Published in Coinmonks

Anton Zagorskii Follow

Jul 18, 2018 · 10 min read · Listen

Operational Transformations as an algorithm for automatic conflict resolution

1. Introduction

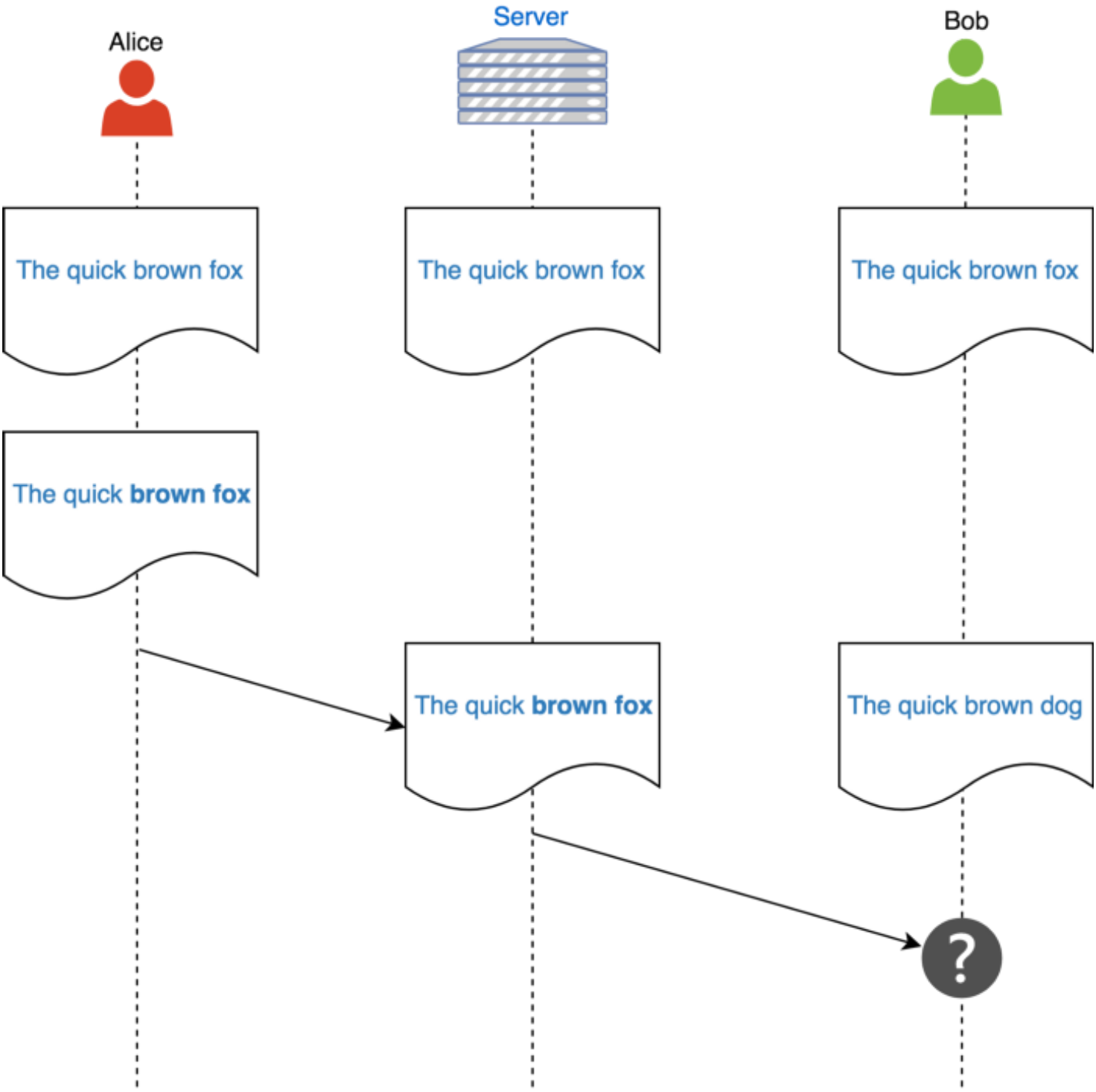
Automatic conflicts resolution algorithms in distributed systems with more than one leader nodes (In this article by leader node we mean a node which accepts data changing requests) is quite interesting research field. There are different approaches and algorithms in this area with their own trade-offs and in this article we will focus on Operational Transformation technology which is aimed to solve conflicts in collaborative editing applications like Google Docs or Etherpad.

Collaborative editing is a difficult task to solve from a development point of view because clients can be doing changes in the same parts of text at almost the same time. To imitate immediate response each client has to maintain its own local copy of the document, otherwise clients would have wait till their changes are synced with other clients and that can and will take a noticeable amount of time. So the main issue in collaborative editing is to ensure *consistency* between local replicas i.e. all replicas have to *converge* to the identical, correct version of the document (Note we can not require all replicas to have an identical copy of the document at some point in time as the editing process can be endless).

1.1 First versions of Google Docs

First versions of Google Docs (as well as other similar applications) used a comparison of document versions. Imagine we have two clients

Quite often this approach doesn't work well. For example, imagine that Alice, Bob, and server start from the synchronized document "The quick brown fox".



Alice bolds the words "brown fox" and at the same time Bob changes "fox" to "dog". Alice's changes arrived first at the server, it applies and sends changes further to Bob. The correct result of the merge should be "The quick **brown dog"**, but merging algorithms do not have enough information to perform the correct merge. From their perspective "The quick **brown fox** dog", "The quick **brown** dog", "The quick **brown** dog **fox"** are all correct and valid results. (In such cases in git you would have merge conflicts and you would have to merge manually). So that is the main problem — we can not rely on the merge with such an approach.

Ok, now we need to understand **when** to apply changes — we need a *collaboration protocol*. In Google Docs all operations are down to 3 possible types:

- Insert text
- Delete text
- Apply style

When you edit a document all changes are appended to the changelog in one of those 3 types and the changelog is replayed from some point when you open a document.

(First (public) Google project with OT support was Google Wave and its set of possible operations was much richer)

1.3 Example

Let Alice and Bob start from a synchronized document “LIFE 2017”.

