Global Sequence Protocol

A Robust Abstraction for Replicated Shared State

(Extended version, Microsoft Research Technical Report, MSR-TR-2015-11)

Sebastian Burckhardt[1], Daan Leijen[1], Jonathan Protzenko[1], and Manuel Fähndrich[2]

1 **Microsoft Research**
2 **Google**

Abstract

In the age of cloud-connected mobile devices, users want responsive apps that read and write shared data everywhere, at all times, even if network connections are slow or unavailable. The solution is to replicate data and propagate updates asynchronously. Unfortunately, such mechanisms are notoriously difficult to understand, explain, and implement.

To address these challenges, we present GSP (global sequence protocol), an operational model for replicated shared data. GSP is simple and abstract enough to serve as a mental reference model, and offers fine control over the asynchronous update propagation (update transactions, strong synchronization). It abstracts the data model and thus applies both to simple key-value stores, and complex structured data. We then show how to implement GSP robustly on a client-server architecture (masking silent client crashes, server crash-recovery failures, and arbitrary network failures) and efficiently (transmitting and storing minimal information by reducing update sequences).

This technical report is an extended version of the ECOOP'15 paper [11].

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases distributed computing, eventual consistency, GSP protocol

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.999

1 Introduction

Many applications can benefit from replicating shared data across devices, because it is often desirable to keep applications responsive even if network connections are slow or unavailable. Unfortunately, the CAP theorem [4, 17, 19] shows that strong consistency (such as linearizability or sequential consistency) requires communication with a reliable server or with a majority partition on each update, which becomes slow or impossible if network connections are slow or unavailable. Since responsiveness is often more important than strong consistency, researchers and practitioners have proposed the use of various forms of eventual consistency [8, 9, 15, 21, 29]. In such systems, update propagation and conflict resolution is lazy, proceeding only when network conditions permit, and replicas may temporarily differ, while converging to the same state eventually.

Although asynchronous update propagation and eventual consistency offer clear benefits, they are also more difficult to understand, both for system implementors and client programmers, motivating the need for simple programming models.

Previous work on replicated data types [7, 23, 24] and cloud types [5, 14] suggests that higher-level data abstractions can mitigate the mental overhead of working with weakly con-

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 999–1052

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

sistent replicas, as they can resolve conflicts automatically and prevent us from accidentally breaking representation invariants due to unexpected races.

To evaluate these ideas in practice, we have implemented the cloud types model [5, 9, 14] in a scripting language for mobile devices. Our experiences suggest that it can indeed provide significant benefits. In particular, the automation of communication, error handling, and replication substantially simplifies the app development. However, data abstraction is not enough, and there remains room for improvement in several aspects:

- *Reasoning.* Client programmers often misunderstand where exactly they risk consistency errors, erring both on the safe and the unsafe side. Moreover, they are generally wary of "magical solutions" that do not convey an intelligible mechanism. Above all, what they need is a simple mental reference model to understand how to use the mechanism appropriately to write correct programs.
  The existing consistency models for cloud types are either too abstract for non-experts in memory consistency (e.g. the axiomatic model in [9]), or too complicated and overly general for the situation at hand (e.g. the revision diagram model in [5, 9]).
- *Judicious Synchronization.* While asynchronous update propagation is sufficient in most situations, for many apps we encountered a few situations where strong synchronization is needed (e.g. finalizing a reservation, ending an auction, or joining a game with an upper limit on the number of players). Thus, it is important that programmers can easily choose between synchronous and asynchronous reads and updates (and pay the cost of synchronicity only when they ask for it).
- *Robust Implementations.* Implementations must be carefully engineered to hide failures of clients, the network, and the server, and to minimize the amount of data stored and transmitted.
  For example, The cloud types implementations in [5, 14] do not discuss failures of any kind, and transmit the entire state in each message, which is impractical unless the amount of data shared is small. Moreover, the pushing and pulling of updates between client and server cannot proceed concurrently but is forced to alternate, which introduces significant delays.

In this paper, we describe several improvements in these areas. Specifically, we make the following contributions:

- We introduce the global sequence protocol (GSP), an operational model describing the system behavior precisely, yet abstractly enough to be suitable as a simple mental model of the replicated store. It is based on an abstract *data model* that can be instantiated to any data type, be it a simple key-value store, or the rich cloud types model. We compare GSP to the TSO (total store order) memory model and discuss its consistency properties.
- We show how GSP supports judicious use of synchronization. *Push* and *pull* operations give programmers precise control over the update propagation, and *flush* allows them to perform reads and updates synchronously whenever desired, thus recovering strong consistency.
- We present a detailed system implementation model of GSP that provides significant advantages:
  Robust Streaming. Updates are streamed continuously in both directions between server and client. We show precisely how clients may crash silently, how the server may fail and recover, how connections are established, how they can fail, and how they can be

reconnected and resume transmission correctly, without disrupting the execution of the client program at any point.

Reduction. Update sequences often exhibit redundancy (for example, if a variable is assigned several times, only the last update matters). We show how to eagerly reduce update sequences, storing them in reduced form in state or delta objects. This means that our implementation stores and propagates a minimal amount of information only.

- We have implemented the ideas presented in this paper as an extension of TouchDevelop, a freely available programming language and development environment. Thus, we have made the cloud types programming model publicly available online for inspection and experimentation, and we provide links to a dozen example applications.

Overall, our work marks a big step forward towards a credible programming model for automatically replicated, weakly consistent shared data on devices, by providing both an understandable high-level system and data model, and a robust implementation containing powerful and interesting optimizations.

2 Overview

To write correct programs, we need a simple yet precise mental model of the store. In a conventional setting, we assume a single-copy semantics where client programs can read and write the shared data atomically. But to tolerate slow and unreliable connections, we must find an alternative model that accounts for the existence of multiple copies, i.e. multiple versions of the shared data.

To this end, we introduce in this paper an operational model of a replicated store, called *Global Sequence Protocol* (GSP). It is based on the simple idea that clients eventually agree on a global sequence of updates, while seeing a subsequence of this final sequence at any given point of time.

We introduce GSP in four stages. First, we clarify how to abstract the data operations (section 3). Then, we introduce and explain Core GSP, a basic version of GSP that does not include transactions or synchronization (section 4), and discuss various aspects of its consistency model. In section 5 we present transactional GSP, and explain the benefits of its transactions and synchronization support. In section 6, we show in detail how the GSP protocol can be realistically implemented on a client-server topology in a way that transparently hides channel failures, silent crashes of clients, and crash-recovery failures of the server. A cornerstone of the implementation is the use of *reduction*, which eliminates redundancy from update sequences.

We then conclude the paper by reporting on our practical experiences with implementing and operating GSP as an extension of TouchDevelop (section 7), and comparing with related work (section 8).

3 Data Models

In our experience, the key mental shift required to understand replicated data is to understand program behavior as a sequence of updates, rather than states. To this end, we characterize the shared data by its set of updates and queries, and represent a state by the sequence of updates that have led to it.

Sequence notations. We write $T*$ for the type of sequences of type T. Furthermore, $[]$ is the empty sequence, $s_1 \cdot s_2$ is the concatenation of two sequences, $s[i]$ is the element

at position i (starting with 0), and for a nonempty sequence s ($s.length > 0$), the expression $s[1..]$ denotes the subsequence satisfying $s = s[0] \cdot s[1..]$.

Rather than fixing the set of update and read operations upfront, we represent them using abstract types for updates, reads, and values.

abstract type *Update*, *Read*, *Value*;

Likewise, we abstractly represent the semantics of operations by a function *rvalue* that takes a read operation and a sequence of updates, and returns the value that results from applying all the updates in the sequence to the initial state of the data:

function $rvalue$: $Read \times Update^* \to Value$

We call a particular binding for *Update*, *Read*, *Value* and *rvalue* a *data model*.

3.1 Examples

Register. We can define a data model for a *register* using the following update and read operations

$$\begin{aligned} Update &= \{\, wr(v) \mid v \text{ in } Value \,\} \\ Read &= \{\, rd \,\} \end{aligned}$$

and define the value returned by a read operation to be the last value written:

$$\begin{aligned} rvalue(rd, s) = \ & \text{match } s \text{ with} \\ & [] \quad \to \text{undefined} \\ & s_0 \cdot wr(v) \to v \end{aligned}$$

Counter. We can define a data model for a *counter* as follows:

$$\begin{aligned} Update &= \{\, inc \,\} \\ Read &= \{\, rd \,\} \\ rvalue(rd, s) &= s.length \end{aligned}$$

where a read simply counts the number of updates.

Key-value Store. Perhaps the most widely used data type in cloud storage is the *key-value store*, which will serve as our main running example. We can define its data model as follows:

$$\begin{aligned} Update &= \{\, wr(k,v) \mid k,v \text{ in } Value \,\} \\ Read &= \{\, rd(k) \mid k \text{ in } Value \,\} \end{aligned}$$

$$\begin{aligned} rvalue(rd(k), s) = \ & \text{match } s \text{ with} \\ & [] \quad \to \text{undefined} \\ & s_0 \cdot wr(k_0,v) \to \text{if } (k = k_0) \text{ then } v \text{ else } rvalue(rd(k),s_0) \end{aligned}$$

Reduction of Update Sequences. For readers who may be alarmed by the prospect of having to store and transmit long update sequences: note that we will introduce state and delta objects in section 6.1, which store update sequences in reduced form (for example, a key-value store needs to store only the last update for a given key).

4 Core GSP

We show a basic version of the global sequence protocol in Fig. 1, which includes the data operations (reads and updates), but omits synchronization and transactions for now.

The protocol specifies the behavior of a finite, but unbounded number of clients, by defining the state of each client, and transitions that fire in reaction to external stimuli. The

```
role Core_GSP_Client {

  known : Round * := []; // known prefix of global sequence
  pending: Round * := []; // sent, but unconfirmed rounds
  round : ℕ := 0; // counts submitted rounds

  // client program interface
  update(u: Update) {
    pending := pending · u;
    RTOB_submit( new Round { origin=this, number=round++ , update=u } );
  }
  read(r: Read) : Value {
    var compositelog := known · pending;
    return rvalue(r, updates(compositelog));
  }

  // network interface
  onReceive(r: Round) {
    known := known · r;
    if (r.origin = this) {
      assert(r = pending[0]); // due to RTOB total order
      pending := pending[1..];
    }
  }

  // rounds data structure
  class Round { origin: Client, number: ℕ, update: Update }
  function updates(s: Round *): Update * { return s[0].update · · · s[s.length - 1].update; }
}
```

Figure 1 Core Global Sequence Protocol (GSP).

transitions fall into two categories: the interface to the client program (from where update and read operations arrive), and the interface to the network (from where messages arrive).

The clients communicate using *reliable total order broadcast* (RTOB), a group communication primitive that guarantees that all messages are reliably delivered to all clients, and in the same total order. RTOB has been well studied in the literature on distributed systems [12, 16], and is often used to build replicated state machines. It can be implemented on various topologies (such as client-server or peer-to-peer) and for various degrees of fault-tolerance. We describe one particular such implementation and important optimizations in Section 6.

Core GSP stores and propagates updates as follows.

- Each client stores a currently known prefix of the global update sequence in *known*, and a sequence of pending updates in *pending*.
- When the client program issues an update, we (1) append this update to the sequence of pending updates, and (2) wrap the update into a *Round* object, which includes the origin and a sequence number, and broadcast the round.

- When receiving a round, we append the contained update to the known prefix of updates. Moreover, if this round is an echo (it originated on the same client), we remove it from the pending queue.

Since RTOB delivers messages in the same order to all recipients, the *known* prefixes in the clients (while not necessarily the same length at any given time) always match. Also, an echo of a round always matches the first (oldest) element of the *pending* queue.

When a client issues a read, we combine the update sequences in the *known* prefix and in the *pending* operations to determine the value returned by the read. Thus, it appears to the client program that its own updates have taken effect, before they are confirmed (i.e. before they are processed and echoed by the RTOB). This consistency property is sometimes called Read-my-Writes [28]).

An important point is that we cannot rely on RTOB being fast: at best, it requires a server roundtrip, and at worst, it can be stalled for prolonged periods by a failure or by a network partition, for example if the client is offline. Thus, making the updates in the pending queue visible to reads is essential for applications to appear responsive.

Example. We can implement a causally consistent key-value store by using the read $rd(k)$ and write $wr(k,v)$ operations defined earlier. Then clients can always read and write any key without waiting for communication. In particular, the store remains operational even on clients that are temporarily offline. If two clients write a different value for the same key, they may temporarily see a different value, but once both updates have gone through the RTOB, their relative order in the global sequence determines the final value: the last writer wins.

4.1 Beware Consistency

By design, Core GSP is not strongly consistent: *updates are asynchronous and take effect with a delay*. Programmers who are not aware of this can easily run into trouble. For example, consider a program that tries to increment a value for a given key by reading it, adding one, and then writing it back:

$\mathsf{var}\, x = rd(\text{"counter"})$
$wr(\text{"counter"}, x + 1)$

This counter implementation does not count correctly if called concurrently. For example, two readers may both read the current value 0, and then both issue an update wr("counter", 1). Thus, the final value (once both updates have gone through RTOB) is 1, not 2 as we would like.

We show in section 5 how to extend Core GSP with synchronization operations that can be used to enforce strong consistency where needed (at the expense of requiring communication, and losing the benefit of offline availability).

However, in many cases, there is a more elegant solution that avoids expensive synchronization. The trick is to use a richer data model that lets us express the update directly, at a higher level. For example, if the data model supports updates of the form $add(k,v)$, we can increment a counter by calling

$add(\text{"counter"}, 1)$

which always counts correctly: all add operations appear in the global sequence, and the read operation can correctly accumulate them. In general, the idea of including application-specific update operations in the data model is a powerful trick that can help to avoid synchronization in many situations.

4.2 Cloud Types

Although key-value stores are a powerful primitive, they are cumbersome and error-prone to work with directly. Productivity is greatly aided by a capability to declare structured data with richer update and query semantics.

Luckily, it turns out we can quite easily define higher level data types on top of the data model abstraction (section 3). In particular, we can implement full Cloud Types as proposed by previous work [5, 14]. Cloud types allow users to define and compose all of the data type examples given earlier (registers, counters, key-value stores), plus tables, which support dynamic creation and deletion of storage.

Cloud types also help to mitigate consistency issues, since concurrent updates are handled in a way that is consistent with the semantics of the type. For example, all integer-typed fields support an $add(n)$ operation.

In appendix B.1 we show how to define cloud types as a data model, and how to implement state and delta objects that optimally reduce the update sequences.

4.3 Eventual Consistency

Although GSP does not provide strong consistency, its consistency guarantees are still as strong as possible for a protocol that remains available under network partitions: it is quiescently consistent, eventually consistent, and causally consistent (as defined in [8], for example).

It is *quiescently consistent*, because when updates stop, clients converge to the same *known* prefix with empty *pending* queue (this is the original definition of eventual consistency as introduced in [29]). It is *eventually consistent* (as defined in [6–9]) because all updates become eventually visible to all clients, and are ordered in the same arbitration order. It is *causally consistent* because an update U by some client C cannot become visible to other clients before all of the updates (let's call them V) that are visible to client C at the time it performs U. The reason is that the updates V consist of (1) the common server prefix, or (2) pending updates, which are all guaranteed to become visible to other clients no later than U.

Comparison to TSO. Core GSP appears conceptually (and even in name) quite similar to TSO [31] (total store order), a widely used relaxed memory model that queues stores performed by a processor in a local store buffer, from where they drain to memory asynchronously. This naturally leads us to ask the question: is Core GSP observationally equivalent to TSO? Interestingly, the answer depends on the notion of observational equivalence. If we assume that the relative order of operations on different clients is not directly observable (which is a common assumption for memory models, where clients are processors that do not communicate directly), the two are indeed equivalent. However, if the relative order of operations on different clients is observable (which is a reasonable assumption for distributed interactive applications with external means of communication), then they are not equivalent, as the following scenario illustrates.

Consider that the key-value store data model represents shared memory in a multiprocessor, which initially stores 0 for each address, and consider two clients performing the following interleaving of operations (where each column shows the operation of one client, and vertical placement defines the observed interleaving of the operations):

$wr(A,2)$	.
.	$wr(B,1)$
.	$wr(A,1)$
.	$rd(A) \rightarrow 2$
$rd(B) \rightarrow 0$	.

