation in its local state, *i.e.*, $\{a\}$. Then, instead of computing $\{a, b, c\}$ as the δ-group that should be sent to node **B** (arrow **1**), it will compute $\{a, c\}$, as we can see in Figure 2.8.

This technique, *removing redundant state in δ-groups*, is presented in Chapter 4.

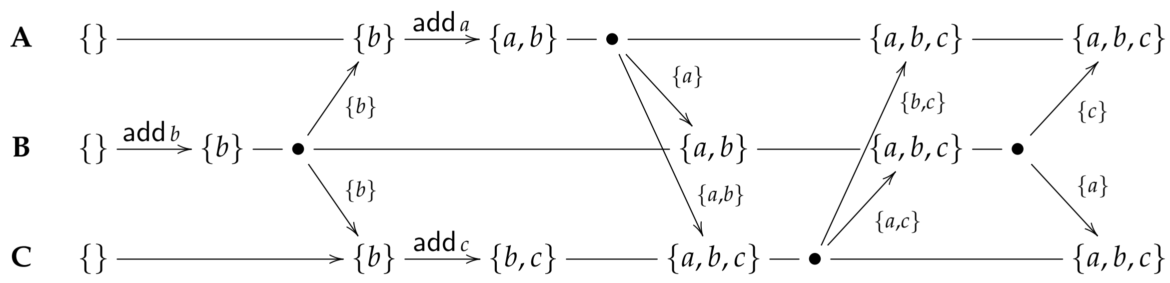

Figure 2.8.: Synchronization of a $\mathsf{GSet}^{\delta}\langle E\rangle$ with three nodes connected in a *ring* topology avoiding back-propagation of δ-groups and removing redundant state present in the received δ-groups

2.4 PORTFOLIO

This section presents a portfolio of state-based CRDTs. In each specification we only define δ-mutators, since the mutator can always be derived using the correspondent δ-mutator, by **Property** 1. Also, δ-mutators are parameterized by node identifier $i \in \mathbb{I}$, even if its behavior does not depend on which replica it is invoked.

TWO-PHASE SET

The only data type introduced so far, $\mathsf{GSet}^{\delta}\langle E\rangle$, only allows elements to be added to the set. It is possible to define a new set data type, $\mathsf{TwoPSet}^{\delta}\langle E\rangle$, that allows additions and removals by pairing two *grow-only set*, using the product $\times$ composition technique [6].

The product $A \times B$ combines two lattices A and B, producing a lattice pair. The join of two pairs $(a, b), (a', b') \in A \times B$, merges each component of the pairs, and it is defined as:

$$(a, b) \sqcup (a', b') = (a \sqcup a', b \sqcup b')$$

The specification of this data type, called *two-phase set*, can be found in Figure 2.9. To add an element to the set, we use mutator add^{δ}, which adds the element to the first component of the pair. Similarly, when we remove an element, we add it to the second component of the pair, using mutator remove^{δ}. Both these mutators resort to the *minimum* δ-mutator add^{δ} defined for $\mathsf{GSet}^{\delta}\langle E\rangle$, being themselves *minimum*. An element is considered to be in the set if it only belongs to the first component.

This simple design allows adding and removing elements, but once an element has been removed, adding it again is not possible (*i.e.*, adding it does not alter the result of value query function).

$$
\begin{aligned}
\mathsf{TwoPSet}^{\delta}\langle E\rangle &= \mathsf{GSet}^{\delta}\langle E\rangle \times \mathsf{GSet}^{\delta}\langle E\rangle \\
\bot &= (\bot, \bot) \\
\mathsf{add}^{\delta}(e, (a, r)) &= (\mathsf{add}^{\delta}(e, a), \bot) \\
\mathsf{remove}^{\delta}(e, (a, r)) &= (\bot, \mathsf{add}^{\delta}(e, r)) \\
\mathsf{value}((a, r)) &= a \setminus r
\end{aligned}
$$

Figure 2.9.: Specification of $\mathsf{TwoPSet}^{\delta}\langle E\rangle$ on replica i

positive counter

A CRDT counter that only allows increments is known as *grow-only counter* or *positive counter* (Figure 2.10), and can be constructed using another composition technique: map $\hookrightarrow$ [6].

A map $K \hookrightarrow V$ maps a set of keys K to a lattice V. The join of two maps $m, m' \in K \hookrightarrow V$ merges the lattice values associated to each key, and it is defined as:

$$m \sqcup m' = \{k \mapsto m(k) \sqcup m'(k) \mid k \in \mathsf{dom}(m) \cup \mathsf{dom}(m')\}$$

When a key $k \in K$ does not belong to some map $m \in K \hookrightarrow V$ (*i.e.* $k \notin \mathsf{dom}(m)$), $m(k)$ returns the bottom value $\bot \in V$; otherwise, it returns the value $v \in V$ associated with k.

In the case of $\mathsf{PCounter}^{\delta}$, the set of node identifiers $\mathbb{I}$ is mapped to the primitive lattice MaxInt presented in Section 2.2. Increments are tracked per node, individually, and stored in a map entry. The value of the counter is the sum of each entry's value in the map.

$$
\begin{aligned}
\mathsf{PCounter}^{\delta} &= \mathbb{I} \hookrightarrow \mathsf{MaxInt} \\
\bot &= \{\} \\
\mathsf{inc}_i^{\delta}(m) &= \{i \mapsto m(i) + 1\} \\
\mathsf{value}(m) &= \sum_{j \in \mathsf{dom}(m)} m(j)
\end{aligned}
$$

Figure 2.10.: Specification of $\mathsf{PCounter}^{\delta}$ on replica i

positive-negative counter

In order to allow increments and decrements, we can store per node a pair of two MaxInt: the first component tracks the number of increments, while the second, the number of decrements. This data type is constructed resorting to the two composition techniques described above.

$$\begin{aligned}
\mathsf{PNCounter}^\delta &= \mathbb{I} \hookrightarrow (\mathsf{MaxInt} \times \mathsf{MaxInt}) \\
\bot &= \{\} \\
\mathsf{inc}^\delta_i(m) &= \{i \mapsto (\mathsf{fst}(m(i)) + 1, \bot)\} \\
\mathsf{dec}^\delta_i(m) &= \{i \mapsto (\bot, \mathsf{snd}(m(i)) + 1)\} \\
\mathsf{value}(m) &= \sum_{j \in \mathsf{dom}(m)} \mathsf{fst}(m(j)) - \mathsf{snd}(m(j))
\end{aligned}$$

Figure 2.11.: Specification of $\mathsf{PNCounter}^\delta$ on replica i

Both these counters suffer from the *identity explosion* problem, having a size linear with the number of nodes that ever manipulated the counter, even when some leave the system. A recent CRDT counter design, called *borrow-counter* [16], addresses this problem by distinguishing transient from permanent nodes and allowing transient nodes to increment the counter as if the increments were performed by a permanent node.

ADD-WINS SET

The $\mathsf{TwoPSet}^\delta\langle E\rangle$ presented has a shortcoming: elements removed cannot be re-added. To circumvent this limitation, some design choices have to be made in order to resolve possible conflicting concurrent operations: operations that are not commutative in their sequential form, *e.g.*, adding and removing the same element from a set, conflict when they occur concurrently. CRDTs solve this conflict deterministically by allowing the element to be in the set (*add-wins*) or not to be in the set (*remove-wins*). An overview of these set semantics can be found in [8].

The *add-wins set* $\mathsf{AWSet}^\delta\langle E\rangle$ belongs to a class of CRDTs called *causal CRDTs* [3]. Causal CRDTs generalize techniques presented in [1, 9] for efficient use of meta-data state. The lattice state of a causal CRDT is formed by a *dot store* and a *causal context*. A causal context is a set of dots $\mathcal{P}(\mathbb{I} \times \mathbb{N})$, where the first component of dot $\mathbb{I} \times \mathbb{N}$ is a node identifier $i \in \mathbb{I}$ and the second a local sequence number $n \in \mathbb{N}$. Function next_i is used by replica i to generate a new dot.

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

Three dot stores are introduced in [3]: DotSet, DotFun and DotMap. $\mathsf{AWSet}^\delta\langle E\rangle$ makes use of two of them:

- $\mathsf{DotSet} : \mathsf{DotStore} = \mathcal{P}(\mathbb{I} \times \mathbb{N})$, a set of dots

- $\mathsf{DotMap}\langle K, V : \mathsf{DotStore}\rangle : \mathsf{DotStore} = K \hookrightarrow V$, a map from a set of keys K to another dot store V

The lattice join for causal CRDTs can be defined as:

$$\mathsf{Causal}\langle T : \mathsf{DotStore}\rangle = T \times \mathsf{CausalContext}$$

$$\textbf{where } T : \mathsf{DotSet}$$
$$(s, c) \sqcup (s', c') = ((s \cap s') \cup (s \setminus c') \cup (s' \setminus c), c \cup c')$$

$$\textbf{where } T : \mathsf{DotMap}\langle \_, \_\rangle$$
$$(m, c) \sqcup (m', c') = (\{k \mapsto v(k) \mid k \in \mathsf{dom}(m) \cup \mathsf{dom}(m') \wedge v(k) \neq \bot\}, c \cup c')$$
$$\textbf{where } v(k) = \mathsf{fst}((m(k), c) \sqcup (m'(k), c'))$$

The intuition here is: if a dot is not present in the dot store but is present in the causal context, it means it was in the dot store before. So, when merging two replicas, a dot is discarded if present in only one dot store and in the causal context of the other:

- $(\{A_1\}, \{A_1\}) \sqcup (\{\}, \{A_1\}) = (\{\}, \{A_1\})$
  A_1 is discarded since it was observed in both, and it is not present in the second dot store
- $(\{A_1\}, \{A_1\}) \sqcup (\{B_1\}, \{A_1, B_1\}) = (\{B_1\}, \{A_1, B_1\})$
  A_1 is again discarded, but B_1 survives because, although it is not present in the first the dot store, it was not observed in its causal context
- $(\{A_1, A_2\}, \{A_1, A_2, B_1\}) \sqcup (\{B_1, B_2\}, \{A_1, B_1, B_2\}) = (\{A_2, B_2\}, \{A_1, A_2, B_1, B_2\})$
  A_1 and B_1 are discarded, while A_2 and B_2 survive
- $(\{k \mapsto \{A_1\}\}, \{A_1\}) \sqcup (\{k \mapsto \{B_1\}\}, \{A_1, B_1\}) = (\{k \mapsto \{B_1\}\}, \{A_1, B_1\})$
  similar to the second example but the DotSet is associated with key k in the DotMap
- $(\{k \mapsto \{A_1\}\}, \{A_1\}) \sqcup (\{\}, \{A_1\}) = (\{\}, \{A_1\})$
  similar to the first example, but since the resulting DotSet is bottom, key k is removed from the DotMap

An $\mathsf{AWSet}^{\delta}\langle E\rangle$ is a DotMap from the set of possible elements E to a DotSet (Figure 2.12). When an element is added to a set, a new dot is created and that element is mapped to a DotSet with only that dot. If the element was already in the set, the dots that were supporting it are removed. To remove an element, we just remove its entry from the DotMap (if concurrently this element is added to the set, the element will survive since a removal only affects the set of dots observed locally). An element is considered to be in the set if it has an entry in the map.

$$
\begin{aligned}
\mathsf{AWSet}^{\delta}\langle E\rangle &= \mathsf{Causal}\langle \mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle\rangle \\
\bot &= \{\} \\
\mathsf{add}^{\delta}_{i}(e,(m,c)) &= (\{e \mapsto \{d\}\}, m(e) \cup \{d\}) \quad \textbf{where} \;\; d = \mathsf{next}_i(c) \\
\mathsf{remove}^{\delta}_{i}(e,(m,c)) &= (\{\}, m(e)) \\
\mathsf{value}((m,c)) &= \mathsf{dom}(m)
\end{aligned}
$$

Figure 2.12.: Specification of $\mathsf{AWSet}^{\delta}\langle E\rangle$ on replica i

2.5 SUMMARY

In this chapter we covered the state-based and delta-state-based algorithms, techniques currently used to synchronize state-based CRDTs. Although the delta-state-based algorithm exploits information nodes have about neighbors, non-naive algorithms to synchronize state-based CRDTs when this information is not available are still missing. We introduce such algorithms in the next chapter.

