Global Sequence Protocol: A Robust Abstraction for Replicated Shared State

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

1 **Microsoft Research, USA**
2 **Google, USA**

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).

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.2015.568

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 consistent 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 citeect), 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 $rvalue : Read \times Update^* \rightarrow Value$

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:

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

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:

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

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

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 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],

```
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).

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 1.** 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:

```
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 the extended technical report [11] 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.

5.1 Discussion

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

```
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).

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*	:	$\mathit{Read} \times \mathit{State} \rightarrow \mathit{Value}$
function *apply*	:	$\mathit{State} \times \mathit{Delta}^* \rightharpoonup \mathit{State}$
const *emptydelta*	:	Delta
function *append*	:	$\mathit{Delta} \times \mathit{Update} \rightharpoonup \mathit{Delta}$
function *reduce*	:	$\mathit{Delta}^* \rightharpoonup \mathit{Delta}$

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

▶ **Example 2.** 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 $\mathit{read}(r,s)$ yields the same result as reading $\mathit{rvalue}(r,u_1 \cdot u_n)$ 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 $\mathit{Delta} \times \mathit{Update}^*$, let $\lhd$ be the smallest relation such that (1) $\mathit{emptydelta} \lhd []$, and (2) $d \lhd a$ implies $\mathit{append}(d,u) \lhd a \cdot u$ for all updates u, and (3) $d_1 \lhd a_1 \wedge \cdots \wedge d_n \lhd a_n$ implies $\mathit{reduce}(d_1 \cdots d_n) \lhd a_1 \cdots a_n$.
- On $\mathit{State} \times \mathit{Update}^*$, let $\lhd$ be the smallest relation such that (1) $\mathit{initialstate} \lhd []$, and (2) $s \lhd a \wedge d_1 \lhd a_1 \wedge \cdots \wedge d_n \lhd a_n$ implies $\mathit{apply}(s,d_1 \cdots d_n) \lhd a \cdot a_1 \cdots a_n$.

```
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.

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 the extended technical report [10, 11] 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 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).

```
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.

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*.

```
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).

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 ($established = \mathsf{true}$), 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 $s.maxround[\mathsf{this}]$ 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 $s.maxround[\mathsf{this}]$ 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 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

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

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*,

```
// class Client (continued)
  // auxiliary functions
  function $curstate()$ : $State$ {
    return $apply(known, deltas(pending) \cdot pushbuf \cdot transactionbuf)$;
  }
  function $deltas(seq : Round*)$ : $Delta$ { return $seq[0].delta \cdot \cdot \cdot seq[seq.length - 1].delta$; }
  procedure $adjust\_pending\_queue(upto : \mathbb{N})$ {
    while ($upto >= pending[0].round$) { $pending := pending[1..]$; }
  }
  // network transitions (nondeterministic)
  $receive()$ {
    requires $channel$ != $null$ && $channel.serverstream.length > 0$;
    $channel.receivebuffer := channel.receivebuffer \cdot channel.serverstream[0]$;
    $channel.serverstream := channel.serverstream[1..]$;
  }
  $drop\_connection()$ { $channel := null$; }
  $send\_connection\_request()$ {
    requires $channel == null$;
    $channel$ := new $Channel$(this);
  }
  $send()$ {
    requires $channel$ != $null$ && $channel.established$ && $rds\_in\_pushbuf > 0$;
    var $r$ := new $Round$ { $origin$ = this, $round = round - 1$, $delta = pushbuf$ };
    $pending := pending \cdot r$;
    $channel.clientstream := channel.clientstream \cdot r$;
    $pushbuf := []$;
    $rds\_in\_pushbuf := 0$;
  }
}
```

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

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 $(\Sigma_S, \sigma^i_S, T_E \cup T_S, \delta_S)$, 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 $\sigma^i = (\sigma^i_I, \sigma^i_S, \sigma^i_A)$
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 $\langle \sigma = \sigma^0_S, t_1, \sigma^1_S, t_2, \ldots, t_n, \sigma^n_S \rangle$ (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: $(\sigma'_I, \sigma^n_S, u_A((\sigma_I, \sigma_S, \sigma_A), t, \sigma_S)) \in R$.

To define this relation and prove the obligations, we construct a "combined transition system" in the extended technical report [11] 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*Z [5, 14] which include tables, indices, and records. We describe the cloud types data model and prove optimal reduction in the extended technical report [11].

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.

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, 2012.

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)*, 2014.

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, 2012.

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 2014.

11 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. Technical Report MSR-TR-2015-11, Microsoft Research, April 2015. Extended version.

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.

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.

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.

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.

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, 2011.

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 2011.

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.

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.