October 22, 2018

Lightweight Idempotent Operational Transform Inference in a Collaborative Editor

Owen Glofcheski

Lightweight idempotent operational transform inference in a collaborative editor

ABSTRACT

Per techniques of this disclosure, clients send opaque blobs of text to a server, rather than full operational transforms. The server uses the revision number received from the client to infer operational transforms by obtaining the shortest edit sequence between the received opaque text and revisions subsequent to the revision number. The techniques are particularly suitable for use in lightweight collaborative editors, such as note-taking software.

An idempotency issue potentially arises when not using full operational transforms in communications between client and server: duplicate requests can sometimes result in duplicate transforms on the server. Therefore, per the disclosed techniques, the server deduplicates commands it generates from those it had generated previously. The inferred OTs are thereby free of idempotency issue.

KEYWORDS

- operational transform
- idempotency
- document editor
- word processor
- note-taking
- collaborative editing
- shortest edit sequence
- delta storing
- delta differencing

BACKGROUND

Data models, e.g., used in collaborative word processors or document editors, can be stored at a server in the form of operational transforms (OTs) that represent changes (deltas). For example, client devices that edit documents send updates to a document in the form of operational transforms to communicate with a server that hosts the document. When the deltas are composed together, the full data model is recreated. Storing data models as OTs is popular since such storage provides full revision history, undo/redo stack, smart text-merging, etc.

DESCRIPTION

The overhead of a client sending operational transforms to a server may be undesirable for lightweight editors, e.g., note-taking software. Per techniques of this disclosure, clients are configured to send full opaque text to a server that infers OTs and merge-operations based on text received from different clients by using a revision number received from each client. To forestall idempotency problems, the server deduplicates commands it generates from those it had generated previously. The techniques thereby infer a stable OT free of idempotency issues.

Data models are stored in the form of operations that represent changes (deltas). Thus, a set of operations defines the data model. Also defined is logic that merges a combination of operations. Although there is in principle no limit to the number (or complexity) of operations, this disclosure advantageously defines two simple operations, e.g., **`insertText`** and **`deleteText`**. Each operation stores relevant data, e.g., the type of operation, the text associated with the operation, the start-position within the document where the operation takes effect, the revision number of the document, etc.

Example: An **`insertText`** operation can be represented as follows.

```
operation {
    type: INSERT
    text: "abc",
    offset: 1
}
```

A revision number is stored within the data structure of an operation such that a specified version of the data model can later be retrieved. The revision number can be any monotonically increasing value. For example, the revision number can be a simple incrementing value. As another example, the revision number can be a timestamp. A timestamp-based revision number enables client queries to be made based on time/date. The revision number enables proper merging of differing versions given by clients.

