A Formal Analysis of the Global Sequence Protocol *

Hernán Melgratti[1,2] and Christian Roldán[1]

[1] Departamento de Computación, FCEyN, Universidad de Buenos Aires.
[2] CONICET.

Abstract. The Global Sequence Protocol (GSP) is an operational model for replicated data stores, in which updates propagate asynchronously. We introduce the GSP-calculus as a formal model for GSP. We give a formal account for its proposed implementation, which addresses communication failures and compact representation of data, and use simulation to prove that the implementation is correct. Then, we use the GSP-calculus to reason about execution histories and prove ordering guarantees, such as read my writes, monotonic reads, causality and consistent prefix. We also prove that GSP extended with synchronous updates provides strong consistency guarantees.

1 Introduction

Cloud infrastructures provide data storages that are virtually unlimited, elastic (i.e., scalable at run-time), highly available and partition tolerant. This is achieved by replicating data over multiple servers. A client may perform update and read operations over any of these replicas and the store is responsible for keeping them synchronised. However, it is known (CAP theorem [7]) that any system cannot simultaneously provide availability, partition tolerance, and consistency. Thus, one of these properties has to be discarded. Today's popular data storages, such as Dynamo [6] and Cassandra [9], ensure availability and offer weaker notions of consistency, called *eventual consistency*. Roughly, eventual consistency guarantees that all updates will be delivered to the different replicas, which will eventually converge to the same state [1]. The storages adopt different strategies to achieve eventual consistency, which impact on the guarantees provided by the system, i.e., on the kind of inconsistencies or anomalies that are allowed to happen. For instance, a storage may resolve automatically conflicts introduced be concurrent updates (e.g., by using timestamps or causality) or may leave the problem to applications that read the database (like in Cassandra). In this way, the consistency model supported by a data store becomes crucial when writing applications.

Consequently, there has been a growing interest on establishing programming abstractions to help developers to deal with eventual consistent stores. For

* Research partially supported by UBACyT project 2014-2017 20020130200092BA.

instance, commutative replicated data types [10] and cloud types [3] provide programmers with suitable data type abstractions that encapsulate issues related to eventual consistency. Recent proposals advocate declarative approaches for programming with eventual consistency, e.g., to automatically select the consistency level required from a store provided with a consistency contract for the application [11] or to prove that a given consistency level is adequate for preserving some data invariant [8]. With similar aims, the Global Sequence Protocol (GSP) [5] proposes an operational model to reason about applications running on top of replicated stores. Basically, the state of a store is represented as the sequence of updates that have led to it. Clients have their own copy of the state which they operate upon: each read and write operation has immediate effect over the local state and the system propagates changes to make all replicas consistent using a reliable total order broadcast protocol (RTOB). The RTOB protocol guarantees that all messages are delivered in the same total order to all clients. Replicas rely on the order generated by RTOB to converge to the same state. In the very basic model, called core GSP, each client interacts with its local state by performing read and write operations. Albeit simple, this model introduces some subtleties when programming because it does not ensure read stability (i.e., two successive reads may return different values) nor atomicity of several updates (i.e., another client may partially observe the effects of a sequence of updates). To overcome these limitations, three synchronisation primitives, namely `pull`, `push` and `confirmed`, allow programmers to control the propagation of changes. It has been shown that this model can be implemented so to handle communication failures and to represent updates efficiently by using two type of objects: states and deltas. Both models, i.e. the idealised one and its implementation, have been defined in terms of a reference implementation.

In this paper, we propose a formal account for each model: the GSP and IGSP calculi (§ 2 and § 3). We prove that the behaviour of a program running over IGSP can be observed over the idealised model. Technically, we show that each IGSP system can be simulated by the corresponding GSP system (§ 4). Then, we study and prove the consistency guarantees ensured by GSP. We rely on the characterisation of consistency guarantees in terms of abstract histories proposed in [2]. Abstract histories capture the visibility relation between actions and the arbitration order of updates in the system. Then, a wide-spectrum of consistency models can be characterised in terms of these two relations. In § 5, we show how to operationally associate abstract histories to concrete computations and prove that GSP enjoys properties such as *Monotonic Read*, *Causal Visibility* and *Consistent prefix*, among others. Finally, in § 6 we study the extension of GSP with synchronous write operations, which ensures strong consistency.

2 Global Sequence Protocol Calculus

2.1 Syntax

Clients interact with a store by performing operations in $\mathcal{U} \cup \mathcal{R}$: an element in $\mathcal{U}$ denotes an update operation, while one in $\mathcal{R}$ stands for a read operation.

$$
\begin{array}{llll}
\text{(NATURALS)} & j,k,n \in \mathbb{N} & & \\
\text{(UPDATE)} & \mathcal{U} = \{u, u', \ldots, u_0, \ldots\} & \text{(UPD SEQ)} & \mathsf{u} ::= \epsilon \mid u^{\mathbb{v}} \cdot \mathsf{u} \\
\text{(READ)} & \mathcal{R} = \{r, r', \ldots, r_0, \ldots\} & \text{(BLOCK SEQ)} & \mathsf{b} ::= \epsilon \mid (\!|\mathsf{u_T}|\!) \cdot \mathsf{b} \\
\text{(EVENT)} & \mathbb{V} = \{\mathtt{v}, \mathtt{v}', \ldots, \mathtt{v}_0, \ldots\} & \text{(SYSTEM)} & N ::= S \parallel C \\
\text{(VAR)} & \mathcal{X} = \{x, x', \ldots, x_0, \ldots\} & \text{(STORE)} & S ::= \mathsf{b} \\
\text{(IDS)} & \mathcal{I} = \{\mathsf{i}, \mathsf{j}, \mathsf{i}', \ldots, \mathsf{i}_0, \ldots\} & \text{(CLIENT)} & C ::= 0 \mid \langle P, \mathsf{u_T}, \mathsf{b}, \mathsf{b}, k, j \rangle_{\mathsf{i}} \mid C \parallel C \\
\text{(PROGRAM)} & P ::= \mathtt{update}(u); P \mid \mathtt{let}\ x = \mathtt{read}(r)\ \mathtt{in}\ P \mid \mathtt{pull}; P \mid & & \\
 & \quad \mathtt{push}; P \mid \mathtt{let}\ x = \mathtt{confirmed}\ \mathtt{in}\ P & &
\end{array}
$$

Fig. 1. Syntax of the GSP calculus

No operation can simultaneously read and update a store, therefore we assume $\mathcal{U} \cap \mathcal{R} = \emptyset$. We write $u, u', u'', \ldots$ for updates and $r, r', r'', \ldots$ for reads.

The state of a store is represented by a sequence of updates. For technical convenience (particularly in § 5), we distinguish different executions of the same operation. Formally, stores associate each update with a fresh event identifier. We assume a set $\mathbb{V}$ of event identifiers $\mathtt{v}$, $\mathtt{v}_0$, ..., $\mathtt{v}'$,... and write $u^{\mathtt{v}}$ for the update u associated with the event $\mathtt{v}$.

We use u to denote sequences of decorated updates and $(\!|\mathsf{u}|\!)$ for an atomic block of updates. We write b for a sequence of blocks. We denote the empty sequence with ϵ and use the usual operations on sequences such as $\mathsf{b}[i]$ to denote the i-th element of b, $\mathsf{b}[i..j]$ for the subsequence of b from position i to j, $|\mathsf{b}|$ for its length and $\mathsf{b} \setminus \mathsf{b}'$ for the relative complement of b in b'. Additionally, $\underline{\mathsf{b}}$ stands for the plain sequence of updates in b (i.e., without any separation in blocks).

We rely on the countable sets $\mathcal{X}$ of program variables $x, x', \ldots$ and $\mathcal{I}$ of client identifiers $\mathsf{i}, \mathsf{i}', \ldots, \mathsf{i}_1, \ldots$.

Definition 2.1 (GSP Language). *The set of* GSP *terms is given by the grammar in Fig. 1.*

A GSP system N consists of a store and zero or more clients. The global store S is completely defined by its state, which consists of a sequence of blocks. The term $\langle P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_{\mathsf{i}}$ stands for a client identified by i and engaged on the execution of the program P. The remaining elements are used to describe the state of the local replica: $\mathsf{u_T}$ contains the updates that have been made locally and are part of an unfinished block; $\mathsf{b_S}$ models the communication buffer, which keeps all blocks completed by the client but not received by the global store; $\mathsf{b_P}$ is the pending buffer, which contains all completed blocks that are unconfirmed by the global store. For simplicity, we do not have an explicit replica of the global store in each client; we use instead a natural number k to indicate the portion of the global state that is known to the client. Specifically, the client i knows the sequence $S[0..k-1]$. Similarly, j indicates the number of updates received by the client that have not been added to the local replica, i.e., the client has received the updates contained in the segment $S[k..k+j-1]$.

