Google Drive Blog

The latest news and updates from the Google Drive team.

What's different about the new Google Docs: Working together, even apart

Tuesday, September 21, 2010

Editor's Note: In May, we walked through some technical details about what's different in the new Google Docs. Beginning today, we'll dive into the collaboration technology behind Google Docs in three parts, starting with a look at the challenges encountered when building a collaborative application. Tomorrow's post will describe how Google Docs uses an algorithm called operational transformation to merge edits in real time. Finally, on Thursday, we'll dive into the collaboration protocol for sending changes between the editors.

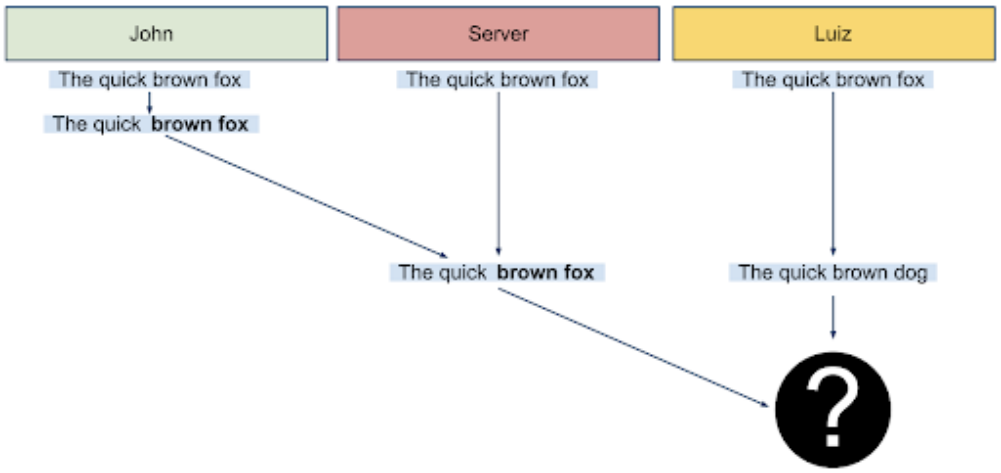
The way people work is changing. Ten years ago, it was too hard to co-author a document, so things took longer, or people just put up with less collaboration. But as our communication tools have become better, it's become more common to have a group of people writing a doc collaboratively.

Collaboration is technically difficult because many people can be making changes to the same content at almost the same time. Since connection speeds aren't instantaneous, when you make a change, you're temporarily creating a local version of the document that is different from the versions

other collaborators see. The core implementation challenge is to make sure that all the editing sessions eventually converge on the same, correct, version of the document.

One approach taken by the old Google documents and by many other collaborative word processors is to compare document versions. Suppose there are two editors: John and Luiz. In the old Google Docs, the server begins with one version of a document and receives an updated version from John. The server finds the differences between its version and John's version and decides out how to merge those two versions as best it can. Then the server sends this merged version to Luiz. If Luiz has changes that have not yet been sent to the server, then he needs to compare the server version with his local version and merge the two versions together. Then Luiz sends this merged local version to the server and the process continues.

But often, this approach doesn't work well. Take the example below. John, Luiz and the server start with the text The quick brown fox. John bolds the words **brown fox**. As he's doing this, Luiz highlights the word fox and replaces it with the word dog. Suppose John's changes arrive at the server first, and then the server sends those changes to Luiz.



The correct way to merge John's style change and Luiz's text substitution is as The quick **brown dog**. But Luiz doesn't have enough information to know what

the correct merge is. From his perspective, The quick **brown fox** dog, The quick **brown** dog, The quick **brown** dog **fox** are all perfectly valid ways of merging the two versions. And that's the problem: if you just compare versions, you can't make sure that changes are merged in the way that an editor would expect.