3

JOIN DECOMPOSITIONS

As we saw in the previous chapter, delta-state-based CRDTs can greatly reduce the amount of information exchanged among nodes during CRDT synchronization. For that, each node should store additional metadata for keeping track of the updates seen by its neighbors in the system (*i.e.* the nodes with which that node directly synchronizes its CRDT replicas). This metadata is stored in a structure called δ-buffer. If nodes are operating over an unstable network where links can fail, this metadata may have to be garbage collected to avoid unbounded growth of the local δ-buffer. When links are restored, nodes can no longer compute a δ-group based on their knowledge, and the full state has to be sent. This is also a problem in highly dynamic overlays, where the set of neighbors is constantly changing.

Δ-CRDTs [29] solve the problem of full state transmission by exchanging metadata used to compute a Δ that reflects the missing updates. In this approach CRDTs need to be extended to maintain the additional metadata for Δ derivation, and if this metadata needs to be garbage collected, the mechanism falls-back to standard full state transmission.

In this chapter we propose an alternative solution that does not require extending current state-based CRDT designs, but instead is able to decompose the local state into smaller states that are selected and grouped in a Δ for efficient transmission. Section 3.1 introduces the concept of Join Decomposition of a state-based CRDT. Section 3.2 proposes two algorithms, *state-driven* and *digest-driven*, that can be used to efficiently synchronize state-based CRDTs when no metadata is available. Section 3.3 presents a portfolio of Join Decompositions for state-based CRDTs and Section 3.4 concludes the chapter.

3.1 JOIN DECOMPOSITIONS OF STATE-BASED CRDTS

Given a lattice state $s \in S$, $D \in \mathcal{P}(S)$ is an (irredundant) join decomposition [10] of s if the join of all elements in the decomposition produces s (**Property 3**) and if each element in the decomposition is not redundant (**Property 4**):

$$\bigsqcup D = s \tag{3}$$

$$\forall d \in D \cdot \bigsqcup (D \setminus \{d\}) \sqsubset s \tag{4}$$

Given $s = \{a, b, c\}$, and the following decomposition examples

✓ $D_1 = \{\{a, b, c\}\}$

✗ $D_2 = \{\{b\}, \{c\}\}$

✗ $D_3 = \{\{a, b\}, \{b\}, \{c\}\}$

✓ $D_4 = \{\{a, b\}, \{c\}\}$

✓ $D_5 = \{\{a\}, \{b\}, \{c\}\}$

$D_2 = \{\{b\}, \{c\}\}$ is not a join decomposition since the join of all its elements does not produce $s = \{a, b, c\}$, *i.e.*, **Property 3** is not respected. $D_3 = \{\{a, b\}, \{b\}, \{c\}\}$ is not a join decomposition because one of its elements, $\{b\}$, is redundant, and thus, it does not respect **Property 4**.

3.1.1 *Join-irreducible states*

An element $s \in S$ is said to be join-irreducible if it cannot result from the join of two elements other that itself:

$$\forall t, u \in S \cdot s = t \sqcup u \Rightarrow (t = s \vee u = s)$$

✓ $s_1 = \{\}$

✓ $s_2 = \{a\}$

✗ $s_3 = \{a, b\}$

$s_2 = \{a\}$ is join-irreducible because when it is obtained by joining two elements in the lattice, one of the elements is itself:

- $s_2 = \{a\} \sqcup \{\}$

- $s_2 = \{\} \sqcup \{a\}$
- $s_2 = \{a\} \sqcup \{a\}$

The same can't be said about $s_3 = \{a, b\}$ since $s_3 = \{a\} \sqcup \{b\}$.

Let $\mathcal{J}(S) \subseteq S$ be the subset of the lattice S containing all the join-irreducible elements of S [11]. If all the elements in a join decomposition are join-irreducible, we have a join-irreducible decomposition:

$$\forall d \in D \cdot d \in \mathcal{J}(S) \tag{5}$$

✗ $D_1 = \{\{a, b, c\}\}$

✗ $D_2 = \{\{a, b\}, \{c\}\}$

✓ $D_3 = \{\{a\}, \{b\}, \{c\}\}$

In this context $D_1 = \{\{a, b, c\}\}$ and $D_2 = \{\{a, b\}, \{c\}\}$ are not join-irreducible decompositions since both have elements that are not join-irreducible, which violates **Property** 5.

Given a state-based CRDT S, its join-irreducible decomposition is given by function $\mathsf{D} : S \to \mathcal{P}(S)$ [17]. Such function for a $\mathsf{GSet}\langle E \rangle$ can be defined as:

$$\mathsf{D}(s) = \{\{e\} \mid e \in s\}$$

3.2 Efficient Synchronization of State-based CRDTs

Consider two nodes, node **A** with $a \in S$, and node **B** with $b \in S$, connected in a *line* topology, such that $\mathbf{A} \to \mathbf{B}$ (Appendix A). At some point the link between the nodes fails, but both keep updating the local state. When the link is restored, what should node **A** send to node **B** so that node **B** observes the updates done on **A** since they stopped communicating? We could try to find a Δ such that:

$$a = b \sqcup \Delta$$

However, if both nodes performed updates while they were offline, in general their local states are concurrent (states $s, t \in S$ are said to be concurrent if $s \not\sqsubseteq t \wedge t \not\sqsubseteq s$) and such Δ does not exist. The trick is how to find a Δ which reflects the updates done in the join of a and b, still missing in b such that:

$$a \sqcup b = b \sqcup \Delta$$

In Algorithm 2 that presents the classic delta-state-based synchronization algorithm, $\Delta = a$ (**line** 25). The goal is to design a protocol that reduces the state transmitted between the two nodes and results in node **B** having the missed updates done on node **A** while they were unable to communicate. This section presents two such algorithms: *state-driven* in Subsection 3.2.1 where node **B** sends its state b to node **A** and **A** computes Δ; and *digest-driven* in Subsection 3.2.2 where **B** sends some information about its state b, smaller that b, but enough for **A** to derive Δ.

3.2.1 *State-driven Synchronization*

The *state-driven* approach can be used to synchronize any state-based CRDT as long as we have its join decomposition. Δ is given by function $\mathsf{min}^{\Delta} : S \times S \to S$ that takes as argument the local state s and the remote state t:

$$\mathsf{min}^{\Delta}((s,t)) = \bigsqcup\{d \in \mathsf{D}(s) \mid t \sqsubset t \sqcup d\}$$

The Δ that results from this function is the join of all states in the join decomposition of the local state s that will strictly inflate the remote state t: a state $s \in S$ is a strict-inflation of t, *i.e.*, $t \sqsubset s$, if $t \sqsubseteq s \wedge t \neq s$. In Section 3.3, when presenting the Join Decompositions portfolio, we will show how to do this inflation checking efficiently for join-irreducible decompositions.

Algorithm 3 presents the *state-driven* synchronization for state-based CRDTs. When a node $i \in \mathbb{I}$ with local state X_i receives a remote state t from $j \in \mathbb{I}$ (**line** 5), it will compute $\Delta = \mathsf{min}^{\Delta}((X_i, t))$, send this Δ to node j, and merge the received state with its local state. When a node receives Δ (**line** 9) it simply merges this Δ with the local state.

Periodically, node $i \in \mathbb{I}$ sends its local state X_i to neighbor $j \in n_i$, if $i > j$ (**line** 15).

1 **inputs:**
2   $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
3 **durable state:**
4   $X_i \in S$, CRDT state, $X_i^0 = \bot$
5 **on** $\mathsf{receive}_{j,i}(\mathsf{state}, t)$
6   $\Delta_i = \mathsf{min}^{\Delta}((X_i, t))$
7   $\mathsf{send}_{i,j}(\mathsf{delta}, \Delta_i)$
8   $X_i' = X_i \sqcup t$
9 **on** $\mathsf{receive}_{j,i}(\mathsf{delta}, \Delta_j)$
10   $X_i' = X_i \sqcup \Delta_j$
11 **on** $\mathsf{operation}_i(\mathsf{m})$
12   $X_i' = \mathsf{m}(X_i)$
13 **periodically** // ship state
14   **for** $j \in n_i$
15   **if** $i > j$
16       $\mathsf{send}_{i,j}(\mathsf{state}, X_i)$

Algorithm 3: *State-driven* synchronization algorithm on replica i

The condition $i > j$ is used to decide which of the two nodes should start the *state-driven* algorithm, since, if both nodes have the initiative, this algorithm will be more bandwidthheavy than the simple state-based approach (Algorithm 1). Condition $i > j$ can be replaced by any predicate $\mathsf{P} : \mathbb{I} \times \mathbb{I} \to \mathbb{B}$ such that:

$$\forall i, j \in \mathbb{I} \cdot \mathsf{P}(i, j) \Rightarrow \neg\, \mathsf{P}(j, i)$$

A relation defined using this predicate as its characteristic function is asymmetric.

In Figure 3.1 we have two nodes, **A** and **B**, connected in a *line* topology, synchronizing a $\mathsf{GSet}\langle E \rangle$. Both start from the same state $\{a, b\}$, node **A** adds x and y to the set, and node **B** adds z. In $\bullet$, the *state-driven* synchronization algorithm starts with **B** sending its full state to **A**, and node **A** replies with a Δ.