A program P is built as a sequence of operations that interacts with the store: $\mathtt{read}(r), \mathtt{update}(u), \mathtt{pull}, \mathtt{push}, \mathtt{confirmed}$ (we postpone their description

until § 2.2). A program $\texttt{let}\ x = \ldots\ \texttt{in}\ P$ introduces a bound variable whose scope is P. The definition of free variables of a program is standard. We say that a process P is *closed* when it does not contain free variables. We keep the language for programs simple. We remark that this choice does not affect the results presented in this paper. Actually, we could just have characterised the behaviour of programs as a labelled transition system, but we prefer to have a syntax throughout the presentation.

Definition 2.2 (Well-formedness). *A* GSP *system* $N = C_0 \| \ldots \| C_m \| S$ *where* $C_l = \langle P_l, \mathsf{u}_{\mathsf{T}l}, \mathsf{b}_{\mathsf{S}l}, \mathsf{b}_{\mathsf{P}l}, k_l, j_l \rangle_{\mathsf{i}_l}$ *for all* $l \in \{0, \ldots, m\}$ *is* well-formed *if the following conditions hold*

1. $\mathsf{i}_l \neq \mathsf{i}_{l'}$ *for all* $l \neq l'$;
2. $k_l + j_l \leq |S|$ *for all* l;
3. $\mathsf{b}_{\mathsf{P}l} = (\!|\mathsf{u}_1|\!) \cdots (\!|\mathsf{u}_\mathsf{p}|\!) \cdot \mathsf{b}_{\mathsf{S}l}$ *and for all* $1 \leq x < y \leq \mathsf{p}$ *there exists* x', y' *s.t.* $S[x'] = (\!|\mathsf{u}_x|\!)$, $S[y'] = (\!|\mathsf{u}_y|\!)$ *and* $k_l \leq x' < y'$; *and*
4. $\mathsf{u} = \underline{S \cdot \mathsf{b}_{\mathsf{S}0} \cdots \mathsf{b}_{\mathsf{S}m} \cdot \mathsf{u}_{\mathsf{T}0} \cdots \mathsf{u}_{\mathsf{T}m}}$, *if* $\mathsf{u}[x] = u^{\mathtt{v}}$, $\mathsf{u}[y] = u^{\mathtt{v}'}$ *and* $x \neq y$ *then* $\mathtt{v} \neq \mathtt{v}'$.

We require identifiers to univocally identify clients (1) and every local state to be consistent with the global store, i.e., a client can see at most every message in the store (2), all unconfirmed blocks in $\mathsf{b}_{\mathsf{P}l}$ are either in the communication buffer $\mathsf{b}_{\mathsf{S}l}$ or in the unseen part of the global store $(\!|\mathsf{u}_1|\!) \cdots (\!|\mathsf{u}_\mathsf{p}|\!)$ (3). Moreover, an event identifier is associated with a unique update in the system (4). Hereafter, we assume every GSP system to be well-formed.

2.2 Operational Semantics

The operational semantics of GSP is given by a labelled transition system over well-formed terms, quotiented by the structural equivalence $\equiv$ defined as the least equivalence such that $\|$ is associative, commutative and has 0 as neutral element. The set of actions is given by the following grammar:

$$\lambda \ ::= \ \tau \mid rd(r) \mid wr(u^{\mathtt{v}}) \mid pull \mid push \mid cfm$$

As usual, τ stands for an internal, unobservable action, while the remaining ones correspond to the interaction of a client with the store. A label (λ, i) indicates that the client i performs the action λ. We write $\xrightarrow{\lambda}_{\mathsf{i}}$ instead of $\xrightarrow{(\lambda,\mathsf{i})}$.

We now comment on the inference rules in Fig. 2. When a client performs an update (rule UPDATE), the change has only local effects: the sequence of local updates u_T is extended with the operation u decorated with a globally fresh identifier $\mathtt{v}$. We remark that decorations are used for technical reasons but they are operationally irrelevant (see § 5).

A client propagates its local changes to the global store by executing `push` (rule PUSH): all local changes in u_T will be transmitted as a block $(\!|\mathsf{u}_\mathsf{T}|\!)$, i.e., as an atomic unit. Nevertheless, these changes are not made available immediately

(UPDATE)
$$\frac{\mathbb{v} \text{ fresh}}{\langle \texttt{update}(u); P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N \xrightarrow{wr(u^{\mathbb{v}})}_\mathsf{i} \langle P, \mathsf{u_T} \cdot u^{\mathbb{v}}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N}$$

(PUSH)
$$\langle \texttt{push}; P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N \xrightarrow{push}_\mathsf{i} \langle P, \epsilon, \mathsf{b_S} \cdot (\!|\mathsf{u_T}|\!), \mathsf{b_P} \cdot (\!|\mathsf{u_T}|\!), k, j \rangle_\mathsf{i} \parallel N$$

(SEND)
$$\langle P, \mathsf{u_T}, (\!|\mathsf{u'_T}|\!) \cdot \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel C \parallel S \xrightarrow{\tau}_\mathsf{i} \langle P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel C \parallel S \cdot (\!|\mathsf{u'_T}|\!)$$

(RECEIVE)
$$\frac{k + j < |S|}{\langle P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel C \parallel S \xrightarrow{\tau}_\mathsf{i} \langle P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j+1 \rangle_\mathsf{i} \parallel C \parallel S}$$

(PULL)
$$\langle \texttt{pull}; P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel C \| S \xrightarrow{pull}_\mathsf{i} \langle P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P} \setminus S[k..k+j-1], k+j, 0 \rangle_\mathsf{i} \parallel C \| S$$

(READ)
$$\frac{rvalue(r, \underline{S[0..k-1] \cdot \mathsf{b_P} \cdot \mathsf{u_T}}) = v}{\langle \texttt{let } x = \texttt{read}(r) \texttt{ in } P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel C \| S \xrightarrow{rd(r)}_\mathsf{i} \langle P\{v/x\}, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel C \| S}$$

(CONFIRM)
$$\frac{v = (\mathsf{b_P} \cdot \mathsf{u_T} == \epsilon)}{\langle \texttt{let } x = \texttt{confirmed in } P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N \xrightarrow{cfm}_\mathsf{i} \langle P\{v/x\}, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N}$$

Fig. 2. Operational semantics for GSP

at the global store because of the asynchronous communication model. In fact, the new block $(\!|\mathsf{u_T}|\!)$ is added to the communication buffer $\mathsf{b_S}$, which contains all blocks that have not reached the global store. Also, $(\!|\mathsf{u_T}|\!)$ is added to the pending messages $\mathsf{b_P}$. Rule SEND stands for a block that finally reaches the global store. Conversely, rule RECEIVE models the reception of a new update. The received update is not immediately added to the local replica. Actually, each client explicitly refreshes its local view by executing `pull` (rule PULL). At this time, all previously received updates j are incorporated to the local copy (i.e., k is changed to $k + j$). Additionally, all pending updates in the new fragment $S[k..k + j - 1]$ are remove from $\mathsf{b_P}$.

The semantics of operations is defined abstractly by the interpretation function $rvalue : \mathcal{R} \times \mathcal{U}^* \to \mathcal{V}$, i.e., a function that takes a read operation and a sequence of updates and returns a value in some domain $\mathcal{V}$. A read operation r is evaluated over the local state of the client (rule READ), i.e., the known prefix of the global store $S[0..k-1]$ and the local updates in $\mathsf{b_P}$ and $\mathsf{u_T}$. The value v is bound to the variable x, and hence all free occurrences of x in the continuation P are substituted by v. A client may perform `confirmed` to check whether its executed updates are already in the global store: this operation returns true only when the local buffers $\mathsf{b_P}$ and $\mathsf{u_T}$ are both empty (rule CONFIRM).

We remark that the operational semantics preserves well-formedness.

