Analysis of Contemporary Methodologies For Near-Real Time Collaboration

Moin Ahmed Qidwai

2021-09-26

Introduction

Near-real time (NRT) collaboration is the goal of many software applications, from collaborative text-editors like Google Docs to modelling applications like Autodesk Maya. In reality most applications could benefit from allowing users to collaborate effectively regardless of the problem domain that the application targets. As such the reason that so few of the mainstream software supports this functionality is generally the result of complexity accompanied with solutions to this problem. While there are a few different methodologies for supporting NRT collaboration, in this paper we shall investigate YJS, a popular library implementing the YATA approach along with Operational Transformation, the approach at the core of Google Docs. We shall then present comparisons of the two aforementioned techniques, along with results of tests conducted in real world environments.

Collaboration Objectives

In order for a solution to be considered for NRT collaboration, it must be able to satisfy certain conditions or objectives.

0.1 Eventual Convergence

Eventual convergence dictates that if two collaborators receive the same set of operations in any order, the end result of those operations must be the same for both the collaborators.
That is, If we have a algorithm A for merging operations into a list and two sets of operations S_1 and S_2 that observe the below relation.

$$\forall o : o \in S_1 \leftrightarrow o \in S_2 \tag{1}$$

Then the following must hold true, where $\mapsto$ represents an input symbol.

$$S_1 \mapsto A \equiv S_2 \mapsto A \tag{2}$$

0.2 Intention Preservation

The idea behind preservation of intention is simple. Any solution that aims to provide for NRT collaboration must ensure the result is in accordance with the intention of all collaborators.

0.3 Interleaving

Interleaving occurs if two or more collaborators insert multiple characters at the same index, upon integrating their insertions the result may have their

inputs mixed.
Example: Collaborators C_1 and C_2 add "vious" and "cious" to "pre". If interleaving occurs the result may be "prevcioiouuss". A program that allows for NRT collaboration must ensure interleaving cannot occur.

YATA

The YATA (Yet Another Transformation approach) is the core specification underlying YJS. This specification consists of two main components, a doubly linked list and a set of rules that all operations must observe.

0.4 Data Representation

The doubly linked list representation used by YATA is in contrast to other algorithms. Another popular algorithm is the RGA, which utilizes a unidirectional linked list. The doubly linked list allows YATA to avoid interleaving at the start of the document or in prepend operations. As such it can cater for a wider range of operations and use cases than RGA by default. Though due to storing a pointer to the successor the data representation in YATA requires more memory.

$$Block_i = (id_i, origin_{left}, origin_{right}, deleted_i, value_i) \tag{3}$$

Equation **??** represents a single element in the linked list of YATA. The origins represent the pointers to predecessor and successor elements.

$$id_i = (replica_i, counter_i) \tag{4}$$

Equation **??** represents the identifier for a single block in the linked list. It consists of the Replica ID (or user id) and the operation counter.

The above representation ensures each block has a unique identifier and a total order.

0.5 Operations

The YATA specification only outlines two types of operations: insertion and deletion. A combination of these operations can also lead to many others, for example the update operation.

0.5.1 Insertion

$$Operation_k = (id_k, origin_k, left_k, right_k, deleted_k, value_k) \tag{5}$$

Equation **??** represents the insertion operation with counter (k). The **origin** represents the predecessor for the block at the time of creation. The **left** and **right** represent the predecessor and successor respectively after the operation has been merged into the linked list. The **deleted** flag indicates if the block representing the operation has been marked for deletion. The **value** is the actual content that is to be inserted.

0.5.2 Deletion

The deletion operation is simply represented by setting the deleted flag of the insertion operation to **true**.

$$Operation_k = (id_k, origin_k, left_k, right_k, true, value_k) \tag{6}$$

0.5.3 Operation Ordering

Every operation block has a total order in the list defined by the natural predecessor relation $<$.

$$O_1 < O_2 \leftrightarrow O_1 is\ a\ predecessor\ of\ O_2 \tag{7}$$

$$O_1 \leq O_2 \leftrightarrow O_1 < O_2 \vee O_1 \equiv O_2 \tag{8}$$

Given the above predecessor relation and insert operation we can represent an insertion between two operations $O_i\ and\ O_j$ as shown below.

$$Operation_{new} = (id_{new}, O_i, O_i, O_j, false, value_{new}) \tag{9}$$

In equation **??** the following relation must hold $O_i < Operation_{new} < O_j$.