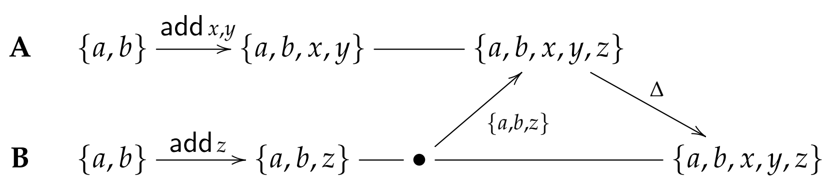

Figure 3.1.: *State-driven* synchronization of a $\mathsf{GSet}\langle E \rangle$ with two nodes connected in a *line* topology

The trivial join decomposition for any state $s \in S$ that does not respect **Property 5** is $\{s\}$, and if the received state is concurrent or less than the local state, we will have full state transmission: Δ would be $\{a, b, x, y\}$ in Figure 3.1. However, with a join-irreducible decomposition, we can reduce Δ to $\{x, y\}$. Without expanding $t = \{a, b, z\}$:

$$\begin{aligned}
\mathsf{min}^{\Delta}((\{a,b,x,y\}, t)) &= \bigsqcup \{ d \in \mathsf{D}(\{a,b,x,y\}) \mid t \sqsubset t \sqcup d \} && (\textbf{def. } \mathsf{min}^{\Delta}) \\
&= \bigsqcup \{ d \in \{\{a\}, \{b\}, \{x\}, \{y\}\} \mid t \sqsubset t \sqcup d \} && (\textbf{def. } \mathsf{D}) \\
&= \bigsqcup \{\{x\}, \{y\}\} && (\textbf{def. } \sqcup, \textbf{def. } \sqsubset) \\
&= \{x, y\} && (\textbf{def. } \textstyle\bigsqcup)
\end{aligned}$$

3.2.2 *Digest-driven Synchronization*

In the *digest-driven* approach, the node that initiates the synchronization procedure, instead of sending its full state as in *state-driven*, only sends a digest $r \in \mathcal{R}$ about its state $s \in S$ that still allows the receiving node to compute a Δ. An immediate consequence is the increased number of messages that have to be exchanged to achieve convergence between the two nodes (Figure 3.2).

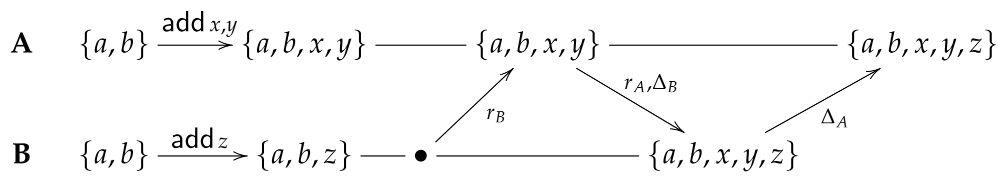

Figure 3.2.: *Digest-driven* synchronization of a $\mathsf{GSet}\langle E\rangle$ with two nodes connected in a *line* topology

Δ is given by function $\mathsf{min}^\Delta : S \times \mathcal{R} \to S$ that takes as argument the local state s and the received digest r:

$$\mathsf{min}^\Delta((s,r)) = \bigsqcup\{d \in \mathsf{D}(s) \mid \mathsf{inf}((d,r))\}$$

The resulting Δ will be the join of all states in the join-irreducible decomposition of the local state that will strictly inflate the remote state: this decision is based on the received digest which is data type specific, and thus a data type specific inflation checking function $\mathsf{inf} : S \times \mathcal{R} \to \mathbb{B}$ is needed. Also, for each state-based CRDT that supports *digest-driven* synchronization, a digest extraction function $\mathsf{digest} : S \to \mathcal{R}$ has to be defined.

Figure 3.3 shows two possible functions for digest extraction and inflation checking for $\mathsf{GSet}\langle E\rangle$. Both functions rely on a third function f that should produce an unique identifier for each element of the set, *i.e.*, function f should be injective.

$$\begin{aligned} \mathsf{digest}(s) &= \{\mathsf{f}(e) \mid e \in s\ \} \\ \mathsf{inf}((e,r)) &= \mathsf{f}(e) \notin r \end{aligned}$$

Figure 3.3.: digest and inf functions for $\mathsf{GSet}\langle E\rangle$

This function f has to be carefully crafted. First, it should further reduce (if possible) the amount of information exchanged between nodes to achieve convergence, when compared to the *state-driven* synchronization. Moreover, the use of non-injective functions, *e.g.*, *hash* functions that are not perfect [28], doesn't guarantee convergence. In order to illustrate this problem let $E = \{a,b,x,y,z\}$, $\mathcal{R} = \mathcal{P}(\mathbb{N})$, and $\mathsf{f} : E \to \mathbb{N}$ such that:

$$\begin{aligned} \mathsf{f}(a) &= 1 \\ \mathsf{f}(b) &= 2 \\ \mathsf{f}(x) &= 2 \\ \mathsf{f}(y) &= 3 \\ \mathsf{f}(z) &= 4 \end{aligned}$$

The same example of Figure 3.2 using function f is depicted in Figure 3.4.

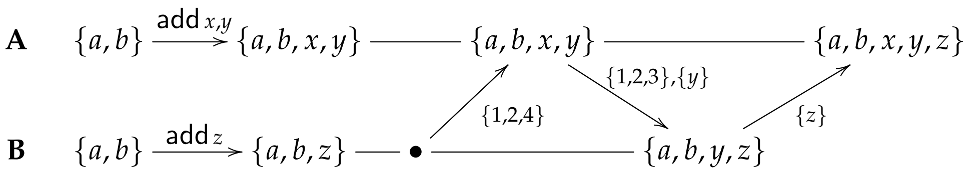

Figure 3.4.: *Digest-driven* synchronization of a $\mathsf{GSet}\langle E\rangle$ with two nodes connected in a *line* topology using a non-injective digest function

We can see that both nodes do not converge to the same state. When node **B** sends $r_B = \{1, 2, 4\}$ as digest, it is implying that it has all the elements $e \in E$ such that $\mathsf{f}(e) \in r_B$, *i.e.*, $\{a, b, x, z\}$, while in fact it only has $\{a, b, z\}$.

Figure 3.4 also hints on another technique that could be designed: the node that computes the first Δ, instead of sending the digest of what it has locally, can instead reply with the digest of what it doesn't have, since it has the digest of the other node.

Algorithm 4 unifies the *state-driven* and *digest-driven* algorithms. It assumes digest is defined for any CRDT (**line** 24): if the data type supports *digest-driven*, it returns the digest, otherwise the CRDT state is returned. When a node receives this digest, it checks (**line** 7) whether it is receiving a CRDT state (*state-driven* algorithm) or a digest (*digest-driven* algorithm). In the first case, the algorithm will proceed as described in Algorithm 3 by sending a Δ to the remote node (**line** 8) and merging the received remote state into the local state (**line** 9). In the second case, the node will send not only this Δ, but also a digest of its local state (**line** 12). When receiving a digest and a Δ (**line** 15), the node computes another Δ given the received digest, sends it, finally merging the received Δ with its local state.

3.3 Portfolio

In this section we will present a portfolio with join decomposition, digest and inflation check functions of the data types presented in Chapter 2. All join decomposition functions produce irreducible decompositions, digest functions are defined even if the data type does not support *digest-driven* synchronization, and inflation checks of join-irreducible states are performed in an efficient way.

Grow-only set

Figure 3.5 defines the join-decomposition function (as seen in Section 3.1) for $\mathsf{GSet}\langle E\rangle$,

inputs:
  $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
durable state:
  $X_i \in S$, CRDT state, $X_i^0 = \bot$
on $\mathsf{receive}_{j,i}(\mathsf{digest}, r)$
  $\Delta_i = \mathsf{min}^{\Delta}((X_i, r))$
  if $r \in S$
    $\mathsf{send}_{i,j}(\mathsf{delta}, \Delta_i)$
    $X_i' = X_i \sqcup r$
  else
    $q = \mathsf{digest}(X_i)$
    $\mathsf{send}_{i,j}(\mathsf{digest}, q, \Delta_i)$
on $\mathsf{receive}_{j,i}(\mathsf{delta}, \Delta_j)$
  $X_i' = X_i \sqcup \Delta_j$
on $\mathsf{receive}_{j,i}(\mathsf{digest}, r, \Delta_j)$
  $\Delta_i = \mathsf{min}^{\Delta}((X_i, r))$
  $\mathsf{send}_{i,j}(\mathsf{delta}, \Delta_i)$
  $X_i' = X_i \sqcup \Delta_j$
on $\mathsf{operation}_i(\mathsf{m})$
  $X_i' = \mathsf{m}(X_i)$
periodically // ship state or digest
  for $j \in n_i$
    if $\mathsf{P}(i, j)$
      $r = \mathsf{digest}(X_i)$
      $\mathsf{send}_{i,j}(\mathsf{digest}, r)$

Algorithm 4: *State-driven* and *Digest-driven* synchronization algorithms on replica i

a digest function that simply returns the lattice state, and an inflation check function for join-irreducible states.

$$
\begin{aligned}
\mathsf{GSet}\langle E\rangle &= \mathcal{P}(E) \\
\mathsf{D}(s) &= \{\{e\} \mid e \in s\} \\
\mathsf{digest}(s) &= s \\
\mathsf{inf}((\{e\}, s)) &= e \notin s
\end{aligned}
$$

Figure 3.5.: Specification of join-decomposition, digest, and inflation check functions for $\mathsf{GSet}\langle E\rangle$

Deciding if merging a lattice state $d \in S$ with another lattice state $s \in S$ will result in an strict inflation can be trivially done by checking if:

$$s \sqsubset s \sqcup d$$

However, if $d \in \mathcal{J}(S)$, *i.e.*, if d is a join-irreducible state, this checking can be done more efficiently. For a $\mathsf{GSet}\langle E\rangle$, we can simply check if the element in the join-irreducible state is already in the set: if it is not in the set, then merging it with the set will result in a strict inflation:

$$s \sqsubset s \sqcup \{e\} \Leftrightarrow e \notin s$$

two-phase set

The join-irreducible decomposition of $\mathsf{TwoPSet}\langle E\rangle$ is presented in Figure 3.6. Each element in the decomposition is irreducible since one component has a singleton set and the other is bottom.

$$\begin{aligned}
\mathsf{TwoPSet}\langle E\rangle &= \mathsf{GSet}\langle E\rangle \times \mathsf{GSet}\langle E\rangle \\
\mathsf{D}((a,r)) &= \{(\{e\},\bot) \mid e \in a\} \cup \{(\bot,\{e\}) \mid e \in r\} \\
\mathsf{digest}((a,r)) &= (a,r) \\
\mathsf{inf}((d,(a,r))) &= \begin{cases} e \notin a & \textbf{if } d = (\{e\},\bot) \\ e \notin r & \textbf{if } d = (\bot,\{e\}) \end{cases}
\end{aligned}$$