Lemma 2.1. *If N is well-formed and $N \xrightarrow{\lambda}_\mathsf{i} N'$, then N' is well-formed.*

(STATE)	$\mathtt{s}, \mathtt{s}', \ldots, \mathtt{s}_1, \ldots \in \mathit{State}$	(SGMT SEQ)	$\mathtt{seg} ::= \epsilon \mid \mathit{seg} \cdot \mathtt{seg}$
(DELTA)	$\delta, \delta', \ldots, \delta_1, \ldots \in \mathit{Delta}$	(IN SRV)	$\mathtt{in_s} \in \mathcal{I} \to \mathtt{r}$
(ROUND)	$r ::= \langle \mathsf{i}, n, \delta \rangle$	(OUT SRV)	$\mathtt{out_s} \in \mathcal{I} \to \mathtt{seg}$
(RND SEQ)	$\mathtt{r} ::= \epsilon \mid r \cdot \mathtt{r}$	(SYSTEM)	$\mathtt{N} ::= \mathtt{S} \parallel \mathtt{C}$
(MAX RND)	$\mathtt{f}, \mathtt{f}', \ldots, \mathtt{f}_1, \ldots \in \mathcal{I} \to \mathbb{N}$	(SERVER)	$\mathtt{S} ::= \langle \mathtt{s}, \mathtt{f}, \mathtt{in_s}, \mathtt{out_s} \rangle$
(SEGMENT)	$\mathit{seg} ::= \langle \delta, \mathtt{f} \rangle \mid \langle \mathtt{s}, \mathtt{f} \rangle$	(CLIENTS)	$\mathtt{C} ::= 0 \mid \langle P, \mathtt{s}, \delta, \delta, n, \mathtt{r}, \mathtt{seg} \rangle_{\mathsf{i}} \mid \mathtt{C} \| \mathtt{C}$

Fig. 3. Syntax of the IGSP calculus

3 Implementation of GSP

The GSP model describes an idealised system that abstracts away from several implementation details, such as non-optimised representation of the state and unreliable communication. This section presents a formal model for the implementation proposed in [5].

3.1 Syntax

The implementation of GSP relies on a compact representation for states and updates. Their precise definition highly depends on the datatype of the values handled by the store, but they are characterised in terms of two abstract types: *State* and *Delta*, which provides the following operations [5]:

$$\begin{array}{ll} \delta_\emptyset : \mathit{Delta} & \emptyset : \mathit{State} \\ \mathit{append} : \mathit{Delta} \times \mathcal{U} \to \mathit{Delta} & \mathit{apply} : \mathit{State} \times \mathit{Delta}^* \to \mathit{State} \\ \mathit{reduce} : \mathit{Delta}^* \to \mathit{Delta} & \mathit{read} : \mathcal{R} \times \mathit{State} \to \mathcal{V} \end{array}$$

Constants $\delta_\emptyset$ and $\emptyset$ denote the empty elements in their respective types. An object $\delta \in \mathit{Delta}$ describes the effects of a sequence of updates and is built by either appending an update to an existing delta (*append*) or combining together several deltas (*reduce*). Operation *read* is the interpretation function for operations (i.e., the implementation counterpart of function $\mathit{rvalue}(\_, \_)$ used by the idealised model) and *apply* corresponds to state transformations.

Clients and the global store exchange δ objects to communicate changes. As each single δ may correspond to several update operations, clients send each δ accompanied by its own identifier and a sequence number n. Precisely, clients send rounds, i.e. triples $r = \langle \mathsf{i}, n, \delta \rangle$. Differently, the global store sends segments $\mathit{seg} = \langle \delta, \mathtt{f} \rangle$, in which δ is accompanied by a function $\mathtt{f} \in \mathcal{I} \to \mathbb{N}$. In this way, the global store confirms all changes from client i until round $\mathtt{f}(\mathsf{i})$. To deal with crashes and recovery, the server may send segments of the form $\langle \mathtt{s}, \mathtt{f} \rangle$, which communicates a complete state instead of a delta object.

Definition 3.1 (GSP Language). *The set of* IGSP *terms is given by the grammar in Fig. 3.*

As for GSP, a system is composed by a global store $\mathtt{S}$ and possibly many clients $\mathtt{C}$. A global store is modelled by a tuple $\langle \mathtt{s}, \mathtt{f}, \mathtt{in_s}, \mathtt{out_s} \rangle$ containing a

state $\mathtt{s}$, a function $\mathtt{f}$ to keep track of processed rounds and the communication buffers $\mathtt{in_s}$ and $\mathtt{out_s}$. There are two dedicated buffers for each client i: $\mathtt{in_s}(\mathsf{i})$ contains the rounds received from i, and $\mathtt{out_s}(\mathsf{i})$ the segments that have been sent to i.

A client is represented by a term $\langle P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i}$. As for GSP, i is its identity and P is its program. Note that the language for programs remains unaltered. The component δ_T is analogous to $\mathtt{u_T}$ in the GSP model, i.e., it keeps all local updates until the client performs `push`. Differently, δ_P keeps all finished blocks that have not been sent. The number n identifies the current round. Buffer $\mathtt{r}$ keeps all sent rounds that have not been confirmed by the global store (similar to $\mathtt{b_P}$ in GSP), while $\mathtt{in_c}$ keeps all received segments (analogous to j).

We also impose the following well-formedness condition on systems.

Definition 3.2 (IGSP well-formedness). *A* IGSP *system* $\mathtt{N} = \mathtt{C}_0 \| \ldots \| \mathtt{C}_m \| \mathtt{S}$ *with* $\mathtt{S} = \langle \mathtt{s}, \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}\rangle$ *and* $\mathtt{C}_l = \langle P_l, \mathtt{s}_l, \delta_{\mathsf{T}l}, \delta_{\mathsf{P}l}, n_l, \mathtt{r}_l, \mathtt{in_c}l\rangle_{\mathsf{i}_l}$ *for* $l \in \{0, \ldots, m\}$ *is* well-formed *if the following conditions hold*

1. $\mathsf{i}_l \neq \mathsf{i}_{l'}$ *for all* $l \neq l'$.
2. $dom(\mathtt{in_s}) = dom(\mathtt{out_s}) \subseteq \{0, \ldots, m\}$.
3. $\mathsf{i}_l \notin dom(\mathtt{out_s})$ *implies* $\mathtt{in_c}l = \epsilon$.
4. *if* $\mathsf{i}_l \in dom(\mathtt{out_s})$ *and* $\mathtt{in_c}l \cdot \mathtt{out_s}(\mathsf{i}_l) \neq \epsilon$ *then either*
   (*i*) $\mathtt{in_c}l \cdot \mathtt{out_s}(\mathsf{i}_l) = \langle \delta_0, \mathtt{f}_0\rangle \cdots \langle \delta_h, \mathtt{f}_h\rangle$; *or*
   (*ii*) $\mathtt{in_c}l \cdot \mathtt{out_s}(\mathsf{i}_l) = \langle \mathtt{s}, \mathtt{f}_0\rangle \cdot \langle \delta_1, \mathtt{f}_1\rangle \cdots \langle \delta_h, \mathtt{f}_h\rangle$

   and $\mathtt{f}_j(\mathsf{i}_l) \leq \mathtt{f}_k(\mathsf{i}_l)$ *for all* $j < k \in \{0..h\}$ *and* $\mathtt{f}_h = \mathtt{f}$.
5. $\mathtt{r}_l = \langle \mathsf{i}_0, n_0, \delta_0\rangle \cdots \langle \mathsf{i}_r, n_r, \delta_r\rangle$, $n_j < n_k$ *for all* $j < k \in \{0..r\}$ *and either*
   (*i*) $\delta_{\mathsf{P}l} = \delta_\emptyset$, $n_r \leq n_l$ *and* $\mathtt{f}(\mathsf{i}_l) \leq n_l$; *or*
   (*ii*) $\delta_{\mathsf{P}l} \neq \delta_\emptyset$, $n_r < n_l$ *and* $\mathtt{f}(\mathsf{i}_l) < n_l$.
6. *either*
   (*i*) $\mathtt{r}_l = \epsilon$, $\mathtt{f}(\mathsf{i}_l) \leq n_l$ *and if* $\mathsf{i}_l \in dom(\mathtt{in_s})$ *then* $\mathtt{in_s}(\mathsf{i}_l) = \epsilon$;
   (*ii*) $\mathtt{r}_l = \mathtt{in_s}(\mathsf{i}_l) = \langle \mathsf{i}_l, n_{fst}, \delta_{fst}\rangle \cdot \mathtt{r}'_l$ *and* $n_{fst} > \mathtt{f}(\mathsf{i}_l)$;
   (*iii*) $\mathtt{r}_l = \mathtt{r}''_l \cdot \langle \mathsf{i}_l, n_{lst}, \delta_{lst}\rangle \cdot \mathtt{in_s}(\mathsf{i}_l)$ *and* $\mathtt{in_c}l \cdot \mathtt{out_s}(\mathsf{i}_l) = \langle \delta_0, \mathtt{f}_0\rangle \cdots \langle \delta'_{lst}, \mathtt{f}_{lst}\rangle$ *with* $\mathtt{f}_{lst}(\mathsf{i}_l) = n_{lst}$; *or*
   (*iv*) $\mathtt{in_s}(\mathsf{i}_l) = \epsilon$, $\mathtt{r}_l = \mathtt{r}''_l \cdot \langle \mathsf{i}_l, n_{lst}, \delta_{lst}\rangle$ *and either* $\mathtt{in_c}l \cdot \mathtt{out_s}(\mathsf{i}_l) = \langle \mathtt{s}, \mathtt{f}'\rangle \cdot \mathtt{seg}$ *with* $\mathtt{f}'(\mathsf{i}_l) \leq n_{lst}$ *or* $\mathtt{in_c}l \cdot \mathtt{out_s}(\mathsf{i}_l) = \epsilon$ *and* $\mathtt{f}(\mathsf{i}_l) \leq n_{lst}$.