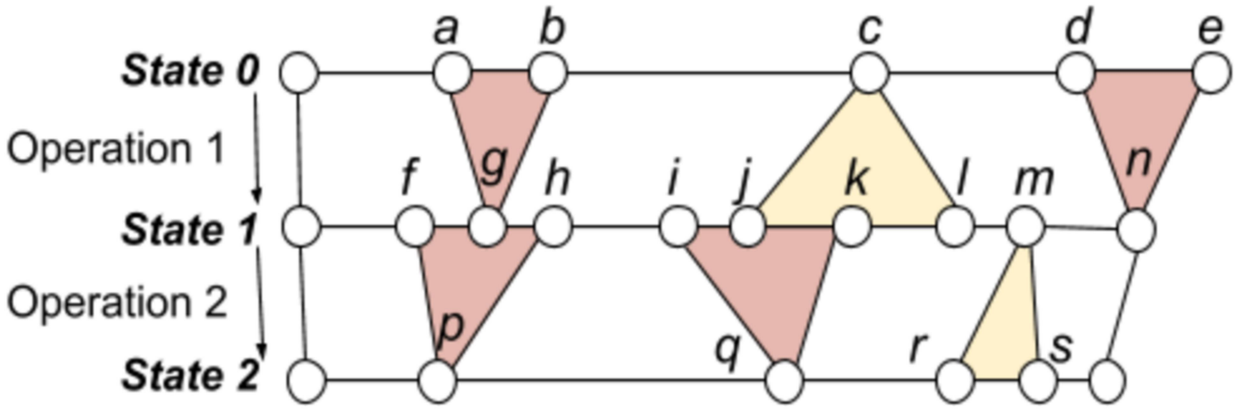

Fig. 1: `insertText` and `deleteText` operations

Fig. 1 illustrates **`insertText`** and **`deleteText`** operations on a document. A document is at a **`state 0`**, with example line (or character) markings **`a`**, **`b`**, **`c`**, **`d`** and **`e`**. An **`operation 1`** is performed that transforms the document to **`state 1`**. **`Operation 1`** comprises

- a **`deleteText`** between **`a`** and **`b`**, with the result marked as **`g`** in **`state 1`**;
- an **`insertText`** at **`c`**, resulting in new text between **`j and l`** in **`state 1`**; and
- a **`deleteText`** between **`d`** and **`e`**, with the result marked **`n`** in **`state 1`**.

The document at **`state 1`** is marked with markings **`f`**, **`g`**, **`h`**, **`i`**, **`j`**, **`k`**, **`l`**, **`m`**, and **`n`**. An **`operation 2`** is performed that transforms the document from **`state 1`** to **`state 2`**. **`Operation 2`** comprises

- a **`deleteText`** between **`f`** and **`h`**, with the result marked as **`p`** in **`state 2`**;
- a **`deleteText`** between **`i`** and **`k`** with the result marked as **`q`** in **`state 2`**; and
- an **`insertText`** at **`m`**, resulting in new text between **`r and s`** in **`state 2`**.