Figure 3.6.: Specification of join-decomposition, digest, and inflation check functions for $\mathsf{TwoPSet}\langle E\rangle$

Given

- $s = (\{a,b\},\{a\})$ as local state and
- $t = (\{a,c\},\{\})$ as remote state:

$$\begin{aligned}
\mathsf{min}^{\Delta}((s,t)) &= \bigsqcup\{d \in \mathsf{D}(s) \mid \mathsf{inf}((d,t))\} && (\textbf{def. } \mathsf{min}^{\Delta}) \\
&= \bigsqcup\{d \in \{(\{a\},\bot),(\{b\},\bot),(\bot,\{a\})\} \mid \mathsf{inf}((d,t))\} && (\textbf{def. } \mathsf{D}) \\
&= \bigsqcup\{(\{b\},\bot),(\bot,\{a\})\} && (\textbf{def. } \mathsf{inf}) \\
&= (\{b\},\{a\}) && (\textbf{def. } \bigsqcup)
\end{aligned}$$

POSITIVE COUNTER

The join-irreducible decomposition of PCounter is defined in Figure 3.7. Each element in the decomposition is irreducible since it is a single map entry, and the values in each entry form a total order. Checking if an irreducible state inflates some lattice state can be done by simply testing if the entry's value in the lattice state is lower than the value in the irreducible state.

$$\begin{aligned}
\mathsf{PCounter} &= \mathbb{I} \hookrightarrow \mathsf{MaxInt} \\
\mathsf{D}(m) &= \{\{i \mapsto p\} \mid (i,p) \in m\} \\
\mathsf{digest}(m) &= m \\
\mathsf{inf}((\{i \mapsto p\},m)) &= m(i) < p
\end{aligned}$$

Figure 3.7.: Specification of join-decomposition, digest, and inflation check functions for PCounter

Given

- $s = \{A \mapsto 2, B \mapsto 1, C \mapsto 17\}$ as local state and
- $t = \{A \mapsto 2, C \mapsto 12\}$ as remote state:

$$
\begin{aligned}
\mathsf{min}^{\Delta}((s,t)) &= \bigsqcup\{d \in \mathsf{D}(s) \mid \mathsf{inf}((d,t))\} && (\textbf{def. } \mathsf{min}^{\Delta})\\
&= \bigsqcup\{d \in \{\{A \mapsto 2\},\{B \mapsto 1\},\{C \mapsto 17\}\} \mid \mathsf{inf}((d,t))\} && (\textbf{def. } \mathsf{D})\\
&= \bigsqcup\{\{B \mapsto 1\},\{C \mapsto 17\}\} && (\textbf{def. } \mathsf{inf})\\
&= \{B \mapsto 1, C \mapsto 17\} && (\textbf{def. } \textstyle\bigsqcup)
\end{aligned}
$$

positive-negative counter

Figure 3.8 defines the join decomposition function of PNCounter. Each element in the decomposition is irreducible since it is a single map entry, and the values in each entry are irreducible as well.

$$
\begin{aligned}
\mathsf{PNCounter} &= \mathbb{I} \hookrightarrow (\mathsf{MaxInt} \times \mathsf{MaxInt})\\
\mathsf{D}(m) &= \{\{i \mapsto (p,\bot)\},\{i \mapsto (\bot,n)\} \mid (i,(p,n)) \in m\}\\
\mathsf{digest}(m) &= m\\
\mathsf{inf}((d,m)) &= \begin{cases} \mathsf{fst}(m(i)) < p & \textbf{if } d = \{i \mapsto (p,\bot)\}\\ \mathsf{snd}(m(i)) < n & \textbf{if } d = \{i \mapsto (\bot,n)\}\end{cases}
\end{aligned}
$$

Figure 3.8.: Specification of join-decomposition, digest, and inflation check functions for PNCounter

Given

- $s = \{A \mapsto (10,5)\}$ as local state and
- $t = \{A \mapsto (3,7)\}$ as remote state:

$$
\begin{aligned}
\mathsf{min}^{\Delta}((s,t)) &= \bigsqcup\{d \in \mathsf{D}(s) \mid \mathsf{inf}((d,t))\} && (\textbf{def. } \mathsf{min}^{\Delta})\\
&= \bigsqcup\{d \in \{\{A \mapsto (10,\bot)\},\{A \mapsto (\bot,5)\}\} \mid \mathsf{inf}((d,t))\} && (\textbf{def. } \mathsf{D})\\
&= \bigsqcup\{\{A \mapsto (10,\bot)\}\} && (\textbf{def. } \mathsf{inf})\\
&= \{A \mapsto (10,\bot)\} && (\textbf{def. } \textstyle\bigsqcup)
\end{aligned}
$$

add-wins set

Join Decomposition function of $\mathsf{AWSet}\langle E\rangle$ is defined in Figure 3.9, along with its digest, and inflation check function. Each element in the join decomposition either represents an addition $(\{e \mapsto \{d\}\},\{d\})$ or a removal $(\{\},\{d\})$ of an element. The digest function produces a pair with the set of *active* dots (dots in the dot store supporting the elements in the set) in the first component and the causal context in the second. A join-irreducible state will strictly inflate an $\mathsf{AWSet}\langle E\rangle$ if it has a dot not observed in the causal context ($d \notin c$) or if it represents a removal and the dot is still *active* ($m = \{\} \wedge d \in a$).

$$
\begin{aligned}
\mathsf{AWSet}\langle E\rangle &= \ \mathsf{Causal}\langle \mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle\rangle \\
\mathsf{D}((m,c)) &= \{(\{e \mapsto \{d\}\}, \{d\}) \mid (e,s) \in m \wedge d \in s\} \\
&\quad \cup \{(\{\}, \{d\}) \mid d \in c \setminus \bigcup \mathsf{range}(m)\} \\
\mathsf{digest}((m,c)) &= (\bigcup \mathsf{range}(m), c) \\
\mathsf{inf}(((m,\{d\}),(a,c))) &= d \notin c \vee (m = \{\} \wedge d \in a)
\end{aligned}
$$

Figure 3.9.: Specification of join-decomposition, digest, and inflation check functions for $\mathsf{AWSet}\langle E\rangle$

Given

- $s = (\{x \mapsto \{A_1\}\}, \{A_1, B_1, B_2\})$ as local state,
- $t = (\{x \mapsto \{A_1\}, y \mapsto \{B_2\}\}, \{A_1, B_1, B_2\})$ as remote state and
- $r = \mathsf{digest}(t) = (\{A_1, B_2\}, \{A_1, B_1, B_2\})$ as the digest of remote state t:

$$
\begin{aligned}
\mathsf{min}^{\Delta}((s,r)) &= \bigsqcup\{d \in \mathsf{D}(s) \mid \mathsf{inf}((d,r))\} && (\textbf{def. } \mathsf{min}^{\Delta}) \\
&= \bigsqcup\{d \in \{(\{x \mapsto \{A_1\}\}, \{A_1\}), (\{\}, \{B_1\}), (\{\}, \{B_2\})\} \mid \mathsf{inf}((d,r))\} && (\textbf{def. } \mathsf{D}) \\
&= \bigsqcup\{(\{\}, \{B_2\})\} && (\textbf{def. } \mathsf{inf}) \\
&= (\{\}, \{B_2\}) && (\textbf{def. } \textstyle\bigsqcup)
\end{aligned}
$$

3.4 SUMMARY

In this chapter we introduced the concept of Join Decomposition of a state-based CRDT and showed how it can be used to efficiently synchronize state-based CRDTs in two novel algorithms: *state-driven* and *digest-driven*. In the next chapter we integrate these two algorithms in the delta-state-based synchronization algorithm, and propose some modifications that further lower the amount of data transmitted during synchronization.

4

DELTA-STATE-BASED SYNCHRONIZATION ALGORITHM REVISITED

Chapter 2 presented some enhancements that can reduce the amount of state transmission required by the delta-state-based synchronization algorithm. These enhancements fall into two categories: *sender-based-knowledge* and *receiver-based-knowledge*. The classic delta-state-based algorithm mainly exploits *sender-based-knowledge* by only sending to a neighbor the δ-groups in its δ-buffer unacknowledged by that neighbor. In Section 4.1 we further improve on this by also avoiding to send the implicitly acknowledged δ-groups.

The original algorithm didn't employ any *receiver-based-knowledge* strategy: when a node received a δ-group, it would add it to the δ-buffer, to be further propagated to its neighbors. In this situation, one single update would lead to an infinite cycle of sending, back and forth, that δ-group. The classic algorithm [2, 3] (Algorithm 2) solves this by only adding to the δ-buffer the δ-groups that strictly inflate the local state. In Section 4.2 we explain why this is still not enough, and present an alternative optimal solution.

There is a third category of optimizations, when nodes have no knowledge about the neighbor they will synchronize with. This situation occurs when peers get partitioned by the network and nodes need to forget their knowledge about those peers to avoid unbounded growth of the δ-buffer, or when synchronizing with a new peer due to membership changes. The delta-state-based algorithm contemplates this situation, but full state is exchanged. In Chapter 3 we addressed this problem with the *state-driven* and *digest-driven* algorithms. Section 4.3 integrates these algorithms in revisited delta-state-based synchronization algorithm, avoiding bidirectional full state transmission. Finally, Section 4.4 concludes this chapter.