We require all clients to have different identifiers (1). Communication channels in the implementation are bidirectional, hence $\mathsf{i}_l \in dom(\mathtt{in_s})$ iff $\mathsf{i}_l \in dom(\mathtt{out_s})$ (2). Moreover, the input buffer of a disconnected client is empty (3). Condition (4) states that $\langle \mathtt{s}, \mathtt{f}'\rangle$ can appear only as the first message in the flow from the store and that the store confirms processed rounds in a non-decreasing order. Similarly, clients send rounds with increasing round number (5). The last condition (6) states a coherence requirement between pending rounds and the segments sent by the store, which can only confirm rounds that are pending.

(I-UPDATE)

$$\langle \mathtt{update}(u); P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N} \xrightarrow{wr(u^{\mathsf{N}})}_\mathsf{i} \langle P, \mathtt{s}, \mathit{append}(\delta_\mathsf{T}, u), \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N}$$

(I-PUSH)

$$\langle \mathtt{push}; P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N} \xrightarrow{push}_\mathsf{i} \langle P, \mathtt{s}, \delta_\emptyset, \mathit{reduce}(\delta_\mathsf{P} \cdot \delta_\mathsf{T}), n+1, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N}$$

(I-SEND)

$$\frac{\delta_\mathsf{P} \neq \delta_\emptyset \qquad \mathsf{i} \in \mathit{dom}(\mathtt{in_s}) \qquad r = \langle \mathsf{i}, n, \delta_\mathsf{P}\rangle \qquad \mathtt{in_c} \cdot \mathtt{out_s}(\mathsf{i}) \neq \langle \mathtt{s}', \mathtt{f}\rangle \cdot \mathtt{seg}}{\begin{array}{l}\langle P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \langle \mathtt{s}', \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}\rangle \parallel \mathtt{C} \xrightarrow{\tau}_\mathsf{i} \\ \qquad \langle P, \mathtt{s}, \delta_\mathsf{T}, \delta_\emptyset, n, \mathtt{r} \cdot r, \mathtt{in_c}\rangle_\mathsf{i} \parallel \langle \mathtt{s}', \mathtt{f}, \mathtt{in_s}[\mathsf{i} \mapsto \mathtt{in_s}(\mathsf{i}) \cdot r], \mathtt{out_s}\rangle \parallel \mathtt{C}\end{array}}$$

(I-RECEIVE)

$$\frac{\mathtt{out_s}(\mathsf{i}) = \mathit{seg} \cdot \mathtt{seg}}{\begin{array}{l}\langle P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \langle \mathtt{s}', \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}\rangle \parallel \mathtt{C} \xrightarrow{\tau}_\mathsf{i} \\ \qquad \langle P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c} \cdot \mathit{seg}\rangle_\mathsf{i} \parallel \langle \mathtt{s}', \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}[\mathsf{i} \mapsto \mathtt{seg}]\rangle \parallel \mathtt{C}\end{array}}$$

(I-PULL$_1$)

$$\frac{\mathtt{r}' = \mathit{filter}(\mathtt{f}_k(\mathsf{i}), \mathtt{r}) \qquad \mathtt{in_c} = \langle \delta_1, \mathtt{f}_1\rangle \ldots \langle \delta_k, \mathtt{f}_k\rangle}{\langle \mathtt{pull}; P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N} \xrightarrow{pull}_\mathsf{i} \langle P, \mathit{apply}(\mathtt{s}, \mathit{reduce}(\delta_1 \cdots \delta_k)), \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}', \epsilon\rangle_\mathsf{i} \parallel \mathtt{N}}$$

(I-READ)

$$\frac{\mathit{read}(r, \mathit{apply}(\mathtt{s}, \Delta(\mathtt{r}) \cdot \delta_\mathsf{P} \cdot \delta_\mathsf{T})) = v}{\langle \mathtt{let}\ x = \mathtt{read}(r)\ \mathtt{in}\ P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N} \xrightarrow{rd(r)}_\mathsf{i} \langle P\{v/x\}, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N}}$$

(I-CONFIRM)

$$\frac{v = (\mathtt{r} \cdot \delta_\mathsf{P} \cdot \delta_\mathsf{T} == \epsilon)}{\langle \mathtt{let}\ x = \mathtt{confirmed}\ \mathtt{in}\ P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N} \xrightarrow{cfm}_\mathsf{i} \langle P\{v/x\}, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \mathtt{N}}$$

(I-BATCH)

$$\frac{\langle \delta, \mathtt{f}'\rangle = \mathit{rnds}(\mathtt{in_s}) \quad \delta \neq \delta_\emptyset \quad \mathtt{s}' = \mathit{apply}(\mathtt{s}, \delta) \quad \forall \mathsf{i}.(\mathtt{out_s}'(\mathsf{i}) = \mathtt{out_s}(\mathsf{i}) \cdot \langle \delta, \mathtt{f}[\mathtt{f}']\rangle \wedge \mathtt{in_s}'(\mathsf{i}) = \epsilon)}{\langle \mathtt{s}, \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}\rangle \parallel \mathtt{C} \xrightarrow{\tau} \langle \mathtt{s}', \mathtt{f}[\mathtt{f}'], \mathtt{in_s}', \mathtt{out_s}'\rangle \parallel \mathtt{C}}$$

(I-DROP-CXN)

$$\frac{\mathsf{i} \in \mathtt{in_s} \qquad \mathsf{i} \in \mathtt{out_s}}{\begin{array}{l}\langle P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \langle \mathtt{s}, \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}\rangle \parallel \mathtt{C} \xrightarrow{\tau} \\ \qquad \langle P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \epsilon\rangle_\mathsf{i} \parallel \langle \mathtt{s}, \mathtt{f}, \mathtt{in_s} \setminus \mathsf{i}, \mathtt{out_s} \setminus \mathsf{i}\rangle \parallel \mathtt{C}\end{array}}$$

(I-ACCEPT-CXN)

$$\frac{\mathsf{i} \notin \mathtt{in_s} \qquad \mathsf{i} \notin \mathtt{out_s}}{\langle \mathtt{s}, \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}\rangle \parallel \mathtt{C_i} \parallel \mathtt{C} \xrightarrow{\tau} \langle \mathtt{s}, \mathtt{f}, \mathtt{in_s}[\mathsf{i} \mapsto \epsilon], \mathtt{out_s}[\mathsf{i} \mapsto \langle \mathtt{s}, \mathtt{f}\rangle]\rangle \parallel \mathtt{C_i} \parallel \mathtt{C}}$$

(I-PULL$_2$)

$$\frac{\mathtt{in_c} = \langle \mathtt{s}''', \mathtt{f}_0\rangle \cdot \langle \delta_1, \mathtt{f}_1\rangle \ldots \langle \delta_k, \mathtt{f}_k\rangle \qquad \mathtt{s}'' = \mathit{apply}(\mathtt{s}''', \mathit{reduce}(\delta_\emptyset \cdot \delta_1 \cdots \delta_k)) \qquad \mathtt{r}' = \mathit{filter}(\mathtt{f}_k(\mathsf{i}), \mathtt{r})}{\begin{array}{l}\langle \mathtt{pull}; P, \mathtt{s}, \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}, \mathtt{in_c}\rangle_\mathsf{i} \parallel \langle \mathtt{s}', \mathtt{f}, \mathtt{in_s}, \mathtt{out_s}\rangle \parallel \mathtt{C} \xrightarrow{pull}_\mathsf{i} \\ \qquad \langle P, \mathtt{s}'', \delta_\mathsf{T}, \delta_\mathsf{P}, n, \mathtt{r}', \epsilon\rangle_\mathsf{i} \parallel \langle \mathtt{s}', \mathtt{f}, \mathtt{in_s}[\mathsf{i} \mapsto \mathtt{in_s}(\mathsf{i}) \cdot \mathtt{r}'], \mathtt{out_s}\rangle \parallel \mathtt{C}\end{array}}$$

Fig. 4. Operational semantics of IGSP

3.2 Operational Semantics

As for the idealised model, the operational semantics is given by a labelled transition system over well-formed terms, up-to structural equivalence. We consider a new label τ without any client annotation for transitions associated with changes in the global store and communication failures. The inference rules are in Fig. 4.