This interleaving is not observable on TSO: since the client on the right sees $rd(A)$ return 2, it must be the case that $wr(A,2)$ has drained to memory after $wr(A,1)$ drained. Since writes by the same client drain to memory in order, this implies that $wr(B,1)$ must have drained to memory sometime before $rd(A)$ returns, and thus before $rd(B)$ is called, thus the $rd(B)$ cannot return 0. However, under GSP, this interleaving is possible, because $rd(B)$ may be called before the RTOB delivers the update for $wr(B,1)$ to the client on the left.

5 Transactional GSP

The Core GSP protocol we introduced in the previous section is already quite useful. However, it can be further improved by adding support for transactions and synchronization.

In this section, we introduce transactional GSP, which adds the synchronization operations *push* and *pull*, and the synchronization query *confirmed*. These additions give the programmer more control. The transactional GSP protocol is shown in Fig. 2. It is derived from Core GSP (Fig. 1), but improves the design in the following three aspects.

Update Transactions. Often, a client program updates several data items at a time, and those updates are meant to be atomic. For example,

wr("items", "[key1,key2]")
wr("key1", "something")
wr("key2", "something else")

In the global sequence model, the updates may arrive at another client at different times, thus that client may see an intermediate state that it was not supposed to observe.

To solve this problem, GSP uses a *transactionbuffer*. Updates performed by the client program go into this buffer. All updates in the buffer are broadcast in a single round when the client program calls *push*, and only then. They effectively form an 'update transaction' that is persisted and transmitted atomically. Updates in the transactionbuffer are included in the composite log, thus they are immediately visible to subsequent reads.

Read Stability. In the global sequence protocol, an update arriving from the network can interleave in unpredictable ways with the locally executing client program. In particular, if a client program performs two reads in a sequence, the second read may return a different value. In our experience, it is very difficult to write correct programs under such conditions (cf. data races in multiprocessor programs).

To solve this problem, GSP uses a *receivebuffer*. Received rounds are stored in this buffer. All rounds in the *receivebuffer* are processed when the client program calls *pull*, and only then. Thus, the client program can rely on read stability - the visible state can change only when issuing *pull*, or when performing local updates.

Confirmation Status. It is often desirable to find out if an update has committed (i.e. is now part of the global sequence constructed by the RTOB). Since this is impossible for client programs to detect in the global sequence protocol, we add a new function *confirmed* to the interface which returns true iff there are no local updates awaiting confirmation.

```
role GSP_Client {
  known : Round * := []; // known prefix of global sequence
  pending: Round * := []; // sent, but unconfirmed rounds
  round : ℕ := 0; // counts submitted rounds
  transactionbuffer : Update * := [];
  receivebuffer: Round * := [];

  // client program interface
  update(u: Update) { transactionbuffer := transactionbuffer · u; }
  read(r: Read) : Value {
    var compositelog := updates(known) · updates(pending) · transactionbuffer;
    return rvalue(r, compositelog);
  }
  confirmed() : boolean { return pending = [] && transactionbuffer = [] }
  push() {
    var r := new Round { origin=this, number=round++, updates=transactionbuffer};
    tob_submit( r );
    pending := pending · r;
    transactionbuffer := [];
  }
  pull() {
    foreach(var r in receivebuffer) {
      known := known · r;
      if (r.origin = this) { pending := pending[1..]; }
    }
    receivebuffer := [];
  }

  // network interface
  onReceive(r: Round) { receivebuffer := receivebuffer · r; }

  // rounds data structure
  class Round { origin: Client, number: ℕ, updates: Update }
  function updates(s: Round *): Update * { return s[0].updates · · · s[s.length-1].updates;}
}
```

Figure 2 Transactional GSP (Global Sequence Protocol)

5.1 Discussion

We now discuss several interesting aspects of the transactional GSP model related to consistency and synchronization.

On-Demand Strong Consistency. GSP is sufficiently expressive to allow client programs to recover strong consistency when desired. To this end, we can write a flush operation that waits for all pending updates to commit (and receives any other updates in the meantime):

```
flush() {
  push();
  while ( ! confirmed()) { pull(); }
}
```

Using *flush*, we can implement linearizable (strongly consistent) versions of any read operation r or update operation u as follows:

```
synchronous_update(u) { update(u); flush(); }
synchronous_read(r)   { flush(); read(r); }
```

These synchronous versions exhibit a single-copy semantics: they behave as if the read or update were executed directly on the server.

In practice, we found that for most applications, the majority of reads and updates need not be strongly consistent. However, there often remain a few situations (e.g. finalizing a reservation, ending an auction, or joining a game with an upper limit on the number of players) where true arbitration is required, and where we are willing to pay the cost of synchronous communication (i.e. wait for the server to respond, or even block if offline). The ability of GSP to handle both synchronous and asynchronous reads and updates within the same framework is thus a major advantage.

Automatic Transactions. A prime scenario for GSP is the development of user-facing reactive event-driven applications, such as web applications or mobile apps. In that setting, we found it advantageous to automate the *push* and *pull* operations. Since the client program is already designed for cooperative concurrency and executes in event handlers, our framework can execute the following yield operation automatically between event handlers, and repeatedly when the event queue is empty:

```
yield() {
  push();
  pull();
}
```

All of the applications we wrote using the TouchDevelop platform rely on automatic transactions.

Comparison of Transactions. Our update transactions are different from conventional transactions (read-committed, serializable, snapshot isolation, or parallel snapshot isolation) since they do not check for any read or write conflicts. In particular, they never fail. The advantage is that they are highly available [2], i.e. progress is not hampered by network partitions. The disadvantage is that unlike serializable transactions (but like read-committed, snapshot, or parallel snapshot transactions), they not preserve all data invariants.

6 Robust Streaming

The GSP model described in the previous sections abstracts away many details that are important when we try to implement it in practice. In particular, it assumes that we have a RTOB implementation, it does not model failures of any kind, and it suffers from space explosion due to ever-growing update sequences.

In this section, we show that all of these issues are fixable without needing to change the abstract GSP protocol. Specifically, we describe a robust streaming server-client implementation of GSP. It explicitly models communication using sockets (duplex streams) and contains explicit transitions to model failures of the server, clients, and the network. Moreover, it eliminates all update sequences, and instead stores current server state and deltas in reduced form. Importantly, it is robust in the following sense:

- *Client programs never need to wait for operations to complete, regardless of failures in the network, server, or other clients.*
- Connections can fail at any time, on any end. New connections can replace failed ones and resume transmission correctly.
- The server may crash and recover, losing soft state in the process, but preserving persistent state. The persisted server state contains only a snapshot of the current state and the number of the last round committed by each client. It *does not store any logs.*
- Clients may crash silently or temporarily stop executing for an unbounded amount of time, yet are always able to reconnect. In particular, there are no timeouts (the protocol is fully asynchronous). Permanent failures of clients cannot disrupt the server, other clients, or violate the consistency guarantees.

Despite the more realistic communication and the possibility of channel and server failures, the streaming protocol remains faithful to the original protocol: we prove that it is a refinement, that is, all of its behaviors correspond to a behavior of the abstract protocol (transactional GSP, see Fig. 2). Thus, programmers may remain blissfully unaware of these complications.

6.1 States and Deltas

The streaming model does not store any update sequences (neither in the client, nor on the server). Instead, it eagerly reduces such sequences, and stores them in reduced form, either as *state objects* (if the sequence is a prefix of the global update sequence) or as *delta objects* (if the sequence is a segment of the global update sequence).

abstract type *State*
abstract type *Delta*

Deltas are produced by appending updates, or by reducing several deltas, and states are produced by applying deltas to the initial state:

const *initialstate* : *State*
function *read* : $Read \times State \to Value$
function *apply* : $State \times Delta^* \rightharpoonup State$
const *emptydelta* : *Delta*
function *append* : $Delta \times Update \rightharpoonup Delta$
function *reduce* : $Delta^* \rightharpoonup Delta$

Note that we define some of these functions as partial, reflecting that some updates may be invalid (in the cloud types model, these include incorrectly typed field updates or creation

```
class Channel {
  client: Client; // immutable
  Channel(c: Client) { client := c; }

  // duplex streams
  clientstream: Round * := [];                    // client to server
  serverstream: (GSPrefix | GSSegment) * := []; // server to client

  // server-side connection state
  accepted: boolean := false; // whether server has accepted connection

  // client-side connection state
  receivebuffer: (GSPrefix | GSSegment) *; // locally buffered packets
  established: boolean := false;              // whether client processed 1st packet
}
```

Figure 3 Channel Objects.

of a row with a duplicate unique identifier, for example).

Example. For the key-value store, the implementation of this abstract interface is straightforward. We can represent both *State* and *Delta* as maps from keys to values, and define *reduce*, *append* and *apply* to simply merge such mappings (where the last write wins).

Correctness. Intuitively, a state-and-delta implementation correctly represents a given data model if the result of reading a state s by means of calling $read(r,s)$ yields the same result as reading $rvalue(r,u_1 \cdot u_n)$ where $u_1 \cdot u_n$ is the combined sequence of updates that led to the state s. More formally, we can define a overloaded *representation* relation $\lhd$ that relates state and delta objects to the update sequences they represent, as follows:

- On *Delta* $\times$ *Update* $*$, let $\lhd$ be the smallest relation such that (1) $emptydelta \lhd []$, and (2) $d \lhd a$ implies $append(d,u) \lhd a \cdot u$ for all updates u, and (3) $d_1 \lhd a_1 \wedge \cdots \wedge d_n \lhd a_n$ implies $reduce(d_1 \cdots d_n) \lhd a_1 \cdots a_n$.
- On *State* $\times$ *Update* $*$, let $\lhd$ be the smallest relation such that (1) $initialstate \lhd []$, and (2) $s \lhd a \wedge d_1 \lhd a_1 \wedge \cdots \wedge d_n \lhd a_n$ implies $apply(s,d_1 \cdots d_n) \lhd a \cdot a_1 \cdots a_n$.

Now, we define a state-and-delta implementation to be *correct* if and only if $s \lhd a$ implies $read(r,s) = rvalue(r,a)$ for all reads r, states s and update sequences a.

Optimality. Subtleties arises when we care about space leaks. For the key-value store, for example, a sloppy implementation of the state object may fail to remove a key whose value is set to undefined from the map. We call such an implementation *non-optimal*, because some states occupy more space than needed (there exists a smaller representation of the same update sequence that is indistinguishable by queries).

Engineering state and delta objects to be optimal can be quite challenging once richer data types are considered, for example regarding dynamic creation and deletion of table rows. In appendix B.4 we show an example of such a nontrivial optimal implementation of state and delta objects for the cloud types data model.

6.2 Channels

We model the network and communication sockets using *Channel* objects (Fig. 3). Channels contain two streams, one for each direction. We model streams using sequences, by adding elements on the right and removing them on the left. Channel objects also contain server-side and client-side connection state that can be read and written only on the respective side (i.e. it is not used for communication).

The sequence *clientstream* contains the rounds that the client sends to the server. The *Round* objects are defined as in GSP, except that they contain a delta object in place of a sequence of updates:

```
struct Round { origin: Client; number: ℕ; delta: Delta; }
```

The sequence *serverstream* contains reduced segments of the global sequence that the server streams to the client. When sending on a channel, a server always starts with a *GSPrefix* object (containing a *State* object), and then keeps sending *GSSegment* objects (containing *Delta* objects).

```
class GSPrefix { // represents a prefix of the global update sequence
  state: State := initialstate;
  maxround: (Client ⇀ ℕ) := {};
  method apply(s: GSSegment) {
    foreach((c,r) in s.maxround) { maxround[c] := r; }
    state := apply(state, s.delta);
} }
class GSSegment { // represents an interval of the global update sequence
  delta: Delta := emptydelta;
  maxround: (Client ⇀ ℕ) := {};
  method append(r: Round): void {
    maxround[r.origin] := r.number;
    delta := reduce(delta · r.delta);
} }
```

The method *apply* extends a prefix with a segment, and the method *append* extends a segment with a round. Both *GSPrefix* and *GSSegment* contain a partial map *maxround* that records the maximal client round of each client that is contained in the segment. Thus a client c receiving a prefix or segment can look at $maxround[c]$ to determine the latest confirmed round.

6.3 Server

(Fig. 4) The server state is separated into persistent state (*serverstate*), which stores the current state and the number of the last round of each client that has been incorporated into the state, and soft state (*connections*) which stores currently active connections. A connection is started by the *accept_connection* transition, which adds it to the active connections *connections* and sets the *accepted* flag. It then sends the current state (i.e. the reduced prefix of the global sequence) on the channel.

During normal operation, the server repeatedly performs the *processbatch* operation. It combines a nondeterministic number of rounds (we use the $*$ in the pseudocode to denote a nondeterministic choice) from each active connection into a single segment. This segment stores all updates in reduced form as a delta object. We then append this segment to the persistent state (which applies the delta to the current state, and updates the maximum

```
class StreamingServer {
  // persistent state
  serverstate: GSPrefix := new GSPrefix();
  // soft state
  connections: (Client ⇀ Channel) := {};
  // transitions
  accept_connection(ch: Channel) {
    requires !ch.accepted && connections[ch.client] = null;
    ch.accepted := true;
    connections[ch.client] := ch;
    ch.serverstream := ch.serverstream · serverstate; // send first packet: current state
  }
  processbatch() {
    var s := new GSSegment();
    // collect updates from all incoming segments
    foreach((c,ch) in connections)
      receive(s, ch, *);
    // atomically commit changes to persistent state
    serverstate := serverstate.apply(s);
    // notify connected clients
    foreach((c,ch) in connections)
      ch.serverstream := ch.serverstream · s;
  }
  drop_connection(c: Client) { connections[c] := null; }
  crash_and_recover() { connections := {}; }
  // auxiliary functions
  receive(s: GSSegment, ch: Channel, count: int) {
    requires count <= ch.clientstream.length;
    foreach(r in ch.clientstream[0..count])
      s.append(r);
    ch.clientstream := ch.clientstream[count..];
  }
}
```

Figure 4 State and Transitions of the Streaming Server.

round number per client), and send it out on all active channels.

The transition *drop_connection* models the abrupt failure or disconnection of a channel at the server side - but not on the client, who may still send and receive packets until it in turn drops the connection. Note that a client may reconnect later, using a fresh channel object, and will resend rounds that were lost in transit when the channel was dropped.

The transition *crash_and_recover* models a failure and recovery of the server, which loses all soft state but preserves the persistent state.

6.4 Client

(Fig. 5,6) The fields of the client are similar to the transactional GSP client (Fig. 2), but with the following differences:

1. We use *State* and *Delta* objects instead of update sequences: *known* is now a *State* object, and *transactionbuf* is a *Delta* object.
2. There is a variable *channel* that contains the current connection (or null if there is none).
3. There is a new *pushbuffer*, which holds updates that were pushed but have not been sent on any channel yet (e.g. because there was no channel established at the time of the push). *rds_in_pushbuf* counts the number of rounds in the pushbuffer.
4. The *receivebuffer* is now stored inside the channel object.

The *read* transition computes the visible state by calling *curstate* which (nondestructively) applies the delta objects in *pending*, *pushbuf* and *transactionbuf* to the state object in *known*.

The *update* transition adds the given update to the transaction buffer, and clears the *tbuf_empty* flag (since delta objects do not have a function to query whether they are empty, we use a flag to determine whether the transaction buffer is empty).
The *confirmed* transition checks whether updates are pending in *pending* or *transactionbuf* or *pushbuffer*.

The *push* transition moves the content of the transactionbuffer to the pushbuffer, by combining and reducing the respective delta objects. The *pull* transition processes all packets in the receive buffer (which we model as part of the channel object), if any. When processing a packet, we track if this is the first packet received on this channel by checking the *established* flag.

- If the packet is not the first packet (*established* = true), it is a GSSegment packet containing a delta object and representing an aggregation of one or more rounds. This delta object is then applied to *known*. Since the segment may also contain (reduced) echoes of one or more unconfirmed rounds, we determine the latest confirmed round *s.maxround*[this] and remove all rounds up to that one from the *pending* queue (in *adjust_pending_queue*).
- If the packet is the first packet, it is a GSPrefix packet containing the latest server state. We assign it to *known* and set *established* to true. However, we need to do some more work: since there may have been other channel objects used previously by this client, and dropped by the server at some point, we need to take care to resume the streaming of rounds with the correct round (to avoid losing or duplicating rounds). Since *s.maxround*[this] tells us the latest committed round on the server, we can ensure this by first removing confirmed rounds from the *pending* queue (using *adjust_pending_queue*), and then resending any rounds remaining in the *pending* queue.

