Search

Code Commit

Understanding and Applying Operational Transformation

17
May
2010

Almost exactly a year ago, Google made one of the most remarkable press releases in the Web 2.0 era. Of course, by "press release", I actually mean keynote at their own conference, and by "remarkable" I mean potentially-transformative and groundbreaking. I am referring of course to the announcement of Google Wave, a real-time collaboration tool which has been in open beta for the last several months.

For those of you who don't know, Google Wave is a collaboration tool based on real-time, simultaneous editing of documents via a mechanism known as "operational transformation". Entities which appear as messages in the Wave client are actually "waves". Within each "wave" is a set of "wavelets", each of which contains a set of documents. Individual documents can represent things like messages, conversation structure (which reply goes where, etc), spell check metadata and so on. Documents are composed of well-formed XML with an implicit root node. Additionally, they carry special metadata known as "annotations" which are (potentially-overlapping) key/value ranges which span across specific regions of the document. In the Wave message schema, annotations are used to represent things like bold/italic/underline/strikethrough formatting, links, caret position, the conversation title and a host of other things. An example document following the Wave message schema might look something like this:

```
<body>
  <line/>Test message
  <line/>
  <line/>Lorem ipsum dolor sit amet.
</body>
```

(assuming the following annotations):

- style/font-weight -> bold
- style/font-style -> italic
- link/manual -> http://www.google.com

You will notice that the annotations for `style/font-style` and `link/manual` actually overlap. This is perfectly acceptable in Wave's document schema. The resulting rendering would be something like this:

> **Test message**
>
> *Lorem ipsum dolor* sit amet.

The point of all this explaining is to give you at least a passing familiarity with the Wave document schema so that I can safely use its terminology in the article to come. See, Wave itself is not nearly so interesting as the idea upon which it is based. As mentioned, every document in Wave is actually just raw XML with some ancillary annotations. As far as the Wave server is concerned, you can stuff whatever data you want in there, just so long as it's well-formed. It just so happens that Google chose to implement a communications tool on top of this data backend, but they could have just as easily implemented something more esoteric, like a database or a windowing manager.

The key to Wave is the mechanism by which we interact with these documents: operational transformation. Wave actually doesn't allow you to get access to a document as raw XML or anything even approaching it. Instead, it demands that all of your access to the document be performed in terms of operations. This has two consequences: first, it allows for some really incredible collaborative tools like the Wave client; second, it makes it *really* tricky to implement any sort of Wave-compatible service. Given the fact that I've been working on Novell Pulse (which is exactly this sort of service), and in light of the fact that Google's documentation on the subject is sparing at best, I thought I would take some time to clarify this critical piece of the puzzle. Hopefully, the information I'm about to present will make it easier for others attempting to interoperate with Wave, Pulse and the (hopefully) many OT-based systems yet to come.

Operations

Intuitively enough, the fundamental building block of operational transforms are operations themselves. An operation is exactly what it sounds like: an action which is to be performed on a document. This action could be inserting or deleting characters, opening (and closing!) an XML element, fiddling with annotations, etc. A single operation may actually perform many of these actions. Thus, an operation is actually made up of a sequence of operation *components*, each of which performs a particular action with respect to the *cursor* (not to be confused with the *caret*, which is specific to the client editor and not at all interesting at the level of OT).

There are a number of possible component types. For example:

- insertCharacters — Inserts the specified string at the current index
- deleteCharacters — Deletes the specified string from the current index
- openElement — Creates a new XML open-tag at the current index
- deleteOpenElement — Deletes the specified XML open-tag from the current index
- closeElement — Closes the first currently-open tag at the current index
- deleteCloseElement — Deletes the XML close-tag at the current index
- annotationBoundary — Defines the *changes* to any annotations (starting or ending) at the current index
- retain — Advances the index a specified number of items

Wave's OT implementation actually has even more component types, but these are the important ones. You'll notice that every component has something to do with the cursor index. This concept is central to Wave's OT implementation. Operations are effectively a stream of components, each of which defines an action to be performed which effects the content, the cursor or both. For example, we can encode the example document from earlier as follows:

1. openElement(`'body'`)
2. openElement(`'line'`)
3. closeElement()
4. annotationBoundary(`startKeys: ['style/font-weight'], startValues: ['bold']`)
5. insertCharacters(`'Test message'`)
6. annotationBoundary(`endKeys: ['style/font-weight']`)
7. openElement(`'line'`)
8. closeElement()
9. annotationBoundary(`startKeys: ['style/font-style'], startValues: ['italic']`)
10. openElement(`'line'`)
11. closeElement()
12. insertCharacters(`'Lorem '`)
13. annotationBoundary(`startKeys: ['link/manual'], startValues: ['http://www.google.com']`)

14. insertCharacters(`'ipsum'`)
15. annotationBoundary(`endKeys: ['style/font-style']`)
16. insertCharacters(`' dolor'`)
17. annotationBoundary(`endKeys: ['link/manual']`)
18. insertCharacters(`' sit amet.'`)
19. closeElement()

Obviously, this isn't the most streamlined way of referring to a document's content for a human, but a stream of discrete components like this is *perfect* for automated processing. The real utility of this encoding though doesn't become apparent until we look at operations which only encode a partial document; effectively performing a particular mutation. For example, let's follow the advice of *Strunk and White* and capitalize the letter 'm' in our title of 'Test message'. What we want to do (precisely-speaking) is delete the 'm' and insert the string 'M' at its previous location. We can do that with the following operation:

1. retain(`8`)
2. deleteCharacters(`'m'`)
3. insertCharacters(`'M'`)
4. retain(`38`)

Instead of adding content to the document at ever step, most of this operation actually leaves the underlying document untouched. In practice, retain() tends to be the most commonly used component by a wide margin. The trick is that every operation *must* span the full width of the document. When evaluating this operation, the cursor will start at index 0 and walk forward through the existing document and the incoming operation one item at a time. Each XML tag (open or close) counts as a single item. Characters are also single items. Thus, the entire document contains 47 items.