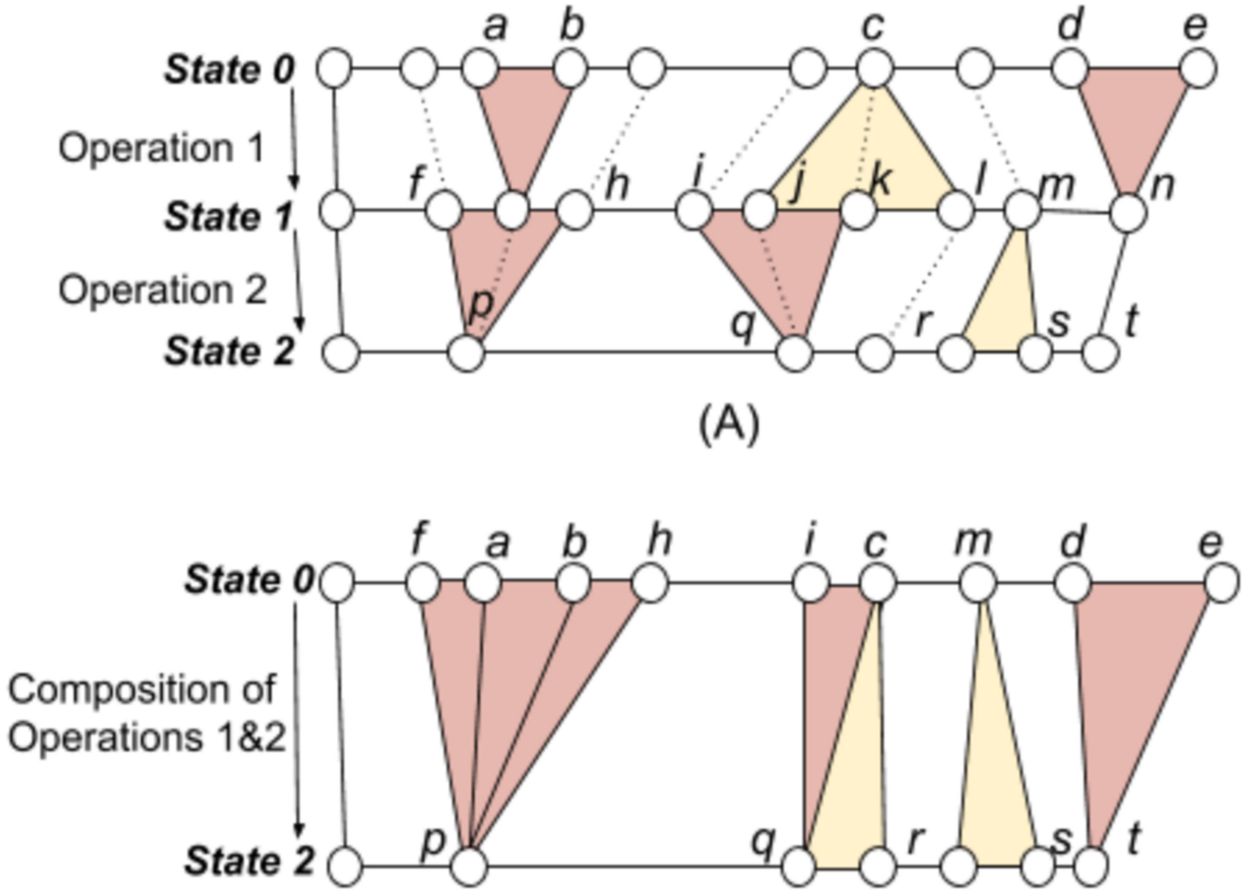

Fig. 2: Composing operations

Operations can be composed together, and the composition accounts for the range of the operations. For example, a deletion performed over an outer range after a deletion over an inner range will include the inner deletion as well. This is illustrated in Fig. 2. An inner **`deleteText`**

between **a** and **b** (**state 0**) and an outer **deleteText** between **f** and **h** (**state 1**) in Fig. 2A are both composed into a single **deleteText** (Fig. 2B) that deletes text between **f** and **h** in **state 0** to reach **state 2** directly. Similarly, a deletion performed over an outer (larger) range after an inner insertion will delete the inner insertion as well. Fig. 2 illustrates other examples of composing two operations into one.

A sequence of operations is stored as a list with monotonically increasing revision numbers.

Example: A three-character **insertText** of **abc** followed by a **deleteText** of the middle character **b** results in **ac**. This sequence of operations is represented in a list with monotonically increasing revision numbers, as follows.

```
revisions: [
   { revision_id: 1, Insert "abc" at 0 }  // "abc"
   { revision_id: 2, Delete [1, 2) }      // "ac"
]
```

Inference of changes

As explained before, the client sends opaque blobs of text to the server, rather than operational transforms. Per techniques of this disclosure, the corresponding operational transforms are inferred by the server from the received opaque blobs of text. The shortest edit sequence (SES) technique [1] is used to infer changes that were made by the client to transform the version at the server. SES does not guarantee retrieval of changes actually made by the client; rather, an approximation of these changes is obtained. The client provides a revision number along with each opaque blob of text. The server infers the operational transforms based on the revision number.

Once the server determines the OTs corresponding to a client, it transforms those operations against all other operations that have occurred since the revision number received from the client. Note that new revisions are not simply inserted below others, since this causes a bump in the revision number of subsequent operations, which other clients rely on to perform their own merging.

Example: In the following example, a client state is merged with a server state using the provided merging logic.

Client State

```
text: "hello world"
revision_id: 123
```

Server State:

```
revisions: [
   { revision_id: 123, Insert "hello" at 0 }   // "hello"
   { revision_id: 124, Delete [0, 5) }         // ""
   { revision_id: 125, Insert "goodbye" at 0 } // "goodbye"
]
```

Merging Logic:

```
1. Retrieve Shortest Edit Sequence (SES) between "hello world" and "hello"
   --> Insert " world" at 5
2. Transform against revisions > 123
   --> Insert " world" at 7

3. Final state becomes:
revisions: [
   { revision_id: 123, Insert "hello" at 0 }   // "hello"
   { revision_id: 124, Delete [0, 5) }         // ""
   { revision_id: 125, Insert "goodbye" at 0 } // "goodbye"
   { revision_id: 126, Insert " world" at 7 }  // "goodbye world"
]
```

Retrieval of a revision of a document

For the purposes of undo/redo, debugging, etc., a user may retrieve text at a specific revision of a document or note. Every time a user requests to read the text for a note, the entire note is not rebuilt from the operational transforms, since this is not scalable as the cost of retrieval is high for documents that are frequently used or updated. For scalability reasons, snapshots of the document are stored at certain intervals within the revision table. The interval between snapshots is data-driven, e.g., determined by frequency of reads/updates, etc. The snapshots store an already-built model at a specific revision, thereby requiring the composition of a relatively small amount of operational transforms during retrieval. Storing snapshots comes at the cost of slightly more memory. For light-weight editors, e.g., note-taking software, documents (or notes) are typically small (< 1 MB), and such memory use is largely negligible.