Alice changes 2017 to 2018 and that actually creates 2 operations:

	Alice								
Indices	0	1	2	3	4	5	6	7	8
	L	I	F	E		2	0	1	7
Delete @8	L	I	F	E		2	0	1	
Insert '8', @8	L	I	F	E		2	0	1	8

At the same time Bob changes the text to “CRAZY LIFE 2017”:

	Bob														
Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	L	I	F	E		2	0	1	7						
Insert 'C' @0	C	L	I	F	E		2	0	1	7					
Insert 'R', @1	C	R	L	I	F	E		2	0	1	7				
Insert 'A', @2	C	R	A	L	I	F	E		2	0	1	7			
Insert 'Z', @3	C	R	A	Z	L	I	F	E		2	0	1	7		
Insert 'Y', @4	C	R	A	Z	Y	L	I	F	E		2	0	1	7	
Insert ' ', @5	C	R	A	Z	Y		L	I	F	E		2	0	1	7

If Bob just applies Alice’s delete operation when he receives it then he gets an incorrect document (the digit 7 should have been deleted

Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	C	R	A	Z	Y		L	I	E		2	0	1	7

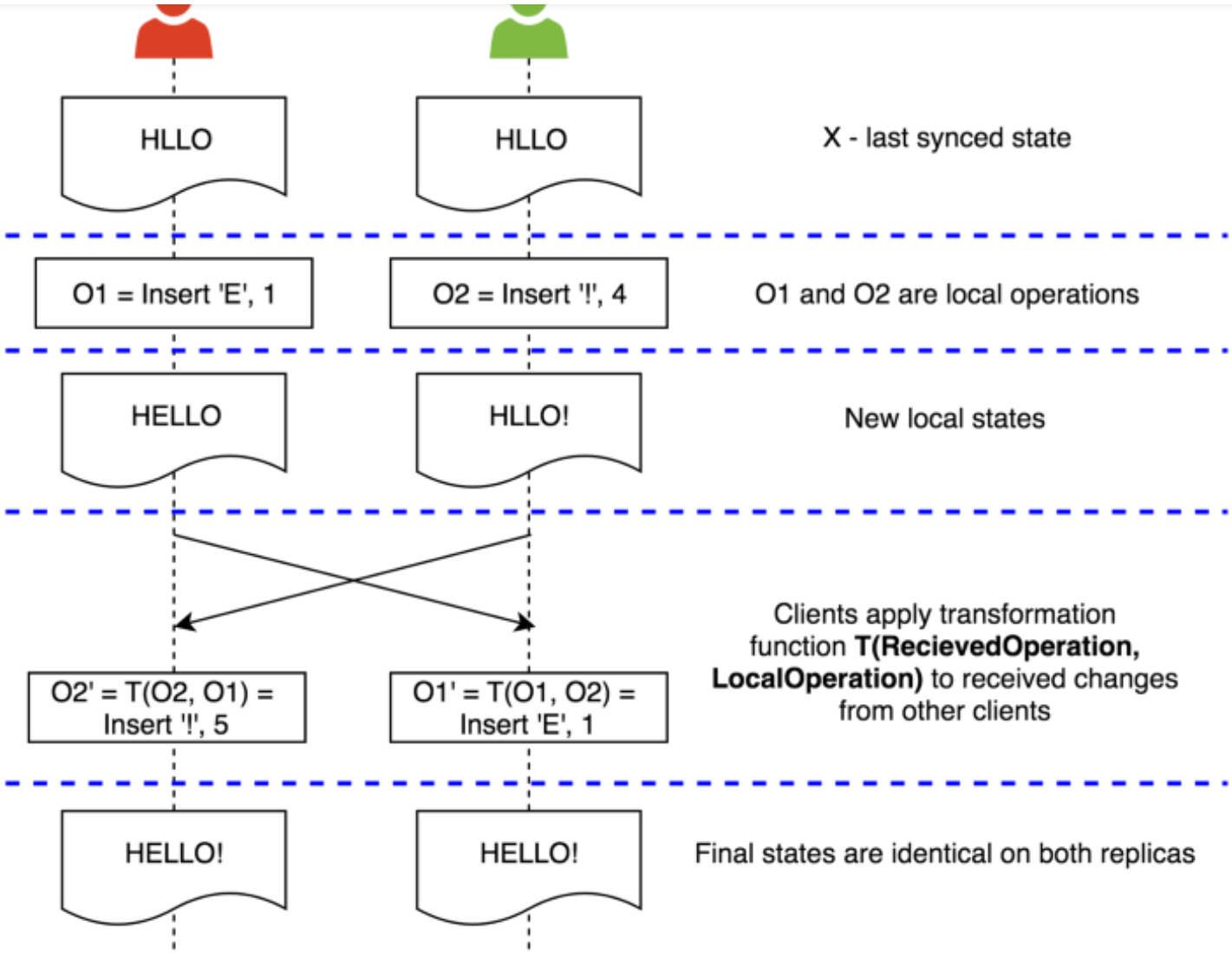
Bob needs to transform the delete operation accordingly to his local changes to get the correct state of the document :

$$\{Delete\ @8\} \rightarrow \{Delete\ @14\}$$

	Bob													
Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	C	R	A	Z	Y		L	I	F	E		2	0	1

Now it is perfect.

To make it more formal, let’s consider the following example:



then

$$O1'(O2(X)) = O2'(O1(X))$$

Voila! Described algorithm is the Operational Transformation.