Our operation above cursors harmlessly over the first eight items (the `<body>` tag, the `<line/>` tag and the string `'Test '`). Once it reaches the `'m'` in `'message'`, we stop the cursor and perform a mutation. Specifically, we're using the deleteCharacters() component to remove the `'m'`. This component doesn't move the cursor, so we're still sitting at index `8`. We then use the insertCharacters() component to add the character `'M'` at precisely our currently location. This time, some new characters have been inserted, so the cursor advances to the end of the newly-inserted string (meaning that we are now at index `9`). This is intuitive because we don't want to have to retain() over the text we just inserted. We do however want to retain() over the remainder of the document, seeing as we don't need to do anything else. The final rendered document looks like the following:

> **Test Message**
>
> *Lorem ipsum dolor* sit amet.

Composition

One of Google's contributions to the (very old) theory behind operational transformation is the idea of operation composition. Because Wave operations are these nice, full-span sequences of discrete components, it's fairly easy to take two operations which span the same length and merge them together into a single operation. The results of this action are really quite intuitive. For example, if we were to compose our document operation (the first example above) with our `'m'`-changing operation (the second example), the resulting operation would be basically the same as the original document operation, except that instead of inserting the text `'Test message'`, we would insert `'Test Message'`. In composing the two operations together, all of the retains have disappeared and any contradicting components (e.g. a delete and an insert) have been directly merged.

Composition is extremely important to Wave's OT as we will see once we start looking at client/server asymmetry. The important thing to notice now is the fact that composed operations *must* be fundamentally compatible. Primarily, this means that the two operations must span the same number of indexes. It also means that we cannot compose an operation which consists of only a text insert with an operation which attempts to delete an XML element. Obviously, that's not going to work. Wave's `Composer` utility takes care of validating both the left and the right operation to ensure that they are compatible as part of the composition process.

Please also note that composition is *not* commutative; ordering is significant. This is also quite intuitive. If you type the character `a` and *then* type the character `b`, the result is quite different than if you type the character `b` and *then* type the character `a`.

Transformation

Here's where we get to some of the really interesting stuff and the motivation behind all of this convoluted representational baggage. Operational Transformation, at its core, is an *optimistic* concurrency control mechanism. It allows two editors to modify the same section of a document at the same time without conflict. Or rather, it provides a mechanism for sanely resolving those conflicts so that neither user intervention nor locking become necessary.

This is actually a harder problem than it sounds. Imagine that we have the following document (represented as an operation):

1. insertCharacters(`'go'`)

Now imagine that we have two editors with their cursors positioned at the end of the document. They *simultaneously* insert a `t` and `a` character (respectively). Thus, we will have two operations sent to the server. The first will retain 2 items and insert a `t`, the second will retain 2 items and insert `a`. Naturally, the server needs to enforce atomicity of edits at some point (to avoid race conditions during I/O), so one of these operations will be applied first. However, as soon as either one of these operations is applied, the retain for the other will become invalid. Depending on the ordering, the text of the resulting document will either be `'goat'` or `'gota'`.

In and of itself, this isn't really a problem. After all, any asynchronous server needs to make decisions about ordering at some point. However, issues start to crop up as soon as we consider relaying operations from one client to the other. Client A has already applied its operation, so its document text will be `'got'`. Meanwhile, client B has already applied *its* operation, and so its document text is `'goa'`. Each client needs the operation from the other in order to have any chance of converging to the same document state.

Unfortunately, if we naïvely send A's operation to B and B's operation to A, the results will *not* converge:

- `'got'` + (retain(`2`); insertCharacters(`'a'`) = `'goat'`
- `'goa'` + (retain(`2`); insertCharacters(`'t'`) = `'gota'`

Even discounting the fact that we have a document size mismatch (our operations each span 2 indexes, while their target documents have width 3), this is obviously not the desired behavior. Even though our server may have a sane concept of consistent ordering, our clients obviously need some extra hand-holding. Enter OT.

What we have here is a simple one-step diamond problem. In the theoretical study of OT, we generally visualize this situation using diagrams like the following:

The way you should read diagrams like this is as a graphical representation of operation application on two documents at the same time. Client operations move the document to the left. Server operations move the document to the right. Both client and server operations move the document

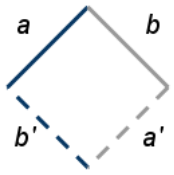
downward. Thus, diagrams like these let us visualize the application of operations in a literal "state space". The dark blue line shows the client's path through state space, while the gray line shows the server's. The vertices of these paths (not explicitly rendered) are points in state space, representing a particular state of the document. When both the client and the server line pass through the same point, it means that the content of their respective documents were in sync, at least at that particular point in time.

So, in the diagram above, operation *a* could be client A's operation (retain(2); insertCharacters('t')) and operation *b* could be client B's operation. This is of course assuming that the server chose B's operation as the "winner" of the race condition. As we showed earlier, we cannot simply naïvely apply operation *a* on the server and *b* on the client, otherwise we could derive differing document states (`'goat'` vs `'gota'`). What we need to do is automatically adjust operation *a* with respect to *b* and operation *b* with respect to *a*.

We can do this using an operational transform. Google's OT is based on the following mathematical identity:

$$\text{xform}(a, b) = (a', b'), \text{ where } b' \circ a \equiv a' \circ b$$

In plain English, this means that the `transform` function takes two operations, one server and one client, and produces a pair of operations. These operations can be applied to their counterpart's end state to produce exactly the same state when complete. Graphically, we can represent this by the following:



Thus, on the client-side, we receive operation *b* from the server, pair it with *a* to produce *(a', b')*, and then compose *b'* with *a* to produce our final document state. We perform an analogous process on the server-side. The mathematical definition of the `transform` function guarantees that this process will produce the *exact* same document state on both server and client.

Coming back to our concrete example, we can finally solve the problem of `'goat'` vs `'gota'`. We start out with the situation where client A has applied operation *a*, arriving at a document text of `'got'`. It now receives operation *b* from the server, instructing it to retain over 2 items and insert character `'a'`. However, before it applies this operation (which would obviously result in the wrong document state), it uses operational transformation to derive operation *b'*. Google's OT implementation will resolve the conflict between `'t'` and `'a'` in favor of the server. Thus, `b'` will consist of the following components:

1. retain(2)
2. insertCharacters('a')
3. retain(1)

You will notice that we no longer have a document size mismatch, since that last retain() ensures that the cursor reaches the end of our length-3 document state (`'got'`).

