Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types

Petru Nicolaescu
RWTH Aachen University
Lehrstuhl Informatik 5
Ahornstr. 55
52074 Aachen, Germany
nicolaescu@dbis.rwth-aachen.de

Kevin Jahns
RWTH Aachen University
Lehrstuhl Informatik 5
Ahornstr. 55
52074 Aachen, Germany
jahns@dbis.rwth-aachen.de

Michael Derntl
Eberhard Karls Universität
Tübingen eScience-Center
Wilhelmstr. 32
72074 Tübingen, Germany
michael.derntl@uni-tuebingen.de

Ralf Klamma
RWTH Aachen University
Lehrstuhl Informatik 5
Ahornstr. 55
52074 Aachen, Germany
klamma@dbis.rwth-aachen.de

ABSTRACT

Near real-time collaboration using Web browsers is becoming rapidly more and more popular for many applications such as text editing, coding, sketching and others. These applications require reliable algorithms to ensure consistency among the participating Web clients. Operational Transformation (OT) and more recently Commutative Replicated Data Types (CRDT) have become widely adopted solutions for this kind of problem. However, most existing approaches are non-trivial and require trade-offs between expressiveness, suitable infrastructure, performance and simplicity. The ever growing number of potential use cases, the new possibilities of cutting-edge messaging protocols that shaped the near real-time Web, and the use of N-way communication between clients (e.g. WebRTC), create a need for peer-to-peer algorithms that perform well and are not restricted to only a few supported data types. In this paper, we present YATA, an approach for peer-to-peer shared editing applications that ensures convergence, preserves user intentions, allows offline editing and can be utilized for arbitrary data types in the Web browser. Using Yjs, its open-source JavaScript library implementation, we have evaluated the performance and multiple usage of YATA in Web and mobile browsers, both on test and real-world data. The promising evaluation results as well as the uptake by many commercial vendors and open-source projects indicate a wide applicability of YATA.

GROUP '16, November 13-16, 2016, Sanibel Island, FL, USA

DOI: http://dx.doi.org/10.1145/2957276.2957310

CCS Concepts

•**Human-centered computing** → **Collaborative and social computing; Computer supported cooperative work; Collaborative and social computing systems and tools; Synchronous editors;** *Collaborative interaction; Open source software;* •**Computing methodologies** → **Concurrent algorithms;**

Keywords

Near real-time collaborative editing; commutative replicated data types; operational transformation; peer-to-peer information systems

1. INTRODUCTION

Near real-time (NRT) collaboration techniques have been intensively studied by the CSCW community during the past three decades [6, 30, 26, 29]. Mostly directed towards shared editing, highly relevant works include OT [6] algorithms and systems for enabling concurrent editing on the Web. These approaches leverage optimistic and reliable concurrency control mechanisms that lead to widely adopted OT-based systems such as Google Docs. A major advantage is that these systems do not require any locking mechanisms to ensure consistency of the shared data [10]. In a NRT setting, the user agents apply edits immediately to their local copy, while concurrently sending notifications of those changes via communication protocols to the remote user agents. All copies have to eventually have the same content, regardless of the order of the received operations or problems that might occur during the message propagation, such as temporary connection loss.

Recent advances on the Web allowed the development of fast and reliable communication protocols, such as WebRTC, Websockets, XMPP over Websockets, Server-Sent Events, which were also quickly adopted by Web communities, industry and academia. Besides enabling the message propagation technology stack required for shared edit-

ing systems, such protocols can leverage the use of peer-to-peer (P2P) shared editing algorithms and offer a fast and reliable communication infrastructure for their implementations. Thus, P2P approaches became a viable alternative to the more traditional client-server approaches [3]. This is due to the fact that each collaborator receives the updates in a N-way communication and applies them locally, without the need to wait for acknowledgment data, orders or transformed operations from a central server. In consequence, existing NRT collaboration algorithms need to adapt to new performance requirements, which is not a trivial task, especially in the case of massive collaboration, with a scaling number of users and changes [20].