Rule (I-UPDATE), which is analogous to rule (UPDATE), adds the operation u to the temporary block $\delta_{\mathtt{T}}$. The decoration $\mathtt{v}$ is irrelevant in this model, hence we do not impose any freshness requirement. A client terminates a block by executing `push` (I-PUSH). At this time, the block $\delta_{\mathtt{T}}$ is appended to the already terminated blocks in $\delta_{\mathtt{P}}$, which will be sent on the next round. Additionally, the block counter n is incremented by 1. By rule (I-SEND), a client sends changes to the global store. This transition takes place whenever the client is connected (i.e., $\mathsf{i} \in \mathit{dom}(\mathtt{in_s})$), there are finished blocks in $\delta_{\mathtt{P}}$ (i.e., $\delta_{\mathtt{P}} \neq \delta_{\emptyset}$) and there is no need for resynchronisation (i.e., $\mathtt{in_c} \cdot \mathtt{out_s}(\mathsf{i}) \neq \langle \mathtt{s}', \mathtt{f} \rangle \cdot \mathtt{seg}$)[3]. The available blocks are sent within the same round $r = \langle \mathsf{i}, n, \delta_{\mathtt{P}} \rangle$, which contains the number n corresponding to the last finished block. The new round r is added to the corresponding input buffer in the store, i.e., $\mathtt{in_s}(\mathsf{i})$ is updated to $\mathtt{in_s}(\mathsf{i}) \cdot r$ (where $\_[\_ \mapsto \_]$ is the update operator for functions). Additionally, r is added to the sequence of pending rounds $\mathtt{r}$ and the buffer $\delta_{\mathtt{P}}$ is reset to $\delta_{\emptyset}$.

Symmetrically, the client i may receive an available segment at any time (I-RECEIVE). The new segment *seg* is removed from the buffer $\mathtt{out_s}(\mathsf{i})$ of the global store and added to the input buffer of the client. As for the idealised model, all received changes are applied to the local replica when i performs `pull`. Rule (I-PULL$_1$) handles the case in which the connection with the global store has not been previously reset. In such case, all received segments are of the form $\langle \delta, \mathtt{f} \rangle$. Therefore, the changes $\delta_1 \cdots \delta_{\mathtt{k}}$ are applied to the local state $\mathtt{s}$ and all rounds confirmed by the received segments are removed from the pending list $\mathtt{r}$. By well-formedness (Def. 3.2,5), it suffices to consider the confirmation $\mathtt{f}_k$, which has the greatest confirmation. Hence, all rounds up-to $\mathtt{f}_k(\mathsf{i})$ are removed from $\mathtt{r}$. This is done by the auxiliary function *filter*$(\_, \_)$, defined as follows

$$\mathit{filter}(n, \mathtt{r}) = \langle \mathsf{i}, n_j, \delta_j \rangle \cdots \langle \mathsf{i}, n_k, \delta_k \rangle \quad \text{if} \quad \mathtt{r} = \langle \mathsf{i}, n_0, \delta_0 \rangle \cdots \langle \mathsf{i}, n_j, \delta_j \rangle \cdots \langle \mathsf{i}, n_k, \delta_k \rangle, \; n_{j-1} \leq n \text{ and } n_j > n$$

Rules (I-READ) and (I-CONFIRM) are analogous the ones in the GSP calculus. We use $\Delta(\_)$ for the function that projects a sequence of rounds into the sequence that contains the corresponding δs. The global store changes its state as prescribed by rule (I-BATCH): it collects all received rounds in $\mathtt{in_s}$ by using the auxiliary function *rnds*$(\_)$, which builds a unique object δ by appending all available rounds, and a function $\mathtt{f}$ that associates each client with the number of the last received round. Let $\mathtt{in_s}$ be such that $\mathit{dom}(\mathtt{in_s}) = \{\mathsf{i}_0, \ldots, \mathsf{i}_m\}$ and $\forall \mathsf{i}_l \in \mathit{dom}(\mathtt{in_s}).\mathtt{in_s}(\mathsf{i}_l) = \mathtt{r}_l \cdot \langle \mathsf{i}_l, n_l^{k_l}, \delta_l^{k_l} \rangle$. Then, *rnds*$(\_)$ is defined as follows

[3] For simplicity we check re-synchronisation by inspecting buffers instead of explicitly adding the condition *channel established* used in the implementation.

$$rnds(\mathtt{in_s}) = \langle \delta, \mathtt{f}' \rangle \quad \text{with} \quad \begin{array}{r} \delta = reduce(\Delta(\mathtt{in_s}(\mathsf{i}_0)) \cdots \Delta(\mathtt{in_s}(\mathsf{i}_m))), \\ dom(\mathtt{f}') = \{\mathsf{i} \mid \mathsf{i} \in dom(\mathtt{in_s}) \text{ and } \mathtt{in_s}(\mathsf{i}) \neq \epsilon\} \text{ and} \\ \forall \mathsf{i}_l \in dom(\mathtt{in_s}).\mathtt{f}'(\mathsf{i}_l) = n_l^{k_l} \end{array}$$

The obtained δ is applied to the current state $\mathtt{s}$ and $\mathtt{f}$ is updated with $\mathtt{f}'$. In addition, the new segment $\langle \delta, \mathtt{f}[\mathtt{f}'] \rangle$ is sent to every connected client, i.e., it is added at the end of every buffer $\mathtt{out_s}(\mathsf{i})$. The input buffers $\mathtt{in_s}(\mathsf{i}_l)$ are emptied because all received rounds have been processed.

The remaining rules deal with connectivity issues: rule (I-DROP-CXN) models a disconnection: the buffers $\mathtt{out_s}(\mathsf{i})$ and $\mathtt{in_s}(\mathsf{i})$ are removed from the global store and also the input buffer of i is set to ϵ. When the client i (re-)establishes its connection (I-ACCEPT-CXN), the store creates the buffers for i and sends a segment containing the current state of the store. Rule (I-PULL_2) is analogous to (I-PULL_1), but handles the first `pull` after a reconnection. The first received segment $\langle \mathtt{s}''', \mathtt{f}_0 \rangle$ contains a state instead of a delta object. The client uses $\mathtt{s}'''$ instead of its local state to resynchronise. The application of successive segments is analogous to rule (I-PULL_1). Moreover, the client resends a round $\mathtt{r}'$ containing all pending segments lost by the server during the disconnection.

The proposed implementation allows for a server to crash, i.e., to close all communication buffers, but we do not model explicitly this behaviour because it can be obtained by applying rule (I-DROP-CXN) several times.

Lemma 3.1. *Let* $\mathtt{N}$ *be a well-formed* IGSP *system. If* $\mathtt{N} \xrightarrow{\lambda}_{\mathsf{i}} \mathtt{N}'$*, then* $\mathtt{N}'$ *is well-formed.*

4 Correctness of the Implementation

We now prove that IGSP is a correct implementation of GSP. We recall in Fig. 5 the requirements stated in [5] for the operations provided by the data types State and Delta. Formally, the relation $\_ \lhd \_$ associates delta and state objects with sequences of updates: $\delta \lhd \mathtt{u}$ (similarly, $\mathtt{s} \lhd \mathtt{u}$) means that δ (correspondingly, $\mathtt{s}$) is a compact representation of $\mathtt{u}$. Then, it is also assumed that $\mathtt{s} \lhd \mathtt{u}$ implies $read(r, \mathtt{s}) = rvalue(r, \mathtt{u})$ for any r. Building on the above relation, we define under which conditions a IGSP system is an implementation of a GSP system.

Definition 4.1. *Let* $N = C_0 \parallel \ldots \parallel C_m \parallel S$ *be a* GSP *system such that* $C_l = \langle P_l, \mathtt{u_{T}}_l, \mathtt{b_{S}}_l, \mathtt{b_{P}}_l, k_l, j_l \rangle_{\mathsf{i}_l}$ *for all* $l \in \{0, \ldots, m\}$*, and* $\mathtt{N} = \mathtt{C}_0 \parallel \ldots \parallel \mathtt{C}_m \parallel \mathtt{S}$ *a* IGSP *system such that* $\mathtt{S} = \langle \mathtt{s}, \mathtt{f_s}, \mathtt{in_s}, \mathtt{out_s} \rangle$ *and* $\mathtt{C}_l = \langle P_l, \mathtt{s}_l, \delta_{\mathtt{T}l}, \delta_{\mathtt{P}l}, n_l, \mathtt{r}_l, \mathtt{in_c}_l \rangle_{\mathsf{i}_l}$*. We say* $\mathtt{N}$ implements N *if the following conditions hold:*