Meanwhile, the server has received our operation *a* and it performs an analogous series of steps to derive operation *a'*. Once again, Google's OT must resolve the conflict between `'t'` and `'a'` in the *same* way as it resolved the conflict for client A. We're trying to apply operation *a* (which inserts the `'t'` character at position 2) to the server document state, which is currently `'goa'`. When we're done, we must have the exact same document content as client A following the application of *b'*. Specifically, the server document state must be `'goat'`. Thus, the OT process will produce the operation *a'* consisting of the following components:

1. retain(3)
2. insertCharacters('t')

Client A applies operation *b'* to its document state, the server applies operation *a'* to its document state, and they *both* arrive at a document consisting of the text `'goat'`. Magic!

It is very important that you really understand this process. OT is all about the `transform` function and how it behaves in this exact situation. As it turns out, this is *all* that OT does for us in and of itself. Operational transformation is really just a concurrency primitive. It doesn't solve every problem with collaborative editing of a shared document (as we will see in a moment), but it does solve this problem very well.

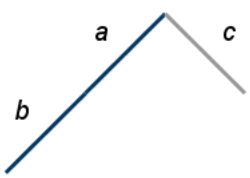
One way to think of this is to keep in mind the "diamond" shape shown in the above diagram. OT solves a very simple problem: given the top two sides of the diamond, it can derive the bottom two sides. In practice, often times we only want one side of the box (e.g. client A only needs operation *b'*, it doesn't need *a'*). However, OT *always* gives us both pieces of the puzzle. It "completes" the diamond, so to speak.

Compound OT

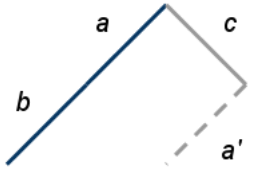
So far, everything I have presented has come pretty directly from the whitepapers on waveprotocol.org. However, contrary to popular belief, this is *not* enough information to actually go out and implement your own collaborative editor or Wave-compatible service.

The problem is that OT doesn't really do all that much in and of itself. As mentioned above, OT solves for two sides of the diamond in state space. It *only* solves for two sides of a simple, one-step diamond like the one shown above. Let me say it a third time: the case shown above is the *only* case which OT handles. As it turns out, there are other cases which arise in a client/server collaborative editor like Google Wave or Novell Pulse. In fact, *most* cases in practice are much more complex than the one-step diamond.

For example, consider the situation where the client performs *two* operations (say, by typing two characters, one after the other) while at the same time the server performs one operation (originating from another client). We can diagram this situation in the following way:



So we have two operations in the client history, *a* and *b*, and only one operation in the server history, *c*. The client is going to send operations *a* and *b* to the server, presumably one after the other. The first operation (*a*) is no problem at all. Here we have the simple one-step diamond problem from above, and as well know, OT has no trouble at all in resolving this issue. The server transforms *a* and *c* to derive operation *a'*, which it applies to its current state. The resulting situation looks like the following:



Ok, so far so good. The server has successfully transformed operation *a* against *c* and applied the resulting *a'* to its local state. However, the moment we move on to operation *b*, disaster strikes. The problem is that the server receives operation *b*, but it has nothing against which to transform it!

Remember, OT *only* solves for the bottom two sides of the diamond given the top two sides. In the case of the first operation (*a*), the server had both top sides (*a* and *c*) and thus OT was able to derive the all-important *a'*. However, in this case, we only have one of the sides of the diamond (*b*); we don't have the server's half of the equation because the server never performed such an operation!

In general, the problem we have here is caused by the client and server diverging by more than one step. Whenever we get into this state, the OT becomes more complicated because we effectively need to transform incoming operations (e.g. *b*) against operations which *never happened!* In this case, the phantom operation that we need for the purposes of OT would take us from the tail end of *a* to the tail end of *a'*. Think of it like a "bridge" between client state space and server state space. We need this bridge, this second half of the diamond, if we are to apply OT to solve the problem of transforming *b* into server state space.

Operation Parentage

In order to do this, we need to add some metadata to our operations. Not only do our operations need to contain their components (retain, etc), they also must maintain some notion of parentage. We need to be able to determine exactly what state an operation requires for successful application. We will then use this information to detect the case where an incoming operation is parented on a state which is not in our history (e.g. *b* on receipt by the server).

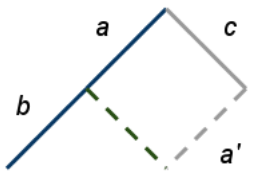
For the record, Google Wave uses a monotonically-increasing scalar version number to label document states and thus, operation parents. Novell Pulse does the exact same thing for compatibility reasons, and I recommend that anyone attempting to build a Wave-compatible service follow the same model. However, I personally think that compound OT is a lot easier to understand if document states are labeled by a hash of their contents.

This scheme has some very nice advantages. Given an operation (and its associated parent hash), we can determine instantly whether or not we have the appropriate document state to apply said operation. Hashes also have the very convenient property of converging exactly when the document states converge. Thus, in our one-step diamond case from earlier, operations *a* and *b* would be parented off of the same hash. Operation *b'* would be parented off of the hash of the document resulting from applying *a* to the initial document state (and similarly for *a'*). Finally, the point in state space where the client and server converge once again (after applying their respective operations) will have a single hash, as the document states will be synchronized. Thus, any further operations applied on either side will be parented off of a correctly-shared hash.

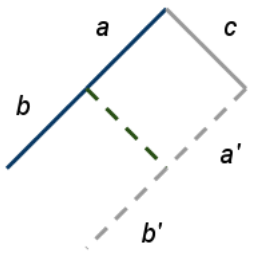
Just a quick terminology note: when I say "parent hash", I'm referring to the hash of the document state *prior* to applying a particular operation. When I say "parent operation" (which I probably will from time to time), I'm referring to the hash of the document state which results from applying the "parent operation" to its parent document state. Thus, operation *b* in the diagram above is parented off of operation *a* which is parented off of the same hash as operation *c*.