With few exceptions, available OT algorithms are designed for client-server architectures. Algorithms that do support N-way P2P message propagation still rely on propagating a state vector (e.g., COT [29]) with each message. Moreover, algorithms that need preprocessing on the server side cannot be used in such P2P settings. In the past decade, CRDT algorithms [20, 2, 21, 23] emerged in the distributed systems specialization area and were also used in collaborative editing [1]. These have a small document update size and do not rely on vector clocks (e.g., the WooT approach [20]), and can therefore suit the new communication protocols and practices on the Web. A short comparison of the different algorithms is given in Section 2.

Regardless of being P2P or a client-server approach, the available NRT collaboration tools (e.g. OT-based) mostly support linear data or other specific ones (e.g. tree-like [11]) for ensuring the above described properties. Applications based on complex models must therefore map the underlying data to the data structure that is supported by the used collaboration framework. However, this process is time consuming, application-specific, and often hard to achieve. Moreover, there are few open-source implementations that work directly in Web browsers, can be used P2P and offer a developer-friendly, easy and intuitive way for rapidly enabling NRT collaboration on custom data types.

To overcome these shortages of existing algorithms, we present a new approach called YATA (Yet Another Transformation Approach) for enabling NRT collaboration on extendable data types and Yjs, its open-source implementation. We have developed an efficient algorithm that ensures the convergence and intention preservation for collaboration on shared data types. By design YATA inherently leads to fewer conflicts, exposes a favorable run time complexity, supports offline editing (i.e. given a peer that is joining the collaboration session with unsynchronized changes) and collaboration on various data types. The algorithm uses an internal linked list representation, similar to the CDRT approaches WooT [20] and WooTH [1]. However, as it has been proven that keeping deleted operations leads to a loss of performance, YATA solves this disadvantage by introducing a garbage collector for avoiding a drastic increase of unnecessary operations.

For exploiting YATA in Web applications, the algorithm was implemented as an open-source framework called Yjs[1] [19], which can be used with multiple communication protocols, including P2P, federated or client-server. By means of Yjs, which is already garnering interest from the open-source community, we have observed YATA's performance and consider it to be very promising for leveraging lightweight applications, where collaboration logic can be easily engineered on the client side. Finally, besides the usage in real-world open-source products, Yjs has been thoroughly evaluated in a simulated distributed environment for performance, testing the conflict resolution and its scalability, with very good results.

The rest of this paper is organized as follows. First, in Section 2 we give a short introduction to consistency maintenance algorithms, their development within the CSCW community and describe their most relevant features. Section 3 describes the YATA approach, offers a formal proof that it converges regardless of the number of peers involved in collaboration and presents a time and space complexity analysis. Furthermore, we offer the details for extending the approach to other data types in Section 4. Section 5 shortly describes the Yjs framework. The consistency tests performed for our approach and the usage of the framework in real applications are presented in Section 6. Finally, Section 7 summarizes the main contributions and provides an outlook for this work.

2. RELATED WORK

Previous studies present various techniques for solving conflicts in collaborative editing settings. Coined during late 1980s, concurrency control [6, 27] includes locking (pessimistic, mutually exclusive mechanism), transactions, single active participation (token-based participation), dependency detection (timestamps for conflict detection), differential synchronization (client-server asynchronous approach) [7] and three-way merge (classic versioning systems).

The most prominent mechanisms for optimistic concurrency control are OT [6, 26, 27, 28] and CRDT [20, 21, 31, 23]. OT algorithms distinguish between a *control algorithm* and a *transformation function* [28]. The former determine the concurrent operations to be transformed against other operations according to concurrency/context relations. The latter determine how to transform a pair of operations according to their type, position and other parameters. The responsibilities of these two components are defined by a set of transformation properties and conditions. An overview of OT control algorithms, their verification and available state of the art systems are given in [28, 32, 17, 13]. In contrast, CRDT algorithms offer optimistic concurrency control by defining commutative operations that do not conflict with each other [24]. In the following, we provide a brief selection of relevant OT and CRDT algorithms.