The *receive* transition straightforwardly moves a packet from the serverstream into the receive buffer. The *drop_connection* transition models loss of connection at the client side. It

```
class StreamingClient {
  known: State := emptystate; // known prefix
  pending: Round * := []; // sent, but unconfirmed rounds
  round: ℕ := 0; // counts submitted rounds
  transactionbuf: Delta := emptydelta;
  tbuf_empty : boolean := true;
  channel: Channel := null;
  pushbuf: Delta := emptydelta; // updates that were pushed, but not sent yet
  rds_in_pushbuf: ℕ := 0; // counts the number of rounds in the pushbuffer
  // client interface transitions
  read(r: Read) : Value { return read(r, curstate()); }
  update(u: Update) {
    transactionbuf := transactionbuf.append(u);
    tbuf_empty := false;
  }
  confirmed() : boolean {
    return pending = [] && rds_in_pushbuf = 0 && tbuf_empty;
  }
  push() {
    pushbuf := pushbuf.append(transactionbuf);
    transactionbuf := []; tbuf_empty := true;
    rds_in_pushbuf := rds_in_pushbuf + 1; round := round + 1;
  }
  pull() {
    while (channel != null && channel.receivebuffer.length != 0) {
      var s := channel.receivebuffer[0];
      channel.receivebuffer := channel.receivebuffer[1..]
      if (channel.established) // not the first packet received on this channel
        assert(s instanceof GSSegment);
        known := known.append(s);
        adjust_pending_queue(s.maxround[this]);
      } else {
        channel.established := true;
        assert(s instanceof GSPrefix); // first packet contains latest server state
        known := s.state; // replace known prefix
        // resume sending rounds (remove confirmed, resend unconfirmed)
        adjust_pending_queue(s.maxround[this]);
        channel.clientstream := channel.clientstream · pending;
      }
    }
  }
// continued in next figure
```

Figure 5 States and transitions of the Streaming Client (part 1 of 2).

```
// class Client (continued)
  // auxiliary functions
  function curstate(): State {
    return apply(known, deltas(pending) · pushbuf · transactionbuf);
  }
  function deltas(seq: Round *): Delta { return seq[0].delta · · · seq[seq.length - 1].delta; }
  procedure adjust_pending_queue(upto: ℕ) {
    while (upto >= pending[0].round) { pending := pending[1..]; }
  }
  // network transitions (nondeterministic)
  receive() {
    requires channel != null && channel.serverstream.length > 0;
    channel.receivebuffer := channel.receivebuffer · channel.serverstream[0];
    channel.serverstream := channel.serverstream[1..];
  }
  drop_connection() { channel := null; }
  send_connection_request() {
    requires channel == null;
    channel := new Channel(this);
  }
  send() {
    requires channel != null && channel.established && rds_in_pushbuf > 0;
    var r := new Round { origin = this, round = round - 1, delta = pushbuf };
    pending := pending · r;
    channel.clientstream := channel.clientstream · r;
    pushbuf := [];
    rds_in_pushbuf := 0;
  }
}
```

Figure 6 States and transitions of the Streaming Client (part 2 of 2).

removes the channel object from the client - but not from the server, who may still send and receive packets on the channel until it in turn drops the connection. The *send* transition requires that there is an established channel (this is is important to handle channel recovery correctly, as explained earlier). It sends one or more rounds (stored in *pushbuf*) as a single cumulative round with the latest pushed round number, and appends it to the *pending* queue. Note that in practice, we found it sensible to add additional preconditions on *send*, to limit the number of rounds in the pending buffer, and to avoid overflowing buffers in the network layer.

6.5 Refinement Proof

We now prove that the streaming client-server protocol (Fig. 3,4,5,6) is a correct implementation, or a *refinement*, of the transactional GSP protocol (Fig. 2). This means that programmers need not worry about the intricacies of the former, but can safely assume that they are writing code for the latter: in particular, channel and server failures remain hidden beneath the protocol abstraction.

We define the set T_E of interface events to contain all expressions

$$\mathit{read}(c,r)(v) \mid \mathit{update}(c,u) \mid \mathit{confirmed}(c)(v) \mid \mathit{push}(c) \mid \mathit{pull}(c)$$

These events represent calls by the client program happening on some client c, and possibly returning a value v. The read and update events take a read operation r or an update operation u as a parameter.

We can now define a *trace* to be a finite or infinite sequence of interface events, and say a protocol *Impl* refines a protocol *Spec* if all traces of *Impl* are also traces of *Spec*. Since the events in the traces capture all interactions between the client program and the storage subsystem, refinement in this sense implies that if we run client programs on protocol *Impl*, they cannot detect a difference, i.e. all observable outcomes are consistent with running on protocol *Spec*.

For our proof, we use standard refinement proof methodology. We formalize a protocol as a labeled transition system $(\Sigma, \sigma^i, T, \delta)$ where Σ is a set of states, $\sigma^i \in \Sigma$ is the initial state, T is a set of transition labels, and $\delta \subset \Sigma \times T \times \Sigma$ is a transition relation. We write $\langle \sigma, t, \sigma' \rangle$ to represent an element of δ, that is, a transition with label t from state σ to state σ', and write $\langle \sigma_0, t_1, \sigma_1, t_2, \ldots, t_n, \sigma_n \rangle$ for a sequence of transitions with labels $t_1, \ldots, t_n$ (note that for $n = 0$, this is an empty transition sequence containing just a single state $\langle t_1 \rangle$).

We define transition systems for the implementation and specification to be $(\Sigma_I, \sigma_I^i, T_E \cup T_I, \delta_I)$ and $(\Sigma_S, \sigma_S^i, T_E \cup T_S, \delta_S)$, respectively, where the sets T_I, T_E, T_S represent categories of transitions, as follows.

We distinguish between *external* transitions T_E, which are exactly the interface events we have already defined above, and *internal* transitions T_S and T_I of the specification and implementation, respectively. Internal transitions usually represent nondeterministic events such as sending, receiving, or processing of messages that are not visible to the client program.

We define the set T_S of internal transitions of the specification to contain all expressions

$$\mathit{onreceive}(c,r) \mid \mathit{process}(r)$$

where c ranges over clients and r ranges over rounds (the rounds data structure). The *onreceive* transition is the handler for receiving an RTOB message in Fig. 2. The *process* transition represents the RTOB commit, i.e. the moment where a round becomes ordered into the global sequence. It is not explicitly listed in Fig. 2, since the RTOB is described

abstractly there.

Naturally, the implementation has many more internal transitions than the specification, since it has more "moving parts". We define the set T_I of internal transitions of the implementation as

$$\begin{aligned} &\mathit{receive}(c) \mid \mathit{send}(c) \mid \mathit{processbatch}() \mid \mathit{crash\_and\_recover}() \\ &\mid \mathit{client\_drop\_connection}(c) \mid \mathit{server\_drop\_connection}(ch) \\ &\mid \mathit{accept\_connection}(ch) \mid \mathit{send\_connection\_request}(c) \end{aligned}$$

where c ranges over clients and ch ranges over channel objects. All of these correspond to procedures with the same name in the code (Fig. 4,5,6).

We would like to emphasize that although we need to carefully consider all of these internal implementation transitions when proving refinement, the end result is that the programmer can be blissfully unaware of them.

6.5.1 Proof structure

To prove refinement, we construct an extended simulation relation

$R \subseteq \Sigma_I \times \Sigma_S \times \Sigma_A$,

where Σ_I is the implementation state, Σ_S is the specification state, and Σ_A is auxiliary state.

In our case, Σ_A is only needed to record history variables (we do not need prophecy variables as introduced in [1]). This means that Σ_A is updated according to some update function u_A that defines an auxiliary state $u_A((\sigma_I, \sigma_S, \sigma_A), t, \sigma_S)$ for each triple $(\sigma_I, \sigma_S, \sigma_A)$ and each transition $\langle \sigma_I, t, \sigma_I' \rangle$.

The following conditions capture the requirements on R to be a simulation. It is easy to see that if such an R exists, it implies trace refinement as desired.

1. R contains the initial state $\sigma^i = (\sigma_I^i, \sigma_S^i, \sigma_A^i)$
2. for all tuples $(\sigma_I, \sigma_S, \sigma_A) \in R$ and implementation transitions $\langle \sigma_I, t, \sigma_I' \rangle$, there exists a specification transition sequence $\langle \sigma = \sigma_S^0, t_1, \sigma_S^1, t_2, \ldots, t_n, \sigma_S^n \rangle$ (where $n \geq 0$) satisfying the following conditions:
   a. If $t \in T_I$ (that is, t is an externally unobservable transition of the implementation), then all the labels $t_1, \ldots, t_n$ must be in T_S (i.e. must be internal transitions of the specification).
   b. If $t \in T_E$ (that is, t is an externally observable transition of the implementation), then there must exist an i such that $t = t_i$ (one specification transition must match), and $t_j \in T_S$ for $j \neq i$ (the other specification transitions must be externally unobservable).
   c. When we move all three state components forward, that is, when we (1) apply the implementation transition to the implementation state, (2) apply the specification transition sequence to the specification state, and (3) update the auxiliary state, we stay in the relation: $(\sigma_I', \sigma_S^n, u_A((\sigma_I, \sigma_S, \sigma_A), t, \sigma_S)) \in R$.

To define this relation and prove the obligations, we construct a "combined transition system" in appendix A whose state is $\Sigma = \Sigma_I \times \Sigma_S \times \Sigma_A$, and which has the transitions described in (2.c) above. The simulation relation R captures the relation between rounds, clients, global sequences, and channels in the specification and the implementation. We can think of R as an *invariant* of this combined transition system, rather than a simulation relation. This helps to organize the proof in a familiar way, by stating invariants and transitions, and proving preservation.

6.6 Further optimizations

In our prototype we implemented a few additional optimizations left out here for simplicity. They include:

- The server caches recent deltas. When clients reconnect, and the server still has the relevant deltas in its cache, the server sends only the deltas needed instead of the whole state.
- The server, when sending segments to a client c, includes not the whole *maxround*, but only *maxround*[*c*].
- As written, reads are potentially inefficient, thus some optimizations may be required. For example, in our implementation of the cloud types model, we store updates of fields inside objects representing the fields, and we cache the result of expensive reads, such as table enumerations.

The implementation presented here uses a single server, which is appropriate for modest read/write loads. The server can be easily made reliable, even on unreliable cloud compute infrastructure, by using reliable cloud storage to store the persistent state. Devising implementations that scale to heavier loads, while certainly possible, is beyond the scope of this paper.

7 Implementation in TouchDevelop

We have realized the ideas presented in this paper and their implementation as an extension of the web-based IDE and runtime system called TouchDevelop [30]. We implemented the streaming model using a Azure cloud service backed by Azure table storage (for the persistent state). Clients are written in JavaScript, run in webbrowsers, persist the local data in HTML5 local storage (and thus remain available offline), and connect to the service using websockets.

Rather than a basic key-value store data model, the TouchDevelop language supports full *cloud types* [5, 14] which include tables, indices, and records. We describe the cloud types data model and prove optimal reduction in appendix B.

For illustration, we give a few examples of TouchDevelop apps that use cloud types below: feel free to run them, inspect and edit their code, and create your own variations! The first two examples below have been contributed by our user community. In all of these examples, the use of cloud types is very simple, with the exception of the Cloud Game Selector which involves tricky synchronization and a flush operation.

- **Relatd** [sic] (http://tdev.ly/ruef) Lets users enter their qualities (either from a list, or freely entered) and finds and displays other users that share them.
- **Chatter Box** (http://tdev.ly/spji) A chat application.
- **TouchDevelop Jr.** (http://tdev.ly/vkrpa) Program a tiny robot using a simple language, then share your scripts with other users.
- **Instant Poll** (http://tdev.ly/nggfa) An app for quickly polling an audience and displaying the responses as a grid of colors.
- **Expense Recorder** (http://tdev.ly/nvoha) Allows easy recording of expenses in a table.
- **Contest Voting** (http://tdev.ly/etww) Used to determine the winner of the "Touch of Summer" coding contest.
- **Cloud List** (http://tdev.ly/blqz) A general-purpose list that can be concurrently edited.
- **Cloud Game Selector** (http://tdev.ly/nvjh) A library for matching multiple players to play games together.

- **Cloud Paper Scissors** (http://tdev.ly/sxjua) A simple rock/paper/scissors game that uses the cloud game selector library.

Other researchers have also experimented with refactoring non-cloud data in TD scripts into cloud types. A formative study shows that refactoring is applicable, relevant, and saves human effort [18].

8 Related Work

Eventual Consistency is motivated by the impossibility of achieving strong consistency, availability, and partition tolerance at the same time, as stated by the CAP theorem [17]. EC across the literature uses a variety of techniques to propagate updates (e.g. general causally-ordered broadcast [24, 26]). For a general high-level comparison of eventual consistency notions appearing in the literature, see [3, 6, 8].

Bayou's weakly consistent replication [29] follows a similar overall system design. However, it does not articulate an abstract reference model like GSP, or a data model. Conflicts are not resolved simply by ordering updates, but require explicit merge functions provided by the user.

As explained earlier, our Global Sequence Protocol (GSP) is an adaptation of a reliable total order broadcast [12, 16]. However, we go beyond prior work on broadcast, as we articulate the concept of a data model, describe how to reduce updates, and discuss optimality of reduction.

A version of core GSP supporting arbitrary replicated data types appears in [8], but without synchronization, transactions, or a robust implementation.

As explained in Section 4.3, GSP is similar, but not equivalent to the TSO memory model. In particular, GSP allows data to be read and updated offline without requiring communication. We could call GSP "the TSO for distributed systems". Neither TSO, nor any other memory consistency model we know of, allows arbitrary data models like GSP, or provides transactional synchronization via *push* and *pull*.

Just like our work, replicated data types and in particular CRDTs [7, 24, 24] provide optimized distributed protocols for certain data types. However, CRDTs are not easy to customize and compose, since the consistency protocol is not cleanly separated from the data model as in GSP, but specialized for a particular, fixed data type.

As explained earlier, the original work on cloud types [5, 14], while providing a comprehensive, composable way to define replicated structured data, falls short of providing either a simple reference model or a robust implementation.

The Jupiter system [22] has a similar system structure (client-server with bidirectional streaming) as our streaming model. However, it uses an *operational transformation* (OT) algorithm to transform conflicting updates with respect to each other, instead of simply ordering updates sequentially as in GSP.

The OT approach [13, 25–27] provides an interesting and powerful (but arguably also somewhat confusing and error-prone [20]) conflict resolution mechanism that has seen successful application and even industrial adoption for collaborative editing applications. However, it comes at the expense of scalability. OT transformations grow quadratically with the number of concurrent updates, and prevent extended offline operation since that requires storing and later processing of update sequences.

In our situation, all the example applications used structured data that was entirely expressible using cloud types, which by design avoid the need for OT. Thus, we were not

inclined to pay the price for providing operational transformations as part of our data model (but may revise this choice in the future).

9 Conclusion

We have motivated, defined, and explained the global sequence protocol (GSP), a simple operational reference model for replicated, eventually consistent shared data. We then showed how to implement it, presenting a robust streaming implementation that can hide network, server, and client failures, and reduces update sequences, while conforming to the GSP reference model.

Both GSP and the streaming implementation are parameterized by an abstract data model, and thus apply to a wide range of data types from simple key-value stores to the full cloud types model.

We hope that our work provides a path for simpler development of distributed applications on mobile devices. In the future, we would like to further investigate the layering of data abstractions and how to best support user-defined data models. We are also considering more work on the refinement proof, such as obtaining a mechanically verified proof, and/or adding fairness and liveness. Finally, we are working on extending GSP to partial replication scenarios.

10 Acknowledgments

We would like to thank Alexey Gotsman for interesting discussions on the GSP consistency model, and Adam Morrison for pointing out a mistake we made in our reasoning when comparing GSP to TSO in an earlier version of this paper.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82 (2), 1991.

[2] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *International Conference on Very Large Databases (VLDB)*, 2014.

[3] P. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 923–928. ACM, 2013.

[4] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC'00*, 2000.

[5] S. Burckhardt, M. Fähndrich, D. Leijen, and B. Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 7313 of *LNCS*, pages 283–307. Springer, 2012a.

[6] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft, 2013.

[7] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Principles of Programming Languages (POPL)*, 2014a.

[8] Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1 (1-2): 1–150, 2014.

[9] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Mooly Sagiv. Eventually Consistent Transactions. In *European Symposium on Programming (ESOP), (extended version available as Microsoft Tech Report MSR-TR-2011-117), LNCS*, volume 7211, pages 64–83, 2012b.