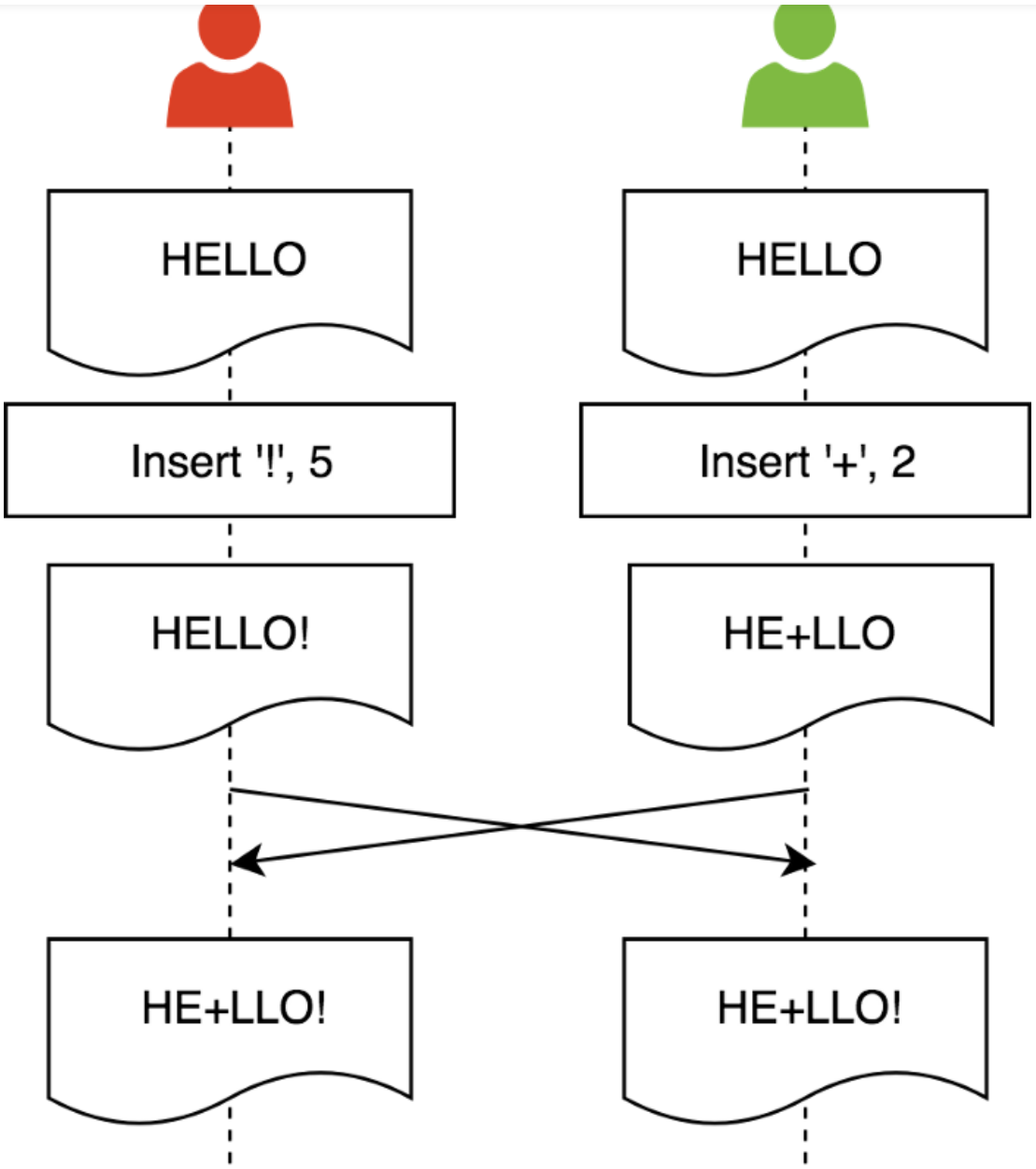
2. Operational Transformation

2.1 Consistency models

Few *consistency models* have been developed to provide consistency for OT. Let's consider one of them — CCI model.

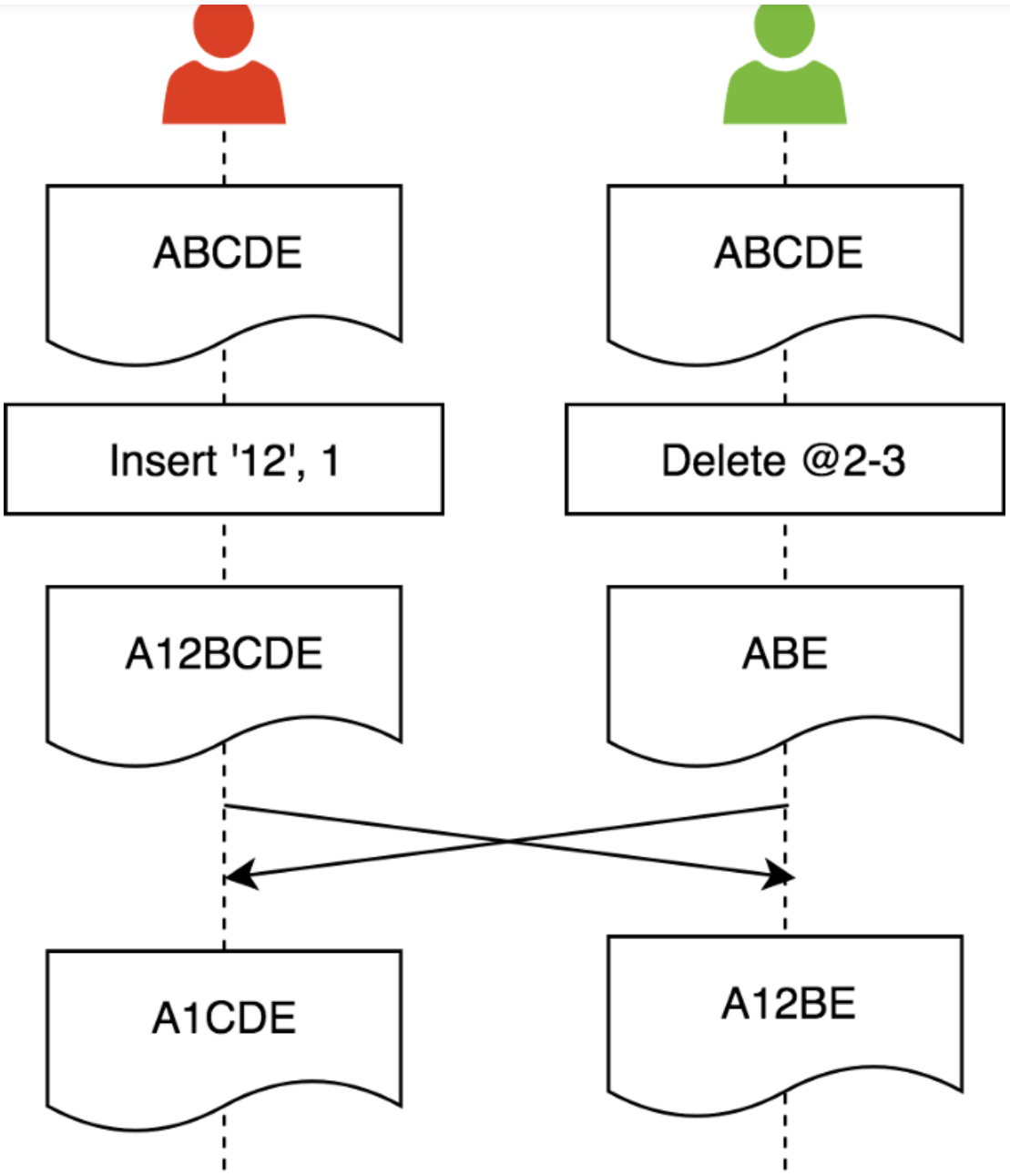
The CCI model consists of the following properties:

1. Convergence: all replicas of the same document must be identical after execution of all operations



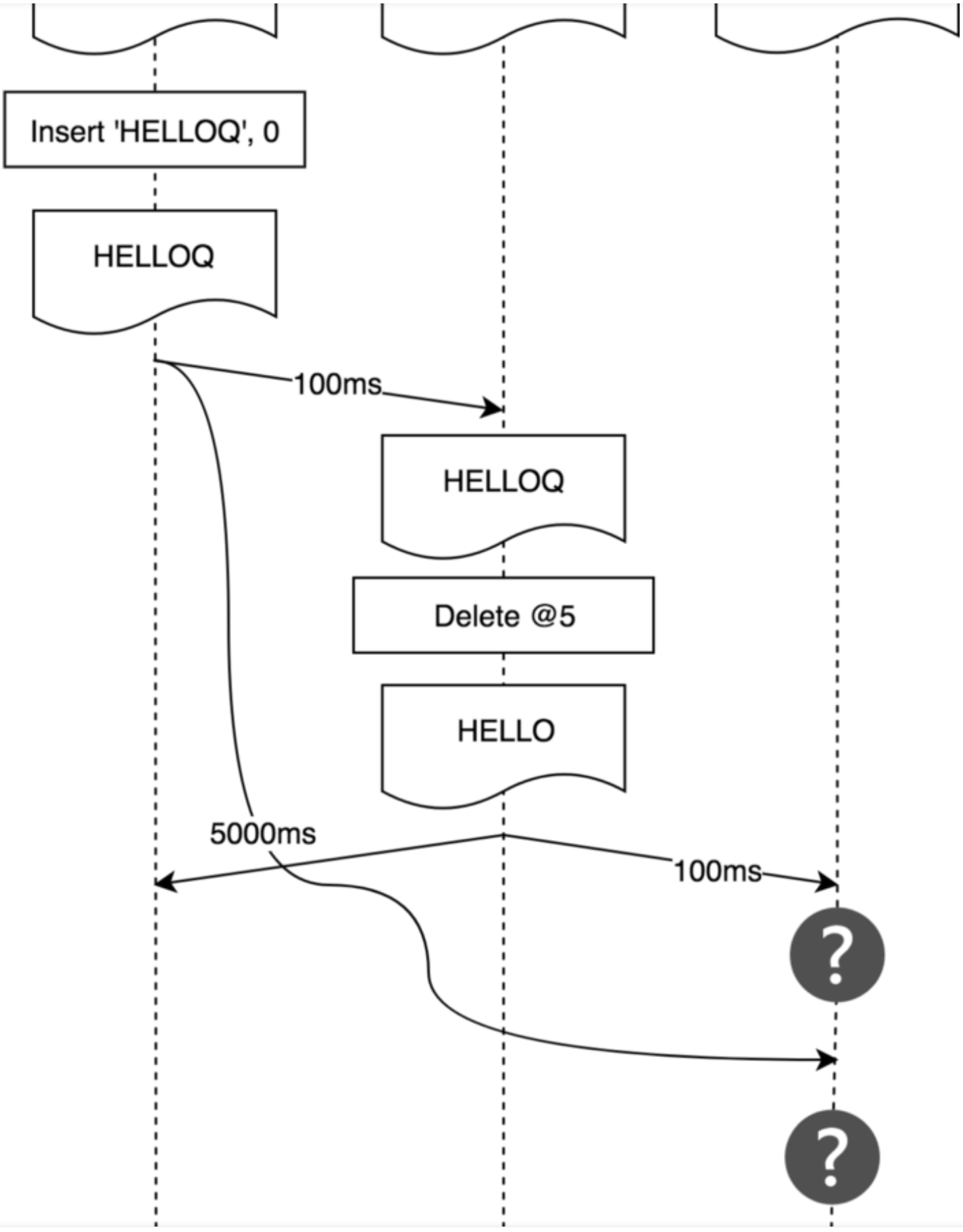
Alice and Bob start from the same document, then they do local changes and replicas diverge (thus high responsiveness is achieved). Convergence property requires Alice and Bob to see the same document after they received changes from each other.

2. Intention preservation: ensures that the effect of executing an operation on any document state will be the same as the *intention of the operation*. An intention of operation is defined as the effect of its execution on a replica it was created.
Let's consider an example where this property is violated:



Alice and Bob start from the same document state, then do their local changes. The intention of Alice's operation is to insert '12' at position 1 and the intention of Bob's operation is to delete 'CD'. After synchronization Bob's intention is violated. We can also observe replicas have diverged, but that is not a requirement for the intention preservation property. Exercise: what is the correct final state in this example?

3. Causality preservation: operations must be executed accordingly to their natural cause-effect order during the process of collaboration (based on the happened-before relation).
Let's consider the example where this property is violated:

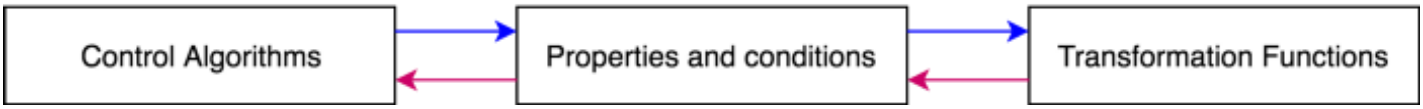


to delete a symbol which doesn't exist yet in his replica.

OT can't ensure this property so other algorithms like Vector clock can be used.

2.2 OT Design

One of the OT design strategies is to split it into 3 parts:



1. Transformation control algorithms: responsible for determining when an operation (transformation target) is ready for transformation, which operations (transformation reference) should be transformed against, and in which order should transformations be carried out
2. Transformation functions: responsible for performing actual transformations on the target operation according to the impact of the reference operation
3. Properties and conditions: define the relationships between transformation control algorithms and functions, and serve as OT algorithm correctness requirement

2.3 Transformation functions

Transformation functions can be divided into two types:

- Inclusion/Forward transformation: to transform operation Oa before Ob so that the effect of Ob is included (e.g. two insertions)
- Exclusion/Backward transformation: to transform operation Oa before Ob so that the effect of Ob is excluded (e.g. insertion after deletion)