Idempotency

In the context of this disclosure, idempotency is a property by which repeated identical requests from the same client result in the same server state. Under idempotency, the second (or third, fourth, etc.) identical request sent from a client to a server does not change the server state.

Idempotency is a desirable property; however, when not using full operational transforms in client-server communications, duplicate requests can sometimes result in duplicate transforms on the server, leading to non-idempotency. This is illustrated with an example.

Example: Idempotency issue.

If we have initial server state

```
revisions: [
   { revision_id: 1, Insert "hello" at 0 }  // "hello"
   { revision_id: 2, Insert " world" at 5 } // "hello world"
]
```

And the client upsyncs "`hello hi`" at `revision:1`.

1. We read a snapshot at `revision:1` which yields "`hello`"
2. Diffing the 2 texts yields the command `Insert " hi" at 5`
3. Transposing against `revision:2` yields `Insert " hi" at 5`

So the final revision state looks like

```
revisions: [
   { revision_id: 1, Insert "hello" at 0 }  // "hello"
   { revision_id: 2, Insert " world" at 5 } // "hello world"
   { revision_id: 3, Insert " hi" at 5 }    // "hello hi world"
]
```

If that same client upsyncs "`hello hi`" at `revision:1` again (perhaps due to not receiving an ACK in a low connectivity environment).

1. We read a snapshot at `revision:1` again which yields "`hello`"
2. Diffing the 2 texts yields the command `Insert " hi" at 5` again
3. Transposing against `revision:2` yields `Insert " hi" at 5`
4. Transposing against `revision:3` yields `Insert " hi" at 5`

So the final revision state looks like

```
revisions: [
   { revision_id: 1, Insert "hello" at 0 }  // "hello"
   { revision_id: 2, Insert " world" at 5 } // "hello world"
   { revision_id: 3, Insert " hi" at 5 }    // "hello hi world"
   { revision_id: 4, Insert " hi" at 5 }    // "hello hi hi world"
]
```

In the above example, in order to make the command idempotent, the fact that the same client already created revision 3 must be tracked.

Creator metadata

In order to facilitate idempotency, the server maintains a state that records the client that wrote a revision to the document. Note that the server does not deduplicate against commands generated by other clients. For example, two clients both writing "hi" at the same location is valid. Additionally, the same client writing "hi" several times at the same location is a valid operation.

Per techniques of this disclosure, the server maintains in a persistent manner a unique identifier for a client, and also the client-generated last-updated timestamp, in order to perform deduplication of commands. The client identifier can be any unique string associated with the client, e.g., MAC address, UIDevice, device-vendor identifier, or combinations thereof. An example data structure that stores creator metadata is shown below.

Example: Data structure for creator metadata, holding client identifier and last-updated timestamp.

```
message CommandMetadata {
  // The id of the client who made the change.
  // This could be any unique client identifier.
  optional string client_id = 1;

  // The time at which the client believes they made the edit.
  optional int64 client_updated_ts = 2;
}
```

Merging procedure

The merging procedure disclosed herein works under the assumption that a client complies with monotonically increasing revision numbers. In the face of delayed packets (which may cause decreasing revisions numbers) the client timestamp stored on the revision provides sufficient information to ignore old packets.

The set of revisions considered during deduplication is bounded by the difference between the client's revision and the maximum revision. As clients synchronize, this window becomes smaller so performance is improved for clients which are more up to date.

However, even in a pathological case where a client creates a large number of changes and sync requests without receiving an acknowledgement, the revisions being used in the read are the same revisions used in the non-idempotent transposition process, so there is no additional read latency.