4.1 AVOIDING BACK-PROPAGATION OF δ-GROUPS

If node **A** sends a δ-group d to node **B**, when **B** decides to synchronize with its neighbors, it should filter out this d when computing the δ-group to be sent to node **A**. This can be achieved by tagging each δ-group in the δ-buffer B with the origin node identifier. Previously, the δ-buffer B was a map with sequence numbers $c \in \mathbb{N}$ as keys and lattice states

$d \in S$ as values. In Algorithm 5, the δ-buffer B is modified in order to keep track of each value's origin in the buffer: keys $c \in \mathbb{N}$ are now mapped to pairs with a lattice state in the first component and node identifier (origin) $j \in \mathbb{I}$ in the second (**line** 8). When receiving a δ-group from some node $j \in \mathbb{I}$, we add to the buffer B a pair formed by the non-redundant state Δ (this optimization is explained in the next section) and the node identifier j (**line** 13). As before, when propagating changes to neighbors, a node first checks if it has enough information to compute a δ-group: if the δ-buffer is empty and the neighbor is missing information ($B_i = \{\} \wedge A_i(j) < c_i$), or the entries in the δ-buffer are in the future of what the node knows about the neighbor ($\mathsf{min}(\mathsf{dom}(B_i)) > A_i(j)$), hence full state has to be sent (**line** 25) (this is addressed in Section 4.3). When the node has enough information, only the unacknowledged entries ($A_i(j) \leq l < c_i$) that are not tagged with this neighbor identifier ($\mathsf{snd}(B_i(l)) \neq j$) are sent (**line** 28).

```
1  inputs:
2    nᵢ ∈ P(I), set of neighbors
3  durable state:
4    Xᵢ ∈ S, CRDT state, Xᵢ⁰ = ⊥
5    cᵢ ∈ N, sequence number, cᵢ⁰ = 0
6  volatile state:
7    Aᵢ ∈ I ↪ N, ack map, Aᵢ⁰ = {}
8    Bᵢ ∈ N ↪ (S × I), buffer, Bᵢ⁰ = {}
9  on receive_{j,i}(delta, d, n)
10   Δ = min^Δ((d, Xᵢ))
11   if ⊥ ⊏ Δ
12     X'ᵢ = Xᵢ ⊔ Δ
13     B'ᵢ = Bᵢ{cᵢ ↦ (Δ, j)}
14     c'ᵢ = cᵢ + 1
15   send_{i,j}(ack, n)
16 on receive_{j,i}(ack, n)
17   A'ᵢ = Aᵢ{j ↦ max(Aᵢ(j), n)}
18 on operationᵢ(m^δ)
19   d = m^δ(Xᵢ)
20   X'ᵢ = Xᵢ ⊔ d
21   B'ᵢ = Bᵢ{cᵢ ↦ (d, i)}
22   c'ᵢ = cᵢ + 1
23 periodically // ship interval or state
24   for j ∈ nᵢ
25     if (Bᵢ = {} ∧ Aᵢ(j) < cᵢ) ∨
           min(dom(Bᵢ)) > Aᵢ(j)
26       send_{i,j}(delta, Xᵢ, cᵢ)
27     else
28       d = ⨆{fst(Bᵢ(l)) | Aᵢ(j) ≤ l < cᵢ
                            ∧ snd(Bᵢ(l)) ≠ j}
29       if ⊥ ⊏ d
30         send_{i,j}(delta, d, cᵢ)
31 periodically // garbage collect deltas
32   l = min{n | (_, n) ∈ Aᵢ}
33   B'ᵢ = {(n, _, _) ∈ Bᵢ | n ≥ l}
```

Algorithm 5: Delta-state-based synchronization algorithm avoiding back-propagation of δ-groups and removing redundant state present in the received δ-groups on replica i

4.2 Removing redundant state in δ-groups

A received δ-group can contain redundant state, *i.e.*, state that has already been propagated to neighbors, or state that is in the δ-buffer B, still to be propagated. This occurs in topologies where the underlying graph is cyclic: nodes can receive the same information from different paths in the graph. In order to detect if a δ-group has redundant state, nodes do not need to keep everything in the δ-buffer or even inspect the δ-buffer: it is enough to compare the received δ-group with the local lattice state X_i. In Algorithm 2, received δ-groups were added to δ-buffer only if they would strictly inflate the local state. In the modified Algorithm 5, we extract from the δ-group what strictly inflates the local state X_i (**line** 10), and if that is different from bottom (**line** 11), then we merge it with X_i and add it to the buffer (**line** 13).

This extraction is achieved with the same technique used in the *state-driven* algorithm, described in Chapter 3. However, instead of selecting which irreducible states from the join decomposition of the local state strictly inflate the received remote state, we select which irreducible states from the join decomposition of the received δ-group strictly inflate the local state.

With this technique, the following property always holds for a given δ-buffer B (if δ-mutators are *minimum*):

$$\bigcap\{\mathsf{D}(b) \mid b \in B\} = \{\}$$

An algorithm, in which δ-buffers respect this property, is *receiver-based* bandwidth-optimal. Further improvements are possible for *sender-based* but require an underlying structured overlay, *e.g.*, a *Plumtree* [21].

4.3 SYNCHRONIZING WITH A NEW NEIGHBOR

Algorithm 6 combines *state-driven* and *digest-driven* algorithms presented in Algorithm 4 with the delta-state-based Algorithm 5 presented previously in this chapter.

If a node has no information about the neighbor it wants to synchronize with, instead of sending its full state, it starts the *state-driven* or *digest-driven* algorithm (**line** 40), depending which algorithm the data type supports (or some configuration, given that *state-driven* is always possible, as long as the join decomposition is defined for that data type).

As in Algorithm 4, when receiving this message, the node checks if it is receiving a lattice state or a digest (**line** 20), to ensure that the correct technique is employed. In the case of *state-driven*, instead of directly merging the received state with the local state, a node triggers the receipt of a delta (**line** 22), adding the received information (what strictly inflates) to the δ-buffer and merging that information with the local state. This trigger also occurs when receiving the second message of the *digest-driven* algorithm (**line** 29).

```
inputs:
  nᵢ ∈ 𝒫(𝕀), set of neighbors
durable state:
  Xᵢ ∈ S, CRDT state, Xᵢ⁰ = ⊥
  cᵢ ∈ ℕ, sequence number, cᵢ⁰ = 0
volatile state:
  Aᵢ ∈ 𝕀 ↪ ℕ, ack map, Aᵢ⁰ = {}
  Bᵢ ∈ ℕ ↪ (S × 𝕀), buffer, Bᵢ⁰ = {}
on receive_{j,i}(delta, d, n)
  Δ = min^Δ((d, Xᵢ))
  if ⊥ ⊏ Δ
    X'ᵢ = Xᵢ ⊔ Δ
    B'ᵢ = Bᵢ{cᵢ ↦ (Δ, j)}
    c'ᵢ = cᵢ + 1
  send_{i,j}(ack, n)
on receive_{j,i}(ack, n)
  A'ᵢ = Aᵢ{j ↦ max(Aᵢ(j), n)}
on receive_{j,i}(digest, r, n)
  Δᵢ = min^Δ((Xᵢ, r))
  if r ∈ S
    send_{i,j}(delta, Δᵢ, cᵢ)
    receive_{j,i}(delta, r, n)
  else
    q = digest(Xᵢ)
    send_{i,j}(digest, q, Δᵢ, cᵢ)
on receive_{j,i}(digest, r, Δⱼ, n)
  Δᵢ = min^Δ((Xᵢ, r))
  send_{i,j}(delta, Δᵢ, cᵢ)
  receive_{j,i}(delta, Δⱼ, n)
on operationᵢ(m^δ)
  d = m^δ(Xᵢ)
  X'ᵢ = Xᵢ ⊔ d
  B'ᵢ = Bᵢ{cᵢ ↦ (d, i)}
  c'ᵢ = cᵢ + 1
periodically // ship interval, state or digest
  for j ∈ nᵢ
    if (Bᵢ = {} ∧ Aᵢ(j) < cᵢ) ∨
        min(dom(Bᵢ)) > Aᵢ(j)
      if P(i, j)
        r = digest(Xᵢ)
        send_{i,j}(digest, r, cᵢ)
    else
      d = ⨆{fst(Bᵢ(l)) | Aᵢ(j) ≤ l < cᵢ
                          ∧ snd(Bᵢ(l)) ≠ j}
      if ⊥ ⊏ d
        send_{i,j}(delta, d, cᵢ)
periodically // garbage collect deltas
  l = min{n | (_, n) ∈ Aᵢ}
  B'ᵢ = {(n, _, _) ∈ Bᵢ | n ≥ l}
```

Algorithm 6: Delta-state-based synchronization algorithm avoiding back-propagation of δ-groups, removing redundant state present in the received δ-groups, and resorting to *State-driven* and *Digest-driven* synchronization algorithms when synchronizing with new neighbors on replica i

4.4 SUMMARY

In this chapter we revisited the delta-state-based algorithm and proposed modifications that improve the state transmission, as we will show in the next chapter when evaluating the thesis contributions. We have also defined a sufficient condition for a *receiver-based* bandwidth-optimal algorithm. Interestingly, the same technique used for Δ derivation in the *state-driven* and *digest-driven* algorithms can also be used to design such algorithm.

5

EVALUATION

In this chapter we evaluate the theoretical contributions presented in Chapters 3 and 4. For that, we set out to answer the following questions:

- How does the *state-driven* and *digest-driven* algorithms compare to the state-based algorithm?
- How does the classic delta-state-based algorithm compare to the state-based algorithm?
- What's the effect of *avoiding back-propagation of δ-groups* and *removing redundant state in δ-groups* in the delta-state-based algorithm?
- What's the impact of the data type and the underlying topology in the proposed modifications?
- If a network partition occurs and nodes are forced to forget what they know about neighbors, how does the *state-driven* and *digest-driven* algorithms compare to bidirectional full-state transmission in the delta-state-based algorithm?

To this end, we have implemented a set of libraries that are presented in 5.1, along with the experimental setup used for the evaluation. In Section 5.2 we show the evaluation results, and this chapter is concluded in Section 5.3.

5.1 EXPERIMENTAL SETUP

In order to evaluate the proposed solutions, we have implemented several libraries.

TYPES

This library [20] of state-based CRDTs[1] implements several data types [3, 5, 6, 27]:

1 *types* is also a library of *pure-op based CRDTs* [7]

- Boolean, MaxInt
- DWFlag, EWFlag
- LWWRegister, MVRegister
- PCounter, PNCounter, LexCounter, BoundedCounter
- GSet, TwoPSet, ORSet, AWSet
- Pair, GMap, AWMap

These data types are all equipped with δ-mutators (and mutators defined through these δ-mutators), a binary join, inflation and strict inflation check, and query functions. Some data types also define join decomposition, digest and Δ derivation functions.

partisan

This library [19] is a scalable peer service prototype designed for Lasp [24]. It provides three different backends:

- *Default* (full membership): $\forall i \in \mathbb{I} \cdot n_i = \mathbb{I} \setminus \{i\}$
- *Client-Server* (star topology): $n_i = \mathbb{I} \setminus \{i\}$ if i is a server, and $|n_i| = 1$ if i is a client (this unique neighbor is a server)
- *HyParView* [22]: a protocol that maintains the invariant $\forall i \in \mathbb{I} \cdot a \leq |n_i| \leq b$, where a and b are the minimum and maximum size of the active view, respectively

We have extended *partisan* with a *Static* backend where connections are performed explicitly between nodes, giving us total control on the topology employed. This backend will be used to run experiments on top of *line* and *ring* topologies.

ldb

This library [12] is a CRDT *key-value store* that leverages both *types* and *partisan* libraries for the implementation of the state-based and delta-state-based synchronization backends[2].