Here is an example of transformation functions which work with character changes:

```
Tii(Ins[p1, c1], Ins[p2, c2]) {
  if (p1 < p2) || ((p1 == p2) && (order() == -1))  // order() - order calculation
        return Ins[p1, c1]; // Tii(Ins[3, 'a'], Ins[4, 'b']) = Ins[3, 'a']
  else
        return Ins[p1 + 1, c1]; // Tii(Ins[3, 'a'], Ins[1, 'b']) = Ins[4, 'a']
}

Tid(Ins[p1, c1], Del[p2]) {
  if (p1 <= p2)
    return Ins[p1, c1]; // Tid(Ins[3, 'a'], Del[4]) = Ins[3, 'a']
  else
    return Ins[p1 - 1, c1]; // Tid(Ins[3, 'a'], Del[1]) = Ins[2, 'a']
}

Tdi(Del[p1], Ins[p2, c1]) {
  // Exercise
}

Tdd(Del[p1], Del[p2]) {
  if (p1 < p2)
    return Del[p1]; // Tdd(Del[3], Del[4]) = Del[3]
  else
    if (p1 > p2) return Del[p1 - 1]; // Tdd(Del[3], Del[1]) = Del[2]
  else
    return Id; // Id - identity operator
```

3. Collaboration protocol in Google Docs

Let's consider how Google Docs collaboration protocol works in more details.

Each client maintains the following information:

- Last synced revision (id)
- All local changes which have not been sent to the server (Pending changes)
- All local changes sent to the server but have not been acknowledged (Sent changes)
- Current document state visible to the user

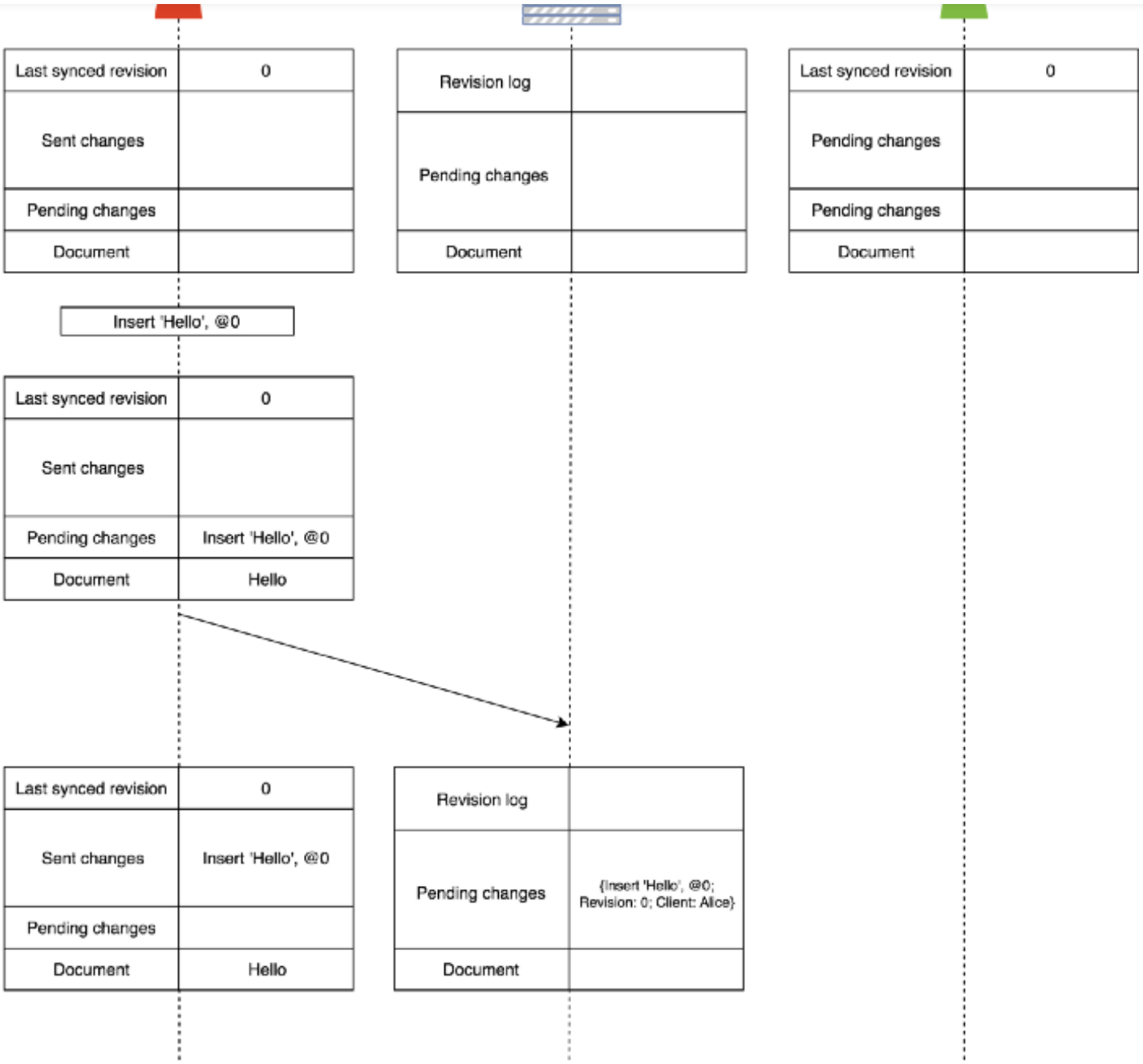
The server maintains the following information:

- List of all received but have not been processed changes (Pending changes)
- Log of all processed changes (Revision log)
- State of the document on the time of last processed change

Assume we have Alice, Server, Bob and they start from a synchronised empty document.