As one may notice the origin and the left operation are the same in the above equation as this represents the operation at the time of it's creations. The origin for the operation is set at the time of the operation creation and does not change thereafter. The left pointer may change during the merge process of operations created by different replicas, if there are conflicts.

0.6 Rules of Conflict Resolution

As mentioned earlier in this paper YATA consists of certain rules that must be observed by operations specially in cases of conflicts. These rules are the cornerstone of the YATA approach as they ensure **eventual convergence** and **intention preservation**.

0.6.1 Conflicting insertions

Insertion operations (O_a, O_b, ...) are in conflict if all of them are to be inserted between O_i and O_j. In the above example, if $Operation_{new}$ is to be integrated in the list of operations $L = [O_i, c_a, c_b, c_c ... O_j]$, then $Operation_{new}$ is in conflict with $[c_a, c_b, c_c, ...]$. The rules resolve these conflicts by calculating the index k for the new insertion. If the rules are observed by all collaborators then each of them calculates the same index. Each rule can be illustrated as a predecessor relation $<_r$. As such if we observe these rules for $Operation_{new}$ then we integrate it between c_i and c_j where $\forall r : c_i <_r Operation_{new} <_r c_j$.

0.6.2 Rule One

The first rule dictates that for two conflicting insertions I_a and I_b that have different origins O_a and O_b respectively, the connection between I_a and O_a must not be intersected by the connection between I_b and O_b, or vice versa. The only instances where the above holds true is illustrated by the below ordered sets (and their opposites, which one can get by swapping the indexes a and b).

$$[O_a, O_b, I_b, I_a] \tag{10}$$

$$[O_a, I_a, O_b, I_b] \tag{11}$$

Set **??** represents the case where a operation and it's origin are inserted in between of the other operation and it's origin. Set **??** represents the case

where a operation and it's origin are inserted after the other operation and it's origin.

Rule one then can be succinctly illustrated by the below equation.

$$O_a <_{r1} O_b \leftrightarrow O_a < Origin_b \vee Origin_b \leq Origin_a \tag{12}$$

0.6.3 Rule Two

Rule two is the standard rule of transitivity and can be illustrated by the below equation.

$$O_a <_{r2} O_b \leftrightarrow \forall O : O_b <_{r2} O \rightarrow O_a \leq O \tag{13}$$

0.6.4 Rule Three

Rule three dictates that if two conflicting insertions have the same origin, then the insertion with the smaller creator ID is to the left. It can be represented by the below equation.

$$O_a <_{r3} O_b \leftrightarrow Origin_a \equiv Origin_b \rightarrow Creator_a < Creator_b \tag{14}$$

0.6.5 Total Order Function

If we combine all the three rules by conjunction and utilize them for insertions we get a total order on the insertion operations. This ensures both **eventual convergence** and **intention preservation**.

0.7 YJS

YJS is an implementation of the YATA specification, though it does couple it with delta-state based operations. In other words it only passes the specific operations that changed per integration, as opposed to the full document as per the YATA specification. This ensures the size of the messages remains small and hence the burden on the network resources is minimized.
The main disadvantage of YJS along with YATA is that when each character is represented as an operation as opposed to a single character, the overall document takes greater space. Though this is compensated with generally lower time complexity and the small size of the propagated messages.

Operational Transformation

At it's core Google Docs utilizes Operational Transformation to provide the ability for NRT to it's users. The idea behind Operational Transformation is quite old yet still actively used, it was pioneered by C. Ellis and S. Gibbs in 1989. It consists of two core ideas as well as outlined below.

- The document state represented as S, is updated using different Operations $O_1(S), O_2(S), ...$ (such as insertion and deletion).
- Conflicts resulting from concurrent updates are resolved using a transform function $O_3 = T(O_1, O_2)$ that takes the two operations in conflict and returns a new operation that can be applied to preserve intention.