You can avoid the merging problem by introducing more restrictions on the editors. For example, you could lock paragraphs so that only one editor was ever allowed to type in a single paragraph at a given time. But locking paragraphs isn't a great solution: you're sidestepping the technical challenges by hampering the collaborative editing experience. Plus, it's always possible for two editors to begin editing a paragraph at the same time. In that case, one of the editors will find out that he didn't actually acquire the paragraph lock and any changes that he made while he thought he had the lock will need to be merged (which has all of the above problems) or discarded.

The new version of Google documents does things differently. In the new editor, a document is stored as a series of chronological changes. A change might be something like `{InsertText 'T' @10}`. That particular change was to insert the letter T at the 10th position in the document. A fundamental difference between the new editor and the old one is that instead of computing the changes by comparing document versions, we now compute the

versions

by playing forward the history of changes.

This approach creates a better collaboration experience, because the editors' intentions are never ambiguous. Since we know the revision of each change, we can check what the editor saw when he made that change and we can figure out how to correctly merge that change with any changes that were made since then.

That's it for today. Tomorrow's post will give an overview of the algorithm for

merging changes — **operational transformation**. Even if we know *how* to properly merge changes, we still need to make sure that each editor knows *when* there are changes that need to be merged. This challenge is handled by the **collaboration protocol** which will be the subject of Thursday's post. Together, these technologies create the character-by-character collaboration in Google Docs.

Posted by: John Day-Richter, Software Engineer

Labels: documents , Google Apps Blog , Google Drive Blog

Google

Google · Privacy · Terms

Google Drive Blog

The latest news and updates from the Google Drive team.

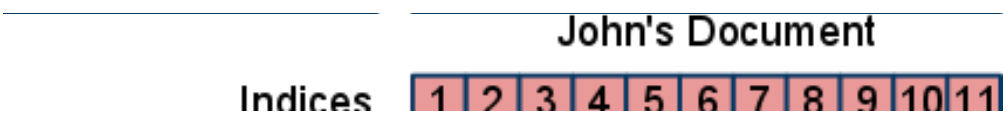
What's different about the new Google Docs: Conflict resolution

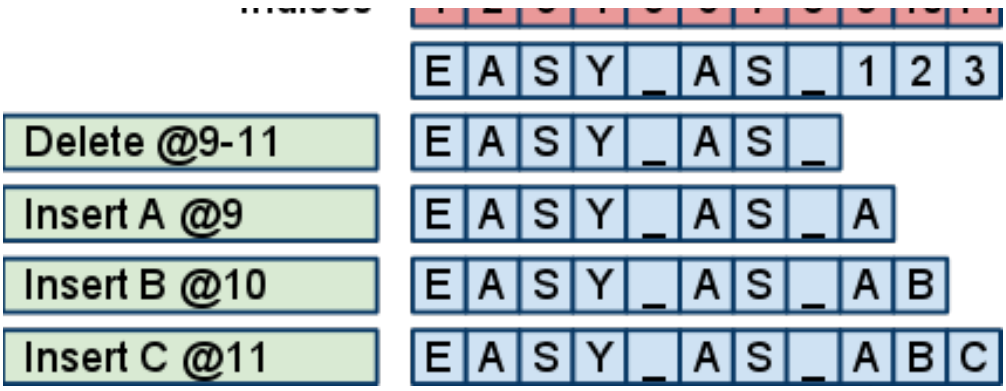
Wednesday, September 22, 2010

Editor's note: This is the second in a series of three posts about the collaboration technology in Google Docs. Yesterday, we explained some of the technical challenges behind real time collaboration.