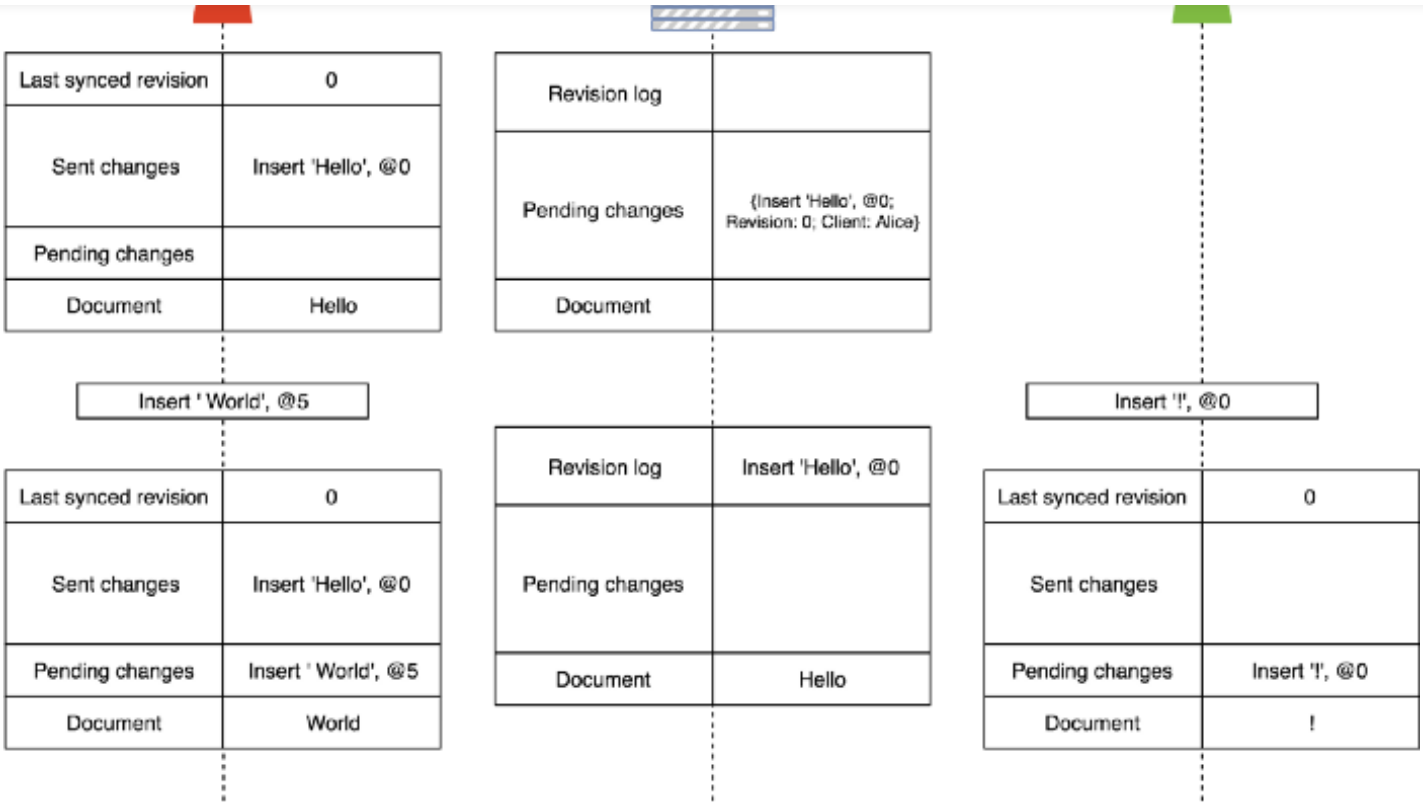
Alice types "Hello":



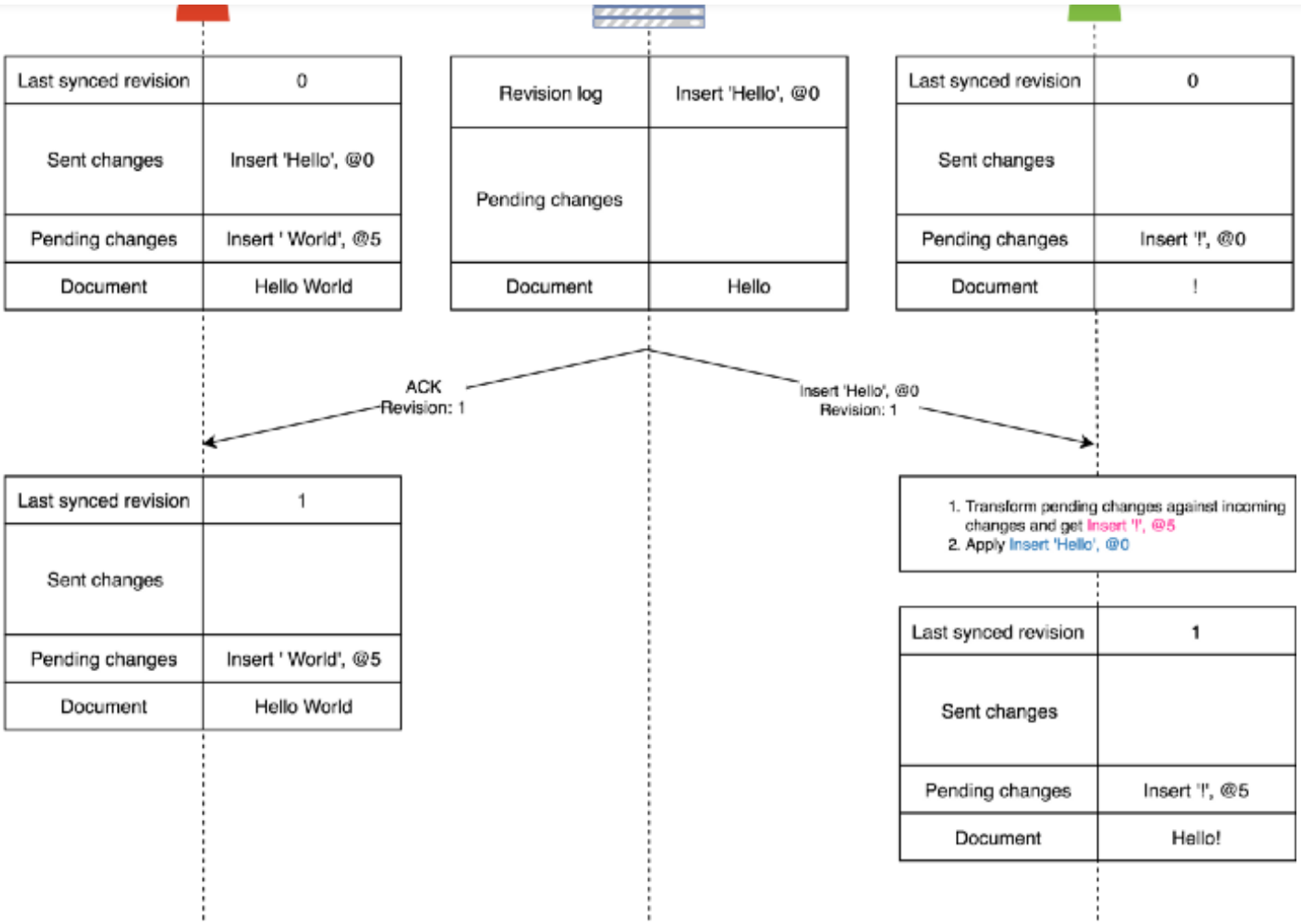



Her change was added to the list of pending changes and then it was sent to the server and moved to the sent changes list.