The *Google OT* approach for Google Wave/Google Docs [32] is based on the *Jupiter* approach [18]. In both cases the server has to preprocess operations before one is executed and propagated to all clients. There is no direct communication between clients. Similarly, the client also preprocesses received operations before it executes them. Since the client has its own history buffer, it can use the same preprocessing algorithm as the server. *Generalized OT* (GOT) [27], *GOT Optimized* (GOTO) [26] and *AnyUndo* [25] are algorithms that do not depend on a server that preprocesses operations. The AnyUndo OT algorithm extends GOTO with undo capabilities, while reusing its transformation algorithm. The space complexity of this algorithm is nearly as good as the Jupiter approach, but its time complexity can be quadratic. Moreover, it has been shown that the GOTO and AnyUndo approaches work if there are transformation functions that

[1]http://y-js.org

fulfill CP1 [22] and CP2 properties [22]. Sun et al. [28] describe these transformation properties—correctness criteria used in OT literature, needed to achieve convergence (CP1, CP2). Violations of these properties result in inconsistent replicas of a shared document. Similar to GOTO/AnyUndo, the *Context-based OT* (COT) [30] algorithm does not need server-side preprocessing. Furthermore, the algorithm enables direct communication between the clients. The idea behind COT is to save every received operation "as is" in a document state and preprocess it afterwards given the operation context. COT has a simplified design, as only one transformation function must be defined. The downside of this approach is that a state vector must be transmitted with every operation, where the size of the state vector is proportional to the number of users in a collaborative session. The *Time Interval Based Operational Transformation* (TIBOT) approach sends operations to all other clients in an N-way communication model. However, each client is only allowed to send the operations at a certain time interval [16]. This significantly reduces complexity, for the price that updates may take longer to be processed. Similarly, the successor approach TIBOT2 maintains a distributed total ordering schema on the execution sequence of operations, allowing N-way communication between clients. Moreover, it avoids both CP1 and CP2 [32] via a more efficient remote processing approach.

All in all, OT approaches still have several drawbacks, such as scaling problems in peer-to-peer and cloud networks with dynamic (join/leave) user behavior [1] and a high complexity in solving convergence. This is a reason why most OT implementations [13] are restricted to linear data structures or specific ones, such as tree-like [9, 4, 11, 12]. Moreover, the complexity was a reason for slowing down the spread of collaborative applications on the Web, which flourished with the uprise of Google and cloud-based systems.

On the other side, CRDT systems are designed for peer-to-peer environments and to perform and achieve consistency for a big number of users. They can be state-based or operation-based [24]. *Without OT* (WooT) [20] is an approach built for consistency preservation in P2P systems. It uses a monotonic linearization function to ensure convergence as a solution to topological sort. According to [20], WooT supports linear structures, lists, block of texts and ordered trees. It generates partial orders for operations, which are represented as elements and assigned a unique identifier. Moreover, it makes use of an unique identifier for each collaborating site and one for each operation, a value for the effect of an operation and the identifiers of previous and next operations for each given "character value" of an insert operation. WooT uses tombstones (i.e. no elements are deleted, they are only marked for deletion), which means that operations are not deleted. Each site maintains besides the identifier a logical clock, a sequence for character values and a structure for pending operations. Similar to YATA, WooT was designed to work in a P2P architecture and it is independent from the order of reception of operations. WooTO and WooTH [1] improve the WooT algorithm in terms of performance [1], via a linked list and a hash table for optimizing retrieval, update and insertion of operations. They were inspired by the replicated growing array (RGA) approach [23, 1], which introduced update operations in addition to the classic insertions and deletions. Lagoot [31] provides a totally-ordered set of elements and was developed for linear data, extended with an undo mechanism based on a PN-Counter [31, 24]. It does not require tombstones, which makes the algorithm more efficient compared to similar mechanisms such as WooT.

3. THE YATA APPROACH

The YATA approach is created to provide a scalable solution for P2P optimistic concurrency control on the Web. The main goals are to allow the P2P collaborative editing of Web pages (DOM elements), graphs, lists, objects and arbitrary types in the Web browser, using cutting-edge protocols for message propagation. Therefore, the algorithm proposes a basic structure using a linked list, which can be extended to achieve collaboration on new shareable data types. YATA's linked list internal representation and a collection of predefined rules limit the number of possible conflicts and ensure intention preservation and convergence. The core idea is enforcing a total order on the shared data types. YATA also supports offline editing, being meant to cope with requirements coming from both Web and mobile clients, such as small operation updates for low bandwidth, on and off connections, random message order at receive time, etc.

YATA currently supports collaboration on linear data, trees, associative arrays and graphs. Using those types, it is possible to create more complex data types.

In the following we formalize our approach and exemplify YATA'S behavior on text (linear data). After giving the assumptions, definitions and describing how convergence is achieved, we extend the linear representation to more data types and explain how those are further realized using YATA.