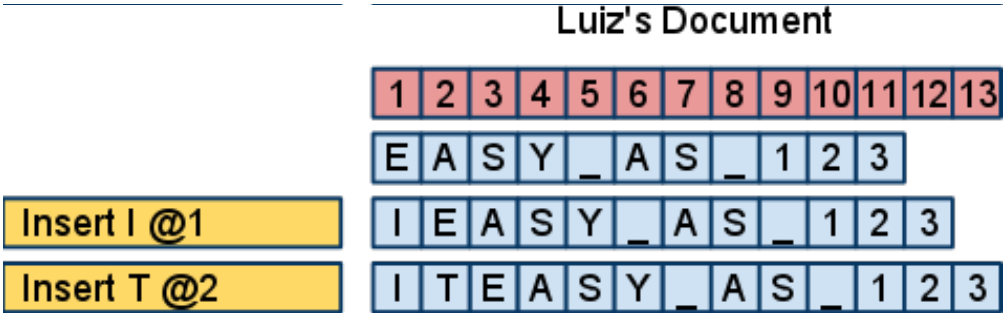
Think of the history of a document as a series of changes. In Google documents, all edits boil down to three basic types of changes: inserting text, deleting text, and applying styles to a range of text. We save your document as a revision log consisting of a list of these changes. When someone edits a document, they're not modifying the underlying characters that represents the document. Instead they are appending their change to the end of the revision log. To display a document, we replay the revision log from the beginning.

To see what these changes look like, suppose that a document edited by John and Luiz initially reads; EASY AS 123. If John (represented by green) changes the document to EASY AS ABC, then he is making four changes:

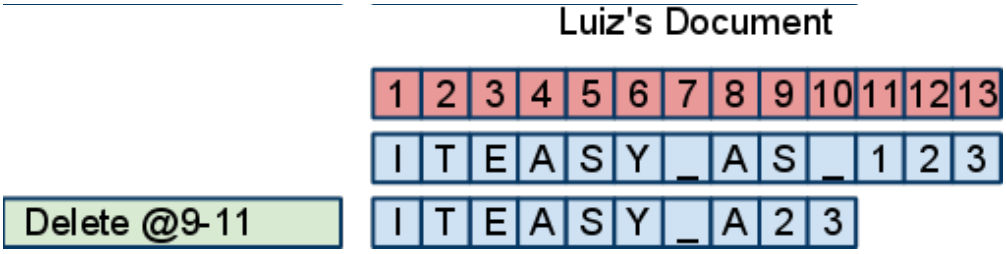




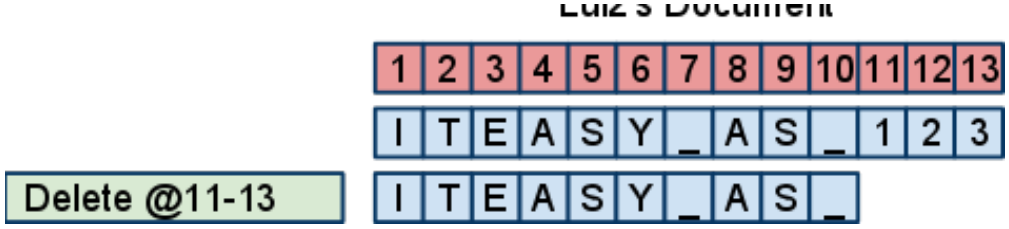
Collaboration is not quite as simple as sending these changes to the other editors because people get out of sync. Suppose as John is typing, Luiz (represented by yellow) begins to change his document to IT'S EASY AS 123. He first inserts the I and the T at the beginning of the document:



Suppose Luiz naively applies John's first change `{DeleteText @9-11}`:



He deleted the wrong characters! Luiz had two characters at the beginning of the doc that John was never aware of. So the location of John's change was wrong relative to Luiz's version of the document. To avoid this problem, Luiz must *transform* John's changes and make them relative to his local document. In this case, when Luiz receives changes from John he needs to know to shift the changes over by two characters to adjust for the IT that Luiz added. Once he does this transformation and applies John's first change, he gets:



Much better. The algorithm that we use to handle these shifts is called operational transformation (OT). If OT is implemented correctly, it guarantees that once all editors have received all changes, everyone will be looking at the same version of the document.