In both backends is possible to synchronize replicas using *state-driven* and *digest-driven* algorithms (if that option is enabled by configuration). In the case of delta-state-based, this synchronization only occurs if the node has no information about the neighbor, then resuming to normal operation by sending δ-groups. The modifications proposed to the delta-state-based algorithm in Chapter 4, *avoiding back-propagation of δ-groups* (BP) and *removing redundant state in δ-groups* (RR), were also implemented, allowing us to measure

2 *ldb* also provides a *pure-op based* backend

their effect, when enabled. Synchronization with neighbors occurs periodically, given some configurable interval.

If enabled, an *ldb* node will collect metrics regarding:

- size of messages sent
- latency creating messages to send
- latency processing messages received

LSIM

This library [13] provides the necessary infrastructure to run *ldb* simulations on top of **Kubernetes**[3]. It provides:

- a set of simulations where each event is a CRDT update, with configurable number of events and its frequency:
  – PCounter, where each event is an *increment*
  – GSet, where each event is an *addition* of a globally unique element to the set
  – AWSet, where 75% of events are *additions* and 25% are *removals*
- different topologies: *line*, *ring* and *HyParView*
- creation of network partitions using **iptables**[4]
- metrics archival in a **Redis**[5] instance
- a special *lsim* node only responsible for orchestrating the experiments:
  – when all nodes are running and connected to neighbors, instruct them to start the simulation
  – if enabled, start and end network partitions
  – when all nodes announce the end of the simulation (finished generating events and observed all the events from other nodes), instruct them to archive the metrics collected during the experiment
  – when all nodes archived their metrics in **Redis**, shutdown nodes

By default, network partitions are disabled, and thus, the topology forms a single connected component. If enabled, network partitions start and end when 50% and 75% of the

3 https://kubernetes.io/
4 https://help.ubuntu.com/community/IptablesHowTo
5 https://redis.io/

events were generated, respectively, with a configurable number of connected components to be created. It is only possible to completely control the number of connected components created if the topology being employed is *Static*.

In order to run *lsim* on **Kubernetes**, this application was containerized using **Docker**[6].

lsim-dash

All the libraries mentioned so far were written in **Erlang**[7]. This library [14] was implemented using **Meteor**[8], a **JavaScript**[9] framework. It periodically fetches information from:

- **Kubernetes**, to know which experiments are currently running, and information (*e.g.* IP and web port) about the nodes in each experiment
- **Running nodes**, to know to which nodes they are connected to (each node exposes a web API with membership information)
- **Redis**, to know which experiments have already ended

This dashboard (Figure 5.1) was specially useful when network partitions were enabled, helping us understand the resulting topology.

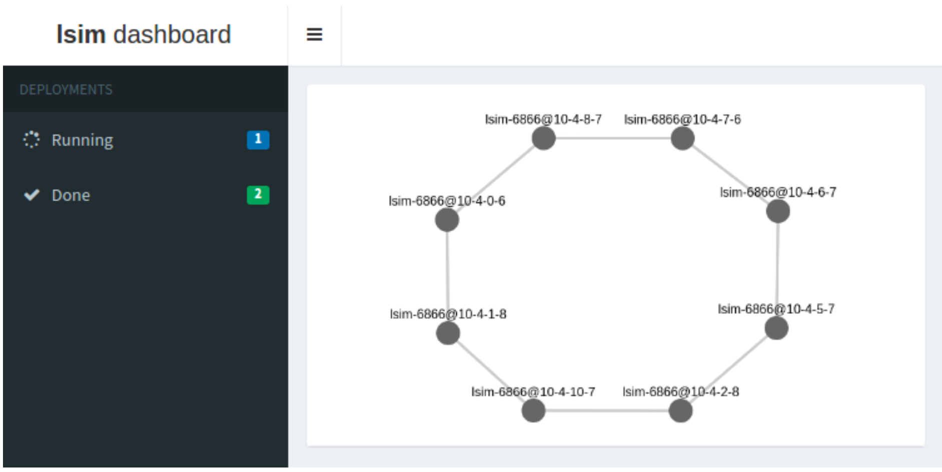

Figure 5.1.: Dashboard

The experiments were run on **Google Container Engine**[10], which uses **Kubernetes** for container orchestration. The machine type used was **n1-standard-1** (1 virtual CPU and 3.75 GB of RAM), with all machines allocated within the same availability zone. The **Kubernetes** cluster size depended on the number of nodes of the experiment, but it was always set

6 https://www.docker.com/
7 http://www.erlang.org/
8 https://www.meteor.com/
9 https://www.javascript.com/
10 https://cloud.google.com/container-engine/

ensuring it was big enough so that two nodes (pods) were not scheduled on the same virtual machine.

5.2 RESULTS

In this section we present the results of the evaluation. All experiments were run with 8 nodes, with the event generation interval and the synchronization interval set at 1 second. In each simulation, the number of generated events per node was 100.

5.2.1 *State-based, state-driven and digest-driven synchronization algorithms*

First we compared the state-based algorithm with *state-driven* and *digest-driven* algorithms on top of three different network topologies, *line*, *ring* and *HyParView*, and with three different simulations GSet, PCounter and AWSet.

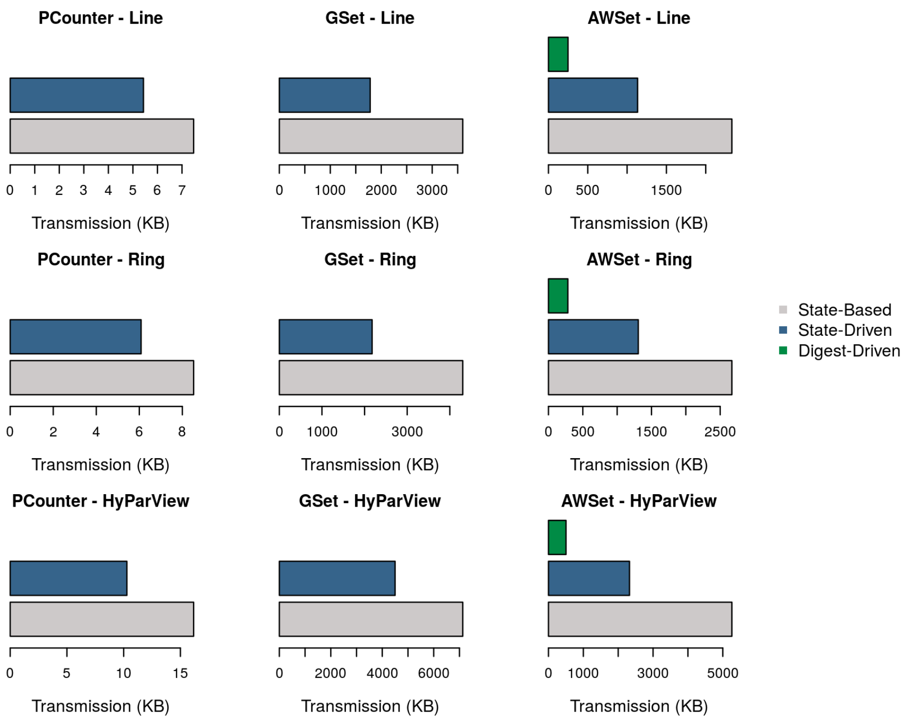

Figure 5.2.: Accumulated transmission of state-based, *state-driven* and *digest-driven* algorithms for *line*, *ring* and *HyParView* topologies

Figure 5.2 shows the accumulated transmission for all the nine configurations. Overall, both *state-driven* and *digest-driven* synchronization algorithms reduce the amount of information exchanged among nodes. In the case of PCounter, *state-driven* is only a small improvement when comparing to *state-driven* since the CRDT state size (linear with the number of nodes in the system) is constant throughout the simulation. In the case of AWSet, the only simulation that supports *digest-driven*, we see that both algorithms reduce information transmission by a considerable amount. However, from these results we cannot say that, *e.g.*, *state-driven* is two times better than state-based, since the absolute values depend on the length of the run. For example, doubling the number of events per node in the simulations would result in a bigger gap between the accumulated transmission of state-based and the two other algorithms.

When comparing GSet and AWSet, one would expect the former to have a smaller accumulated transmission since the data type does not require extra information for causality tracking and conflict-resolution to be stored in the lattice state. In these results, this is not observable and that comes from the fact that the AWSet simulation performs removals throughout the experiment.

In terms of the topologies employed, we can see that the ones with higher number of links (*HyParView* > *ring* > *line*) have higher total transmission, as expected.

Figure 5.3 shows the local latency (time it takes to create a message to send) and remote latency (time it takes to process a message received) CDF, with logarithmic scale on the Latency axis, for state-based, *state-driven* and *digest-driven* synchronization algorithms, with the same three simulations on top of the *HyParView* topology.

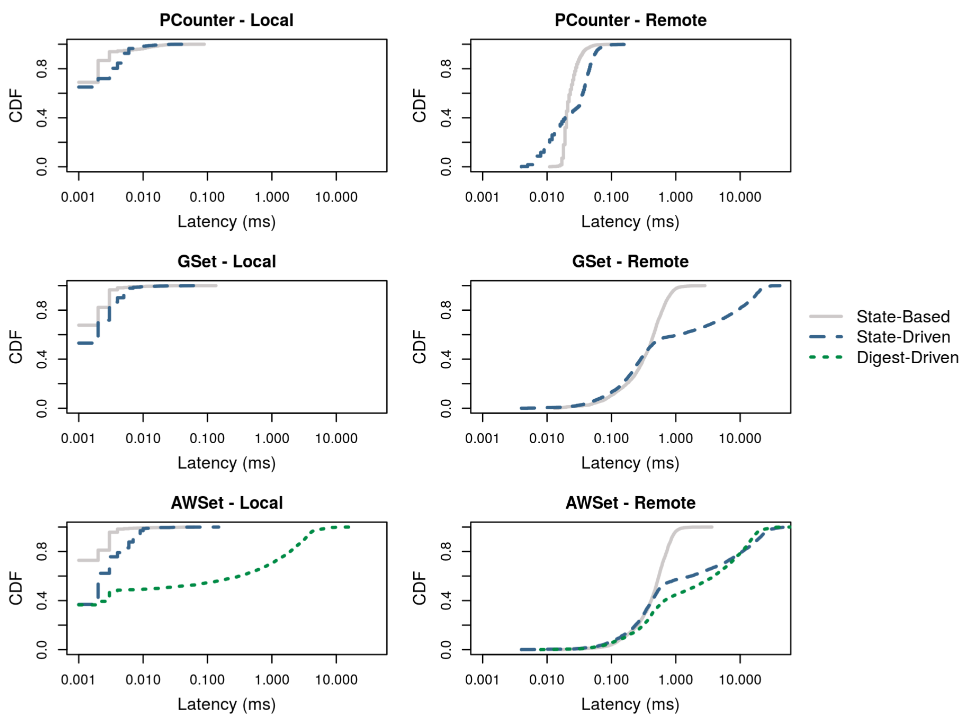

Figure 5.3.: Local and remote latency CDF of state-based, *state-driven* and *digest-driven* algorithms for *HyParView* topology

Both *state-driven* and *digest-driven* incur a penalty in terms of computation required to execute the algorithm. Locally, this penalty is more noticeable in the AWSet simulation with *digest-driven* synchronization algorithm due to the computation required to compute the digest of the lattice state. Remotely, the penalty results from the extra computation needed to calculate the Δ. This penalty is fairly the same either using a lattice state or a digest, in the case of AWSet.