The merging procedure is as follows. Determine which commands the client has already committed without knowing. Apply those commands to the snapshot at the client revision in order to generate a snapshot of what the client is actually transforming against since their last successful write. This in turn requires finding all revisions committed by the same client (with revision-id > client-revision) and transposing these against the inverse of the commands which came before them.

Once a snapshot is obtained of what the client is actually transforming against, generate diffs against this model. Since these diffs are based on this new snapshot (not the snapshot at the client revision) it is transformed back to the original context so it can be transposed properly against all commands created by other clients. This is performed by transposing it against the inverse of the commands generated by the same client. The diffs are then transposed against all later commands as they would normally.

At each step of the transposition, if a command previously generated (by the same client) is encountered, the procedure stops and regenerates a new snapshot right below the command. It then applies the inbound command and the set of contiguous previous commands, creating two new snapshots that are diffed against each other. Going forward, the difference between these

commands is regarded as the new truth of what the inbound text looked like. The merging procedure is illustrated with an example.

Example: Merging procedure.

If we have initial server state caused by client upsyncing "`hello hi`" at `revision:1` against the same existing client state from the previous example, then we have revision state:

```
revisions: [
   { revision_id: 1, Insert "hello" at 0, client_id: 1 }  // "hello"
   { revision_id: 2, Insert " world" at 5, client_id: 1 } // "hello world"
   { revision_id: 3, Insert " hi" at 5, client_id: 2 }    // "hello hi world"
]
```

Client 2 upsyncs "`hello hi`" at `revision:1` again.

1. We read a snapshot at `revision:1` which yields "`hello`"
2. Diffing the 2 texts yields the command `Insert " hi" at 5` again
3. We begin transposing against `revision_id:2` which yields `Insert " hi" at 5`
4. We detect we're transposing against `revision:3` created by the same client. We rebuild a snapshot of the data right below that revision yielding "`hello world`".
5. We apply `revision:3` and the new command against this new snapshot yielding "`hello hi world`" and "`hello hi world`"
6. Diffing the 2 texts yields a `NoopCommand` and we break, knowing this change has already been applied.

So the final revision state still looks like

```
revisions: [
   { revision_id: 1, Insert "hello" at 0, client_id: 1 }  // "hello"
   { revision_id: 2, Insert " world" at 5, client_id: 1 } // "hello world"
   { revision_id: 3, Insert " hi" at 5, client_id: 2 }    // "hello hi world"
]
```

Client 2 adds a character and upsyncs "`hello his`" at `revision:1`.

1. We read a snapshot at `revision:1`, which yields "`hello`"
2. Diffing the 2 texts yields the command `Insert " his" at 5`
3. We begin transposing against `revision_id:2` which yields `Insert " his" at 5`
4. We detect we're transposing against `revision:3` created by the same client. We rebuild a snapshot of the data right below that revision yielding "`hello world`".
5. We apply `revision:3` and the new command against this new snapshot yielding

"`hello hi world`" and "`hello his world`" (It's basically as if the client sent their original text at this revision)

6. Diffing the 2 texts yields the command `Insert s at 8` which can be placed on the revision list.

So the final revision state still looks like

```
revisions: [
   { revision_id: 1, Insert "hello" at 0, client_id: 1 }  // "hello"
   { revision_id: 2, Insert " world" at 5, client_id: 1 } // "hello world"
   { revision_id: 3, Insert " hi" at 5, client_id: 2 }    // "hello hi world"
   { revision_id: 4, Insert "s" at 8, client_id: 2 }      // "hello his world"
]
```

Client 2 removes a character and upsyncs "`hello hi`" at `revision:1`

1. We read a snapshot at `revision:1`, which yields "`hello`"
2. Diffing the 2 texts yields the command `Insert " hi" at 5`
3. We begin transposing against `revision_id:2` which yields `Insert " hi" at 5`
4. We detect we're transposing against a `revision:3` and `revision:4` created by the same client. We rebuild a snapshot of the data right below that command yielding "`hello world`".
5. We apply `revision:3` and `revision:4` in storage and the new command against this new snapshot yielding "`hello his world`" and "`hello hi world`"
6. Diffing the 2 texts yields the command `Delete [8, 9)` which can be placed on the revision list.