3.1 Requirements

Unique identifiers. Each user is represented by an unique identifier (*userid*). Additionally, each user gets an *operation counter* which gets incremented every time a user creates an operation. Upon its creation, the operation thus gets assigned a unique identifier which is composed of the *userid* and the current *operation counter*.

Operations. YATA represents linear data (e.g. text) as a doubly linked list. We define only two types of changes on this representation: *insert* and *delete*. As it is shown in Figure 1, every element in the linked list is represented by an insert operation (also named *insertion*). When an insertion is deleted, it is just marked as such and not removed from the list (i.e. tombstone approach). Therefore, delete operations do not have an effect on our insert algorithm. In Section 3.5 we define a garbage collection mechanism that, in combination with our insert algorithm 3.4, can remove deleted insertions.

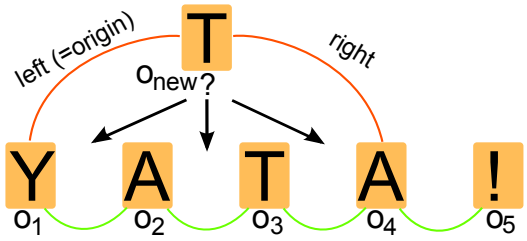

Figure 1: Integration example on text (linear) type.

We denote an insert operation as $o_k(id_k, origin_k, left_k, right_k, isDeleted_k, content_k)$, where id_k is o_k's unique identifier, $content_k$ is the content (e.g. a character), $isDeleted_k$ is a flag that marks an insertion as deleted, and $origin_k$, $left_k$, $right_k$ are references to other already existing insertions. We represent linear data as a doubly linked list S of insertions. Therefore, $left_k$, and $right_k$ reference to the previous node, respectively next node in the list. $origin_k$ denotes the direct predecessor at creation time (i.e., the node after which it was originally integrated to).

We define $<$ as the natural predecessor relation on S.

$$o_1 < o_2 \quad \Leftrightarrow \quad o_1 \text{ is a predecessor of } o_2 \tag{1}$$

$$o_1 \leq o_2 \quad \Leftrightarrow \quad o_1 < o_2 \vee o_1 \equiv o_2 \tag{2}$$

Example. When a user creates a new insertion at a local site, this is integrated between two insertions o_i, and o_j. The newly created insertion is therefore defined as: $o_{new}(id_k, o_i, o_i, o_j, \ false, content_{new})$. Note that $left_{new}$, and $right_{new}$ are defined when an insertion has been applied to the list and may change when new insertions are integrated into S, but $origin_{new}$, defined at insertion creation time is never modified. After the user integrates a new insertion at his local site he sends it (via broadcast), as is, to all users.

A special case occurs when an insertion is performed at the beginning or at the end of S, because there is no insertion to refer to as $left_*$, respectively $right_*$. This can be fixed by using special *delimiters*, which denote the beginning and the end of S, respectively. Therefore, we assume without loss of generality that an insertion always defines $origin_k$, $left_k$, and $right_k$.

3.2 YATA

The example in Figure 1 shows how a received operation o_{new} is integrated in S. Here, the red connections reference the intention of the insertion, which is defined through $left_{new}$ and $right_{new}$ - i.e. insert the letter between these two letters. When the insertion is integrated, YATA assures that it will be placed somewhere between these letters. Convergence is therefore ensured, unless one or more remote operations were already inserted between $left_{new}$ and $right_{new}$, which then leads to a conflict that needs to be solved.

Definition: ***Intention Preservation.***

The intention of an insertion o_i is preserved if and only if the insertion is integrated somewhere between $left_i$ and $right_i$. This notion of intention preservation conforms to the natural perception for the intention of text insertions and it is similar to other definitions found in the literature. In [2], the intention preservation is defined when each character inserted by a user between two other characters in a document keeps its relative position between its neighbors during the editing process.

The concurrent insertion problem.

In the example in Figure 1, the intention of the insertion of "T" is that should be inserted between the characters "Y" and the "A" - e.g., at creation time, "T" sees only "YA". However, a letter sequence "AT" has been already inserted between these two letters. In the example, o_2, and o_3 conflict with o_{new}.