Compound OT

Now that our operations have parent information, our server is capable of detecting that operation *b* is not parented off of any state in its history. What we need to do is derive an operation which will take us from the parent of *b* to some point in server state-space. Graphically, this operation would look something like the following (rendered in dark green):



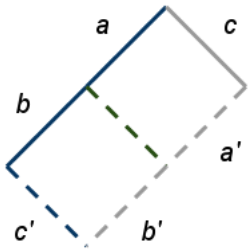
Fortunately for us, this operation is fairly easy to derive. In fact, we already derived and subsequently threw it away! Remember, OT solves for *two* sides of the diamond. Thus, when we transformed *a* against *c*, the resulting operation pair consisted of *a'* (which we applied to our local state) and another operation which we discarded. That operation is precisely the operation shown in green above. Thus, all we have to do is re-derive this operation and use it as the second top side of the one-step diamond. At this point, we have all of the information we need to apply OT and derive *b'*, which we can apply to our local state:



At this point, we're *almost* done. The only problem we have left to resolve is the application of operation *c* on the client. Fortunately, this is a fairly easy thing to do; after all, *c* is parented off of a state which the client has in its history, so it should be able to directly apply OT.

The one tricky point here is the fact that the client must transform *c* against not one but *two* operations (*a* and *b*). Fortunately, this is fairly easy to do. We could apply OT twice, deriving an intermediary operation in the first step (which happens to be exactly equivalent to the green intermediary operation we derived on the server) and then transforming that operation against *b*. However, this is fairly inefficient. OT is fast, but it's still *O(n log n)*. The better approach is to first compose *a* with *b* and then transform *c* against the composition of the two operations. Thanks to Google's careful definition of operation composition, this is guaranteed to produce the same operation as we would have received had we applied OT in two separate steps.

The final state diagram looks like the following:



Client/Server Asymmetry

Technically, what we have here is enough to implement a fully-functional client/server collaborative editing system. In fact, this is very close to what was presented in the 1995 paper on the Jupiter collaboration system. However, while this approach is quite functional, it isn't going to work in practice.

The reason for this is in that confusing middle part where the server had to derive an intermediary operation (the green one) in order to handle operation *b* from the client. In order to do this, the server needed to hold on to operation *a* in order to use it a second time in deriving the intermediary operation. Either that, or the server would have needed to speculatively retain the intermediary operation when it was derived for the first time during the transformation of *a* to *a'*. Now, this may sound like a trivial point, but consider that the server must maintain this sort of information essentially indefinitely for *every* client which it handles. You begin to see how this could become a serious scalability problem!

In order to solve this problem, Wave (and Pulse) imposes a very important constraint on the operations incoming to the server: any operation received by the server *must* be parented on some point in the server's history. Thus, the server would have rejected operation *b* in our example above since it did not branch from any point in server state space. The parent of *b* was *a*, but the server didn't have *a*, it only had *a'* (which is clearly a different point in state space).

Of course, simply rejecting any divergence which doesn't fit into the narrow, one-step diamond pattern is a bit harsh. Remember that practically, *almost all* situations arising in collaborative editing will be multi-step divergences like our above example. Thus, if we naïvely rejected anything which didn't fit into the one-step mold, we would render our collaborative editor all-but useless.

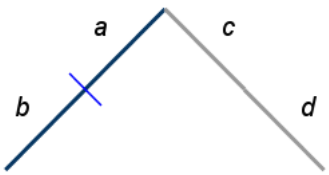
The solution is to move all of the heavy lifting onto the client. We don't want the server to have to track every single client as it moves through state space since there could be thousands (or even millions) of clients. But if you think about it, there's really no problem with the client tracking the *server* as it moves through state space, since there's never going to be any more than one (logical) server. Thus, we can offload most of the compound OT work onto the client side.

Before it sends any operations to the server, the client will be responsible for ensuring those operations are parented off of some point in the server's history. Obviously, the server may have applied some operations that the client doesn't know about yet, but that's ok. As long as any operations sent by the client are parented off of *some* point in the server's history, the server will be able to transform that incoming operation against the composition of anything which has happened since that point without tracking any history other than its own. Thus, the server never does anything more complicated than the simple one-step diamond divergence (modulo some operation composition). In other words, the server can always directly apply OT to incoming operations, deriving the requisite operation extremely efficiently.

Unfortunately, not all is sunshine and roses. Under this new regime, the client needs to work twice as hard, translating its operations into server state space and (correspondingly) server operations back into its state space. We haven't seen an example of this "reverse" translation (server to client) yet, but we will in a moment.

In order to maintain this guarantee that the client will never send an operation to the server which is not parented on a version in server state space, we need to impose a restriction on the client: we can never send more than one operation at a time to the server. This means that as soon as the client sends an operation (e.g. *a* in the example above), it must wait on sending *b* until the server acknowledges *a*. This is necessary because the client needs to somehow translate *b* into server state space, but it can't just "undo" the fact that *b* is parented on *a*. Thus, wherever *b* eventually ends up in server state space, it has to be a descendant of *a'*, which is the server-transformed version of *a*. Literally, we don't know *where* to translate *b* into until we know *exactly* where *a* fits in the server's history.

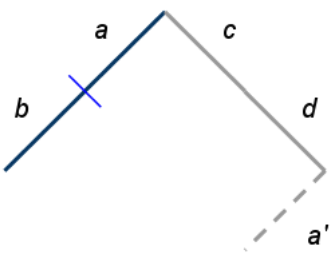
To help shed some light into this rather confusing scheme, let's look at an example:



In this situation, the client has performed two operations, *a* and *b*. The client immediately sends operation *a* to the server and buffers operation *b* for later transmission (the lighter blue line indicates the buffer boundary). Note that this buffering in no way hinders the *application* of local operations. When the user presses a key, we want the editor to reflect that change *immediately*, regardless of the buffer state. Meanwhile, the server has applied two other operations, *c* and *d*, which presumably come from other clients. The server still hasn't received our operation *a*.