So the final revision state still looks like

```
revisions: [
   { revision_id: 1, Insert "hello" at 0, client_id: 1 }  // "hello"
   { revision_id: 2, Insert " world" at 5, client_id: 1 } // "hello world"
   { revision_id: 3, Insert " hi" at 5, client_id: 2 }    // "hello hi world"
   { revision_id: 4, Insert "s" at 8, client_id: 2 }      // "hello his world"
   { revision_id: 5, Delete [8, 9), client_id: 2 }        // "hello hi world"
]
```

Client 1 comes back and swaps "`hello`" with "`bye`" yielding the new revision state.

```
revisions: [
   { revision_id: 1, Insert "hello" at 0, client_id: 1 }  // "hello"
   { revision_id: 2, Insert " world" at 5, client_id: 1 } // "hello world"
   { revision_id: 3, Insert " hi" at 5, client_id: 2 }    // "hello hi world"
   { revision_id: 4, Insert "s" at 8, client_id: 2 }      // "hello his world"
```

```
   { revision_id: 5, Delete [8, 9), client_id: 2 }        // "hello hi world"
   { revision_id: 6, Delete [0, 5), client_id: 1 }        // " hi world"
   { revision_id: 7, Insert "bye" at 0, client_id: 1 }    // "bye hi world"
]
```

Client 2 finally receives `revision:2-3` in the down-sync and merges it with their local storage creating "`hello hi world`". Client 2 adds 2 characters and upsync "`hello hist world`" at `revision:3`.

1. Then we'll read a snapshot at `revision:3`, which yields "`hello hi world`"
2. Diffing the 2 texts yields the command `Insert "st" at 8`
3. We detect we're transposing against `revision:4` and `revision:5` created by the same client. We rebuild a snapshot of the data right below that command yielding "`hello hi world`".
4. We apply `revision:4` and `revision:5` and the new commands against this new snapshot yielding "`hello hi world`" and "`hello hist world`"
5. Diffing the 2 texts yields the command `Insert "st" at  8`
6. We transpose against `revision:6` yielding `Insert " st" at 3`
7. We transpose against `revision:7` yielding `Insert " st" at 6` which can be placed on the revision list.

So the final revision state still looks like

```
revisions: [
   { revision_id: 1, Insert "hello" at 0, client_id: 1 }  // "hello"
   { revision_id: 2, Insert " world" at 5, client_id: 1 } // "hello world"
   { revision_id: 3, Insert " hi" at 5, client_id: 2 }    // "hello hi world"
   { revision_id: 4, Insert "s" at 8, client_id: 2 }      // "hello his world"
   { revision_id: 5, Delete [8, 9), client_id: 2 }        // "hello hi world"
   { revision_id: 6, Delete [0, 5), client_id: 1 }        // " hi world"
   { revision_id: 7, Insert "bye" at 0, client_id: 1 }    // "bye hi world"
   { revision_id: 8, Insert "st" at 6, client_id: 2 }     // "bye hist world"
]
```

Which is equivalent to the state where the client only sent the one upsync "`hello hist`" at `revision:1` or "`hello hist world`" at `revision:2`.

```
revisions: [
   { revision_id: 1, Insert "hello" at 0, client_id: 1 }  // "hello"
   { revision_id: 2, Insert " world" at 5, client_id: 1 } // "hello world"
   { revision_id: 3, Delete [0, 5), client_id: 1 }        // " world"
   { revision_id: 4, Insert "bye" at 0, client_id: 1 }    // "bye world"
   { revision_id: 5, Insert " hist" at 3, client_id: 2 }  // "bye hist world"
```

```
]
```

Idempotency at the client device

Unless they drop responses to old requests, clients also have the potential for idempotency issues, e.g., when receiving multiple responses from server. They also have some issues with merging operational transforms since they do not store full history on the server.