1. $\mathtt{s} \lhd \underline{S}$;
2. $\mathtt{s}_l \lhd \underline{S[0..k_l - 1]}$;
3. $\delta_{\mathtt{T}l} \lhd \mathtt{u_{T}}_l$;
4. $reduce(\Delta(\mathtt{r}_l) \cdot \delta_{\mathtt{P}l}) \lhd \underline{\mathtt{b_{P}}_l}$;

$$(\lhd\text{-}\delta_\emptyset)\quad \delta_\emptyset \lhd \epsilon \qquad (\lhd\text{-}\emptyset)\quad \emptyset \lhd \epsilon$$

$$(\lhd\text{-}append)\ \frac{\delta \lhd \mathtt{u}}{append(\delta, u) \lhd \mathtt{u} \cdot \breve{u}} \qquad (\lhd\text{-}read)\ \frac{\mathtt{s} \lhd \mathtt{u}}{read(r, \mathtt{s}) = rvalue(r, \mathtt{u})}$$

$$(\lhd\text{-}apply)\ \frac{\mathtt{s} \lhd \mathtt{u} \quad \delta_1 \lhd \mathtt{u}_1 \ldots \delta_\mathtt{n} \lhd \mathtt{u}_\mathtt{n}}{apply(\mathtt{s}, \delta_1 \cdots \delta_\mathtt{n}) \lhd \mathtt{u} \cdot \mathtt{u}_1 \cdots \mathtt{u}_\mathtt{n}} \qquad (\lhd\text{-}reduce)\ \frac{\delta_1 \lhd \mathtt{u}_1 \ldots \delta_\mathtt{n} \lhd \mathtt{u}_\mathtt{n}}{reduce(\delta_1 \cdots \delta_\mathtt{n}) \lhd \mathtt{u}_1 \cdots \mathtt{u}_\mathtt{n}}$$

Fig. 5. Coherence requirements for *Delta* and *State* operators

5. *if* $\mathtt{i}_l \in dom(\mathtt{in_s})$ *and* $\mathtt{in_c} \cdot \mathtt{out_s}(\mathtt{i}_l) \neq \langle \mathtt{s}', \mathtt{f} \rangle \cdot \mathtt{seg}$ *then* $reduce(\Delta(\mathtt{in_s}(\mathtt{i}_l)) \cdot \delta_{\mathtt{P}l}) \lhd \underline{\mathtt{b}_{\mathtt{S}l}}$;
   $\breve{a}$
6. *if* $\mathtt{i}_l \in dom(\mathtt{out_s})$ *then either*
   i. $\mathtt{in}_{\mathtt{c}l} \cdot \mathtt{out_s}(\mathtt{i}_l) = \epsilon$, $k_l = |S|$;
   ii. $\mathtt{in}_{\mathtt{c}l} = \langle \delta', \mathtt{f} \rangle \cdot \mathtt{seg}$, $reduce(\Delta(\mathtt{in}_{\mathtt{c}l})) \lhd \underline{S[k_l..k_l + j_l - 1]}$, *and* $reduce(\Delta(\mathtt{out_s}(\mathtt{i}_l))) \lhd \underline{S[k_l + j_l..|S| - 1]}$;
   iii. $\mathtt{in}_{\mathtt{c}l} = \langle \mathtt{s}', \mathtt{f}_0 \rangle \cdot \mathtt{seg}$, *there exists* t *s.t.* $k_l \leq t \leq k_l + j_l$ *s.t.* $\mathtt{s}' \lhd \underline{S[0..t-1]}$, $reduce(\Delta(\mathtt{seg})) \lhd \underline{S[t..k_l + j_l - 1]}$ *and* $reduce(\Delta(\mathtt{out_s}(\mathtt{i}_l))) \lhd \underline{S[k_l + j_l..|S| - 1]}$; *or*
   iv. $\mathtt{in}_{\mathtt{c}l} = \epsilon$, $\mathtt{out_s}(\mathtt{i}_l) = \langle \mathtt{s}', \mathtt{f} \rangle \cdot \mathtt{seg}$ *there exists* $t \geq k_l + j_l$ *s.t.* $\mathtt{s}' \lhd \underline{S[0..t-1]}$ $reduce(\Delta(\mathtt{seg})) \lhd \underline{S[t..|S| - 1]}$;
7. *for all* $\mathtt{f}$ *s.t* $\mathtt{f} = \mathtt{f_s}$ *or* $\langle \_, \mathtt{f} \rangle \in \mathtt{in}_{\mathtt{c}l} \cdot \mathtt{out_s}(\mathtt{i}_l)$, *for all* $\langle \mathtt{i}, n, \delta' \rangle \in \mathtt{r}_l$ *if* $n \leq \mathtt{f}(\mathtt{i}_l)$ *then* $\delta \lhd \underline{S[x..x']}$ *and* $\delta' \lhd \underline{S[y..y']}$ *with* $y' \leq x'$.

The first three conditions are self-explanatory. Condition (4) states that the pending blocks in $\underline{\mathtt{b}_{\mathtt{P}l}}$ correspond either to rounds in the pending list $\mathtt{r}_l$ or to blocks ready to be sent, i.e., in $\delta_{\mathtt{P}l}$. By condition (5), if a client is synchronised with the store (i.e., $\mathtt{i}_l \in dom(\mathtt{in_s})$ and $\mathtt{in_c} \cdot \mathtt{out_s}(\mathtt{i}_l) \neq \langle \mathtt{s}', \mathtt{f} \rangle \cdot \mathtt{seg}$) then all blocks in the sending list $\mathtt{b}_{\mathtt{S}l}$ are either rounds that have been sent, i.e., in $\mathtt{in_s}(\mathtt{i}_l)$, or ready blocks in $\delta_{\mathtt{P}l}$. Condition (6) establishes the relation between the received messages in both models. Basically, the local replica is complete when there are no segments for the client (i). When the first received segment is a delta object (ii), the content in the input buffer $\mathtt{in}_{\mathtt{c}l}$ corresponds to the received messages in $S[k_l..k_l + j_l - 1]$ and the the output buffer $\mathtt{out_s}(\mathtt{i}_l)$ contains the updates in the sequence $S[k_l + j_l..|S| - 1]$. In the remaining two cases, the first segment contains a state. When the segment is in the input buffer of the client (iii), the received state $\mathtt{s}'$ corresponds to a prefix of the sequence S whose length lies in between of the updates already received by the client in the idealised model, i.e., $S[0..t-1]$ with $k \leq t \leq k_l + j_l$, while the remaining conditions are analogous to the previous case. Differently, when the first segment is still on the output buffer of the store (iv), $\mathtt{s}'$ corresponds to a prefix that contains at least all updates in S already known to the client, i.e., $t \geq k_l + j_l$ because the store confirmations are monotonic.

Condition (7) states that in any segment $\langle \delta, \mathtt{f} \rangle$ sent by the store, δ corresponds to a contiguous sequence of updates in S, i.e., $S[x..x']$. Moreover, all confirmed rounds are also within the prefix $S[0..x']$.

We now show that IGSP is a correct implementation of GSP by proving that N weakly simulates $\mathtt{N}$ when $\mathtt{N}$ implements N. We use standard simulation but technically we take into account the fact that GSP associates a fresh event identifier to each update while IGSP does not. Take $\rightarrow = \xrightarrow{\tau} \bigcup_{\mathsf{i} \in \mathcal{I}} \xrightarrow{\tau}_{\mathsf{i}}$, $\Rightarrow$ as the reflexive and transitive closure of $\rightarrow$, i.e.$\Rightarrow = \rightarrow^*$, and $\overset{\lambda}{\Rightarrow}_{\mathsf{i}} = \Rightarrow; \xrightarrow{\lambda}_{\mathsf{i}}; \Rightarrow$.

Definition 4.2 (Simulation). *$\mathcal{R}$ is an* implementation simulation *if for all $(\mathtt{N}, N) \in \mathcal{R}$ we have:*

1. *If $\mathtt{N} \xrightarrow{wr(u^{\mathtt{v}})}_{\mathsf{i}} \mathtt{N}'$ then $\exists N', \mathtt{w}$ s.t. $N \overset{wr(u^{\mathtt{w}})}{\Longrightarrow}_{\mathsf{i}} N'$ and $(\mathtt{N}', N') \in \mathcal{R}$;*
2. *If $\mathtt{N} \xrightarrow{\lambda}_{\mathsf{i}} \mathtt{N}'$ and $\lambda \neq wr(u^{\mathtt{v}}), \tau$ then $\exists N'$ s.t. $N \overset{\lambda}{\Rightarrow}_{\mathsf{i}} N'$ and $(\mathtt{N}', N') \in \mathcal{R}$;*
3. *If $\mathtt{N} \rightarrow \mathtt{N}'$ then $\exists N'$ s.t. $N \Rightarrow N'$ and $(\mathtt{N}', N') \in \mathcal{R}$;*

As usual, we write $\mathtt{N} \precsim N$ if there exists a simulation $\mathcal{R}$ s.t. $(\mathtt{N}, N) \in \mathcal{R}$.

Theorem 4.1. *If $\mathtt{N}$ implements N, then $\mathtt{N} \precsim N$.*

Proof. We show that $\mathcal{R} = \{(\mathtt{N}, N) \mid \mathtt{N} \textit{ implements } N\}$ is a simulation.