Note that we were able to send *a* immediately because we are preserving every bit of data the server sends us. We still don't know about *c* and *d*, but we do know that the last time we heard from the server, it was at the same point in state space as we were (the parent of *a* and *c*). Thus, since *a* is already parented on a point in server state space, we can just send it off.

Now let's fast-forward just a little bit. The server receives operation *a*. It looks into its history and retrieves whatever operations have been applied since the parent of *a*. In this case, those operations are *c* and *d*. The server then composes *c* and *d* together and transforms *a* against the result, producing *a'*.



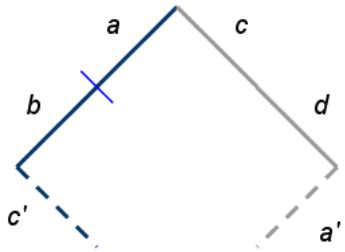
After applying *a'*, the server broadcasts the operation to all clients, including the one which originated the operation. This is a very important design feature: whenever the server applies a transformed operation, it sends that operation off to all of its clients without delay. As long as we can guarantee

strong ordering in the communication channels between the client and the server (and often we can), the clients will be able to count on the fact that they will receive operations from the server in *exactly* the order in which the server applied them. Thus, they will be able to maintain a locally-inferred copy of the server's history.

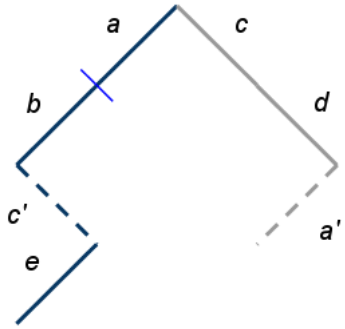
This also means that our client is going to receive *a'* from the server just like any other operation. In order to avoid treating our own transformed operations as if they were new server operations, we need some way of identifying our own operations and treating them specially. To do this, we add another bit of metadata to the operation: a locally-synthesized unique ID. This unique ID will be attached to the operation when we send it to the server and *preserved* by the server through the application of OT. Thus, operation *a'* will have the same ID as operation *a*, but a very different ID from operations *c* and *d*.

With this extra bit of metadata in place, clients are able to distinguish their own operations from others sent by the server. Non-self-initiated operations (like *c* and *d*) must be translated into client state space and applied to the local document. Self-initiated operations (like *a'*) are actually server acknowledgements of our currently-pending operation. Once we receive this acknowledgement, we can flush the client buffer and send the pending operations up to the server.

Moving forward with our example, let's say that the client receives operation *c* from the server. Since *c* is already parented on a version in our local history, we can apply simple OT to transform it against the composition of *a* and *b* and apply the resulting operation to our local document:



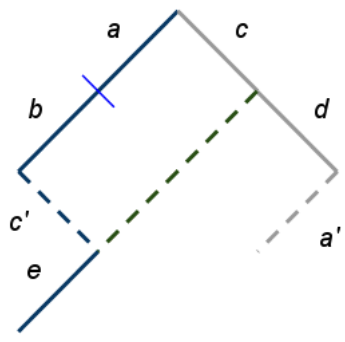
Of course, as we always need to keep in mind, the client is a live editor which presumably has a real person typing madly away, changing the document state. There's nothing to prevent the client from creating *another* operation, parented off of *c'* which pushes it even further out of sync with the server:



This is really getting to be a bit of a mess! We've only sent one of our operations to the server, we're trying to buffer the rest, but the server is trickling in more operations to confuse things and we still haven't received the acknowledgement for our very first operation! As it turns out, this is the most complicated case which can ever arise in a Wave-style collaborative editor. If we can nail this one, we're good to go.

The first thing we need to do is figure out what to do with *d*. We're going to receive that operation before we receive *a'*, and so we really need to figure out how to apply it to our local document. Once again, the problem is that the incoming operation (*d*) is not parented off of any point in our state space, so OT can't help us directly. Just as with *b* in our fundamental compound OT example from earlier, we need to infer a "bridge" between server state space and client state space. We can then use this bridge to transform *d* and slide it all the way down into position at the end of our history.

To do this, we need to identify conceptually what operation(s) would take us from the parent of *d* to the the most recent point in our history (after applying *e*). Specifically, we need to infer the green dashed line in the diagram below. Once we have this operation (whatever it is), we can compose it with *e* and get a single operation against which we can transform *d*.



The first thing to recognize is that the inferred bridge (the green dashed line) is going to be composed exclusively of client operations. This is logical as we are attempting to translate a server operation, so there's no need to transform it against something which the server already has. The second thing to realize is that this bridge is traversing a line parallel to the composition of *a* and *b*, just "shifted down" exactly one step. To be precise, the bridge is what we would get if we composed *a* and *b* and then transformed the result against *c*.

Now, we could try to detect this case specifically and write some code which would fish out *a* and *b*, compose them together, transform the result against *c*, compose the result of *that* with *e* and finally transform *d* against the final product, but as you can imagine, it would be a mess. More than that, it would be dreadfully inefficient. No, what we want to do is proactively maintain a bridge which will always take us from the absolute latest point in server state space (that we know of) to the absolute latest point in client state space. Thus, whenever we receive a new operation from the server, we can directly transform it against this bridge without any extra effort.

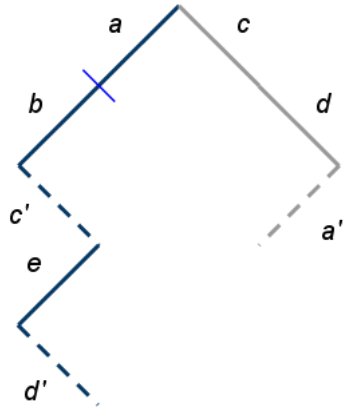
Building the Bridge

We can maintain this bridge by composing together all operations which have been synthesized locally since the point where we diverged from the server. Thus, at first, the bridge consists only of *a*. Soon afterward, the client applies its next operation, *b*, which we compose into the bridge. Of course, we inevitably receive an operation from the server, in this case, *c*. At this point, we use our bridge to transform *c* immediately to the correct point in client

state space, resulting in *c'*. Remember that OT derives *both* bottom sides of the diamond. Thus, we not only receive *c'*, but we also receive a new bridge which has been transformed against *c*. This new bridge is precisely the green dashed line in our diagram above.