Clients need a way to differentiate if a sync includes the changes they've made or not. Otherwise, it is difficult to infer which portion of the text sent in down-sync is due to the client's own up-sync or from some collaborator. An example of this situation is illustrated below.

Initial client text: ""

Client updates text to "`hello`" and sends sync request 1. The response is delayed.

Client updates text to "`hello hi`" and sends sync request 2.

1) Client receives server response to sync request 2 "`hello hi world`"
2) Client diffs the request text "`hello hi`" against response text "`hello hi world`" yielding transform `Insert " world" at 8`
3) Applying command to client text yields "`hello hi world`"
4) Client receives response to sync request 1 "`hello world`"
5) Client has forgotten what text it sent with that sync request so it either doesn't know what to diff against or it incorrectly uses the last sync text "`hello hi world`" which results in an incorrect deletion of "`hi`"

In order to facilitate true idempotency at the client device, clients keep track of their in-flight syncs (or at least the most recent one) and ignore any responses received which don't already match. Under a client guarantee that responses from previous sync will be ignored, the clients can make use of the same 3-way merge used by the server. The web client keeps track of a unique identifier per request and the rebind callback handlers.

Therefore, the clients can use the following merge logic without issue:

```
// 1) When a client sends a sync keep track of the text sent in that sync,
//    this will be called BASE.
//
//    NOTE: A client can continue to make changes while the sync is in flight,
//          this version of the text is called CLIENT_NEW.
//
// 2) If a client sends a new sync, update BASE. When a client receives a
//    down-sync from a previous sync it will be ignored and state will not change.
//
// 3) When a client receives the down-sync from their most recent sync they will
//    extract the text and call it SERVER_NEW.

// Create commands that transform BASE into CLIENT_NEW.
List<Command> clientCommands = diff(BASE, CLIENT_NEW);

// Create commands that transform BASE into SERVER_NEW.
List<Command> serverCommands = diff(BASE, SERVER_NEW);

// Transpose clientCommands against serverCommands.
// Note, technically we can transpose the serverCommands against the clientCommands
// and invert the next step. However, this is more in line with how the server
// would actually produce the diffs, which has the nice benefit of not creating as
// many redundant commands on the server.
List<Command> transposedCommands = transpose(clientCommands, serverCommands);

// Apply the transposedCommands to SERVER_NEW.
String finalText = apply(transposedCommands, SERVER_NEW);
```

Cursor Position

With slight modifications, the above code can also be made to indicate how much a client should offset their current cursor position within the stream. It is essentially the same transposition logic as the **`insertText`** command; however, instead of generating no-op commands, in some circumstances the cursor is shifted. In this case it might be easier to transpose server against client instead of client against server. The following is the logic used to determine cursor position.

1. If text was inserted at **`offset <= cursor_position`** then **`cursor_position = cursor_position + text.length`**
2. If text was deleted at **`end <= cursor_position`** then **`cursor_position = cursor_position - (deletion.end - deletion.start)`**
3. If text was deleted surrounding the **`cursor_position`** then **`cursor_position = cursor_position - (deletion_before_cursor_position.length)`**

Undo Behavior

Generally speaking, a client undoing a revision is essentially the same as the client having written an undo operation. Therefore the idempotent procedures of this disclosure robustly handle the undo operation. In the examples above, the idempotent procedures treat undo operations in the same way as the client writing "**`hello hi`**" followed by "**`hello`**".

CONCLUSION

Per techniques of this disclosure, clients send opaque blobs of text to a server, rather than full operational transforms. The server uses the revision number received from the client to infer operational transforms by obtaining the shortest edit sequence between the received opaque text and revisions subsequent to the revision number. The techniques are particularly suitable for use in lightweight collaborative editors, such as note-taking software.

An idempotency issue potentially arises with not using full operational transforms in communications between client and server: duplicate requests can sometimes result in duplicate transforms on the server. Therefore, per the disclosed techniques, the server deduplicates commands it generates from those it had generated previously. The inferred OTs are thereby free of idempotency issue.

REFERENCES

[1] Myers, Eugene W. "An O(ND) difference algorithm and its variations." *Algorithmica* 1, no. 1-4 (1986): 251-266.