The OT logic in documents must handle all of the different ways that InsertText, DeleteText, and ApplyStyle changes can be paired and transformed against each other. The example above showed DeleteText being transformed against InsertText. To get a feel for how this works, here are a couple more examples of simple transformations:

- Style ranges expand when they are transformed against text insertions: `{ApplyStyle bold @10-20}` transformed against `{InsertText 'ABC' @15}` results in `{ApplyStyle Bold @10-23}`.
- Sometimes changes don't conflict and there's no need to transform anything. For example when a style change is transformed against a different type of style change, there is no conflict: `{ApplyStyle italic @10-20}` transformed against `{ApplyStyle font-color=red @0-30}` results in the same `{ApplyStyle italic @10-20}` because the range of text can be both red and italic simultaneously.

Collaboration in Google Docs consists of sending changes from one editor to the server, and then to the other editors. Each editor transforms incoming changes so that they make sense relative to the local version of the document.

Tomorrow's post will outline the protocol for deciding when each editor uses operational transformation.

Posted by: John Day-Richter, Software Engineer

Labels: documents , Google Apps Blog , Google Drive Blog

Google Drive Blog

The latest news and updates from the Google Drive team.

What's different about the new Google Docs: Making collaboration fast

Thursday, September 23, 2010

This is the final post in a three part series about the collaboration technology in Google Docs. On Tuesday, we explained some of the technical challenges behind real time collaboration. Yesterday, we showed how operational transformation can be used merge editors' changes.

Imagine that you're doing a jigsaw puzzle with a bunch of friends and that everyone is working in the same corner of the puzzle. It's possible to solve a puzzle like this, but it's hard to keep out of each other's way and to make sure that when multiple pieces are added at once, that they all fit together perfectly. Making a document collaborative is a little like that: one challenge is coming up with a method to let multiple people edit in the same area without conflicting edits. A second problem is to ensure that when many changes happen at the same time, each change is merged properly with each other changes. In Google Docs, the first problem is handled by operational transformation and the second problem is handled by the collaboration protocol, which is the subject of this post.

To open a Google document, you need code running in two places: your browser and our servers. We call the code that's running in your browser a client. In the document editor, the client processes all your edits, sends them to the server, and processes other editors' changes when it receives them from the server.

To collaborate in Google Docs, each client keeps track of four pieces of information:

1. The number of the most recent revision sent from the server to the client.
2. Any changes that have been made locally and not yet sent to the server.
3. Any changes that have been made locally, sent to the server, but not yet acknowledged by the server.
4. The current state of the document as seen by that particular editor.

The server remembers three things:

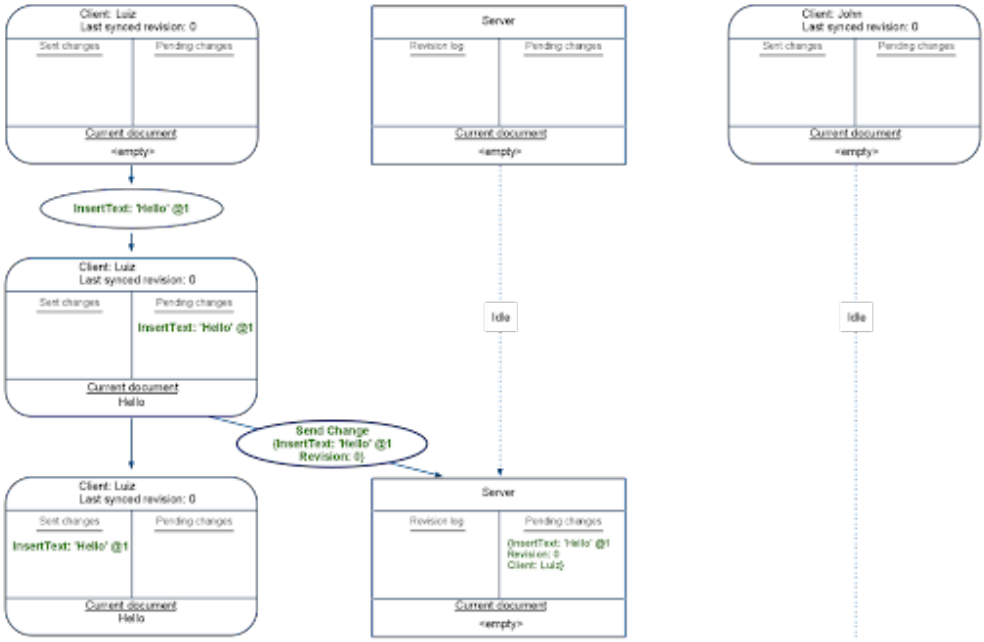
1. The list of all changes that it has received but not yet processed.
2. The complete history of all processed changes (called the revision log)./li>
3. The current state of the document as of the last processed change./li>