Meanwhile, Alice was typing and she added “ world”. At the same time Bob has typed “!” in his empty document (he has not received Alice’s changes):

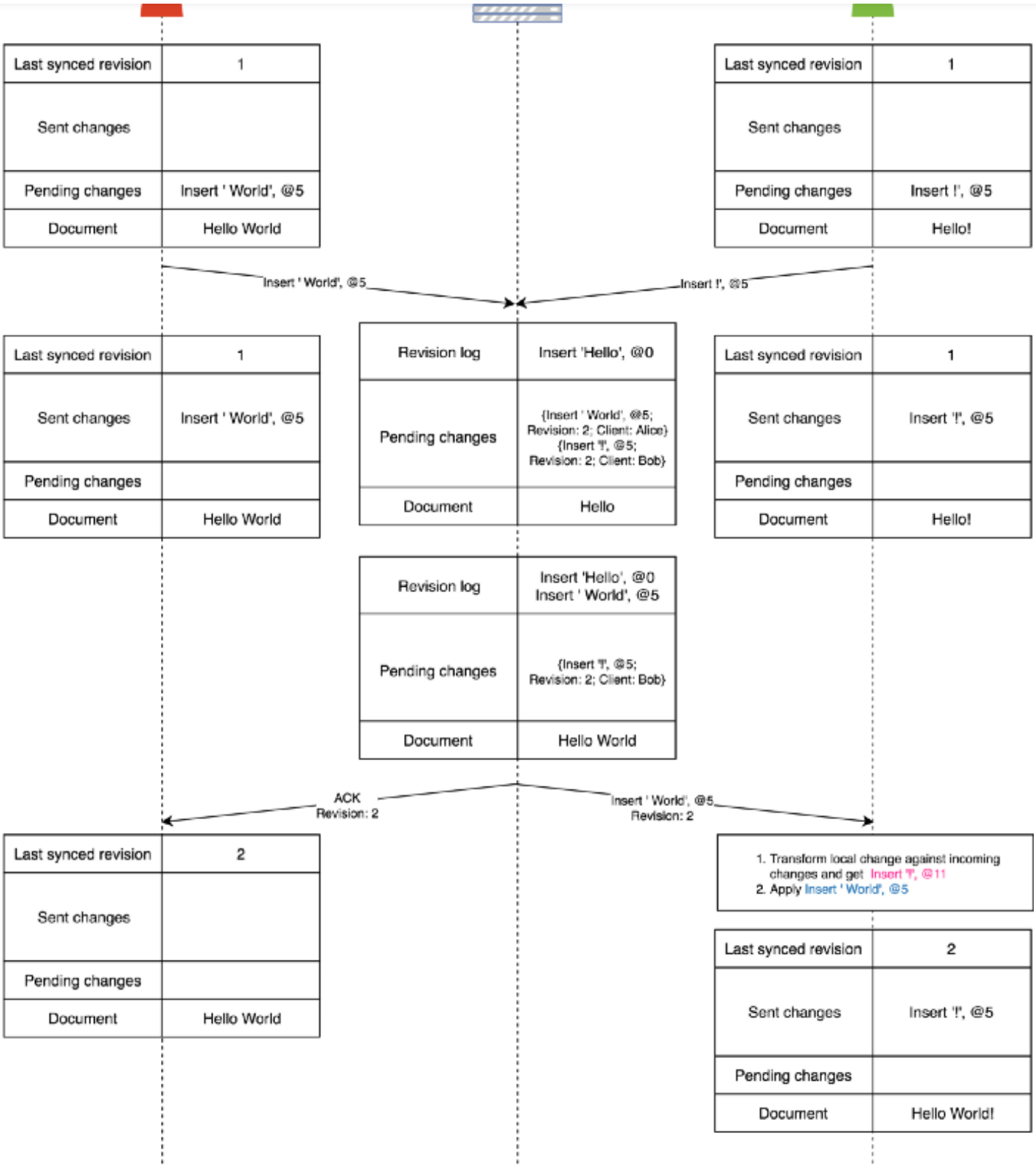


Alice's {Insert ' World', @5} was added to the pending changes but it wasn't sent because her first change has not been acknowledged and **we send only one change at a time**. Note also the server has processed Alice's first change and moved it into revision log. Then it sends an acknowledgement to Alice and propagates her change to Bob:



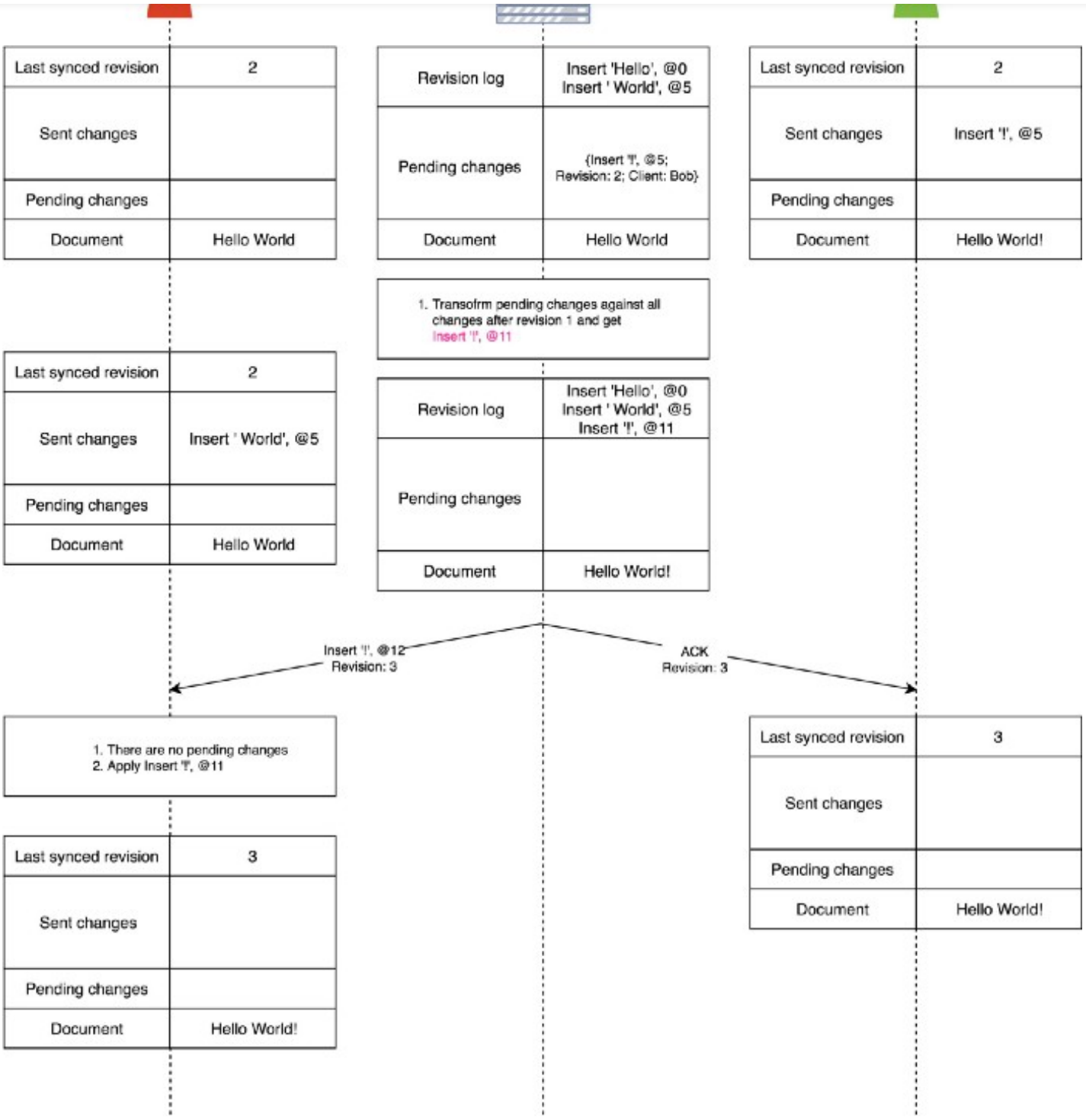
Bob receives the change and applies the transformation function to it and as the result, the index in its pending change was shifted from 0 to 5. Note both Alice and Bob updated their last synced revision to 1. Alice finally removed her first change from sent changes.