For brevity, we only presented the results for the *HyParView* topology. However, the conclusions presented here could have been established using the results of the two other topologies.

5.2.2 *Delta-state-based synchronization algorithm*

With the first question answered in the previous subsection (How does the *state-driven* and *digest-driven* algorithms compare to the state-based algorithm?), in this subsection we are targeting the next three:

- How does the classic delta-state-based algorithm compare to the state-based algorithm?
- What's the effect of *avoiding back-propagation of δ-groups* (BP) and *removing redundant state in δ-groups* (RR) in the delta-state-based algorithm?
- What's the impact of the data type and the underlying topology in the proposed modifications?

The experiments were run on top of *line*, *ring* and *HyParView* topologies, with GSet, PCounter and AWSet simulations, and with state-based and delta-state-based synchronization algorithms, enabling/disabling the optimizations presented in the previous chapter. In total, we have 5 different possible configurations:

- State-based
- Delta-state-based
- Delta-state-based *avoiding back-propagation of δ-groups* (BP)
- Delta-state-based *removing redundant state in δ-groups* (RR)
- Delta-state-based BP + RR

Figure 5.4 shows the accumulated transmission for this set of experiments. We can observe that the classic delta-state-based synchronization algorithm has almost the same total amount of transmission exchanged among nodes as the state-based algorithm.

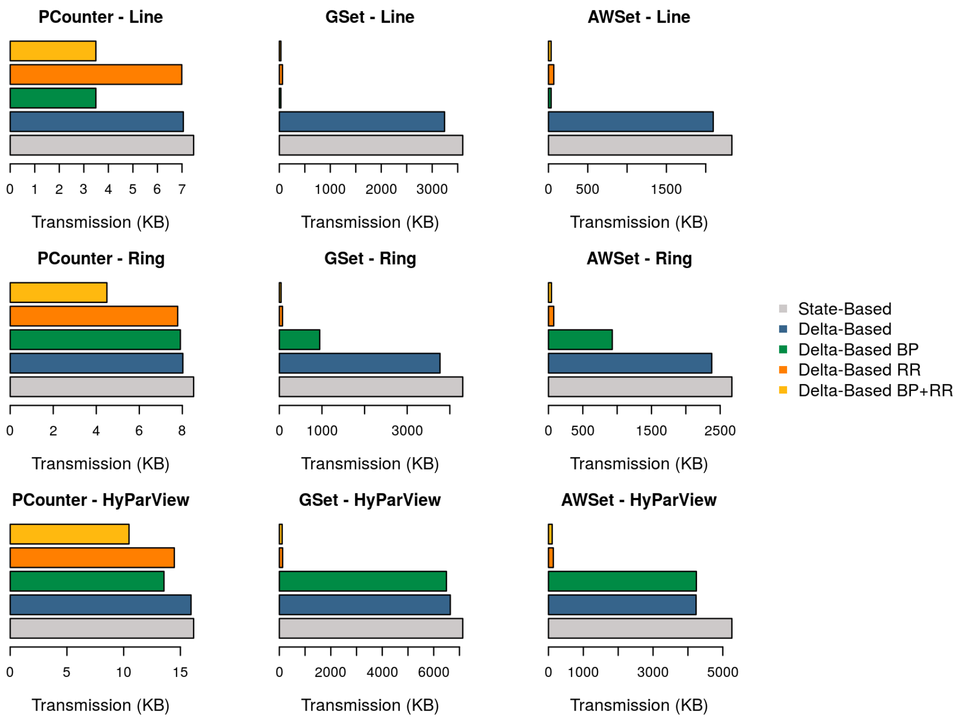

Figure 5.4.: Accumulated transmission of state-based, delta-based, delta-based BP, delta-based RR and delta-based BP + RR algorithms for *line*, *ring* and *HyParView* topologies

Employing our proposed modifications (BP and RR) greatly reduces the amount of information exchanged. The BP optimization is enough for acyclic topologies (see for example **GSet - Line**).

When the topology is cyclic, as is the case of *ring* and *HyParView*, it is necessary to discard redundant information that may be received by some node from different paths in the topology. This can be done with the RR optimization. The more cycles the topology has, the more relevant this optimization is, when comparing to the BP optimization (see for example **GSet - Ring** vs **GSet - HyParView**).

Once again, the constant size of the CRDT state in the PCounter simulation makes these optimizations less relevant.

Figure 5.5 shows the local and remote latency CDF, with logarithmic scale on the Latency axis, for the same set of 5 possible configurations on top of the *HyParView* topology.

Locally it's not easy to do better than state-based since there's no computation performed on this algorithm when a message is sent. In the case of delta-state-based, in all the configurations, it is necessary to compute the δ-group to be sent to each neighbor. This implies selecting which of entries of the δ-buffer haven't been effectively received by a neighbor, and in the end merge all these entries to compute the δ-group. Both BP and RR optimizations imply less or smaller entries to be merged, respectively, and thus these variants perform

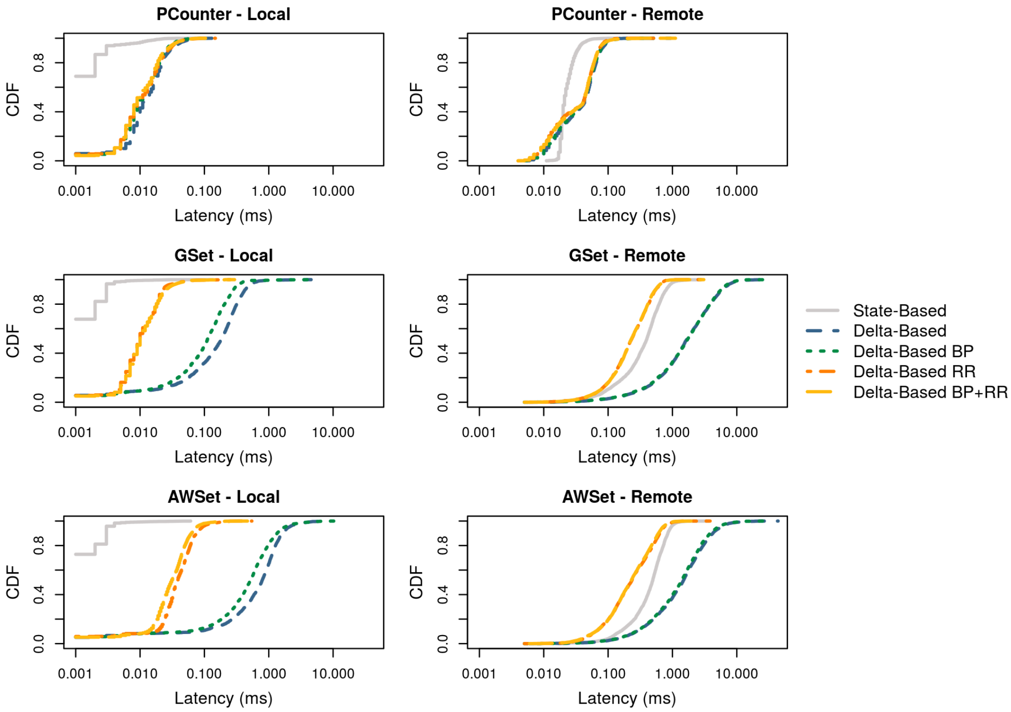

Figure 5.5.: Local and remote latency CDF of state-based, delta-based, delta-based BP, delta-based RR and delta-based BP + RR algorithms for *HyParView* topology

better locally than the classic delta-state-based algorithm: the BP optimization performs slightly better, while RR greatly reduces the computation time required.

Remotely, in the case of the state-based, the only computation required is to merge the received lattice state with the local state. In the case of delta-state-based and delta-state-based BP, first both check if the received δ-group will provoque a strict-inflation in the local state, and if it will, this δ-group is merged with the local state and added to the δ-buffer. As we can see in Figure 5.5, these two variants have equivalent performance and that comes from the fact that the amount of state transmitted (which is what has an impact on inflation-checking and merging computation time) in the *HyParView* topology is the same (as we can observe in Figure 5.4). And since the amount of state transmitted of these two variants is similar to state-based, these variants actually perform worse than state-based (since state-based only has to merge, while these variants have to check for strict-inflation before merging).

One consequence of sending less state, is that merging is also less expensive. The RR optimization is not only better that delta-state-based and delta-state-based BP, but also better than state-based.

5.2.3 *Delta-state-based with state-driven and digest-driven synchronization algorithms*

At this point, there's only question left to be answered:

- If a network partition occurs and nodes are forced to forget what they know about neighbors, how does the *state-driven* and *digest-driven* algorithms compare to bidirectional full-state transmission in the delta-state-based algorithm?

In order to answer this question, we designed a controlled experiment where we induce network partitions in the middle of the execution (when 50% of the events were generated). We have total control on the resulting overlay, since the experiment was run using a *Static* topology, namely a *ring* topology. As the other experiments, this one was run with 8 nodes and the partitions were induced in order to create either 2 connected components (each component with 4 nodes) or 4 connected components (each component with 2 nodes), as shown in Figure 5.6. When 75% of the events were generated, partitions were healed, and again a single connected components was created, in the form of a *ring*.