We remark that $\mathcal{R}^{-1}$ is not a simulation because the implementation cannot mimic the behaviour in which a client have completed two consecutive blocks (i.e., two `push` commands) without sending the first block. In GSP it is still possible to interleave the two blocks with blocks sent by other clients but in IGSP they are treated as atomic because they will be sent as a unique δ object.

5 Consistency Guarantees

In this section we study the consistency properties offered by GSP. We rely on the characterisation of properties in terms of abstract executions [4], execution histories enriched with information about visibility and arbitration of actions.

Definition 5.1. *Let N be a well-formed* GSP *system, an abstract history for N is a tuple $\mathbb{A} = \langle N, \mathrm{OP}, \mathrm{SS}, \mathrm{SO}, \mathrm{VIS}, \mathrm{AR} \rangle$ where:*

- $\mathrm{OP} : \mathbb{V} \rightarrow \mathcal{R} \cup \mathcal{U}$ *maps events to operations;*
- $\mathrm{SS} : \mathcal{I} \rightarrow \mathbb{V}$ *associates events with a session (i.e., a client);*
- $\mathrm{SO} \subseteq \mathbb{V} \times \mathbb{V}$ *describes the order of operations within a session;*
- $\mathrm{VIS} \subseteq \mathbb{V} \times \mathbb{V}$ *indicates whether the effects of an update are visible to a read;*
- $\mathrm{AR} \subseteq \mathbb{V} \times \mathbb{V}$ *resolves concurrent update conflicts.*

We write $\_\downarrow\_$ for function/relation restriction. For a given abstract history $\mathbb{A}$, we write $\mathbb{U}$ (similarly, $\mathbb{R}$) for the codomain restriction of OP to $\mathcal{U}$ (correspondingly, $\mathcal{R}$), i.e., $\mathbb{U} = \{\mathtt{v} \mid \mathtt{v} \in \mathrm{OP}, \mathrm{OP}(\mathtt{v}) \in \mathcal{U}\}$ ($\mathbb{R} = \{\mathtt{v} \mid \mathtt{v} \in \mathrm{OP}, \mathrm{OP}(\mathtt{v}) \in \mathcal{R}\}$).

Definition 5.2 (Well-formed history). *Let* $N = C_0 \| \ldots \| C_m \| S$ *be a* GSP *system where* $C_l = \langle P_l, \mathsf{u}_{\mathsf{T}l}, \mathsf{b}_{\mathsf{S}l}, \mathsf{b}_{\mathsf{P}l}, k_l, j_l \rangle_{\mathsf{i}_l}$. *A history* $\mathbb{A} = \langle N, \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR} \rangle$ *is well-formed if the following conditions hold:*

1. *for all* $\mathsf{i} \in dom(\text{SS})$, $\text{SS}(\mathsf{i}) \subseteq dom(\text{OP})$;
2. $(\mathtt{v}, \mathtt{w}) \in \text{SO}$ *then exist* $\mathsf{i} \in dom(\text{SS})$ *s.t.* $\{\mathtt{v}, \mathtt{w}\} \in \text{SS}(\mathsf{i})$.
3. *for all* $\mathsf{i} \in dom(\text{SS})$, $\text{SO} \downarrow_{\text{SS(i)}}$ *is a total order;*
4. $\text{VIS} \subseteq \mathbb{U} \times \mathbb{R}$;
5. $\text{AR} \subseteq \mathbb{U} \times \mathbb{U}$ *is a prefix order.*
6. $(\mathtt{v}, \mathtt{w}) \in \text{AR}$ *iff*
   - $\underline{S}[i] = u^{\mathtt{v}}$ *and* $\underline{S}[j] = u^{\mathtt{w}}$ *and* $i < j$; *or*
   - $u^{\mathtt{v}} \in \underline{S}$ *and* $u^{\mathtt{w}} \in \underline{\mathsf{b}_{\mathsf{S}l} \cdot \mathsf{u}_{\mathsf{T}l}}$;
7. *if* $\underline{\mathsf{b}_{\mathsf{S}l} \cdot \mathsf{u}_{\mathsf{T}l}}[i] = u^{\mathtt{v}}$ *and* $\underline{\mathsf{b}_{\mathsf{S}l} \cdot \mathsf{u}_{\mathsf{T}l}}[j] = u^{\mathtt{w}}$ *and* $i < j$, *then* $(\mathtt{v}, \mathtt{w}) \in \text{SO}$ *and* $\{\mathtt{v}, \mathtt{w}\} \in \text{SS}(\mathsf{i}_l)$.

The above conditions ensure that events in SS are associated with an operation by OP (1). Besides, SO only relates events belonging to the same session (2), which are totally ordered within each session (3). Differently from the definition in [2], we restrict visibility to keep track of dependencies between updates and read events(4). We do not require AR to be a total order but instead to be a prefix order (5). In this way the updates in different replicas are arbitrated when they reach the global store. The remaining two conditions require the abstract history to be consistent with the state of the system.

Rules in Fig. 6 provides an operational way to associate abstract executions with GSP computations. Rules (A-UPDATE) and (A-READ) add new events to the history and corresponds to the execution of a read or update operation by a client. In both cases OP is extended with a new event $\mathtt{v}$ (i.e., $\mathtt{v} \notin dom(\text{OP})$), which is associated with the corresponding operation (either r or u). The new event $\mathtt{v}$ is added to the corresponding session i, and SO is updated to make $\mathtt{v}$ the maximal event for the session i. Rule (A-UPDATE) amends AR by capturing the fact that all updates that are already in the global state took place before the new event. Rule (A-READ) instead augments VIS with the pairs associating the new event with all events that are seen by the read action, namely, the local view of the global state $S[0..k_{\mathsf{i}} - 1]$ and the local buffers $\mathsf{b}_{\mathsf{P}\mathsf{i}}$ and $\mathsf{u}_{\mathsf{T}\mathsf{i}}$. Rule (A-ARB) handles the changes in the state of the global store (due to a send transition in one client) and amends AR by arbitrating (i) the new events by respecting the relative order in which they are added to the store (i.e., $\{(\mathtt{v}_i, \mathtt{v}_j) \mid i, j \in \{0, \ldots, n\}, i < j\}$) and (ii) all updates in the local state of the clients after the new ones (i.e., $(\{\mathtt{v}_0, \ldots, \mathtt{v}_n\} \times \{\mathtt{w} \mid \mathtt{w} \in \mathbb{U} \; \forall u.u^{\mathtt{w}} \notin S \cdot \mathsf{u}\}$). The remaining transitions of the system are considered as internal changes that do not affect the history and are handled by rule (A-INT).

Lemma 5.1. *Let* $\mathbb{A}$ *be a well-formed history. If* $\mathbb{A} \xrightarrow{\lambda}_{\mathsf{i}} \mathbb{A}'$, *then* $\mathbb{A}'$ *is well-formed.*

We use histories to analyse the ordering guarantees offered by the GSP model. (Due to space limitation, we refer the interested reader to see the characterisations provided in [2, Ch. 5]).

(A-UPDATE)

$$\frac{N \xrightarrow{wr(u^{\mathbb{v}})}_{\mathsf{i}} N' \qquad \mathbb{v} \notin dom(\text{OP}) \qquad \text{OP}' = \text{OP}[\mathbb{v} \mapsto u] \qquad \text{SS}' = \text{SS}[\mathsf{i} \mapsto \text{SS}(\mathsf{i}) \cup \{\mathbb{v}\}] \quad \text{SO}' = \text{SO} \cup (\text{SS}(\mathsf{i}) \times \{\mathbb{v}\}) \qquad \text{AR}' = \text{AR} \cup (\{\mathbb{w} \mid u^{\mathbb{w}} \in S\} \times \{\mathbb{v}\})}{\langle N, \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR}\rangle \xrightarrow{wr(u^{\mathbb{v}})}_{\mathsf{i}} \langle N', \text{OP}', \text{SS}, \text{SO}', \text{VIS}, \text{AR}'\rangle}$$

(A-READ)

$$\frac{N \xrightarrow{rd(r)}_{\mathsf{i}} N' \qquad \mathbb{v} \notin dom(\text{OP}) \qquad \text{OP}' = \text{OP}[\mathbb{v} \mapsto r] \qquad \text{SS}' = \text{SS}[\mathsf{i} \mapsto \text{SS}(\mathsf{i}) \cup \{\mathbb{v}\}] \quad \text{SO}' = \text{SO} \cup (\text{SS}(\mathsf{i}) \times \{\mathbb{v}\}) \qquad \text{VIS}' = \text{VIS} \cup (\{\mathbb{w} \mid u^{\mathbb{w}} \in S[0..k_{\mathsf{i}} - 1] \cdot \mathsf{b}_{\mathsf{P}\mathsf{i}} \cdot \mathsf{u}_{\mathsf{T}\,\mathsf{i}}\} \times \{\mathbb{v}\})}{\langle N, \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR}\rangle \xrightarrow{rd(r^{\mathbb{v}})}_{\mathsf{i}} \langle N', \text{OP}', \text{SS}', \text{SO}', \text{VIS}', \text{AR}\rangle}$$