By carefully making use of this information, it's possible to design the client-server communication such that all editors are capable of rapidly processing each other's changes in real time. Let's walk through a straightforward example of how client-server communication is handled in a document.

In the diagrams below, the two outer columns represent the editors: Luiz and

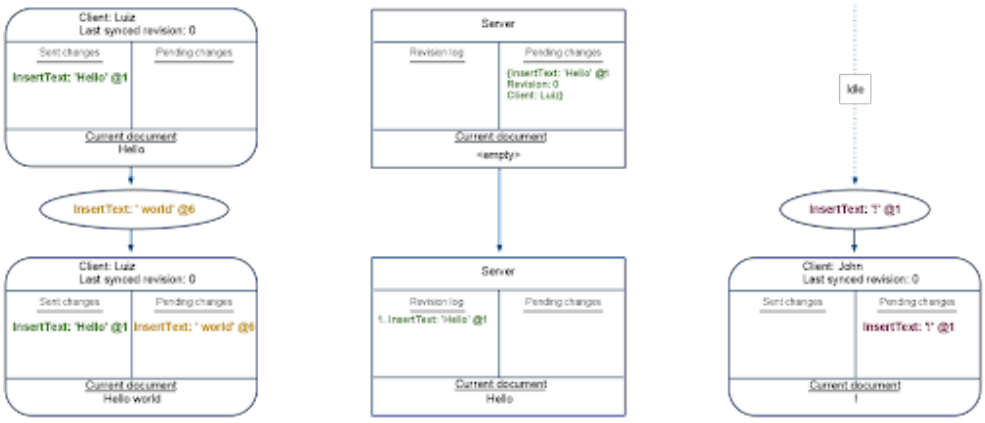
John. The middle column is the server. The oval shapes represent changes inputted by the editors and sent between the clients and the server. The diamonds represent transformations.

Let's say Luiz starts by typing the word Hello at the beginning of the document.



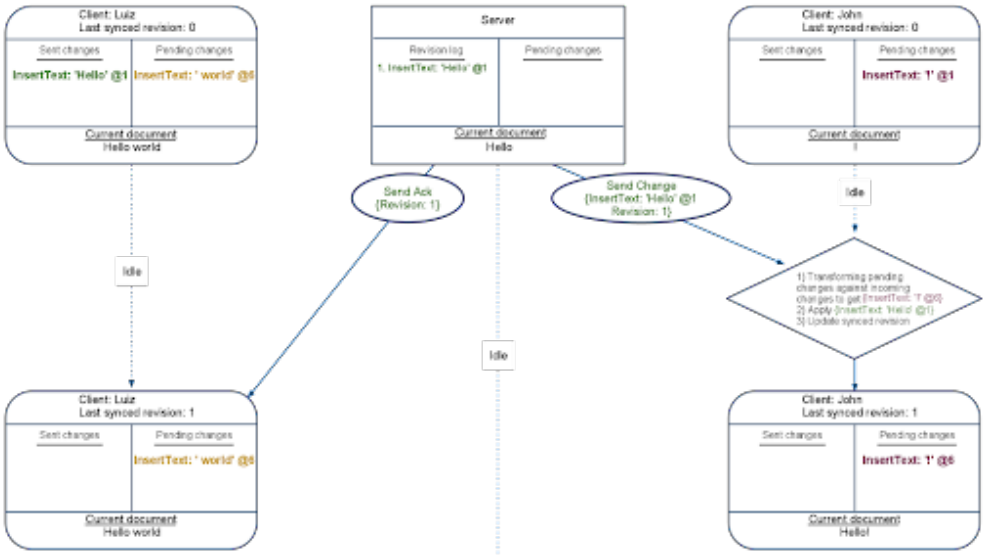
Luiz's client added the edit to his list of pending changes. He then sent the change to the server and moved the change into his list of sent changes.