Meanwhile, the client has performed another operation, *e*. Just as before, we immediately compose this operation onto the bridge. Thanks to our bit of trickery when transforming *c* into *c'*, we can rest assured that this composition will be successful. In other words, we know that the result of applying the bridge to the document resulting from *c* will be precisely the document state before applying *e*, thus we can cleanly compose *e* with the bridge.

Finally, we receive *d* from the server. Just as with *c*, we can immediately transform *d* against the bridge, deriving both *d'* (which we apply to our local document) as well as the new bridge, which we hold onto for future server translations.

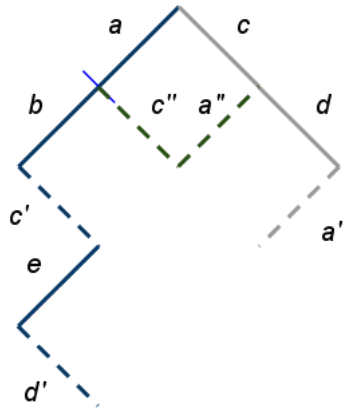


With *d'* now in hand, the next operation we will receive from the server will be *a'*, the transformed version of our *a* operation from earlier. As soon as we receive this operation, we need to compose together any operations which have been held in the buffer and send them off to the server. However, before we send this buffer, we need to make sure that it is parented off of some point in server state space. And as you can see by the diagram above, we're going to have troubles both in composing *b* and *e* (since *e* does not descend directly from *b*) and in guaranteeing server parentage (since *b* is parented off of a point in client state space not shared with the server).

To solve this problem, we need to play the same trick with our buffer as we previously played with the translation bridge: any time the client or the server does anything, we adjust the buffer accordingly. With the bridge, our invariant was that the bridge would always be parented off of a point in server state space and would be the one operation needed to transform incoming server operations. With the buffer, the invariant must be that the buffer is always parented off of a point in server state space and will be the one operation required to bring the server into perfect sync with the client (given the operations we have received from the server thus far).

The one wrinkle in this plan is the fact that the buffer *cannot* contain the operation which we have already sent to the server (in this case, *a*). Thus, the buffer isn't really going to be parented off of server state space until we receive *a'*, at which point we should have adjusted the buffer so that it is parented precisely on *a'*, which we now know to be in server state space.

Building the buffer is a fairly straightforward matter. Once the client sends *a* to the server, it goes into a state where any further local operations will be composed into the buffer (which is initially empty). After *a*, the next client operation which is performed is *b*, which becomes the first operation composed into the buffer. The next operation is *c*, which comes from the server. At this point, we must somehow transform the buffer with respect to the incoming server operation. However, obviously the server operation (*c*) is not parented off of the same point as our buffer (currently *b*). Thus, we must *first* transform *c* against *a* to derive an intermediary operation, *c"*, which is parented off of the parent of the buffer (*b*):



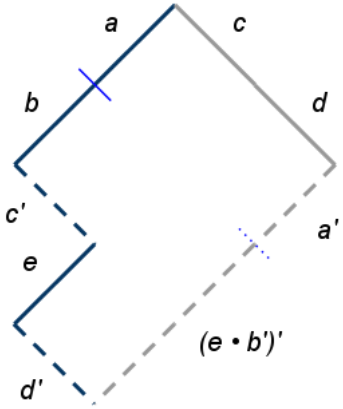
Once we have this inferred operation, *c"*, we can use it to transform the buffer (*b*) "down" one step. When we derive *c"*, we also derive a transformed version of *a*, which is *a"*. In essence, we are anticipating the operation which the server will derive when it transforms *a* against its local history. The idea is that when we finally do receive the real *a'*, it should be exactly equivalent to our inferred *a"*.

At this point, the client performs another operation, *e*, which we immediately compose into the buffer (remember, we also composed it into the bridge, so we've got several things going on here). This composition works because we already transformed the buffer (*b*) against the intervening server operation (*c*). So *e* is parented off of *c'*, which is the same state as we get were we to apply *a"* and then the buffer to the server state resulting from *c*. This should sound familiar. By a strange coincidence, *a"* composed with the buffer is precisely equivalent to the bridge. In practice, we use this fact to only maintain one set of data, but the process is a little easier to explain when we keep them separate.

Checkpoint time! The client has performed operation *a*, which it sent to the server. It then performed operation *b*, received operation *c* and finally performed operation *e*. We have an operation, *a"* which will be equivalent to *a'* if the server has no other intervening operations. We also have a buffer which is the composition of a transformed *b* and *e*. This buffer, composed with *a"*, serves as a bridge from the very latest point in server state space (that we know of) to the very latest point in client state space.

Now is when we receive the next operation from the server, *d*. Just as when we received *c*, we start by transforming it against *a"* (our "in flight" operation). The resulting transformation of *a"* becomes our new in flight operation, while the resulting transformation of *d* is in turn used to transform our buffer down another step. At this point, we have a new *a"* which is parented off of *d* and a newly-transformed buffer which is parented off of *a"*.

Finally, we receive *a'* from the server. We could do a bit of verification now to ensure that *a"* really is equivalent to *a'*, but it's not necessary. What we do need to do is take our buffer and send it up to the server. Remember, the buffer is parented off of *a"*, which happens to be equivalent to *a'*. Thus, when we send the buffer, we know that it is parented off of a point in server state space. The server will eventually acknowledge the receipt of our buffer operation, and we will (finally) converge to a shared document state:



The good news is that, as I mentioned before, this was the most complicated case that a collaborative editor client ever needs to handle. It should be clear that no matter how many additional server operations we receive, or how many more client operations are performed, we can simply handle them within this general framework of buffering and bridging. And, as when we sent the *a* operation, sending the buffer puts the client back into buffer mode with any new client operations being composed into this buffer. In practice, an actively-editing client will spend most of its time in this state: very much out of sync with the server, but maintaining the inferred operations required to get things back together again.