(A-ARB)

$$\frac{S \parallel C \xrightarrow{\lambda}_{\mathsf{i}} S \cdot \mathsf{u} \parallel C' \qquad \lambda \neq wr(u^{\mathbb{v}}), rd(r) \qquad \mathsf{u} = (\!| u_0^{\mathbb{v}_0} \cdots u_n^{\mathbb{v}_n} |\!) \quad \text{AR}' = \text{AR} \cup \{(\mathbb{v}_i, \mathbb{v}_j) \mid i, j \in \{0, \ldots, n\}, i < j\} \cup (\{\mathbb{v}_0, \ldots, \mathbb{v}_n\} \times \{\mathbb{w} \mid \mathbb{w} \in \mathbb{U},\ \forall u.u^{\mathbb{w}} \notin S \cdot \mathsf{u}\})}{\langle S \parallel C, \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR}\rangle \xrightarrow{\lambda}_{\mathsf{i}} \langle S \cdot \mathsf{u} \parallel C', \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR}'\rangle}$$

(A-INT)

$$\frac{S \| C \xrightarrow{\lambda}_{\mathsf{i}} S \| C' \qquad \lambda \neq wr(u^{\mathbb{v}}), rd(r)}{\langle S \| C, \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR}\rangle \xrightarrow{\lambda}_{\mathsf{i}} \langle S \| C', \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR}\rangle}$$

Fig. 6. Computation of abstract executions

Theorem 5.1. *If* $\langle N, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle \to_{\mathsf{i}}^{*} \langle N', \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR}\rangle$ *then*

(1) Read My Writes: $\text{SO}{\downarrow_{\mathbb{U}\times\mathbb{R}}} \subseteq \text{VIS}$
(2) Monotonic Read: $\text{VIS}; \text{SO}{\downarrow_{\mathbb{U}\times\mathbb{R}}} \subseteq \text{VIS}$.
(3) No Circular Causality: $(\text{SO} \cup \text{VIS})^{+}$ *is acyclic.*
(4) Causal Visibility: $(\text{SO} \cup \text{VIS})^{+}{\downarrow_{\mathbb{U}\times\mathbb{R}}} \subseteq \text{VIS}$.
(5) Causal Arbitration: $((\text{SO} \cup \text{VIS})^{+} \setminus \text{SO}){\downarrow_{\mathbb{U}\times\mathbb{U}}} \subseteq \text{AR}$.
(6) Consistent prefix: $\text{AR}; (\text{VIS} \setminus \text{SS}) \subseteq \text{VIS}$.

The following example shows that the GSP model exhibits the Dekker anomaly, hence it does not enjoy *sequential consistency* [2].

Example 5.1 (Dekker anomaly). Consider the following system consisting of two clients and the empty store $N = \epsilon \parallel C_1 \parallel C_2$ where

$$C_1 = \langle \texttt{update}(u_1); \texttt{let } y = \texttt{read}(r_1) \texttt{ in } P, \epsilon, \epsilon, \epsilon, 0, 0\rangle_{\mathsf{i}_1}$$
$$C_2 = \langle \texttt{update}(u_2); \texttt{let } y = \texttt{read}(r_2) \texttt{ in } Q, \epsilon, \epsilon, \epsilon, 0, 0\rangle_{\mathsf{i}_2}$$

Since the updates are made locally, none of the clients see the update performed by the other and this is the essence of the Dekker anomaly which is ruled out by strong consistency models like sequential consistency or linearizability.

(SYNC-UPD)
$$\langle \texttt{syncUpd}(u); P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N \quad \xrightarrow{\tau}_\mathsf{i} \quad \langle \texttt{update}(u); \texttt{push}; \texttt{wait}; P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N$$

(WAIT)
$$\langle \texttt{wait}; P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N \quad \xrightarrow{\tau}_\mathsf{i} \quad \langle \texttt{let } x = \texttt{confirmed in } x \triangleright (\texttt{pull}; \texttt{wait}; P); P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N$$

(GUARD-TRUE)
$$\frac{e \downarrow \mathit{true}}{\langle e \triangleright (P); Q, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N \quad \xrightarrow{\tau}_\mathsf{i} \quad \langle Q, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N}$$

(GUARD-FALSE)
$$\frac{e \downarrow \mathit{false}}{\langle e \triangleright (P); Q, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N \quad \xrightarrow{\tau}_\mathsf{i} \quad \langle P, \mathsf{u_T}, \mathsf{b_S}, \mathsf{b_P}, k, j \rangle_\mathsf{i} \parallel N}$$

Fig. 7. Semantics of GSP with atomic updates

6 GSP with atomic updates

In this section we study the atomic updates proposed in [5]. We extend the language of programs as follows:

$$\text{(PROGRAM)} \qquad P \ ::= \ \ldots \mid \texttt{syncUpd}(u); P$$

The execution of a program $\texttt{syncUpd}(u); P$ remains blocked until the update u is performed over the global store. This is achieved by continuously pulling (i.e., a busy-waiting) until the updates are confirmed by global store. In order to provide the formal semantics of the language, we consider the following runtime syntax for programs.

$$\text{(RUN-TIME-PROGRAM)} \qquad P \ ::= \ \ldots \mid \texttt{wait}; P \mid e \triangleright (P); P$$

The operational semantics for the new primitives is given by the rules in Fig. 2. Rule (SYNC-UPD) rewrites each synchronous update as the sequence consisting of an asynchronous update followed by `pull` and `wait`. Processes `wait` continuously checks whether local changes have been confirmed by the global store. As described by rule (WAIT), it is implemented as a busy-waiting loop that first checks the local buffers by executing `confirmed` and then performs the conditional jump $x \triangleright (\texttt{pull}; \texttt{wait}); P$. If the condition x is true, then it follows as P otherwise it continues as $\texttt{pull}; \texttt{wait}$, as described by rules (GUARD-TRUE) and (GUARD-FALSE).

Single order is characterised, essentially, by imposing arbitration and visibility to coincide [2]. Since our definition for AR and VIS makes them disjoint, we use an alternative characterisation of single order guarantee, which disregards the arbitration order of updates that are not observed. Hence, we use the following characterisation for single order:

$$\text{AR}; \text{VIS} \subseteq \text{VIS} \quad \text{and} \quad \text{AR}^{-1}; \neg\text{VIS} \subseteq \neg\text{VIS}$$

The following result shows that any well-formed GSP system, whose programs are free from asynchronous updates enjoy the single order guarantee.

Theorem 6.1 (Single Order). *Let N be a well-formed system s.t.* `update`(u) *does not appear in N. If* $\langle N, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \rightarrow_{\mathsf{i}}^{*} \langle N', \text{OP}, \text{SS}, \text{SO}, \text{VIS}, \text{AR} \rangle$ *then* $\text{AR}; \text{VIS} \subseteq \text{VIS}$ *and* $\text{AR}^{-1}; \neg\text{VIS} \subseteq \neg\text{VIS}$.

7 Conclusions

We have proposed a formal model for the Global Sequence Protocol and its proposed implementation. We use our formal model to provide a simplified proof (that relies on standard simulation) that the proposed implementation is correct. We remark that our proof does not require to exhibit an auxiliary state for the simulation and that several invariants are trivially ensured by the definition of the model (e.g., the fact that clients have a consistent view of the global sequence) and the well-formed conditions imposed over systems. We have formally studied the consistency guarantees ensured by the model by relying on the operational semantics of the calculus to incrementally compute (a relaxed version of) abstract histories. We have also shown how GSP can be used to formally study programming patterns, like synchronous update operations, that provide stronger consistency guarantees at the expenses of efficiency and availability. We plan to use the GSP calculus as a formal basis for developing programming techniques to enable the fine-tuning of consistency levels in applications.

Acknowledgments We thank the anonymous reviewers of Coordination 2016 for their careful reading of our paper and detailed comments.

References

1. P. Bailis and A. Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013.
2. S. Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.
3. S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP 2012*, pages 283–307. Springer, 2012.
4. S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP 2012*, pages 67–86. Springer, 2012.
5. S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *ECOOP 2015*, pages 568–590, 2015.
6. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07*, pages 205–220. ACM, 2007.
7. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

8. A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause i'm strong enough: reasoning about consistency choices in distributed systems. In *POPL 2016*, pages 371–384, 2016.
9. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
10. M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS 2011*, pages 386–400, 2011.
11. K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI 2015*, pages 413–424. ACM, 2015.