[10] Sebastian Burckhardt, Daan Leijen, and Manuel Fähndrich. Cloud types: Robust abstractions for replicated shared state. Technical Report MSR-TR-2014-43, Microsoft Research, March 2014b. URL http://research.microsoft.com/apps/pubs/default.aspx?id=211340.

[11] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A robust abstraction for replicated shared state. In *ECOOP'15*, July 2015.

[12] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[13] M. Cart and J. Ferrie. Asynchronous reconciliation based on operational transformation for p2p collaborative environments. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007)*, pages 127–138, Nov 2007.

[14] Tim Coppieters, Laure Philips, Wolfgang De Meuter, and Tom Van Cutsem. An open implementation of cloud types for the web. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 2:1–2:2, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2716-9. doi:10.1145/2596631.2596640.

[15] 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 *Symposium on Operating Systems Principles*, pages 205–220, 2007. doi:10.1145/1294261.1294281.

[16] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36 (4): 372–421, December 2004.

[17] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33: 51–59, June 2002. ISSN 0163-5700. doi:http://doi.acm.org/10.1145/564585.564601.

[18] M. Hilton, A. Christi, D. Dig, M. Moskal, S. Burckhardt, and N. Tillmann. Refactoring local to cloud data types for mobile apps. In *MobileSoft '14*. ACM, 2014.

[19] IEEE Computer. CAP retrospective edition. *IEEE Computer*, 45 (2), 2012.

[20] A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351: 167–183, 2006. doi:10.1016/j.tcs.2005.09.066.

[21] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In _SOSP'11_, 2011.

[22] D. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *User interface and software technology (UIST)*, 1995.

[23] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report Rapport de recherche 7506, INRIA, 2011a.

[24] Mark Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Grenoble, France, October 2011b.

[25] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Conference on Supporting Group Work*, GROUP '97, pages 435–445. ACM, 1997.

[26] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work*, CSCW '98, pages 59–68. ACM, 1998.

[27] D. Sun and C. Sun. Operation context and context-based operational transformation. In *Conference on Computer Supported Cooperative Work*, CSCW '06, pages 279–288. ACM, 2006.

[28] D. Terry, A. Demers, K. Petersen, M. Spreitzer M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[29] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29: 172–182, December 1995. ISSN 0163-5980. doi:http://doi.acm.org/10.1145/224057.224070.

[30] N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *ONWARD '11 at SPLASH (also available as Microsoft TechReport MSR-TR-2011-49)*, 2011.

[31] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

A Refinement Proof

As outlined in section 6.5.1, we now prove refinement by constructing a combined transition system, whose invariant is the simulation relation. We first describe the combined state, and then show the combined transitions. We then formulate the invariants and explain how they are preserved. Since this is a manual rather than a mechanical proof, we took liberty to discharge the individual obligations (i.e. how some specific invariant is preserved by some transition) using informal language in some cases, rather than heavy notation. However, we were careful to structure the proof in such a way as to make it possible to track proof obligations and invariants.

A.1 The Combined State

We define the state $\Sigma = \Sigma_I \times \Sigma_S \times \Sigma_A$ of the combined transition system as shown in Fig. 9. Note that the components are separated at the field level, not at the global level. All components belonging to the implementation, specification, and auxiliary state are prefixed with I, S, and A, respectively.

For example, for some state $\sigma \in \Sigma$, the expression $\sigma.Client[c].I\_round$ refers to the *round* field that the implementation stores in each client, while $\sigma.Client[c].S\_round$ is the corresponding specification field.

A.1.1 Completing the Specification Model

The specification is built on top of RTOB, which is only informally described in the high-level description of the algorithm. To prove refinement, we had to complete the specification model, which we did as follows:

- in each client, *S_sent* stores the sequence of all the rounds sent by this client to the RTOB. It is initially empty, and updated in the $push(c)$ transition, by appending the sent round.
- the global variable *S_gs* records all the rounds processed by the RTOB. It is updated by a new internal transition $process(r)$, which appends a round r satisfying that r is the first round in $Client[r.origin].S\_sent$ that is not already in *S_gs*.
- in each client, *S_delivered* stores the sequence of rounds delivered to this client by the RTOB. We model the *onReceive* handler using a transition $receive(c)$ which delivers the next round to client c, that is, the next round r in *S_gs* that is not already in *S_delivered*, and appends r to *S_delivered*.

A.1.2 Auxiliary State

We record the following auxiliary state:

In each *IRound* object, we record the multiplicity of this round in the field *A_multiplicity*. If this is larger than 1, it means this implementation round represents several specification rounds. The field is immutable - it is set only when constructing the *IRound* in the $send(c)$ transition, at which point we set it to $Client[c].I\_rds\_in\_pushbuf$.

In each client, *A_sent* records all the implementation rounds that were sent. It is initially empty, and updated in the $send(c)$ transition, where the newly constructed *IRound* is appended.

In each channel, we record *A_cs_next_in* and *A_cs_next_out* numbers to track the round sequence numbers that are streaming from client to server. Initially, they are both 0.

```
struct State {
  I_serverstate: GSPrefix
  I_connections: (Client ⇀ Channel)
  S_gs: Round *
  Client: array [Client] of {
    I_known: State              S_known : Round *
    I_pending: IRound *         S_pending: Round *
    I_round: ℕ                  S_round : ℕ
    I_transactionbuf: Delta     S_transactionbuf : Update *
    I_channel: Channel          S_receivebuffer: Round *
    I_rds_in_pushbuf: ℕ         S_sent : Round *
    I_pushbuf: Delta            S_delivered : Round *
    I_tbuf_empty: boolean       A_sent: IRound *
  }
  Channel: array [Channel] of {
    I_client: Client            A_cs_next_in: ℕ
    I_accepted: boolean         A_cs_next_out: ℕ
    I_established: boolean      A_ss_next_in: ℕ u {-1}
    I_clientstream: IRound *    A_rb_next_out: ℕ
    I_serverstream: (GSPrefix | GSSegment)
    I_receivebuffer: (GSPrefix | GSSegment)
  }
}
struct IRound {                 struct GSSegment {
  I_origin: Client   I_delta: Delta
  I_number: ℕ        I_maxround: (Client ⇀ ℕ)
  I_delta: Delta     A_from : ℕ
  A_multiplicity: ℕ  A_to: ℕ
}                               }
struct GSPrefix {
  I_state: State
  I_maxround: (Client ⇀ ℕ)
  A_to : ℕ
}
```

Figure 7 Representation of Implementation, Specification, and Auxiliary State

In a pull transition that establishes a channel, *A_cs_next_in* is set to *round - rds_in_pushbuf*, and *A_cs_next_out* is set to the same if the channel is empty, or to the oldest round represented in the stream (respecting multiplicities).

In each channel, we record *A_ss_next_in* and *A_rb_next_out* numbers to track the round sequence numbers that are represented by the objects in *receivebuffer·serverstream* (we consider this concatenation to represent the stream). Initially, they are both 0.

In each *GSPrefix* and *GSSegment*, we record exactly what prefix or segment of the global sequence this object corresponds to, by means of the fields *A_from* and *A_to* (*A_from* is inclusive and *A_to* is exclusive).

```
class GSPrefix { // represents a prefix of the global update sequence
  I_state: State := initialstate;
  I_maxround: (Client ⇀ ℕ) := {};
  A_to: ℕ := 0;
  method apply(s: GSSegment) {
     assert(s.A_from = A_to);
     foreach((c,r) in s.I_maxround) { I_maxround[c] := r; }
     I_state := apply(I_state, s.I_delta);
     A_to := s.A_to;
} }
class GSSegment { // represents an interval of the global update sequence
  I_delta: Delta := emptydelta;
  I_maxround: (Client ⇀ ℕ) := {};
  A_from: ℕ;
  A_to: ℕ;
  constructor GSSegment(from: ℕ) {
    A_to := A_from := from;
  }
  method append(m: IRound): void {
     I_maxround[m.I_origin] := m.I_round;
     I_delta := reduce(I_delta · m.I_delta);
     A_to := A_to + m.I_multiplicity;
} }
```

A.2 Invariants

We now state the invariant of the combined transition system (which constitutes the simulation relation). The global invariant is a conjunction of many individual conditions (also called "the invariants"), each of which has a label. It is quite easy to check for all of the invariants stated in this section that they hold on the initial state (generally, everything works out fine as long as notations are interpreted correctly for empty sequences etc.), so we do not explain that aspect in detail.

The nontrivial part is to show that the invariants are preserved by the combined transitions, which requires some reasoning, and is done in section A.3 (organized by transition).

Before stating the invariants, we need extra notation that helps us formalize how implementation state corresponds to specification state. To this end, we further overload the representation relation $\lhd$ which was defined in section 6.1 on *Delta* × *Update* * and *State* × *Update* *, so it tracks how the implementation represents multiple rounds within a single round *IRound*:

- On *IRound* × *Round**, let $\lhd$ be the relation such that $m \lhd r_1 \cdots r_n$ iff (1) *m.origin*= r_i.*origin* for all i, (2) *m.number* - (n-i) = r_i.*number* for all i, (3) *m.delta*$\lhd r_1$.*updates*$\cdots r_n$.*updates*, and (4) *m.multiplicity*= n. Note that we allow $n = 0$ as a borderline case, interpreting notations as usual.
- On *IRound** × *Round**, let $\lhd$ be the relation such that $m_1 \cdots m_n \lhd a$ iff there exists a decomposition $a = a_1 \cdots a_n$ such that $m_i \lhd a_i$ for all i.

A.2.1 Round Invariants

For each structure r : *Round* appearing anywhere in the state:

- (r1) *Client*[*r.origin*].*S_sent*[*r.number*] = *r*

Condition (r1) means that all rounds that are stored anywhere are always 'a proper record', that is, match the corresponding round as recorded in *S_sent* by the respective client.

For each structure m : *IRound* anywhere in the state:

- (m1) m $\lhd r_1 \cdots r_n$ where n = *m.A_multiplicity* and r_i = *Client*[*m.I_origin*].*S_sent*[*m.I_round*-(n - i)]
- (m2) m $\in$ *Client*[*m.I_origin*].*A_sent*

Condition (m1) means that the round represents the corresponding (multiple) rounds. Condition (m2) means it is a proper record, i.e. this was an actual round that was sent.

A.2.2 Client Invariants

On each client *Client*[c]:

- (c1) *I_known* $\lhd$ *updates*(*S_known*)
- (c2a) *I_pushbuf* $\lhd$ *updates*(*suffix*(*S_pending*, *I_rds_in_pushbuf*))
- (c2b) *I_pending* $\lhd$ *prefix*(*S_pending*, |*S_pending*| - *I_rds_in_pushbuf*)
- (c2c) *m.I_origin* = c for all $m \in$ *I_pending*
- (c3) *I_round* = *S_round*
- (c4a) *I_transactionbuf* $\lhd$ *S_transactionbuf*
- (c4b) *I_tbuf_empty* if and only if *S_transactionbuf* = []
- (c5) *S_known* is a prefix of *S_gs*
- (c6) *S_known*$|_c \cdot$*S_pending* = *S_sent*
- (c7) |*S_sent*| = *S_round* and *S_sent*[i].*number* = i for all i
- (c8) *r.origin* = c for all $r \in$ *S_sent*
- (c9) if *I_channel* != *null* then *I_channel.I_client* = c
- (c10) *S_delivered* = *S_known*
- (c11) *A_sent* $\lhd$ *updates*(*prefix*(*S_sent*, |*S_sent*| - *I_rds_in_pushbuf*))

(c1) says that the state object in *I_known* represents the sequence of all updates in *S_known*. (c2a) characterizes the role of the pushbuffer: what is in the pushbuffer represents a number of rounds (counted by *I_rds_in_pushbuf*) that are, from a specification viewpoint, considered "pushed" already and thus part of *S_pending*. (c2b),(c2c) specify the contents of the pending buffer. Note that (c2b) implicitly states that *I_rds_in_pushbuf* < |*S_pending*|. Condition (c3) means that rounds are counted synchronously in specification and implementation. Conditions (c4a,b) describe that the transactionbuffer of the implementation represents the transactionbuffer of the specification. Conditions (c5) and (c6) are general

invariants of the GSP specification model: (c5) says that the known prefix is a prefix of the global sequence, and (c6) says that the pending sequence contains exactly the sent updates that have not been included in *S_known* (for a sequence of rounds w and a client c, the expression $w\,|_c$ denotes the projection of w onto rounds by c, that is, the largest subsequence of w containing only rounds by c). (c7) states that *S_round* correctly counts the sent rounds and that they are all entered into *S_sent*. (c8) states that the origin of rounds in *S_sent* matches the client. (c9) says that if a client has a channel attached, this channel's client field matches. (c10) says that the delivered prefix is the same as the known prefix, which reflects that in our simulation, the spec receives in the pull transition.

A.2.3 Global Sequence Invariants

The global sequence is always an interleaving of some prefix of the client sequences:

- (g1) the global sequence *S_gs* is an interleaving of the sequences *Client*[c].*S_sent*[0..*maxround*(*S_gs*,c)]
- (g2) *I_serverstate.A_to* = |*S_gs*|

where *maxround*(*S_gs*,c) is defined to be the maximum round of client c appearing in *S_gs*, or -1 if no such round appears.

For each structure p : *GSPrefix* anywhere in the state:

- (p1) *I_state* $\lhd$ *S_gs*[0..*A_to* - 1]
- (p2) for each client, *I_maxround* matches maximum round by that client in *S_gs*[0..*I_state.A_to* - 1], or is undefined if there are none
- (p3) for all $c \in$ dom *I_maxround*, *I_maxround*(c) = m.*I_round* for some $m \in$ *Client*[c].*A_sent*

Condition (p1) means the state correctly represents the update sequences of the corresponding prefix of the global sequence. (p2) says that maxround captures, for each client, the maximal round contained in the global sequence. (p3) ensures that the client rounds respect multiplicities of rounds.

Equivalently, for each structure s : *GSSegment* anywhere in the state:

- (s1) *I_delta* $\lhd$ *S_gs*[*A_from*..*A_to* - 1]
- (s2) for each client, *I_maxround* matches maximum round by that client in *S_gs*[*A_from*..*A_to* - 1], or is undefined if there are none
- (s3) for all $c \in$ dom *I_maxround*, we have *I_maxround*(c) = m.*I_round* for some $m \in$ *Client*[c].*A_sent*

A.2.4 Channel Invariants

On each channel *Channel*[*ch*]:

- (io1) either *clientstream* = [] and *A_cs_next_in* = *A_cs_next_out*, or *clientstream* contains a sequence of rounds by client *I_client* such that (1) if m is the first round in the sequence, then *A_cs_next_out* = m.*I_round* - m.*A_multiplicity* + 1, and (2) if x,y are adjacent rounds in the sequence, then x.*I_round* = y.*I_round* - y.*A_multiplicity*, and (3) if y is the last round of the sequence, then y.*I_round* = *A_cs_next_in* - 1.
- (io2) either *I_receivebuffer* = *I_serverstream* = [] and *A_ss_next_in* = *A_rb_next_out*, or *receivebuffer*·*serverstream* contains (*GSPrefix* | *GSSegment*) * sequence such that (1) if x is the first object in the sequence, then either *A_rb_next_out* = 0 and x is a GSPrefix, or *A_rb_next_out* > 0 and x is a GSSegment satisfying x.*A_from* = *A_rb_next_out*, and (2) if x,y are adjacent objects in the sequence, then x.*A_to* = y.*A_from* (in particular, y can only be a GSSegment), and (3) if x is the last object of the sequence, then x.*A_to* = *A_ss_next_in*

These invariants capture the in-order guarantees of the stream and capture what has last been queued or dequeued. The following channel invariants are more temporal, i.e. can change with setup and failures. On each channel *Channel*[*ch*] for client $c = ch.I\_client$:

- (x1) if *I_accepted* and *I_established* and *Client*[*c*].*I_channel* = *ch*, then we have *A_rb_next_out* = | *Client*[*c*].*S*
- (x2) if *I_accepted* and not *I_established* and *Client*[*c*].*I_channel* = *ch*, then *I_receivebuffer*·*I_serverstream* contains as its first element a *GSPrefix* x such that $x.A\_to \geq |\, Client[c].S\_known \,|$, and such that $x.I\_maxround(c) = maxround(S\_gs, c)$.
- (x3) if not *I_established*, then *I_clientstream* = []
- (x4) if *I_established*, then *I_accepted*.
- (x5) if *I_established* and *I_connections*[*c*] = *ch*, then $A\_cs\_next\_out = maxround(S\_gs, c) + 1$.
- (x6) if *I_established* and *Client*[*c*].*I_channel* = *ch*, then *A_cs_next_in* = *Client*[*c*].*I_round* - *Client*[*c*].*I_rds_in_pushbuf*.
- (x7) if *I_accepted* and *I_connections*[*c*] = *ch*, then *A_ss_next_in* = *I_serverstate*.*A_to*.
- (x8) if not *I_accepted*, then *I_serverstream* = *I_receivebuffer* = [] and *ch* is not in *I_connections*.
- (x9) if *I_established*, then the sequences *I_serverstream* and *I_receivebuffer* contain only GSSegment objects.