Luiz continues to type, adding the word world to his document. At the same time, John types an ! in his empty version of the document (remember he has not yet received Luiz's first change).



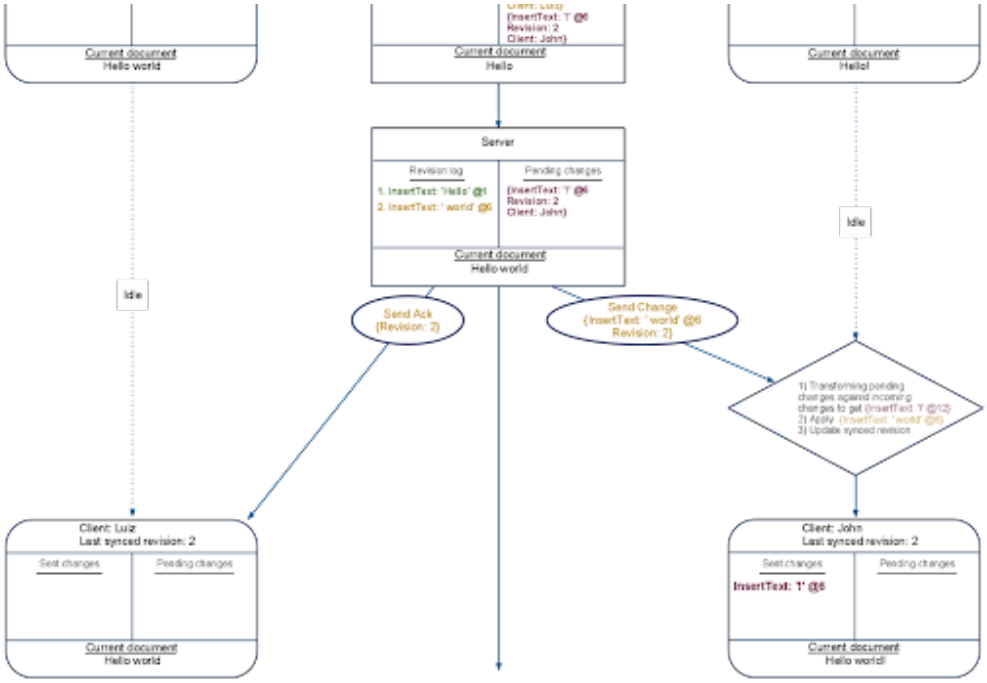
Luiz's `{InsertText ' world' @6}` change was placed in the pending list

and wasn't sent to the server because we never send more than one pending change at a time. Until Luiz recieves an acknowledgement of his first change, his client will keep all new changes in the pending list. Also notice that the server stored Luiz's first change in its revision log. Next, the server will send John a message containing Luiz's first change and it will send Luiz a message acknowledging that it has processed that first change.