0.8 Transformation Function

The transformation function mentioned above can take one of two different forms.

0.8.1 Inclusion Transformation

The inclusion transformation function denoted $IT(O_1, O_2) \rightarrow O_3$ takes two operations O_1, O_2 and returns O_3, which effectively applies operation O_1 as if O_2 is included.

As an example lets say we have a document state (S) = "ACEF" at time T and we receive two concurrent updates to this document state represented by the below operations.

$$O_1(S) \equiv Insert(S, 1, B). \tag{15}$$

$$O_2(S) \equiv Insert(S, 2, D). \tag{16}$$

Operation shown in **??** will add B after the character at position = 1 (A). Operation shown in **??** will add D after the character at position = 2 (C).

Now if we apply the first operation to the state, the new state is equal to NS = "ABCEF". Applying the second operation to this will provide us with a final state FS = "ABDCEF". Clearly we lost the intention of the users, the first user intended for B to be added between A and C, the second user intended for D to be added between C and E. Instead, D was added between B and C. In order to rectify this we will apply the transformation function $IT(O_2, O_1)$ after applying O_1 to S, which transforms O_2 to the below operation.

$$O_3(S) \equiv Insert(S, 3, D). \tag{17}$$

Operation shown in **??** will add D after the character at position = 3 (C).

In general for a pair of character-wise operations **Insert(S, P, C)** (Insert character C after position P in state S) and **Delete(S, P)** (Delete character at position P in state S), four IT functions, denoted as $T_{ii}, T_{id}, T_{di}, T_{dd}$, can be defined as follows (I represents insert operations and D is for deletions).

$$T_{ii}(I(P1, C1), I(P2, C2)) = \begin{cases} I(P1, C1) & \text{if } P1 < P2 \ OR \ U_g \\ I(P1 + 1, C1) & \text{otherwise} \end{cases} \tag{18}$$

U_g in equation **??** is equal to $P1 == P2 \ AND \ U1 > U2$, where U1 and U2 are user identifiers used to break the tie.

$$T_{id}(I(P1, C1), D(P2)) = \begin{cases} I(P1, C1) & \text{if } P1 \leq P2 \\ I(P1 - 1, C1) & \text{otherwise} \end{cases} \tag{19}$$

$$T_{di}(D(P1), I(P2, C2)) = \begin{cases} D(P1) & \text{if } P1 < P2 \\ D(P1 + 1) & \text{otherwise} \end{cases} \tag{20}$$

$$T_{dd}(D(P1), D(P2)) = \begin{cases} D(P1) & \text{if } P1 < P2 \\ D(P1 - 1) & \text{if } P1 > P2 \\ I & \text{otherwise} \end{cases} \tag{21}$$

I in equition **??** is the special identity operator, and it returns the state as is since the deletion had already occurred. It is used as a tie-breaker for the delete/delete pair.

0.8.2 Exclusion Transformation

The exclusion transformation function denoted $ET(O_1, O_2) \rightarrow O_3$ takes two operations O_1, O_2 and returns O_3, which effectively applies operation O_1 as if O_2 is excluded.

Lets revisit the example from the previous section with the final document state F(S) = "ABCDEF". The operations are shown in **??** and **??**.

In this case if we apply transformation function $ET(O_2, O_1)$ after applying O_1 to S, it will transform O_2 to the below operation.

$$O_3(S) \equiv Insert(S, 2, D). \tag{22}$$

Operation shown in **??** will add D after the character at position = 2 (B).

Exclusion transformation is useful when we wish to perform undo of operations. In the above O_3 we are simply applying O_2 as if we had undone O_1.

Generally if we have three operations O_a, O_b, O_c and we were to undo O_a, we would need to apply the following transformations to the original state S to get the final state.

$$O_{b\_final}(S) \equiv ET(IT(O_b, O_c), O_a) \tag{23}$$

$$O_{c\_final}(S) \equiv ET(IT(O_c, O_b), O_a) \tag{24}$$

0.9 The Undo Procedure

The undo algorithm must satisfy the following conditions.

