Citrea and Swarm: partially ordered op logs in the browser

Victor Grishchenko
Citrea LLC
victor.grishchenko@gmail.com

25 March 2014

Abstract

Principles of eventual consistency are normally applied in large-scale distributed systems. I report experiences applying EC in Web app development. Citrea is a collaborative rich text editor employing the Causal Trees[4] technology of concurrency/version control (note: CT is *not* an OT flavor). CT employs symbol ids and trees and, generally, belongs to the same family as WOOT, Logoot or Treedoc [1, 3, 2]. CT makes the ids-and-trees approach production-practical by limiting itself to simple and lightweight algorithms and data structures. Swarm is a JavaScript object sync library that works in the browser, in real time. Swarm fully employs client-side storage and works well under intermittent connectivity. Swarm employs "pure" op-based model. Our top finding is a *specifier*, a serialized event description format that enables straightforward operation storage, caching and exchange.

This paper briefly describes our experiences and findings made while implementing an in-browser collaborative editor Citrea (currently in semi-open beta at `letters.yandex.ru`) and a real-time JavaScript object sync library Swarm.js (unreleased as of Feb 2014). Partially ordered log of immutable operations (POLO) is a known approach in the eventual consistency land. Normally, POLO is employed by large scale systems once operation log linearization becomes unfeasible. We had some experience applying POLO at a smaller scale, namely to a real-time collaborative editor system, where a "dataset" is just a single document. Indeed, a collaborative editor faces the same issues "large" systems face, but for a different reason. In a "large" system, due to its scale, at any moment some components fail. At the same time, bringing all the transactions to a single point for linearization is not feasible due to their volume. Hence, such systems adopt some form of eventual consistency, sometimes POLO.

In a collaborative editor, users exchange edits in real time over potentially faulty high-lag end-user links. That leads to the same problem: linearization is impossible, so changes have to be applied to different replicas in different orders. The established approach to the problem is Operational Transformation, a theory of rewriting operations in flight to retrofit them to de-facto orderings. OT has a reputation for complexity of understanding, development and use. Because of our modest numbers, we chose the more deterministic POLO approach. Citrea decomposes a document into a stream of "atomic" edits (same as OT), but these operations are immutable all the way along. Instead, change merge algorithms adapt to the fact that orders might vary slightly from replica to replica, (as long as) no causality violations are allowed.

The key enabler of our approach was proper unique identification of events using logical clocks (also known as Lamport timestamps). Once reliably identified, events can be exchanged, cached, stored, ordered and, most importantly, can reference preceding events thus forming causal structures. That saved us from volatile positional addressing OT relies on; each operation may reliably identify which part of the text it applies to, as every letter (atom) has an id. The tough case of same-point concurrent insertion is similarly resolved based on Lamport timestamps (Fig. 1). So, in the theory domain, the main finding is that per-letter Lamport timestamps are entirely sufficient to have a simple brute-force solution for all the consistency and convergence problems OT struggles with.

The causal tree model represents a text as a tree of atoms; that makes it similar to WOOT, TreeDoc or Logoot. The main difference is seen from the practical angle: Logoot/TreeDoc employ relatively complex data structures, such as trees and variable-length "dense" ids. Maintaining a tree node for every text's letter is prohibitively expensive. In practice, a CT text is still maintained as a string (of atoms), so a "causal tree" is an *implicit* mathematical abstraction, not an actual data structure. Lamport timestamp is serialized as four Unicode symbols; so the text rests in an interleaved string of characters, their Lamport ids and tombstones. That lowers the overhead by orders of magnitude. Namely, CT-encoded versioned text is x5-x10 the size of the plain text. As text download size is dwarfed by images and, especially, video, the factor of 10 is entirely acceptable. As a useful bonus, per-letter Lamport ids provide versioning and authorship information on every single letter.

The same can be said about dealing with partial order issues. While many EC/CRDT models explicitly employ vector timestamps, CT works as implicitly as possible. Operation delivery channels are guaranteed to introduce no violations of causality (relay-in-order rule) and that is the only guarantee a replica gets. Full version vectors are rarely used; their top usecase is patch exchange after a reconnection.

That leads to very relaxed infrastructure requirements; Citrea needs no central "serializer" servers, works well offline, may store changes locally to resync later (Google Docs can only do that with some caveats) and may work client-to-client, which is valuable in the context of WebRTC.

Swarm is a JavaScript real-time object sync library, which started as a supplementary project for the Citrea editor. Swarm distributes a partially ordered operation log (POLO) to model's replicas that synchronize their state by CRDT-like algorithms.

In our "reactive" interpretation of object-oriented programming, the state is still encapsulated by objects. Every change of state must be wrapped as a *method*, which is almost synonymous to "change", "operation" and "event". Every method invocation is assigned a Lamport timestamp and asynchronously propagated to other replicas (like an op). Also, every method can be listened to (like an event).

As long as we limit ourselves to the case of uninterrupted "steady-state" Alice-to-Bob sync session, Swarm perfectly fits the prepare-effect scheme from the CRDT literature [6, 7] and qualifies as "pure" op-based [5], as the object's state does not affect operation preparation.