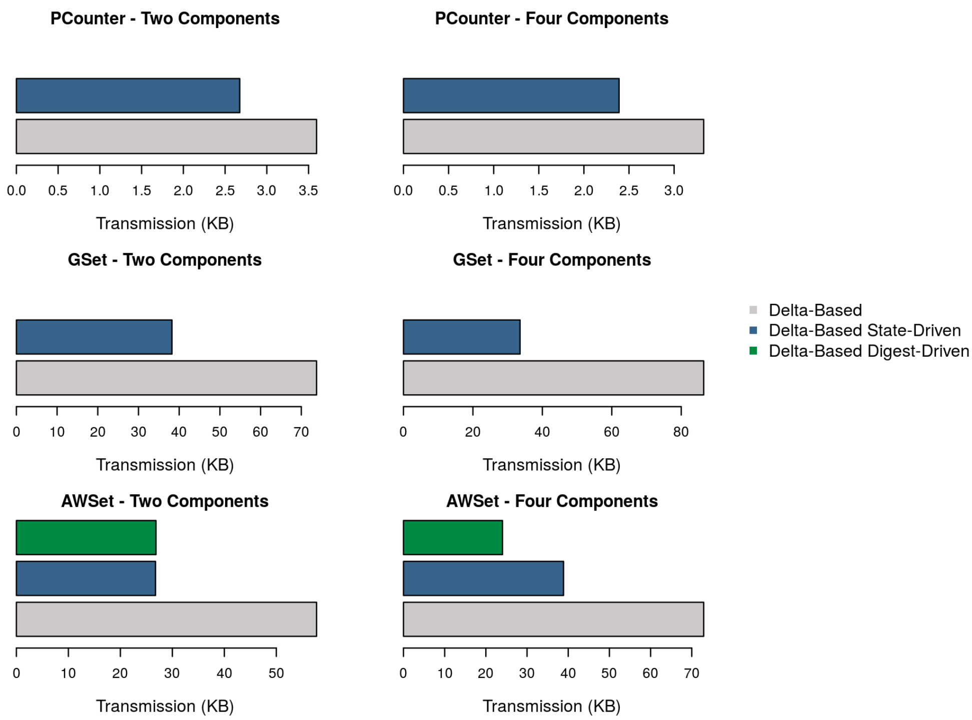

Figure 5.6.: Accumulated transmission of delta-based BP + RR, delta-based BP + RR with *state-driven* and delta-based BP + RR with *digest-driven* algorithms for *ring* topology with induced partitions

When partitions heal, nodes from different partitions need to synchronize, and this synchronization was performed in three different ways:

- bidirectional full state transmission (as in the classic delta-state-based algorithm)

- *state-driven* synchronization algorithm

- *digest-driven* synchronization algorithm

In Figure 5.6 we observe that these novel synchronization algorithms are very effective, and don't incur a noticeable penalty in terms of computation time, as shown in Figure 5.7.

To be noted that this experiment was performed with the optimization RR in the delta-state-based algorithm, and that benefits the first two synchronization strategies, but mostly the first (bidirectional full state transmission), in the long run. In the classic delta-state-based algorithm, full state transmission as a synchronization mechanism means that the whole state would be added to the δ-buffer to be further propagated. This would result in an increased total transmission.

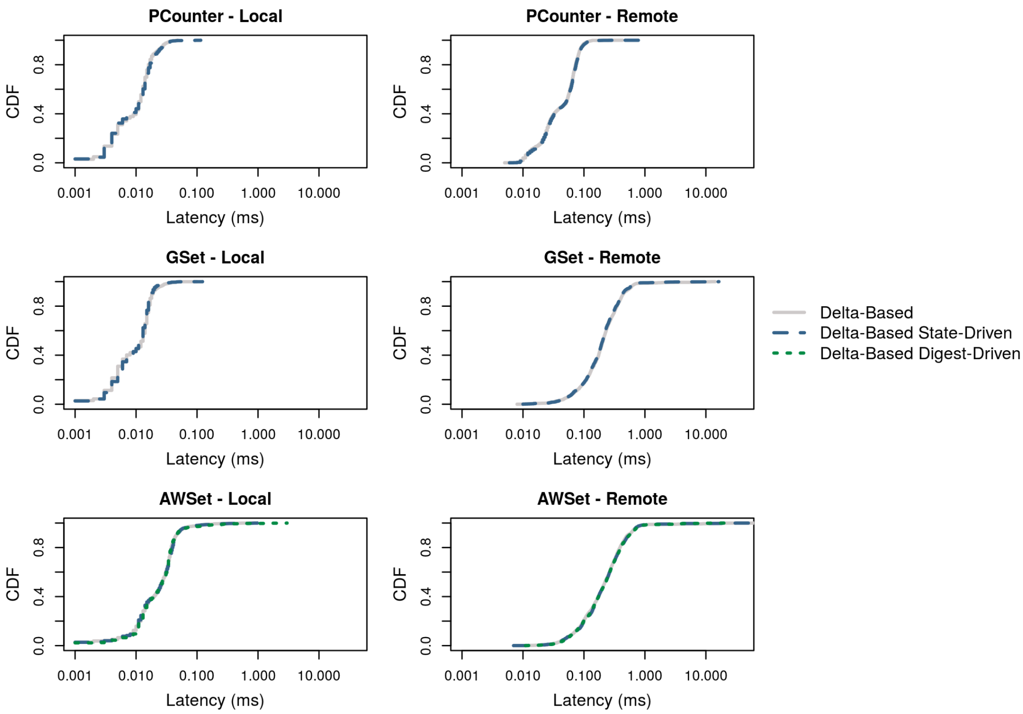

Figure 5.7.: Local and remote latency CDF of algorithms delta-based BP + RR, delta-based BP + RR with *state-driven* and delta-based BP + RR with *digest-driven* for *ring* topology with induced partitions

5.3 SUMMARY

In this chapter we have introduced the set of libraries implemented in order to validate our theoretical contributions. With these libraries we have designed a set of experiments that show the benefits of integrating in the classic delta-state-based algorithm the novel algorithms presented in Chapter 3 and the effect of the optimizations *avoiding back-propagation of δ-groups* and *removing redundant state in δ-groups* presented in Chapter 4.

6

CONCLUSION

In this thesis we revisited the delta-state-based algorithm and defined three categories of optimizations in terms of state transmission: *sender-based-knowledge*, *receiver-based-knowledge* and *lack-of-knowledge*.

In the first, *sender-based-knowledge*, a node exploits the knowledge it has about its neighbors in order to not send information that it has been acknowledged (explictly and impliclty) by a neighbor. In this category, we have presented an optimization, *avoiding back-propagation of δ-groups*, that avoids to send implictly acknowledged entries in the δ-buffer, *i.e.*, entries that were received from that neighbor.

The second category, *receiver-based-knowledge*, includes actions performed by a node when it receives a δ-group from a neighbor, and it was practically not addressed by previous versions of the delta-state-based algorithm. We have introduced an optimization, *removing redundant state in δ-groups*, that ensures optimal additions to the δ-group, *i.e.*, a node only adds to the buffer new information, and thus it avoids sending repeated information to neighbors. This optimization was proved to be very effective during the evaluations performed.

The third category of optimizations, *lack-of-knowledge*, includes worst case scenario situations where nodes don't have any knowledge about the neighbor they will synchronize with. In this category, we have presented two novel algorithms, *state-driven* synchronization algorithm and *digest-driven* synchronization algorithm, and integrated them in the delta-state-based algorithm.

The optimizations presented in the last two categories are possible due to the concept of join decompositions of state-based CRDTs introduced in Chapter 3.

As future work, we intend to explore a bandwith-optimal solution for the *sender-based-knowledge* category, as we did for the *receiver-based-knowledge* category. The classic delta-state-based synchronization algorithm has been shown to respect per-object causal consistency, and we plan to do the same for *state-driven* and *digest-driven* algorithms, and for their inclusion in the delta-state-based algorithm as a *lack-of-knowledge* synchronization mechanism.

BIBLIOGRAPHY

[1] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno M. Preguiça, and Victor Fonte. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. In *Distributed Applications and Interoperable Systems*, 2014.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-Based CRDTs by Delta-Mutation. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, pages 62–76, 2015.

[3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta State Replicated Data Types. *CoRR*, abs/1603.01529, 2016.

[4] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, 2014.

[5] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno M. Preguiça. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *34th IEEE Symposium on Reliable Distributed Systems*, 2015.

[6] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. Composition of State-based CRDTs. Technical report, 2015.

[7] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 126–140, 2014.

[8] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief Announcement: Semantics of Eventually Consistent Replicated Sets. In *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, pages 441–442, 2012.

[9] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.

[10] Garrett Birkhoff. Rings of sets. *Duke Mathematical Journal*, 3(3):443–454, 1937.

[11] Robert P. Dilworth. Lattices with Unique Irreducible Decompositions. *Annals of Mathematics*, 41(4):771–777, 1940.

[12] Vitor Enes. ldb. URL https://github.com/vitorenesduarte/ldb, Retrieved 17-jun-2017.

[13] Vitor Enes. lsim. URL https://github.com/vitorenesduarte/lsim, Retrieved 17-jun-2017.

[14] Vitor Enes. lsim-dash. URL https://github.com/vitorenesduarte/lsim-dash, Retrieved 17-jun-2017.

[15] Vitor Enes, Paulo Sérgio Almeida, and Carlos Baquero. The Single-Writer Principle in CRDT Composition. In *Second Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2017, Barcelona, Spain, June 20*, 2017.

[16] Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and João Leitão. Borrowing an Identity for a Distributed Counter: Work in Progress Report. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC'17, pages 4:1–4:3. ACM, 2017.

[17] Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Join Decompositions for Efficient Synchronization of CRDTs after a Network Partition: Work in progress report. In *First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17*, page 6, 2016.

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

[19] Lasp. partisan. URL https://github.com/lasp-lang/partisan, Retrieved 17-jun-2017.

[20] Lasp. types. URL https://github.com/lasp-lang/types, Retrieved 17-jun-2017.

[21] João Leitão, José Pereira, and Luís E. T. Rodrigues. Epidemic Broadcast Trees. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, October 10-12, 2007*, pages 301–310, 2007.

[22] João Leitão, José Pereira, and Luís E. T. Rodrigues. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 419–429, 2007.

[23] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 467–483, 2016.

[24] Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 184–195, 2015.

[25] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[26] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 386–400, 2011.

[27] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and Commutative Replicated Data Types. *Bulletin of the EATCS*, 104:67–88, 2011.

[28] Renzo Sprugnoli. Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. *Commun. ACM*, 20(11):841–850, 1977.

[29] Albert van der Linde, João Leitão, and Nuno M. Preguiça. Δ-CRDTs: making Δ-CRDTs delta-based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 12:1–12:4, 2016.

A

TOPOLOGIES

Let $\mathbb{I}$ be the set of node identifiers and $i, j, k \in \mathbb{I}$. Let $n_i \in \mathcal{P}(\mathbb{I})$ be the set of neighbors $\forall i \in \mathbb{I}$. Let $\mathbb{T} \subseteq \mathbb{I} \times \mathbb{I}$ be a binary relation used to represent an overlay network topology. If i is connected to j then $(i, j) \in \mathbb{T}$ (**notation:** $i \rightarrow j$). If i is not connected to j then $(i, j) \notin \mathbb{T}$ (**notation:** $i \nrightarrow j$).

$\mathbb{T}$ is assumed to be:

- **irreflexive** : $\forall i \in \mathbb{I} \cdot i \nrightarrow i$
- **symmetric** : $\forall i, j \in \mathbb{I} \cdot i \rightarrow j \Rightarrow j \rightarrow i$

To define a topology where i, j and k are connected defining a *line*, it's enough to say $i \rightarrow j \wedge j \rightarrow k$, or simplify by saying $i \rightarrow j \rightarrow k$.