(x1) says that if the channel is established and has not failed on the client side, then the next unprocessed segment in the receivebuffer starts with the next round in the global sequence after the known prefix of this client. (x2) says that after a server accepts the channel and before the client establishes it or drops it, the channel contains a state object representing a prefix of the global sequence that is at least as large as the known prefix stored in the client. (x3) simply expresses that the client does not start streaming updates to the server before the channel is established. (x4) says that a channel is established only after it is accepted. (x5) says that when a channel has been established and not failed at the server, then the next client round to deliver to the server has the correct sequence number.

A.3 The Combined Transitions

We now examine all transitions of the combined transition system in turn. For each transition of the implementation, we give a snippet of code describing the combined transition, which (1) executes the implementation transition on the implementation state σ_I, (2) executes zero or more specification transitions on the specification state (subject to the conditions explained in 6.5.1), and (3) updates the auxiliary state. Note that when reading auxiliary state, the result may not influence updates to the implementation state, but it can influence updates to the auxiliary state and the specification state.

For each transition, we check that it preserves all relevant invariants (by relevant, we mean that the invariant contains at least one state variable that is modified by this transition), and that it does not violate any implicit or explicit assertions (e.g. index ranges, null pointers, assert instructions).

A.3.1 client read

The read transition performs a read both in the implementation and the specification, and we assert that the results match.

```
transition read(r: Read) : Value {
  var visstate := apply(I_known, deltas(I_pending) · I_pushbuf · I_transactionbuf);
  var rval := read(r, visstate);
  assert rval = spec_read(r);
  return rval;
}
procedure spec_read(r: Read) : Value {
  return rvalue(r, updates(S_known) · updates(S_pending) · S_transactionbuf);
}
```

To prove that the assertion is indeed satisfied, use the invariants to see that:

- by (c1), $I\_known \lhd updates(S\_known)$
- by (c2b), $I\_pending[i].delta \lhd a_i$ for some update sequences $a_0, \ldots a_n$ (where $n = |I\_pending| - 1$) satisfying $a_0 \cdot a_n = updates(prefix(S\_pending, |S\_pending| - I\_rds\_in\_pushbuf))$
- by (c2a), $I\_pushbuf \lhd updates(suffix(S\_pending, I\_rds\_in\_pushbuf))$
- by (c4a), $I\_transactionbuf \lhd S\_transactionbuf$

This then implies that $visstate \lhd updates(S\_known) \cdot updates(S\_pending) \cdot S\_transactionbuf$, which implies (by the correctness of the state-and-delta implementation) that the values returned by *read* and by *rvalue* are the same.

The transition trivially preserves all invariants because no state is modified.

A.3.2 client confirmed

```
transition confirmed() {
  var rval := (I_pending = [] && I_rds_in_pushbuf = 0 && I_tbuf_empty);
  assert rval = spec_confirmed();
  return rval;
}
procedure spec_confirmed() : boolean {
  return S_pending = [] && S_transactionbuf = [];
}
```

To prove that the assertion is indeed satisfied, consider that *rval* is true iff *spec_confirmed* returns true:

- if *rval* is true, then $I\_pending = []$ implies $|S\_pending| = I\_rds\_in\_pushbuf = 0$ by (c2b), and I_tbuf_empty implies $|S\_transactionbuf| = []$ by (c4b), thus *spec_confirmed* returns true as well.
- conversely, if *spec_confirmed* returns true, then $|S\_pending| = 0$ implies $I\_rds\_in\_pushbuf = 0$ and $I\_pending = []$ by (c2b), and $S\_transactionbuf = []$ implies I_tbuf_empty by (c4b), thus *rval* is also true.

The transition trivially preserves all invariants because no state is modified.

A.3.3 client update

```
transition update(u: Update) {
  I_transactionbuf := I_transactionbuf.append(u);
  I_tbuf_empty := false;
  spec_update(u);
}
procedure spec_update(u : Update) {
  S_transactionbuf := S_transactionbuf · u;
}
```

Preserves relevant invariants:

- (c4a) because the same update is appended.
- (c4b) because it is established by the updates.

A.3.4 client push

```
transition push() {
  I_pushbuf := I_pushbuf.append(I_transactionbuf);
  I_transactionbuf := [];
  I_tbuf_empty := true;
  I_rds_in_pushbuf := I_rds_in_pushbuf + 1;
  I_round := I_round + 1;
  spec_push();
}
procedure spec_push() {
  var r := new Round{ origin=this, number=S_round++,
    updates=S_transactionbuf, Client[c].I_rds_in_pushbuf};
  S_sent := S_sent · r;
  S_pending := S_pending · r;
  S_transactionbuf := [];
}
```

Preserves relevant invariants:

- (r1) is established for r in *spec_push* thanks to (c7).
- (m1) cannot be broken by the update of *S_sent* in *spec_push* because it only appends.
- (c2 a,b) because *I_transactionbuf* is appended to *I_pushbuf* and *S_transactionbuf* is appended to *S_pending*, and *I_round*, *S_round*, *I_rds_in_pushbuf* are incremented, and because of (c4)
- (c2c) because origin field is set correctly
- (c3) because both sides are incremented
- (c4a,b) because all fields are reset to initial state
- (c6) because the same round is appended to *S_pending* and *S_sent*
- (c7) because one element is appended to *S_sent* and *S_round* is incremented
- (c8) because the origin field is set to the client
- (c11) because the examined prefix of *S_sent* remains the same
- (g1) because it cannot be broken by appending to *S_sent*
- (x6) because both *I_round* and *I_rds_in_pushbuf* are incremented

A.3.5 client pull

The pull transition processes all objects in the receivebuffer. The first object received on a channel is always a GSPrefix, and allows the client to determine where to resume the stream of updates if a previous connection was interrupted.

```
transition pull() {
  while (I_channel != null && I_channel.I_receivebuffer.length != 0) {
    var s := I_channel.I_receivebuffer[0];
    if (I_channel.I_established) // not the first packet received on this channel
        assert(s instanceof GSSegment);
        assert I_channel.A_rb_next_out := s.from;
        I_known := I_known.append(s);
        adjust_pending_queue(s.I_maxround[this]);
        for (int i = 0; i < s.A_to - s.A_from; i++)
          spec_receive();
    } else {
        I_channel.I_established := true;
        assert(s instanceof GSPrefix); // first packet contains latest server state
        assert I_channel.A_rb_next_out := 0
        I_known := s.I_state;
        adjust_pending_queue(s.I_maxround[this]);
        // resend
        I_channel.I_clientstream := I_channel.I_clientstream · I_pending;
        I_channel.A_cs_next_in := I_round - I_rds_in_pushbuf;
        I_channel.A_cs_next_out := I_pending = [] ? I_round - I_rds_in_pushbuf :
          I_pending[0].I_round - I_pending[0].A_multiplicity;
        // receive additional rounds in spec
          for (int i = |S_known|; i < s.A_to; i++)
    }
    I_channel.I_receivebuffer := I_channel.I_receivebuffer[1..]
    I_channel.A_rb_next_out := s.to;
  }
  spec_pull();
}
procedure adjust_pending_queue(upto: ℕ) {
  while (upto >= I_pending[0].I_round) { I_pending := I_pending[1..]; }
}
procedure spec_receive() {
  var r := S_gs[S_delivered]; // next undelivered
  S_delivered := S_delivered + 1;
  S_receivebuffer := S_receivebuffer · r;
}
procedure spec_pull() {
  foreach(var r in S_receivebuffer) {
    S_known := S_known · r;
    if (r.origin = this) {
      assert S_pending[0] == r;
      S_pending := S_pending[1..];
    }
  }
  S_receivebuffer := [];
}
```

For the purposes of proving invariant preservation, we split the pull transition into several

transitions, one per iteration of the while loop. Specifically, we replace the pull transition with the two pull transitions below (adding more behaviors to the implementation is o.k. if we can still prove it correct). Note that the *spec_pull* is now executed after each iteration, not just at the end. However, it is easy to see that this does not influence the result, since a single pull at the end has the same effect.

A.3.6 client pull-segment

```
transition pull_segment() {
  requires I_channel != null && I_channel.I_receivebuffer.length != 0
    && I_channel.I_established;
  var s := I_channel.I_receivebuffer[0];
  assert (s instanceof GSSegment && s.A_from = |S_known|);
  I_known := I_known.append(s);
  adjust_pending_queue(s.I_maxround[this]);
  for (int i = 0; i < s.A_to - s.A_from; i++)
    spec_receive();
  spec_pull();
  I_channel.I_receivebuffer := I_channel.I_receivebuffer[1..];
  I_channel.A_rb_next_out := s.to;
}
```

The important invariants in this case are (x1) and (x9), because they ensure that we are receiving a segment starting with the correct round, as stated in the first assertion. We apply this delta to the known state, then receive and pull the corresponding number of rounds in the specification.

In *spec_pull*, the assertion states that in the specification state, whatever round we remove from the pending queue matches what we append to the known sequence. This follows from (c6),(c7),(c8), and because at the time *spec_pull* is called, receivebuffer contains a prefix of *S_gs* that is at least as large as *S_known*.

A somewhat subtle aspect is that the *adjust_pending_queue* does remove exactly the acknowledged rounds from *I_pending*, because a single segment may acknowledge many rounds at once. The invariant (s3) is needed for this: it ensures that the *upto* parameter does *exactly* match a round in *A_sent*. Since the rounds in *I_pending* must represent a contiguous sequence of client rounds by (c2b),(c6),(c7) and since they must also contain only rounds in *A_sent*, this is indeed the case.

Preserves relevant invariants:

- (c5) because by (c10), calling *spec_receive* and then *spec_pull* preserves (c5).
- (c1) by (x1) and (io2), $s.A\_from = |S\_known|$, thus by (s1), $s.I\_delta \lhd S\_gs[|S\_known|..|S\_known'|\text{-}1]$. By (c5), we get $I\_known' \lhd S\_gs[0..|S\_known'|\text{-}1]$ and thus by (c5) applied to post-state, $I\_known' \lhd S\_known$.
- (c2a) because *suffix*(*S_pending*, |*S_pending*|-*I_rds_in_pushbuf*) cannot change, because as argued above, the confirmed rounds are in *A_sent*, thus by (c11) they do not include rounds in the pushbuffer.
- (c2b) as discussed above, the rounds removed from *I_pending* represent the corresponding rounds from S_pending.
- (c2c) because things are only removed from the pending queue
- (c6) follows from the assertion inside *spec_pull*
- (c10) because *spec_receive* is followed by *spec_pull*

- (io2) because *A_rb_next_out* is updated as needed.
- (x1) because it holds prior to the transition, and both $|S\_known|$ and *A_rb_next_out* are each increased by the same amount (*s.A_to* - *s.A_from*).
- (x2) because the precondition *I_established* rules out interference.
- (x8) because the precondition *I_established* plus (x4) rules out interference.
- (x9) because elements are only removed.

A.3.7 client pull-prefix

```
transition pull_prefix() {
  requires I_channel != null && I_channel.I_receivebuffer.length != 0
    && ! I_channel.I_established;
  var s := I_channel.I_receivebuffer[0];
  I_channel.established := true;
  assert (s instanceof GSPrefix);
  I_known := s.I_state;
  adjust_pending_queue(s.I_maxround[this]);
  // resend
  I_channel.I_clientstream := I_channel.I_clientstream · I_pending;
  I_channel.A_cs_next_in := I_round - I_rds_in_pushbuf;
  I_channel.A_cs_next_out := I_pending = [] ? (I_round - I_rds_in_pushbuf) :
    I_pending[0].I_round - I_pending[0].A_multiplicity;
  // receive additional rounds in spec
  for (int i = |S_known|; i < s.A_to; i++)
    spec_receive();
  spec_pull();
  I_channel.I_receivebuffer := I_channel.I_receivebuffer[1..];
  I_channel.A_rb_next_out := s.to;
}
```

The important invariant in this case is (x2), because it ensures that we are indeed receiving a GSPrefix as stated in the assertion, which represents a prefix of the global sequence that is (1) at least as long as the already known prefix $|S\_known|$, and (2) tells us exactly which round of this client is the latest to have committed.

As in the *pull-segment* transition, we reason that *adjust_pending_queue* does remove exactly the acknowledged rounds from *I_pending* (using (p3),(c2b),(c6),(c7)) which, along with (x2), establishes the following important property: (*) At the location of the comment \\ *resend*, either $I\_pending = []$, or the first element $m = I\_pending[0]$ satisfies $m.I\_round - m.A\_multiplicity = maxround(S\_gs,c)+1$.

In *spec_pull*, the assertion is satisfied because of (c6),(c7),(c8), and because at the time *spec_pull* is called, receivebuffer contains a prefix of *S_gs* that is at least as large as *S_known*.

Preserves relevant invariants:

- (c5) because by (c10), calling *spec_receive* and then *spec_pull* preserves (c5).
- (c1) because *I_known* is set to *s.I_state*, and thus represents $S\_gs[0..A\_to-1]$ by (p1)
- (c2a) because $suffix(S\_pending, |S\_pending| - I\_rds\_in\_pushbuf)$ cannot change, because as argued above, the confirmed rounds are in *A_sent*, thus by (c11) they do not include rounds in the pushbuffer.

- (c2b) discussed above, the rounds removed from *I_pending* represent the corresponding rounds from S_pending.
- (c2c) because things are only removed from the pending queue
- (c6) follows from the assertion inside *spec_pull*
- (c10) because *spec_receive* is followed by *spec_pull*
- (io1) because: by (c2b),(c6),(c7),(c2d) we know *I_pending* (at the end, or equivalently, at the location of *resend*) represents a consecutive sequence of rounds by *c*. By (x3), *I_clientstream* starts out empty, thus contains in the end what was *I_pending* at the location of *resend*. Therefore, and since we explicitly assign *A_cs_next_in* and *A_cs_next_out*, it satisfies (io1).
- (io2) because *A_rb_next_out* is updated as needed.
- (x1) having established (c1), we know
- (x2),(x3) vacuously satisfied because *I_channel.established* is set to true.
- (x4) since receivebuffer is not empty, and by (x8), the channel must be accepted.
- (x5) by property (*)
- (x6) because *A_cs_next_in* is directly assigned the expected value.
- (x8) because the precondition *I_established* plus (x4) rules out interference.
- (x9) by (io2), only the first element in the receivebuffer can be a GSPrefix object, and this element is removed - thus only GSSegments can remain.

The specification transitions are the sequence containing, in that order, (1) for a received *GSPrefix* object, a sequence of internal transitions $onreceive(c,Round(c,n_i,u_i))$ where n_i range from $|S\_known|+1$ to $s.to-1$, and $u_i = S\_gs[n\_i](2)$ for all received *GSSegment* objects s, a sequence of internal transitions $onreceive(c,Round(c,n_i,u_i))$ where n_i range from $s.from$ to $s.to-1$, and $u_i = S\_gs[n\_i]$, and (3) an external transition $pull(c)$. This means that in our simulation, rounds are received in the specification only right before they are pulled. This is necessary because in the implementation, the receivebuffer is not truly received until pulled (the receivebuffer can be lost when the channel is dropped).

Note that these transitions are valid because of the invariants we get to assume. For an established channel, (x1) along with (io2) shows that any *GSSegment* must contain the "right"" rounds, i.e. the next ones to be appended to the known prefix. For a non-established channel, (x2) shows that the channel starts with a *GSPrefix* of the global sequence, which must be equal to or larger than the known prefix. Thus, we can replace the current known prefix with the received one, and deliver any extra rounds to the specification.

Also note that if this transition establishes a channel, it validates (x5) because (x2) guarantees that the maxround(c) received in the GSPrefix is up-to-date with respect to the maximal round by c contained in *S_gs*.