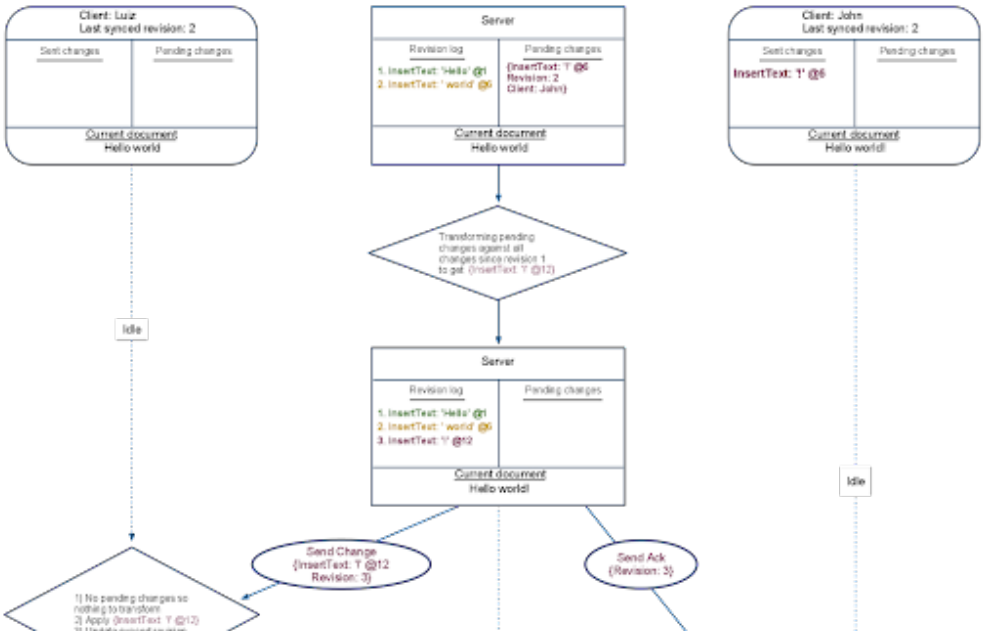
John received Luiz's edit from the server and used operational transformation (OT) to transform it against his pending `{InsertText '!' @1}` change. The result of the transformation was to shift the location of John's pending change by 5 to make room at the beginning of the document for Luiz's Hello. Notice that both Luiz and John updated their last synced revision numbers to 1 when they received the messages from the server. Lastly, when Luiz received the acknowledgement of his first change, he removed that first change from the list of sent changes.

Next, both Luiz and John are going to send their unsent changes to the server.



The server got Luiz's change before John's so it processed that change first. An acknowledgement of the change was sent to Luiz. The change itself was sent to John, where his client transformed it against his still pending `{InsertText '!' @1}` change.

What comes next is important. The server received John's pending change, a change that John believes should be Revision 2. But the server has already committed a Revision 2 to the revision log. The server will use OT to transform John's change so that it can be stored as Revision 3.



The first thing the server did, was to transform John's sent change against all the changes that have been committed since the last time John synced with the server. In this case, it transformed John's change against Luiz's `{InsertText ' world' @6}`. The result shifted the index of John's change over by 6. This shift is identical to the transformation John's client made when it first received Luiz's `{InsertText 'Hello' @1}`.

The example above ends with Luiz and John receiving John's change and the acknowledgement of that change respectively. At this point the server and both editors are looking at the same document — Hello world!.

The main advantages of this collaboration protocol are:

1. Collaboration is fast. At all times, every editor can optimistically apply their own changes locally without waiting for the server to acknowledge those changes. This means that the speed or reliability of your network connection doesn't influence how fast you can type.
2. Collaboration is accurate. There is always enough information for each client to merge collaborators' changes in the same deterministic way.
3. Collaboration is efficient. The information that is sent over the network is always the bare minimum needed to describe what changed.
4. Collaboration complexity is constant. The server does not need to

know anything about the state of each client. Therefore, the complexity of processing changes does not increase as you add more editors.

5. Collaboration is distributed. Only the server needs to be aware of the document's history and only the clients need to be aware of uncommitted changes. This division spreads the workload required to support real time collaboration between all the parties involved.

When we switched to the new document editor, we moved from a very simple collaboration algorithm based on comparing versions to a much more sophisticated algorithm powered by operational transformation and the protocol described above. The results are dramatic: there are no more collaboration conflicts and editors can see each other's changes as they happen, character-by-character.

Well that's all folks: we hope by reading this series you learned a bit more about what's under the hood in Google Docs, and the kinds of things you need to think about to make a fast collaboration experience. You can try collaboration yourself, without signing in, by visiting the Google Docs demo.

Posted by: John Day-Richter, Software Engineer

Labels: documents , Google Apps Blog , Google Drive Blog