Next, both Alice and Bob send their unsent changes to the server:



The server receives Alice's change first so it processes it and sends an acknowledgement to Alice and propagates it to Bob. Bob has to apply again the transformation function and shift his local change to index 11.

Next comes an important moment. The server starts processing Bob's change but because Bob's revision id is obsolete (2 vs actual 3) the server transforms his change against all changes Bob doesn't know about yet and saves it as revision 3.



The process is finished now for Alice, Bob still needs to receive the transformed changed and to send the acknowledgement.

4. Etherpad

Let's have a look at another collaborative editing application which uses OT technique — http://etherpad.org

Etherpad uses a bit different transformation functions — it sends changes to the server as a *changeset* in the following format:

(p1 -> p2)[c_1, c_2, …],

- p2 — length of the document after the change
- c_i — definition if the document:
  if c_i — number or range of numbers then it means indices of *retained* character, or
  if c_i — character or string then it means insertion

Example:

- “” + (0 -> 5)[“Hello”] = “Hello”
- “Hllo World” + (10 -> 14)[0, ‘e’, 1–9, “!!!”] = “Hello World!!!”

A document is formed as a chronological sequence of such changesets applied to an empty document.

We can note that the server can’t just apply changes from clients (it is possible in Google Docs though) because lengths of the documents can be different, for example, if clients A and B started from the same document X of the length n and the do following changesets respectively:

A: (n -> n_a)[…],

B: (n -> n_b)[…]

then B(A(X)) is impossible because B requires a document of length n but it has lengths n_a after A. To solve this problem etherpad introduces so-called *merge* function which takes 2 changesets that apply **to the same document state** and computes a new single changeset. It is required that

$$m(A, B) = m(B, A)$$

It is not very useful to compute m(A, B) when client A receives changes from client B because m(A, B) applies to the state X but A is in the state A(X). Instead of that we should compute A’ and B’ such that

$$B'(A(X)) = A'(B(X)) = m(A, B)(X)$$

$$(i.e.\ we\ need\ to\ transform\ operation)$$

For simplicity let’s define a *follow* function *f*:

$$f(A, B) = B'$$
$$f(B, A) = A'$$
$$f(A, B)(A) = f(B, A)(A) = m(A, B) = m(B, A)$$

- Insertions in B become insertions
- Retain all retained characters in both A and B

Example

$$X = (0 \to 8)[\text{“}baseball\text{”}]$$
$$A = (8 \to 5)[0\text{–}1, \text{“}si\text{”}, 7] \ // == \text{“}basil\text{”}$$
$$B = (8 \to 5)[0, \text{“}e\ \text{”}, 6, \text{“}ow\ \text{”}] \ // == \text{“}below\ \text{”}$$

Then

$$A' = f(B, A) = (5 \to 6)[0, 1, \text{“}si\ \text{”}, 3, 4]$$
$$B' = f(A, B) = (5 \to 6)[0, \text{“}e\text{”}, 2, 3, \text{“}ow\ \text{”}]$$
$$m(A, B) = m(B, A) = A(B') = B(A') = (8 \to 6)[0, \text{“}esiow\ \text{”}]$$

Try to calculate now the final state m(A, B)(X) after merging.

5. Critique of OT

- All document changes must be saved in a set of operations
- OT implementation could be a very complex task, quoting from wikipedia:
  Similarly, Joseph Gentle who is a former Google Wave engineer and an author of the Share.JS library wrote, “Unfortunately, implementing OT sucks. There’s a million algorithms with different tradeoffs, mostly trapped in academic papers. The algorithms are really hard and time consuming to implement correctly. […] Wave took 2 years to write and if we rewrote it today, it would take almost as long to write a second time.”

6. References

- Concurrency Control in Groupware Systems
- What’s different about the new Google Docs: Working together, even apart
- What’s different about the new Google Docs: Conflict resolution
- Understanding and Applying Operational Transformation
- Wikipedia: Operational transformation
- High-latency, low-bandwidth windowing in the Jupiter collaboration system
- What’s different about the new Google Docs: Making collaboration fast

- Google TechTalks: Issues and Experiences in Designing Real-time Collaborative Editing Systems
- Operational Transformation Frequently Asked Questions and Answers

Join Coinmonks Telegram Channel and Youtube Channel get daily Crypto News

Also, Read

- Copy Trading | Crypto Tax Software
- Grid Trading | Crypto Hardware Wallet
- Crypto Telegram Signals | Crypto Trading Bot
- Best Crypto Exchange | Best Crypto Exchange in India
- Best Crypto APIs for Developers
- Best Crypto Lending Platform
- An ultimate guide to Leveraged Token