Conclusion

The OT scheme presented in this article is precisely what we use on Novell Pulse. And while I've never seen Wave's client code, numerous little hints in the waveprotocol.org whitepapers as well as discussions with the Wave API team cause me to strongly suspect that this is how Google does it as well. What's more, Google Docs recently revamped their word processing application with a new editor based on operational transformation. While there hasn't been any word from Google on how exactly they handle "compound OT" cases within Docs, it looks like they followed the same route as Wave and Pulse (the tell-tale sign is a perceptible "chunking" of incoming remote operations during connection lag).

None of the information presented in this article on "compound OT" is available within Google's documentation on waveprotocol.org (unfortunately). Anyone attempting to implement a collaborative editor based on Wave's OT would have to rediscover all of these steps on their own. My hope is that this article rectifies that situation. To the best of my knowledge, the information presented here should be everything you need to build your own client/server collaborative editor based on operational transformation. So, no more excuses for second-rate collaboration!

Resources

- To obtain Google's OT library, you must take a Mercurial clone of the wave-protocol repository:

  ```
  $ hg clone https://wave-protocol.googlecode.com/hg/ wave-protocol
  ```

  Once you have the source, you should be able to build everything you need by simply running the Ant build script. The main OT classes are `org.waveprotocol.wave.model.document.operation.algorithm.Composer` and `org.waveprotocol.wave.model.document.operation.algorithm.Transformer`. Their use is exactly as described in this article. Please note that `Transformer` does not handle compound OT, you will have to implement that yourself by using `Composer` and `Transformer`. Operations are represented by the `org.waveprotocol.wave.model.document.operation.DocOp` interface, and can be converted into the more useful `org.waveprotocol.wave.model.document.operation.BufferedDocOp` implementation by using the `org.waveprotocol.wave.model.document.operation.impl.DocOpUtil.buffer` method.

  All of these classes can be found in the **fedone-api-0.2.jar** file.

- Google's Own Whitepaper on OT
- The original paper on the Jupiter system (the primary theoretical basis for Google's OT)
- Wikipedia's article on operational transformation (surprisingly informative)

Comments

1. Sorry I really love your post but to be sincere:

   tl;dr

   Pablo Fernandez Monday, May 17, 2010 at 12:54 am

2. It is long. That's mainly because I wanted to be as comprehensive as possible. OT is a really extensive topic; you'll notice that I didn't even get into the precise algorithm detailing *how* operations are transformed!

   If you're not interested in OT, then feel free to not read the article. However, I know that when we were first sitting down, thinking about how we would implement co-edit in Pulse, an article like this would have been really invaluable. That's why I wrote it.

   Daniel Spiewak Monday, May 17, 2010 at 6:40 am

3. Thanks for this very useful post, the most informative that I have seen on this topic.

   All the other ones I've seen are superficial treatments, catering to the tl;dr; "info snack" crowd.

   charles02139 Monday, May 17, 2010 at 8:50 am

4. Outstanding entry. I find this very interesting as I've been implementing something like this from the opposite direction; something more like dropbox, which has no knowledge of how a document is edited, and instead must do synchronization after the fact. Clearly when you have total control of the document (e.g. it's not just some file that someone can go in and edit with any old editor), OT is superior.

   Tom Monday, May 17, 2010 at 12:47 pm

5. One small thing… shouldn't the xform equation read: xform(a, b) = (a', b'), where a' ∘ b === b' ∘ a ? (a' composed with the result of b equals b' composed with the result of a)

   Tom Monday, May 17, 2010 at 2:13 pm

6. @Tom

   No, because that would defeat the purpose. Remember, b' represents the transformed server operation. If we have to apply b' before we apply a, then we're dead before we start (because we've already applied a by the time we receive b from the server).

Daniel Spiewak Monday, May 17, 2010 at 2:15 pm

7. Then again, maybe I'm reading the composition operator backwards… I'll correct the article.

   Daniel Spiewak Monday, May 17, 2010 at 2:16 pm

8. While I'll admit to skimming through some details (which is fine since I do not see myself implementing an OT collaboration tool anytime soon), I find this really fascinating and instructive.

   Thanks very much for sharing…

   Dror

   Dror Harari Monday, May 17, 2010 at 3:31 pm

9. Great article. I'm trying to build an OT server in Node.js, and you've just described exactly how hard the job is going to be!

   Andrew Jessup Tuesday, May 18, 2010 at 3:56 am

10. After quickly scanning the text, I wonder: how is this not just a fancy implementation of the command pattern?

    Buddy Casino Thursday, May 20, 2010 at 8:23 am

11. @Buddy

    In a sense, yes, it is a fancy implementation of the command pattern (as most collaborative systems tend to be). The part which is interesting is the way in which commands can apply their effects separately in differing orders, converging to the same state.

    Daniel Spiewak Saturday, May 22, 2010 at 4:19 pm

12. @Daniel: this is indeed an interesting property, thanks for the explanation.

    Buddy Casino Monday, May 24, 2010 at 2:47 am

13. First off, another great post Daniel.

    Google wave real-time collaboration is impressive and makes a great demo, I question if this is this the best approach from an end user point of view. Am i alone in thinking that locking small parts of a document would provide a better user experience ? I guess i really dont want someone editing the same sentence that i am currently working on.

    Jimmy Monday, May 31, 2010 at 3:25 pm

14. Thanks for the great and thorough explanation!

    Though when I finally thought I grasped it all, it seems google is planning to change things a bit.
    In the new client-server protocol, the server won't send updates of your own submits.
    This means the client never gets a' back, but has to figure out from the SubmitResponse's "version after application" when to send new data.

    While I can probably figure out what to change in your approach, revisiting/updating this post would be very helpful.

    Please see my post on google groups here:
    http://groups.google.com/group/wave-protocol/browse_thread/thread/1d9caa71c063098e

    Relevant proto-changes:
    http://code.google.com/p/wave-protocol/source/browse/src/org/waveprotocol/wave/examples/fedone/waveserver/waveclient-rpc.proto?repo=io2010

    Mathijs Kwik Sunday, June 20, 2010 at 11:21 pm

15. @Mathijs

    I'm not sure I understand you. Are you saying that the Wave server will no longer send SubmitResponse? That seems impossible, since OT would not be sound under such a scheme. If you're referring to the ordering issue, we did run into that at Novell. I can't remember exactly how we solved it, but I do know that we found a way to correlate SubmitResponse + WaveletOperation into an Operation + id.