A.3.8 client send_connection_request

```
transition send_connection_request() {
  requires I_channel == null;
  I_channel := new Channel(this);
}
```

Preserves relevant invariants:

- (c9) because client field is set accordingly
- (io1),(io2) because the initial field values satisfy it
- (x1),(x2) vacuously

- (x3) because the stream is initially empty
- (x4),(x5),(x6) vacuously

A.3.9 client drop_connection

transition *drop_connection*() { *I_channel* := *null*; }

Preserves relevant invariants:

- (x1),(x2),(x6) because it strictly weakens the condition

A.3.10 client send

```
transition send() {
  requires I_channel != null && I_channel.I_established && I_rds_in_pushbuf > 0;
  var m := new IRound { I_origin = this, I_round = I_round - 1, I_delta = I_pushbuf,
    A_multiplicity = I_rds_in_pushbuf};
  I_pending := I_pending · m;
  I_channel.I_clientstream = I_channel.I_clientstream · m;
  I_channel.A_cs_next_in := m.I_round;
  A_sent := A_sent · m;
  I_pushbuf := [];
  I_rds_in_pushbuf := 0;
}
```

Preserves relevant invariants:

- (m1) because it holds for the newly created *IRound* object m. Distinguish cases:
  a. *I_rds_in_pushbuf* = 0: Then $m.multiplicity = 0$ and by (c2a), *I_pushbuf* = *emptydelta*, therefore (m1) because $emptydelta \lhd []$ by definition.
  b. *I_rds_in_pushbuf* > 0: We have to find suitable $r_1, \ldots r_n$ as required by (m1). To this end, let $r_1 \cdots r_n = suffix(S\_pending, I\_rds\_in\_pushbuf)$. By (c6), the latter is equal to $suffix(S\_sent, I\_rds\_in\_pushbuf)$, and by (c7) this implies $r_i.number = I\_round - (n - i + 1)$. By (r1) and (c8) this implies $r_i = Client[m.origin].S\_sent[r_i.number]$, and using (c7) we get $r_i.number = I\_round - (n - i + 1)$ and therefore $r_i.number = m.I\_round - (n - i)$ as needed. Also, $m.A\_multiplicity = n$ because we set $m.A\_multiplicity = I\_rds\_in\_pushbuf$. Thus (m1) holds for m.
- (m2) because m is appended to *A_sent*.
- (c11) because *I_rds_in_pushbuf* is reduced by the number of rounds represented by m, and the round m represents the corresponding rounds in *S_sent*.
- (p3) because it cannot break by adding more elements to *A_sent*
- (s3) because it cannot break by adding more elements to *A_sent*
- (io1) because by (x6) we know *A_cs_next_in* is *I_round* - *I_rds_in_pushbuf*, thus appending m to the clientstream preserves (io1).
- (x3) precondition *I_established* rules out interference
- (x6) because *A_cs_next_in* is assigned *I_round* and *I_rds_in_pushbuf* is assigned 0.

A.3.11 client receive

```
transition receive() {
  requires I_channel != null && I_channel.I_serverstream.length > 0;
  I_channel.I_receivebuffer := I_channel.I_receivebuffer · channel.I_serverstream[0];
  I_channel.I_serverstream := I_channel.I_serverstream[1..];
}
```

Preserves relevant invariants:

- (io2) because it does not alter the sequence *receivebuffer* · *serverstream*.
- (x8) because the precondition *I_serverstream.length* > 0 rules out interference.
- (x9) because elements are simply moved from serverstream to receivebuffer.

A.3.12 server processbatch

The transition receives a nondeterministic number of *IRound* objects from each connected channel. For each received object m, it calls *spec_process* a number of times (equal to the multiplicity of m) to receive the corresponding objects.

```
transition processbatch() {
  var s := new GSSegment(I_state.A_to);
  foreach((c,ch) in I_connections)
    receive(s, ch, *);
  I_serverstate := I_serverstate.apply(s);
  foreach((c,ch) in I_connections) {
    ch.I_serverstream := ch.I_serverstream · s;
    ch.A_ss_next_in := s.A_to;
  }
}
procedure receive(s: GSSegment, ch: Channel, count: int) {
  requires count <= ch.I_clientstream.length;
  foreach(m in ch.I_clientstream[0..count]) {
    assert(m.I_client == ch.client);
    s.append(m);
    ch.A_cs_next_out := m.I_round + 1;
    for (int i = 0; i < m.A_multiplicity; i++)
      spec_process(m.origin);
  }
  ch.I_clientstream := ch.I_clientstream[count..];
}
procedure spec_process(c: Client) {
  requires maxround(S_gs,c) < Client[c].S_sent - 1;
  S_gs := S_gs · Client[c].S_sent[maxround(S_gs,c) + 1];
}
```

Note that when processing a round m taken from a channel *ch* by a client c, we know by (x5) and (m1) and (io1) that this round represents the next *m.A_multiplicity* rounds by client c to be added to the global sequence. Thus, by calling *spec_process*(c) that many times, we do the equivalent thing for the specification (and the precondition of that procedure is satisfied).

Preserves relevant invariants:

- (c5) because appending an element to *S_gs* cannot invalidate it.
- (g1) because of (c7),(c8) the appended round becomes the new *maxround*(*S_gs*,*c*).
- (g2) *s.A_to* is initially equal to *I_state.A_to*, gets added each *m.A_multiplicity* in *s.append*(*m*), and then becomes *I_state.A_to*. Thus it is increased by the same amount as $|S\_gs|$.
- (p1) each *m* appended to impl causes the corresponding rounds to be added to spec.
- (p2) is maintained by implementation of apply
- (p3) any entry is caused by some *m* taken from a channel, thus it follows from (m2)
- (s1) because by (g2), *S_gs* is appended the same rounds as *s* starting from index *I_state.A_to*
- (s2) is maintained by implementation of append
- (s3) any entry is caused by some *m* taken from a channel, thus it follows from (m2)
- (io1) because *A_cs_next_out* is updated correctly.
- (io2) because of (x5)
- (x2) if a channel satisfies (x2) it remains true because: we cannot be receiving anything from an unestablished channel by (x3).
- (x5) if we receive anything on a channel *ch* with *ch.I_client* = *c*, then by (x3) and (x4) it is both established and accepted, so by (x5) we know *cs_next_out* = *maxround*(*S_gs*,*c*) + 1. Thus by (io1), *m.I_round* = *cs_next_out* + *m.A_multiplicity* - 1, thus by calling *spec_process* a number of times equal to *m.A_multiplicity* (which adds the corresponding rounds of *c* to *S_gs*, by (c7),(c8)) preserves x5.
- (x7) directly established.
- (x8) because (*c*,*ch*) in *I_connections* rules out interference.
- (x9) because only segments are pushed.

A.3.13 server accept_connection

```
transition accept_connection(ch: Channel) {
  requires !ch.I_accepted && I_connections[ch.I_client] = null;
  ch.I_accepted := true;
  I_connections[ch.I_client] := ch;
  ch.I_serverstream := ch.I_serverstream · I_serverstate;
  ch.A_ss_next_in := I_serverstate.A_to;
}
```

Preserves relevant invariants:

- (io2) because (x3),(x4) imply that clientstream is empty before, it afterwards contains a single GSPrefix satisfying the conditions.
- (x1) vacuously because by (x4), cannot be established
- (x2) because by (x4), the channel starts out empty, thus contains *I_serverstate* in the end, which satisfies the first condition because of (c5),(g2),(p1) and the second condition because of (g2),(p1).
- (x4) because accepted is set to true
- (x5) because the precondition !*ch.I_accepted* implies !*I_established* by (x4) and thus (x5) is vacuously true for this channel.
- (x7) directly established.
- (x8) because *I_accepted* is set to true.
- (x9) because the precondition !*ch.I_accepted* implies !*I_established* by (x4), thus the invariant is not threatened.

A.3.14 server crash_and_recover, drop_connection

```
crash_and_recover() {
  I_connections := {};
}

drop_connection(c : Client) {
  I_connections[c] := null;
}
```

Preserves relevant invariants:

- (c9) since it is set back to null.
- (x5),(x7),(x8) because it strictly weakens the condition.

B Cloud Types and Optimal Reduction

This appendix summarizes results that appeared in an earlier technical report [10] on cloud types, adjusted to the current context.

B.1 Data Model for Cloud Types

In general, programmers choose data models appropriate for location, size, and lifetime of the data. For example, persistent data is often modeled using relational databases or scalable key-value stores, while memory-resident data is typically represented by variables, arrays, and records, sometimes using a garbage-collected heap.

Choosing the right data model for persistent, replicated, shared data is paramount to keep the complexity of conflict resolution and garbage collection under control. Replicated data types [7, 23, 24] encapsulate those challenges within simple abstract data types such as counters, sets, or lists. Cloud types [5] go one step further, allowing programmers to express interrelated data structures.

Cloud types are more powerful than a simple key-value store but only incorporate a subset of relational features in order to handle conflict resolution and consistency. Nonetheless, in our own experience, that model has been powerful enough to cover most cases we encountered. To illustrate cloud types in general, and how they integrate with the GSP model, we now walk through a series of examples, demonstrating common consistency errors (anti-patterns) and how they can be solved.

B.1.1 Bird Watch Example

Let's write a program that keeps track of bird sightings. To start, we just count the overall bird sightings using a global cloud number (*nr*):

```
cloud nr birdcount;

function sighting() {
  birdCount.set(birdCount.get() + 1)
}
```

In the example, we assume there is a UI that invokes the *sighting* function. Moreover, as discussed in Section 5, we assume that the outer event loop calls *yield* after each event is

handled. This ensures that all operations done in an event are part of one update transaction, and that individual operations are never interleaved with other distributed updates; i.e. a programmer can always reason sequentially over the (cloud) state within each event handler. Also, every cloud value comes with a *default value* and they never have to be initialized. For numbers, the default is 0.

Nevertheless, the example is still wrong as it fails to count bird sightings reliably. Suppose two clients both have a sighting at the same time and both increment the count to, say, 1. After yielding, both update transactions, set(1), are appended to the global log and the final value of the bird count will be just 1. The anti-pattern here is that updates to a cloud value must make sense even if some 'earlier' updates are not yet visible to the local client.

To address this issue, cloud types generally come with a richer set of operations than just set. In particular, the *nr* type has an *add* operation which works incrementally:

```
cloud nr birdCount;

function sighting() {
  birdCount.add( 1 )
}
```

In this case, a concurrent sighting appends two $add(1)$ update transactions to the log, resulting in a correct global count of sightings.

We now extend our application to keep track of both the name and count of each bird sighting. Our data model supports *cloud tables* to maintain rows of cloud values:

```
cloud table Birds {
  name : str;
  count: nr;
}

function sighting(name: string) {
  var bird = find(name);
  bird.count.add(1);
}

function find( name : string ) {
  var bird;
  foreach( bird in Birds ) {
    if (bird.name.get() == name)
      return bird;
  }
  bird = Birds.new();
  bird.name.set( name );
  return bird;
}
```

On a sighting, we first call the *find* function to see if there already exists an entry for the particular bird in the cloud table. If no such entry is found, we create a new row using the new function and return that instead. Just like before, we then invoke $add(1)$ to add reliably to the bird count.

Again though, there is a problem with our example. It is possible that two concurrent clients both create a new row for a new sighting of the same bird name because they cannot

not see each others' updates yet. The anti-pattern here is the test if (*not exist*) *create* which is generally a problem because the element may already exist but is not yet visible to the local client. For this situation, our data model provides *cloud indices* which are conceptually infinite key-value dictionaries where all entries are pre-initialized. In our case, the *Birds* cloud index is keyed by the bird name:

```
cloud index Birds[name : string] {
  count : nr;
}

function sighting( name : string ) {
  Birds[name].count.add(1);
}
```

Now the example works as expected. In particular, as shown in Figure 8, each update transaction will consist of the entire *Birds*[*name*].*count.add*(1) expression. The server atomically resolves the first *Birds*[*name*] part, potentially creating a fresh entry, and then perform the field operation *count.add*(1).

Note the contrast between cloud tables and indices: the cloud tables give the ability to create (globally) *fresh* and *ordered* rows addressed by unique identities, while cloud indices provide unordered records of fields addressed by one or more key values. Global cloud variables as in our first example are internally implemented using a designated cloud index without keys, thus with one entry containing all globals as fields.

Even though a cloud table was not the right data structure for tracking the bird count, it is useful in other cases. If the bird log were to also keep track of individual sightings of birds, a table would be the correct structure for keeping track of rows of sightings containing the bird name, the person doing the sighting, the place, date and time.

This concludes our birding example. Note how concise and robust the final example is. Even though this code offers full distributed operation with eventual consistency and robustness under disconnected operation, there is almost no noise: no special error handling code, retrying of transactions, checking of connections, special server code, etc. All of this is taken care of by the implementation of the GSP model. As we saw with the anti-patterns, we still need to carefully consider the implications of those issues that are intrinsic to distributed operation, but the GSP model gives the programmer a robust mental model to think about these.

B.1.2 The data model syntax

Formally, the complete syntax of our data model is defined in Figure 8. For our purposes, we define three basic cloud types: numbers, booleans, and strings. The first entry of each $\mathsf{Val}_{\mathsf{type}}$ declaration defines the default value for each type. Also, each type comes with a set of valid operations $\mathsf{Fop}_{\mathsf{type}}$ that can be performed on values of that type.

The *Uid* type is for unique identifiers that are used to identify table rows. We use a single *Rid* type to identify records of cloud values which can be either a table name indexed by a *uid* (as *tname*(*uid*)), or an index name indexed by keys.

Finally the various *Update* and *Read* operations are defined. Note how fields are indexed uniformly in updates and reads, using the syntax *rid.fname.ftype*. In particular, the field type is included. We do this because we want to avoid needing an explicit schema for the server data, but we do want to all operations to be type safe. We can allow for every field name to be indexed by its type because it incurs no performance or storage overhead. This

Set	Variable	Definition	Meaning
Val	v	$= n \mid b \mid s$	value
$\mathsf{Val}_{\mathsf{nr}}$	n	$= 0 \mid \ldots$	number
$\mathsf{Val}_{\mathsf{bool}}$	b	$=$ false $\mid$ true	boolean
$\mathsf{Val}_{\mathsf{str}}$	s	$=$ ""$\mid \ldots$	string
Uid	uid	$= \ldots$	unique identifier
Fname	$fname$	$= \ldots$	field name
Tname	$tname$	$= \ldots$	table name
Iname	$iname$	$= \ldots$	index name
Ftype	$ftype$	$=$ nr $\mid$ bool $\mid$ str	field types
$\mathsf{Fop}_{\mathsf{nr}}$	fop_{nr}	$=$ set$(n) \mid$ add(n)	number field updates
$\mathsf{Fop}_{\mathsf{str}}$	fop_{str}	$=$ set$(s) \mid$ setifempty(s)	string field updates
$\mathsf{Fop}_{\mathsf{bool}}$	fop_{bool}	$=$ set(b)	boolean field updates
Rid	rid	$=$	record identifier
		$\mid tname(uid)$	table row
		$\mid iname[key_1, \ldots, key_n]$	index entry
Key	key	$= uid \mid s \mid n \mid b$	index key
Update	upd	$=$	update operation
		$\mid$ clr	clear all data
		$\mid$ new$(uid, tname)$	create table row
		$\mid$ del uid	delete table row
		$\mid rid.fname.ftype.fop_{ftype}$	field update
Read	rd	$=$	read operation
		$\mid$ rows $tname$	enumerate rows
		$\mid$ fread $rid.fname.ftype$	field read

Figure 8 The syntax of the data model.

becomes apparent when we define the exact semantics of the data model in next section.

B.2 Semantics

We formally define the semantics of our data model by defining read and update operations on state objects (Fig. 9). State objects store the current state, represented by (1) the current (i.e. created and not deleted) row identifiers, which are stored in *rows*, separately for each table, and sorted by creation order, and (2) all non-default field values, which are stored in the map *fields*. State objects
implement three methods, *read* for performing read operations on the current state, *update* for performing updates, and *targets_deleted_data* which can check if an update is redundant (i.e. targets data that is already deleted and thus has no effect).

State objects are of minimal size, i.e. they do not hold on to irrelevant information: they do not store fields that have the default value, and when a row is deleted, its field values are also removed. In particular, the state object does not contain any tombstones. The only exception is the *used* field which does indeed store all previously used identifiers. However, this field is a *ghost* field, used for proof purposes only: it does not affect control flow (other than assertions) and is not present in the physical implementation.

We now discuss the state object implementation in Figure 9 in some more detail. The state contains two properties, namely *rows* and *fields*. The *rows* is used for tables and maps table names to an ordered sequence of *Uid*'s (and an empty sequence by default). The *fields* stores all the cloud values for both tables and indices, and maps any triple of a record identifier, field name, and field type ($Rid \times Fname \times Ftype$) to either undefined or a cloud value.