Further on, we had to extend the classic toolset with a more powerful concept of a *specifier*. As our apps exchange fine-grained changes in real time, we had to concisely describe the context of every small change as it is likely to be delivered, processed and stored separately from the rest of the related state. For every atomic operation, a specifier contains its class, object id, method name and, most importantly, its Lamport timestamp which is local time plus replica id. In practice, a serialized specifier looks like:

`/TodoItem#PaPEC!7lTX1+gritzko~e4.done` where `7lTX1` is a Base64 timestamp for Feb 14 2014 23:12:01 GMT, `gritzko~e4` is the author/replica id, `TodoItem` and `done` are class and method names respectively and `PaPEC` is an object id. So, every method invocation is serialized into a 3-parameter signature of the event's *specifier*, *value* (method parameters) and *source* (essentially, a callback). As this programming model is inherently asynchronous, a method can not simply return a value or throw an exception the regular way, so any returns are supplied back to the source through the callback.

This programming model may be described as uber-reactive and even (tongue-in-cheek) "reactionary" as we do our best to program extremely distributed concurrent systems more or less the way we programmed good old local MVC apps. The issues of eventual consistency are isolated inside a distributed event bus. The bus delivers events to replicas asynchronously, leaving other steps of the MVC cycle more or less the same.

The advantage of the approach we enjoyed the most, apart from the fact that it works, is the fact that it works well offline and under intermittent connectivity. It is sufficient for replicas to exchange "patches" time to time. Correct POLO synchronization is possible to interrupt, but impossible to break as long as causality order is maintained, which practically amounts to TCP-like in-order delivery guarantees in most cases. The design is generally intuitive and understandable, as opposed to OT.

The top shortcoming of POLO is the overhead of running a live replica of the model at the client, including a significant portion of the logic, as opposed to flattened results/views (like in classic Web apps) or a proxied dataset (like in e.g. Meteor). That is likely an inevitable price to pay for responsive and offline-ready apps.

From the point of view of an implementer, predictably the most difficult situation is a "blast from the past"; once a batch of changes arrives from a reconnected client, it needs to be merged into a replica. Algorithms tend to be application-dependent and that, in particular, prevents us from implementing POLO stack at the database layer (unless we are ready to tolerate version forks, like in Dynamo or CouchDB/pouchdb). Our approach was to leave the storage layer "dumb" and generic. Operations expressed as specifier-value pairs are conveniently stored in any key-value storage, including HTML5 WebStorage. On top of the storage goes the general POLO layer implementing "difficult" routines: handshakes, resynchronization, version vectors, op log maintenance and suchlike. On top of that goes application-dependent POLO logic, including the actual "payload" logic of operations, merge algorithms and state maintenance. Given that all, an implementer may write the rest of an app in the regular MVC fashion, which was the objective.

A CT – concurrent insertion

In the Causal Trees model, a text is understood as a growing tree. A symbol is considered to be a child of its preceding symbol at the time of insertion. As symbols are marked with unique Lamport timestamps, OT's transformations ("offset magic") are no longer needed. The only remaining non-trivial case is the concurrent insertion at the same location.

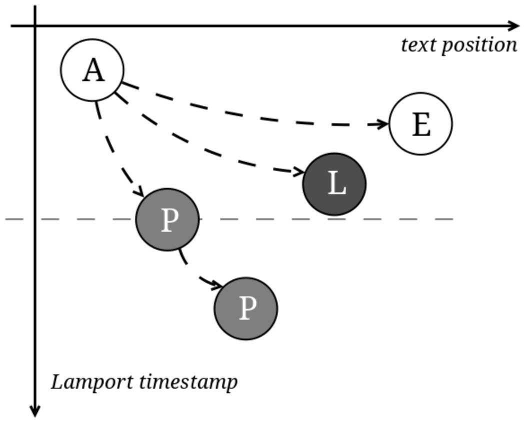

Figure 1: Causal trees: concurrent insertion

The Figure illustrates the case of a string AE changed by Alice into $APPE$ and, concurrently, by Bob into ALE. Hence, Bob and Alice receive PP and L in different orders. Alice gets the hard case: the newly arrived L should not be put after its parent A. Alice detects the case of concurrent insertion by checking that the A's following symbol P has greater timestamp value than the inserted symbol L. Alice skips the *causal subtree* of P, which is PP, and inserts L before *apparently* preexisting E. Whether E actually preexisted L is not even relevant for convergence. For example, in the case of Bob inserting PP between A and L, L is treated as preexisting because of its lesser timestamp. Essentially, sibling subtrees are sorted by Lamport timestamps of their root symbols. Notably, the approach allows not to maintain the tree as an actual data structure. Instead, subtrees are detected by timestamp values. The text is still kept in a string, where every symbol is followed by its Lamport timestamp.

References

[1] G. Oster et al. Data consistency for P2P collaborative editing. CSCW'2006, Banff, Canada, 2006

[2] M. Shapiro and N. Preguia, Designing a commutative replicated data type INRIA, INRIA Tech report RR-6320, 2007

[3] Stphane Weiss, Pascal Urso and Pascal Molli. Logoot: a P2P collaborative editing system. INRIA Tech Report N6713, 2008

[4] Victor Grishchenko. Deep hypertext with embedded revision control implemented in regular expressions. WikiSym'2010, Gdansk, Poland

[5] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. Making Operation-based CRDTs Operation-based. PaPEC'2014, Amsterdam, Netherlands

[6] Mihai Letia, Nuno Preguia and Marc Shapiro. CRDTs: Consistency without concurrency control. INRIA N6956, 2009

[7] Marc Shapiro et al. Conflict-free replicated data types. SSS'2011, Grenoble, France, 2011