    Daniel Spiewak Monday, June 21, 2010 at 7:53 am

16. @Daniel

    Nope, you will get SubmitResponse.
    You just won't get WaveletUpdates for updates that were submitted by yourself.

    I think the response (which gives you "version after application") is enough indeed, since that number + the number new ops you got from the server, will tell you when a' was applied.

    Nothing unsolvable, but it does make the internals somewhat different.

    Mathijs Kwik Monday, June 21, 2010 at 10:46 am

17. It looks like the client/server protocol is somewhat different from the federation protocol in this regard. Federation *must* provide all WaveletUpdates, because multiple clients may be hanging off of the remote federated server. The client/server protocol would only need to provide SubmitResponse. In this case, you don't really get a', just its id. The logic is exactly the same, you just need to be a bit more explicit about detecting acknowledgment operations.

    Daniel Spiewak Monday, June 21, 2010 at 11:11 am

18. @Daniel

    Very good article about OT, very well explained and nice level of detail, I really liked to read well how everything works internally in Google Wave OT.

    @Mathijs, In fact you don't need server to reply to you with a WaveletUpdate for your own local change as long as you have all previous WaveletUpdates and your already accepted by server Update you can reconstruct a valid state for your local document that will match Server state, so your next operation or next waveletupdate received will be send/processed in the right document sequence

    I think one of the most complex scenario, Daniel can correct me if I'm wrong, become when you are buffering several operations to send to the server, as actual Google Wave client/server specification only permits you to be waiting for an ACK for just one operation at time, so is correct to think, (and it happens), that meanwhile you are waiting for an ACK for current sent operation you can receive more WaveletUpdates from different users, that also has a side effect of invalidate your buffered-pending-to-send operations forcing you to apply OT to each operation in the pending local-operations buffer or discard all operations at all, well, in fact OT will discard certain components on your pending operations bases on source WaveletUpdates operations for OT.

So you need to have OT capabilities also on Client side, as this scenario is also possible on server side, client and server logic matchs but as Daniel explained on client you have to take *more* care about ACKs send from server to see if you out-of-sync and make everything match server state on your local copy. So clients are really a bit more complex than federation implementation…

Jesus Salas

Jesus Salas Friday, June 25, 2010 at 5:47 pm

19. You say “two operations must span the same number of indexes” is a prerequisite to composition.

    But what about the following scenario:
    a = write(“a”)
    b = retain(1) ; write(“b”)

    Aren’t a and b composable as write(“ab”), even though b spans 2 indexes and a spans only 1?

    Ben Noland Wednesday, July 7, 2010 at 7:04 am

20. @Ben Noland

    I think you are right.
    I think the answer is that every operation has a “start index size” and an “end index size”
    some components (like write) affect start end sizes, while others (like retain) don’t.

    so in your example:
    a is an operation from 0 to 1 (size-wise)
    b is an operation from 1 to 2

    a and b are composeable since a’s end-size (1) is the same as b’s start-size.

    more complex example:
    a = retain(3); delete(“ab”); write(“fgh”); retain 2 (size 7 to 😎
    b = delete(“q”) ; retain (6); delete(“p”); write(“xyz”) (size 8 to 9)
    composing them gives
    c = delete(“q) ; retain(2); delete(“ab”); write(“fgh”); retain(1); delete(“p”); write(“xyz”) (size 7 to 9)

    basically:
    retain(x) means size x to x
    write(s) means size 0 to len(s)
    delete(s) means size len(s) to 0
    to calculate the start/end size of an operation you just add the numbers of the components.

    Hope this helps, but this is how I understood it

    Mathijs Kwik Monday, July 12, 2010 at 12:23 am

21. @Mathijs

    That’s a very helpful explanation. I did a little research myself, and had come to the same conclusion, but your writeup helps confirm. Thanks

    Ben Noland Saturday, July 17, 2010 at 6:59 am

22. The paper on OT from the Jupiter team was incredible!

    After reading this article and that paper I feel fairly confident enough to implement a very simple (think html textarea), non-xml, real time collaborative editor.

    Thanks so much for spending the time writing this documentation!

    Nick Fitzgerald Sunday, September 5, 2010 at 5:32 pm

23. hi, first i want to thank you for your great research and the way you discribed the whole thing…(on the google draft spezifikation and whitepapers it is not half as easy to understand)……..

    but by reading through all this, the question come up: how do they connect a operation to a paricipant/user?! does anyone have a answer on that?! because in all the whitepapers an spezifikations they never go into that….

    would be very interested in that….
    maybe you have an answer?!

    thanks freddy(sorry for my english, i’m from germany)

    freddy Monday, September 27, 2010 at 8:47 am

24. This is the blog post I’ve been looking for. Bummed I missed it before, but damn, first time I’ve seen it explained clearly.

    I’m coding as I read along (the third read-through) and I’m up to the compound OT.

    Really thumbs up on the post.

    Collin Miller Friday, June 17, 2011 at 6:11 pm

25. Hi, great post! It was one of the best pages I’ve read about OT.

    You help me to build a very simple simulator for my js OT algorithm. Take a look at http://vitotafuni.com/ot/jsottest.html. The first test case is the one you talk about in your post.
    Let me know what you think and feel free to fork my project on https://github.com/vitotafuni/JSOTTEST.

    Thank you

    Vito Tafuni Thursday, June 30, 2011 at 12:13 am

26. Dan this is the best article I have read on the subject and it really helped put OT into perspective for me. Thank you! I now feel confident I could implement my own OT server/client from scratch, and am reassured that I should for my specific application, rather than trying to bend the gigantic one-size-fits-all abstractions to my will, just because they already available. That seems to be where everyone is failing in their implementation; thanks for keeping it simple.

    Mike Smullin Saturday, July 9, 2011 at 1:11 pm