The *read* method defines how *Read* operations are handled. If it is a field read (*fread*(*rid*,*fname*,*ftype*)) the value is read from the *fields*. If the value is undefined, we return the default value of that cloud type. This is very important: we never store default values and this allows us to efficiently represent for example 'infinite' indices in bounded storage.

The *update* method defines *Update* operations. There are two assertions that require update sequences to be *well-formed*. In particular:

- **A1.** When creating a new row using new(*uid*,*tname*), the *uid* must be fresh in the sense that it has not been used before in the update sequence.
- **A2.** For any field update *update*(*rid*,*fname*,*ftype*,*fop*), the operation *fop* must be a valid operation for the field type *ftype*.

Generally, it is straightforward to ensure that clients can only generate well-formed update sequences.

In the *update* method, the *clr* operations simply resets the *rows* and *fields*. The new operation is interesting since it takes a unique identifier as an argument where it is asserted that this *uid* is indeed not used already. Having a *uid* as an argument allows each client to generate unique identifiers locally without synchronization with the server which is crucial for disconnected operation for example.

The *del*(*uid*) operation deletes a particular row in a table and the corresponding fields in that table. Note though that the expression *key.contains*(*uid*) deletes both fields in the table row (indexed by *tname*(*uid*)) *and* any fields that happen to be indexed by that *uid* of the form *iname*[...,*uid*,...].
The *update*(*rid*,*fname*,*ftype*,*fop*) operation is the most interesting and performs an update operation on a particular field. First there is a check that the particular *rid* does not refer to a record that has been deleted already (perhaps by some other concurrent client). If there

```
class State
{
  rows : Tname → Uid * = {};
  fields : Rid × Fname × Ftype ⇀ Val = {};
  function size() { return rows.count + fields.count; }
  method read(r : Read) {
    match(r) with {
      fread(rid,fname,ftype) → {
        var val = fields[(rid,fname,ftype)];
        return (val == undefined) ? defaultval(ftype) : val;
      }
      rows(tname) → return rows[tname];
  } }
  method update(u: Update) {
    match(u) with {
      clr() → rows = fields = {};
      new(uid,tname) → {
        assert(! used.contains(uid)); //A1
        rows[tname].append(uid);
      }
      del(uid) → {
        foreach (tn in rows.keys)
          if (rows[tn].contains(uid)) rows[tn].remove(uid);
        foreach (key in fields.keys)
          if (key.contains(uid)) fields.remove(key);
      }
      update(rid, fname, ftype, fop) → {
        assert(fop in Fop_tname); //A2
        if (exists uid in rid : ! rows.contains(uid))
          return; // update on nonexisting record is no-op
        var curval = read(rid,fname,ftype);
        var newval = match (fop) with {
          set(v) → v
          add(n) → curval + n
          setifempty(s) → (curval = "" ? s : curval)
        }
        if (newval = defaultval(ftype))
          fields.remove((rid,fname,ftype));
        else
          fields[(rid,fname,ftype)] = newval;
    } }
    foreach(uid in u)
      used.append(uid); //track uids to detect freshness violations
  }
  method targets_deleted_data(u : Update): boolean {
    return exists uid in u : ! rows.contains(uid);
  }
}
```

Figure 9 The semantics of the data model.

is any *uid* in the *rid*, either of the form *tname*(*uid*) or *iname*[...,*uid*,...], where the *uid* is not in the *rows*, then we return immediately as this update is now a no-op. This is important for the optimality of storage: if we allow the update to happen on a deleted entry, this may result in a non-default value which would take up storage space. After this check, we simply *read* the current value, apply the operation, and write back the new value. Again, if the new value happens to be the default value of that type, we remove the field to minimize storage requirements.

B.3 Optimal Delta Reduction

In any system that tracks sequences of updates, it is important to keep the length of such sequences under control. In particular, update sequences often exhibit redundancy; for example, if the same field is assigned a new value several times, it is sufficient to store and propagate the last update only. In this section, we show how to optimally reduce update sequences. This is not an easy problem, since it requires us to thoroughly remove deleted data without compromising the semantics. Many systems in practice take the easy route and use tombstones for deleted data, and are thus not optimal.

We formally express the log reduction concept by defining a reduction relation $w \rhd w'$ on update sequences; it captures whether we can replace an update sequence w with another sequence w' without any observable effect.

▶ **Definition 1.** Given an update sequence $w_1 \in \mathsf{Update}^*$ and an update sequence or undefined value $w_2 \in (\mathsf{Update}^* \cup \bot)$, we say that $w_1 \rhd w_2$ (read: w_1 may reduce to w_2) iff for all $r \in \mathsf{Read}$ and $a, b \in \mathsf{Update}^*$ such that $\mathsf{rvalue}(a \cdot w_1 \cdot b, r) \neq \bot$, we have $\mathsf{rvalue}(a \cdot w_1 \cdot b, r) = \mathsf{rvalue}(a \cdot w_2 \cdot b, r)$. For notational convenience, we use $w_2 \lhd w_1$ (read: w_2 may replace w_1) interchangeably with $w_1 \rhd w_2$.

For example, we can prove that deletion is idempotent, i.e. for any $i \in \mathsf{Uid}$, we have $\mathsf{del}\ i \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i$. As another example, it is easy to see that $\mathsf{new}(i,a) \cdot \mathsf{new}(i,a) \rhd \bot$ because $\mathsf{rvalue}(a \cdot \mathsf{new}(i,a) \cdot \mathsf{new}(i,a) \cdot b, r) = \bot$ for all r, a, b (because the update sequence is not well-formed).

Note that the reduction relation is reflexive ($w \rhd w$), transitive ($w_1 \rhd w_2 \wedge w_2 \rhd w_3 \Rightarrow w_1 \rhd w_3$), and congruent ($w_1 \rhd w_2 \Rightarrow \forall a, b : a w_1 b \rhd a w_2 b$). However, it is not symmetric, because the right-hand side may cause fewer well-formedness violations than the left-hand side. For example, although $\mathsf{new}(i,a) \cdot \mathsf{del}\ i \rhd []$ (we prove this below), the converse $[] \rhd \mathsf{new}(i,a) \cdot \mathsf{del}\ i$ is not true: $\mathsf{rvalue}(\mathsf{new}(i,a), \mathsf{rows}\ a) = i$ but $\mathsf{rvalue}(\mathsf{new}(i,a) \cdot \mathsf{new}(i,a) \cdot \mathsf{del}\ i, \mathsf{rows}\ a) = \bot$.

Although reduction presents a great opportunity for saving space and reducing network consumption, many such reductions are possible, and it is not immediately clear what reductions to apply, in what order, and to what effect. The following definition sheds light on what we expect from a good reduction function.

▶ **Definition 2.** Let *reduce* : *Update** $\rightharpoonup$ *Update** be a partial function on update sequences. For a subset $W \subseteq \mathsf{Update}^*$ of update sequences, we say

- *reduce* is a *correct reduction function* on W if, for all $w \in W$, we have $w \rhd \mathsf{reduce}(w)$.
- *reduce* is *optimal reduction function* on W if, for all $w_1 \in W$ and $w_1 \in \mathsf{Update}^*$ such that $w_1 \rhd w_2$, we have $|\mathsf{reduce}(w_1)| \leq |w_2|$.

Optimality implies that when clients operate offline, the delta grows only as much as needed to accommodate the data. For example, even if clients repeatedly update the same location, or create and then delete many objects, the delta stays the same size. We can understand this

phenomenon as follows: since the minimal number of updates needed to get from database state *last* to database state *current* is bounded by last.size() + current.size(), we know that the delta is never larger than the size of the snapshot of the database when last connected, plus the current size of the database.

In the remainder of this section, we describe our reduction implementation, and then prove that it is (1) correct on all update sequences, and (2) optimal for NDU-free sequences. NDU-free means that a sequence does not contain a pattern of new, followed by delete, followed by update:

▶ **Definition 3.** A sequence of updates $w \in \mathsf{Update}^*$ is called NDU-free if it does not contain a subsequence of the form $\mathsf{new}(i, a) \cdot \mathsf{del}\ i \cdot m$ where $i \in \mathsf{Uid}$ and $a \in \mathsf{Tname}$, and where $m \in \mathsf{Update}$ is an update that contains i.

Although this restriction to NDU-free sequences may appear to weaken the optimality statement, this is easily fixed by adding the following check to the update operation in clients:

```
update(u: Update) {
  if (! curstate().targets_deleted_data(u)) { transactionbuf := transactionbuf.append(u); }
  tbuf_empty := false;
}
```

▶ **Lemma 4.** *All update sequences in all states of all executions of the streaming model (adjusted by the above check) are NDU-free.*

▶ **Corollary 5.** *Our delta reduction is optimal in all executions of the streaming model.*

B.3.1 Delta Objects

Our reduction implementation consists of a *Delta* class that stores update sequences in reduced form and allows us to efficiently append and reduce updates individually. It has the following abstract signature:

```
type Delta = ...
const emptydelta : Delta
function append : Delta × Update ⇀ Delta
function seq : Delta → Update *
```

The implementation is shown in Fig. 10. Note that the append function is partial because some updates may cause an assertion violation (creating a row with a *uid* that is already in use triggers assertion A3, and using an operation that is of the wrong type triggers assertion A4). The last function above, *seq*, reads back the reduced sequence from the delta.

B.3.2 Correctness and Optimality

▶ **Theorem 6.** *The following reduction function is correct for all w, and is optimal for all NDU-free well-formed w:*

```
function reduce(w: Delta *): Delta * {
  var d = new Delta();
  foreach(u in w)
    d.append(u);
  return d.seq();
}
```

```
class Delta {
  cleared : boolean;
  deleted : Uid *;
  created : (Uid × Tname) *;
  updated : (Rid × Fname × Ftype) ⇀ Fop;
  Delta() {
    cleared := false; deleted := [];
    created := []; updated := {}; }
  function seq() : Update * {
    return (cleared ? [clr] : [])
           · deleted.map((uid) => del(uid))
           · created.map((uid,tname) => new(uid,tname))
           · updated.map(((r,f,t),o) => fupd(r,f,t,o));
  }
  method append(u : Update) {
    match(u) with {
      clr() → {
        deleted := created := [];
        updated := {}; cleared := true; }
      new(uid,tname) → {
        assert(uid does not appear in
          deleted, created, or updated); //A3
        created.append((uid,tname));
      }
      del(uid) → {
        if ( ! deleted.contains(uid)) {
          if (exists tname : created.contains(uid,tname))
            created.remove((uid,tname));
          else if ( ! cleared)
            deleted.add(uid);
          foreach (key in updated.keys)
            if (key.contains(uid))
              updated.remove(key);
      } }
      update(rid, fname, ftype, fop) → {
        assert(fop in Fop_tname);//A4
        if (fop = add(0) || fop = setifempty(0)
            || exists uid in deleted s.t. uid occurs in rid)
          return; // update has no effect
        var op := match (fop) with {
          set(v) → set(v); // last writer wins
          add(n)) → {
            match updated[rid,fname,ftype] with {
              undefined → add(n);
              add(m) → add(m+n);
              set(m) → set(m+n);
            } }
          setifempty(s)) {
            match updated[rid,fname,ftype] with {
              undefined → setifempty(s);
              set("") → set(s); // succeed
              set(v) → set(v); // fail
              setifempty(v) → setifempty(v); // fail
        } } }
        updated[rid,fname,ftype] :=
```

A complete proof is included in appendix B.4. Note that without NDU-freedom, *reduce* is not guaranteed to be optimal. For example,

$$\mathsf{new}(i,a)\cdot\mathsf{del}\ i\cdot\mathsf{del}\ i \quad \triangleright \quad \epsilon$$

but because *reduce* always proceeds from left to right, it does not produce the optimal result:

$$\begin{aligned}&\mathsf{reduce}(\mathsf{new}(i,a))\cdot\mathsf{del}\ i\cdot\mathsf{del}\ i\\ &=\ \mathsf{reduce}(\mathsf{reduce}(\mathsf{new}(i,a)\cdot\mathsf{del}\ i)\cdot\mathsf{del}\ i)\\ &=\ \mathsf{reduce}(\epsilon\cdot\mathsf{del}\ i)\\ &=\ \mathsf{del}\ i\ \neq\epsilon\end{aligned}$$

We now show that all delta sequences appearing in executions of the streaming model are NDU-free, which concludes our optimality proof.

B.3.2.1 Proof of NDU-freedom.

We need to examine the two places where the *Delta.append* method is called: (1) when processing an update u by the user code. In this case, the transition calls *targets_deleted_data*(u) and skips the update if it contains an update that contains a deleted uid. Thus *current.delta* is always NDU-free, and therefore all the deltas in all the rounds. Note that once a client deletes a row, it will forever appear deleted to this client; therefore, this check enforces that a delete of some row by some client can never be followed by an update by the same client that contains that same row. (2) When the server creates a new segment and adds multiple client rounds. If one of the rounds contains a $\mathsf{new}(i,a)$, then there cannot be any $\mathsf{del}\ i$ except by the same client, because the new is not visible to other clients yet. However, then there cannot be an update containing i by the same client, for the reason just explained.

B.4 Proof of Optimality of delta reduction

We now prove Theorem 6. We start with a few basic invariants that are useful later on. Then we show optimality (§B.4.1), and finally correctness (§B.4.2).

▶ **Lemma 7.** *All $d \in \Delta$ satisfy the following conditions:*

- (Δ1) *Each $i \in$* Uid *occurs at most once within d.*created *and d.*deleted.
- (Δ2) *If i appears in d.*deleted*, then it does not appear in d.*updated.
- (Δ3) *If $d.\mathsf{updated}(k,f,t) = o$ with $o \neq \bot$, then $o \in \mathsf{Fop}_t$.*
- (Δ4) *If $d.\mathsf{updated}(k,f,t) = o$, then $o \neq \mathit{add}(0)$ and $o \neq \mathit{setifempty}(\text{""})$.*
- (Δ5) *If $d.$*cleared *then $d.$*deleted $= []$.

Proof. By induction. It is easy to verify that all conditions are true on emptydelta, and remain true under any append. ◀

□

B.4.1 Optimality Proof

To prove optimality, we need to show that if w_1 is well-formed and NDU-free, and if $w_1 \triangleright w_2$, then $\mathsf{reduce}(w_1)$ is defined and $|\mathsf{reduce}(w_1)| \leq |w_2|$. First, we argue that $\mathsf{reduce}(w_1) \neq \bot$, because otherwise it would have to trigger an assertion (A3 / A4 in Fig. 10). But if w_1 triggered A3 (A4) it would have to also trigger A1 / A2 in Fig. 9, contradicting well-formedness. Second, we prove two lemmas (8, 9) that together imply that w_2 can be no shorter than $\mathsf{reduce}(w_1)$.

▶ **Lemma 8.** *Let $w_1 \in \mathsf{Update}^*$ be an update sequence. Then:*

- ***(i)*** $\mathsf{reduce}(w_1)$ *contains at most one occurrence of* clr.
- ***(ii)*** $\mathsf{reduce}(w_1)$ *contains at most one occurrence of* del i *for each* i.
- ***(iii)*** $\mathsf{reduce}(w_1)$ *contains at most one occurrence of* new$(i, _)$ *for each* i.
- ***(iv)*** $\mathsf{reduce}(w_1)$ *contains at most one occurrence of* $r.f.t.\_$ *for each* (r, f, t)

Proof. (i) is obvious since we store a simple boolean to record whether clr happened. (ii) is guaranteed because we explicitly check for duplicates before recording deletions. (iii) is guaranteed by assertion A3. (iv) is guaranteed because we store field updates in a partial map, keyed by (r, f, t). ◀

□

▶ **Lemma 9.** *Let w_1 be well-formed and NDU-free, and let $w_1 \rhd w_2$. Then:*

- ***(i)*** *If* $\mathsf{reduce}(w_1)$ *contains* clr*, then* w_2 *contains* clr.
- ***(ii)*** *If* $\mathsf{reduce}(w_1)$ *contains* del i*, then* w_2 *contains* del i.
- ***(iii)*** *If* $\mathsf{reduce}(w_1)$ *contains* new(i, a)*, then* w_2 *contains* new(i, a).
- ***(iv)*** *If* $\mathsf{reduce}(w_1)$ *contains* $r.f.t.o$*, then* w_2 *contains* $r.f.t.o'$ *for some* o'.

Proof. For each claim, we proceed indirectly: assuming the claim is false, we find r and w such that $w \cdot w_1$ is well-formed (which implies $\mathsf{rvalue}(w \cdot w_1, r) = \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), r)$) but for which $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), r) \neq \mathsf{rvalue}(w \cdot w_2, r)$ which then contradicts $w_1 \rhd w_2$.