- Undoing an operation O should transform the original state S into the final state FS, such that FS is the result of applying all operations besides O to S.
- Undoing all operations applied to S should bring the state back to S.

Formally we can represent the above conditions as below, given initial state S, operations O_a, O_b applied to it to get final state FS.

$$Undo(O_a) \rightarrow ET(O_b, O_a)(S) \tag{25}$$

$$Undo(O_a, O_b) \equiv S \tag{26}$$

State Storage in OT

At it's core Operational Transformation does not dictate how or where one should store the document state. The state could be managed through a peer-2-peer or client-server architecture, we discuss the two approaches below.

0.10 Peer To Peer

In a peer to peer implementation of operational transformation, there is no single source of truth for the document state. Every peer maintains a local copy of the document state and broadcasts their operations to every other peer along with a state vector containing the local copy of state of every peer. The transformation of incoming operations is done at each peer's node rather than at a central server.

0.10.1 Advantages

- There is no single point of failure as this approach does not rely on a central server.
- Offline collaboration (local network connections) can be supported with greater ease.

0.10.2 Disadvantages

- As the operations and state need to be broadcasted to each and every peer, this approach places high strain on network resources.
- The storage requirements are high as each peer needs to store the state vector containing the number of changes of each peer along with their identifiers.
- The transformation algorithm can be a lot more complicated given the set of divergent possible states between different peers can be high.

0.11 Client-Server

In a client-server implementation of operational transformation, there is a single source of truth for the document state (the server). Every peer maintains a local copy of the document state and it applies the local operations to this copy without need of locking the state. Each peer caches the locally applied operations that have not yet been sent to the server and at appropriate intervals it sends the operations to the server. In some client-server algorithms the client does not wait for an acknowledgement from the server but in Google Docs, the client waits for the server's acknowledgement before sending further operations to it. In our discussion we will assume that the client does wait for acknowledgements from the server.

0.11.1 Advantages

- The locally applied operations are instantaneous. (though depending on the algorithm this may be true of P2P implementations as well).
- The strain on the storage and network resources is low.
- The complexity of the transformation algorithm is much lower since the client only needs to reconcile the local state with that of the server.

0.11.2 Disadvantages

- If the server crashes then the whole system breaks down and may even lead to data loss.
- Offline collaboration over a local network can be more difficult to implement as one or more of the peers needs to act as a server.

Popular OT Algorithms

TBC

CRDT and OT Comparison

We have explored two different methodologies along with their implementations for Near-real time collaboration, CRDT (YATA or YJS) and Operational Transformation (Google Docs). Below we discuss some of the advantages and disadvantages of each of these approaches.

0.12 CRDT

We will present a comparison solely for YATA as our choice of CRDT specification as comparing numerous specifications is out of scope for this report.

0.12.1 Advantages

- There is no need to wait for acknowledgement from other participants/server, increasing performance.
- It is peer to peer by default, hence sharing in all the advantages of P2P architecture.

0.12.2 Disadvantages

- CRDT enforces certain conditions on operations as discussed previously, as such not all operations are compatible with CRDT.
- Since CRDTs are data types, they need to be created for each type of data that our users interact with, hence leading to high initial complexity.

0.13 Operational Transformation

We will compare operational transformation with a central server as it's the most common real world implementation.

0.13.1 Advantages

- Any operation can be supported as long as it's transformations can be defined.
- Due to their greater prevalence in real world applications at present they have broader support.

0.13.2 Disadvantages

- As it requires a central server, it is susceptible to single point of failure.
- The requirement for acknowledgement reduces the performance for clients.
- The transformation functions can get quite complex and proving their validity in all scenarios is difficult.

0.14 Performance Analysis and Testing

TBC

Conclusion

The YATA approach resolves many of the short-comings of other NRT collaboration approaches. It can be utilized to represent a multitude of data types and hence can cater for a wide range of applications. YJS is the most popular implementation of YATA, though it adds many optimizations beyond the core.