- **(i)** Let $w = \mathsf{new}(i_1, a) \cdots \mathsf{new}(i_n, a)$ where the i_j are pairwise distinct and do not appear in w_1, and where $n > |w_2| + |\mathsf{reduce}(w_1)|$. Since clr is the only update that can remove more than one row at a time, and is not contained in w_2, $|\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a)| \geq n - |w_2| > |\mathsf{reduce}(w_1)|$. Since clr is contained in $\mathsf{reduce}(w_1)$ and no update can add more than one row at a time, $|\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)| < |\mathsf{reduce}(w_1)|$. Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)$.
- **(ii)** Since del $i \in \mathsf{reduce}(w_1)$, we must have del $i \in w_1$. Distinguish cases (using the first matching case below):
  - [clr $\in w_1$ or clr $\in w_2$]. Then clr $\in w_1$ and thus clr $\in \mathsf{reduce}(w_1)$. But by ($\Delta 5$) this implies del $i \notin \mathsf{reduce}(w_1)$, contradicting the assumption.
  - [$\mathsf{new}(i, a) \in w_1$ for some a]. Then it must occur only once, and before del i in w_1 (otherwise w_1 is not well-formed). Also, there cannot occur a second del i in w_1, because w_1 is assumed NDU-free, nor can there occur an intervening clr in w_1(first case). But this implies that $\mathsf{reduce}(w_1)$ does not contain del i (since $\mathsf{new}(i, a)$ and del i cancel out during reduction), contradicting the assumption.
    - [otherwise]. Let $w = \mathsf{new}(i, a)$. Then $w \cdot w_1$ is well-formed (because $\mathsf{new}(i, a) \notin w_1$, which would have been the preceding case). Since del $i \in \mathsf{reduce}(w_1)$, $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a) \not\ni i$. But $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \ni i$ because w_2 contains neither clr (first case) nor del i (which we assumed for the purposes of deriving a contradiction to the claim). Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)$
- **(iii)** Let $w = []$. If $\mathsf{new}(i, a) \notin w_2$, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \not\ni i$, but since $\mathsf{new}(i, a) \in \mathsf{reduce}(w_1)$, $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a) \ni i$. Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{rows}\ a) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{rows}\ a)$.
- **(iv)** Pick w based on the value of o:

1. if $o = \mathsf{set}(v)$ for some v, pick $w = r.f.t.\mathsf{set}(x)$ for some $x \neq v$.
2. if $o = \mathsf{add}(n)$ for some $n \neq 0$, pick $w = r.f.t.\mathsf{set}(0)$
3. if $o = \mathsf{setifempty}(v)$ for some $v \neq$ "", pick $w = r.f.t.\mathsf{set}("")$

Note that these cases are exhaustive, because $\mathsf{reduce}(w_1)$ contains no other field updates by (Δ4). Also, $w \cdot w_1$ must be well-formed, otherwise w_1 must create an identifier used in r which would contradict the assumption that w_1 is NDU-free. Now, if w_2 contains no updates for (r, f, t), then $\mathsf{rvalue}(w \cdot w_2, \mathsf{fread}\ r.f.t)$ is x or 0 or "", respectively. But on the other hand, $\mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{fread}\ r.f.t)$ is v or n or v, respectively. Thus, $\mathsf{rvalue}(w \cdot w_2, \mathsf{fread}\ r.f.t) \neq \mathsf{rvalue}(w \cdot \mathsf{reduce}(w_1), \mathsf{fread}\ r.f.t)$.

◀

□

B.4.2 Correctness Proof

To get started with proving the correctness of the reduction function as claimed in Theorem 6 we first assemble all the reductions that are needed for the proof, collected in the Lemma below. None of these reductions increase the length of the sequence, and most of them decrease it.

▶ **Lemma 10.** *For all* $w \in \mathsf{Update}^*$, $u \in \mathsf{Update}$, $i \in \mathsf{Uid}$, $a \in \mathsf{Tname}$, $k, k' \in \mathsf{Key}$, $f, f' \in \mathsf{Fname}$, $t, t' \in \mathsf{Ftype}$, $o \in \mathsf{Fop}$, $s, s' \in \mathsf{Val}_{str}$, $n, m \in \mathsf{Val}_{nr}$, *the following are true:*

i. ***(i)*** $w \cdot \mathsf{clr} \rhd \mathsf{clr}$
ii. ***(ii)*** $w \cdot \mathsf{new}(i, a) \rhd \mathsf{new}(i, a) \cdot w$
iii. ***(iii)*** *if* i *occurs in* u *then* $(u \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i)$ *else* $(u \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i \cdot u)$
iv. ***(iv)*** $\mathsf{new}(i, a) \cdot \mathsf{del}\ i \rhd []$
v. ***(v)*** $\mathsf{del}\ i \cdot \mathsf{del}\ i \rhd \mathsf{del}\ i$ *and* $\mathsf{clr} \cdot \mathsf{del}\ i \rhd \mathsf{clr}$
vi. ***(vi)*** $(i$ *occurs in* $u) \Rightarrow (u \cdot \mathsf{new}(i, a) \rhd \bot)$
vii. ***(vii)*** $((k, f, t) \neq (k', f', t')) \Rightarrow (k.f.t.o \cdot k'.f'.t'.o' \rhd k'.f'.t'.o' \cdot k.f.t.o)$
viii. ***(viii)*** $k.f.t.o \cdot k.f.t.\mathsf{set}(v) \rhd k.f.t.\mathsf{set}(v)$
ix. ***(ix)*** $k.f.t.\mathsf{add}(m) \cdot k.f.t.\mathsf{add}(n) \rhd k.f.t.\mathsf{add}(m + n)$
x. ***(x)*** $k.f.t.\mathsf{set}(m) \cdot k.f.t.\mathsf{add}(n) \rhd k.f.t.\mathsf{set}(m + n)$
xi. ***(xi)*** $k.f.t.\mathsf{set}("") \cdot k.f.t.\mathsf{setifempty}(s) \rhd k.f.t.\mathsf{set}(s)$
xii. ***(xii)*** $(s \neq "") \Rightarrow (k.f.t.\mathsf{set}(\mathsf{s}) \cdot k.f.t.\mathsf{setifempty}(s') \rhd k.f.t.\mathsf{set}(s))$
xiii. ***(xiii)*** $(s \neq "") \Rightarrow (k.f.t.\mathsf{setifempty}(\mathsf{s}) \cdot k.f.t.\mathsf{setifempty}(s') \rhd k.f.t.\mathsf{setifempty}(s))$
xiv. ***(xiv)*** $k.f.t.\mathsf{add}(0) \rhd []$ *and* $k.f.t.\mathsf{setifempty}("") \rhd []$
xv. ***(xv)*** $(o \notin \mathsf{Fop}_t) \Rightarrow (k.f.t.o \rhd \bot)$

Proof. Using Def. 1 directly to prove claims of the form $w_1 \rhd w_2$ is unwieldy because of the large number of quantifiers. Instead we use Lemma 11 below, which allows us to check a condition that quantifies over states only. The proofs of the claims are straightforward, we show the first four only.

- **(i)** Let $apply(s, w \cdot \mathsf{clr}) \neq \bot$. Note that (1) clr clears rows and fields, and (2) since used can only grow, $apply(s, w \cdot \mathsf{clr}).\mathsf{used} \supseteq apply(s, \mathsf{clr})$. Thus $apply(s, w \cdot \mathsf{clr}) \sqsupseteq apply(s, \mathsf{clr})$.
- **(ii)** Let $apply(s, w \cdot \mathsf{new}(i, a)) \neq \bot$. Then i cannot appear in neither $s.\mathsf{used}$ nor w. Thus all updates in w commute with $\mathsf{new}(i, a)$.
- **(iii)** If u does not contain i, then u is either clr, del i' with $i' \neq i$, $\mathsf{new}(i', a)$ with $i' \neq i$ or $r.f.t.o$ with $i \notin r$. In all of those cases the delete operation commutes. If u does contain i, it is either (1) del i : see claim (v), or (2) $\mathsf{new}(i, a)$: deleting a nonexisting uid is the

same as creating then deleting it, or (3) $r.f.t.o$ with $i \in r$: then the deletion means the update turns into a no-op since it targets a nonexisting record.

- **(iv)** Let $apply(s, \mathsf{new}(i,a) \cdot \mathsf{del}\ i) \neq \bot$. Since $\mathsf{new}(i,a)$ adds a row that gets immediately removed again by $\mathsf{del}\ i$, rows and fields are the same as if nothing was done at all, but used contains one more element. Thus $apply(s, \mathsf{new}(i,a) \cdot \mathsf{del}\ i) \sqsupseteq apply(s, [])$.

◀

□

▶ **Lemma 11.** *Define a binary relation $\sqsupseteq$ on states as*

$$(s_1 \sqsupseteq s_2) \Leftrightarrow ((s_1.\mathsf{rows} = s_2.\mathsf{rows}) \wedge (s_1.\mathsf{fields} = s_2.\mathsf{fields}) \wedge (s_1.\mathsf{used} \supseteq s_2.\mathsf{used}))$$

Then the following condition is sufficient to imply $w_1 \rhd w_2$:

$$\forall s \in \mathsf{State} : apply(s, w_1) \neq \bot \Rightarrow apply(s, w_1) \sqsupseteq apply(s, w_2).$$

Proof. It is easy to see that $s_1 \sqsupseteq s_2$ implies both

$$\forall r \in \mathsf{Read} : \mathsf{read}(s_1, r) \neq \bot \Rightarrow \mathsf{read}(s_1, r) = \mathsf{read}(s_2, r)$$

$$\forall b \in \mathsf{Update}^* : \mathsf{apply}(s_1, b) \neq \bot \Rightarrow \mathsf{apply}(s_1, b) \sqsupseteq \mathsf{apply}(s_2, b)$$

from which it is easy to deduce the claim.

◀

□

We now proceed to prove the correctness claim in Thm. 6. We show that $\mathsf{reduce}(w) \lhd w$. by induction over the number of elements in w. For $|w| = 0$, the claim is trivially satisfied ($[] \lhd []$). For the induction step, we can assume $\mathsf{reduce}(w) \lhd w$, and we let d be the state of d at the end of the execution of $\mathsf{reduce}(w)$. Then, using (a) Lemma 12 below and (b) the induction hypothesis, we get that for all $u \in \mathsf{Update}$,

$$\mathsf{reduce}(w \cdot u) = d.\mathsf{append}(u).\mathsf{seq}() \lhd^{(a)} d.\mathsf{seq}() \cdot u = \mathsf{reduce}(w) \cdot u \lhd^{(b)} w \cdot u,$$

which concludes the correctness proof.

▶ **Lemma 12.** *For all $d \in \Delta$ and $u \in \mathsf{Update}$, we have $d.\mathsf{append}(u).\mathsf{seq}() \lhd d.\mathsf{seq}() \cdot u$.*

Proof. We let $lhs = d.\mathsf{append}(u).\mathsf{seq}$ and $rhs = d.\mathsf{seq} \cdot u$, and thus need to show that $lhs \lhd rhs$. We write $d.\mathsf{seq} = C \cdot D \cdot N \cdot U$ where C, D, N, U are sequences of clear, delete, new, and field updates. Now, we do a case distinction (case conditions shown in brackets, applying the first case that matches), and make use of the reductions proved in Lemma 10, labeling $\lhd$ with a subscript indicating the clause used.

- $[u = \mathsf{clr}]$. Then $lhs = \mathsf{clr} \lhd_{(i)} C \cdot D \cdot N \cdot U \cdot \mathsf{clr} = rhs$.
- $[u = \mathsf{new}(i,a)]$.

- [i occurs in $D \cdot N \cdot U$]. Then $lhs = \bot \lhd_{(vi)} C \cdot D \cdot N \cdot U \cdot \mathsf{new}(i, a) = rhs$.
- [otherwise]. $lhs = C \cdot D \cdot N \cdot \mathsf{new}(i, a) \cdot U \lhd_{(ii)} C \cdot D \cdot N \cdot U \cdot \mathsf{new}(i, a) = rhs$.

- [$u = \mathsf{del}\ i$]. Let U', N' be the subsequences of U, N obtained by removing updates containing i.
  - [$D = D_1 \cdot \mathsf{del}\ i \cdot D_2$]. By $\Delta 5$, $C = []$. Then $lhs = D \cdot N \cdot U \lhd_{(v)} \cdot D_1 \cdot \mathsf{del}\ i \cdot \mathsf{del}\ i \cdot D_2 \cdot N \cdot U \cdot N \cdot U \lhd_{(iii),(\Delta 1),(\Delta 2)} D_1 \cdot \mathsf{del}\ i \cdot D_2 \cdot N \cdot U \cdot \mathsf{del}\ i = rhs$.
  - [$N = N_1 \cdot \mathsf{new}(i, a) \cdot N_2$ for some a]. Then $lhs = C \cdot D \cdot N_1 \cdot N_2 \cdot U' \lhd_{(iv)} C \cdot D \cdot N_1 \cdot \mathsf{new}(i, a) \cdot \mathsf{del}\ i \cdot N_2 \cdot U' \lhd_{(iii)} C \cdot D \cdot N_1 \cdot \mathsf{new}(i, a) \cdot N_2 \cdot U \cdot \mathsf{del}\ i = rhs$.
  - [$C = \mathsf{clr}$]. Then $lhs = \mathsf{clr} \cdot D \cdot N \cdot U' \lhd_{(v)} \mathsf{clr} \cdot \mathsf{del}\ i \cdot D \cdot N \cdot U' \lhd_{(iii)} C \cdot D \cdot N \cdot U \cdot \mathsf{del}\ i = rhs$.
  - [otherwise]. Then $lhs = C \cdot D \cdot \mathsf{del}\ i \cdot N \cdot U' \lhd_{(iii)} C \cdot D \cdot N \cdot U \cdot \mathsf{del}\ i = rhs$.

- [$u = k.f.t.o$]. For convenience, we define a function Φ on field updates as: $\Phi((k'.f'.t'.o')) =$

$$\begin{cases} [] & \text{if } o' = \mathsf{add}(0) \text{ or } o' = \mathsf{setifempty}(0) \\ k'.f'.t'.o' & \text{otherwise} \end{cases}$$

  - [$o \notin \mathsf{Fop}_t$]. Then $lhs = \bot \lhd_{(xv)} C \cdot D \cdot N \cdot U \cdot k.f.t.o = rhs$.
  - [$(k, f, t) \notin U$]. Then $lhs = C \cdot D \cdot N \cdot U_1 \cdot \Phi(u) \cdot U_2 \lhd_{(vii)} C \cdot D \cdot N \cdot U \cdot \Phi(u) \lhd_{(xiv)} rhs$.
  - [$U = U_1 \cdot u' \cdot U_2$ where $u' = k.f.t.o'$]. Then
    - [$o = \mathsf{add}(0)$ or $o = \mathsf{setifempty}(\text{""})$]. Then $lhs = C \cdot D \cdot N \cdot U \lhd_{(xiv)} rhs$.
    - [$o = \mathsf{set}(v)$]. Then $lhs = C \cdot D \cdot N \cdot U_1 \cdot u \cdot U_2 \lhd_{(viii)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot u \cdot U_2 \lhd_{(vii)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot U_2 \cdot u = rhs$.
    - [$o' = \mathsf{add}(m), o = \mathsf{add}(n)$]. Then $lhs = C \cdot D \cdot N \cdot U_1 \cdot \Phi(k.f.t.\mathsf{add}(m+n)) \cdot U_2 \lhd_{(xv)} C \cdot D \cdot N \cdot U_1 \cdot k.f.t.\mathsf{add}(m+n) \cdot U_2 \lhd_{(ix)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot u \cdot U_2 \lhd_{(vii)} C \cdot D \cdot N \cdot U_1 \cdot u' \cdot U_2 \cdot u = rhs$.
    - [$o' = \mathsf{set}(m), o = \mathsf{add}(n)$]. Analogously, using (x) instead of (ix).
    - [$o' = \mathsf{set}(\text{""})), o = \mathsf{setifempty}(s)$]. Analogously, using (xi).
    - [$o' = \mathsf{set}(v), v \neq \text{""}, o = \mathsf{setifempty}(s)$]. Analogously, using (xii).
    - [$o' = \mathsf{setifempty}(v), o = \mathsf{setifempty}(s)$]. Analogously, using (xiii) and ($\Delta 4$) (the latter implies $v \neq \text{""}$).

◀

□