Definition: ***Conflicting insertions.***

Keeping the above notations, assuming $S = left_{new} \cdot c_1 \cdot c_2 \cdot .. \cdot c_n \cdot right_{new}$, then we say that o_{new} conflicts with $c_1..c_n$.

In the following we define a function $<_c$ that specifies a position for o_{new} in the set of conflicting insertions $(c_1..c_n)$. As such o_{new} is integrated between c_i, and c_{i+1} when $c_i <_c o_{new} <_c c_{i+1}$. Furthermore, we show that every site converges when integrating with $<_c$ (i.e., we prove that $<_c$ is a strict total order function).

In the following we will frequently refer to the graphical representation of insertions as it is shown in Figure 1. The insertion o_{new} has three references/connections. On the left hand site of o_{new} there is a $origin_{new}$ connection, and a $left_{new}$ connection to o_1. On the right hand site of o_{new} there is a $right_{new}$ connection to o_4. While $left_{new}$, and $right_{new}$ define the usual predecessor/successor relation in a linked list. The $origin_{new}$ connection will never change and is employed to find the strict total order function $<_c$.

We compose the following three rules in order to find a strict total order $<_c$ on conflicting operations.

Rule 1 We forbid crossing of *origin* connections (red lines in the graphical representation) between conflicting insertions. This rule is easily explained using the graphical representation of insertions in the linked list. As we stated before, every insertion has an origin connection to an insertion to the left (to a predecessor). Only when two operations are concurrently inserted after the same insertion, they will have the same origin.

Figure 2: No line crossing.

Figure 2 illustrates the two cases that are allowed when line crossing is forbidden. Either, one operation is between the other operation and its origin, or the origin of the one operation is a successor of the other operation. Therefore, the following formula must hold for conflicting insertions o_1 and o_2:

$$o_1 <_{rule1} o_2 \Leftrightarrow o_1 < origin_2 \ \vee \ origin_2 \leq origin_1 \tag{3}$$

Rule 2 Specifies transitivity on $<_c$. Let $o_1 <_c o_2$. Then following rule ensures, that there is no o that is greater than o_2, but smaller than o_1, with respect to $<_c$

$$o_1 <_{rule2} o_2 \Leftrightarrow \forall o : o_2 <_c o \rightarrow o_1 \leq o \Leftrightarrow \nexists o : o_2 <_c o < o_1 \tag{4}$$

Rule 3 When two conflicting insertions have the same origin, the insertion with the smaller creator id is to the left. We borrow this rule from the OT approach. But in OT this rule is applied when the position parameters are equal.

$$o_1 <_{rule3} o_2 \Leftrightarrow origin_1 \equiv origin_2 \ \rightarrow \ creator_1 < creator_2 \tag{5}$$

We get retrieve the total order function $<_c$ by enforcing all three rules:

$$\begin{aligned} o_1 <_c o_2 \quad &\Longleftrightarrow \quad o_1 <_{rule1} o_2 \wedge o_1 <_{rule2} o_2 \wedge o_1 <_{rule3} o_2 \\ &\Leftrightarrow \quad o_1 < origin_2 \vee origin_2 \leq origin_1 \\ &\wedge \quad \nexists o : o_2 <_c o < o_1 \\ &\wedge \quad origin_1 \equiv origin_2 \rightarrow creator_1 < creator_2 \\ o_1 \leq_c o_2 \quad &\Leftrightarrow \quad o_1 <_c o_2 \vee o_1 \equiv o_2 \end{aligned} \tag{6}$$

3.3 Correctness

$<_c$ only depends on the $origin_*$ connection, and we specified above, that $origin_*$ never changes. We can conclude that whenever two sites compare conflicting insertions, they will find the same order for insertions. Furthermore, this implies that all sites will eventually converge. Finally, we prove that $<_c$ is a strict total order function, i.e. $\leq_c$ is a total order on conflicting operations. Therefore, we have to show that for all conflicting insertions o_1, o_2, and o_3 the ordering function $\leq_c$ is *antisymmetric*, *transitive*, and *total*.

$$o_1 \leq_c o_2 \wedge o_2 \leq_c o_1 \quad \Rightarrow \quad o_1 \equiv o_2 \text{ (antisymmetry)} \quad (7)$$
$$o_1 \leq_c o_2 \wedge o_2 \leq_c o_3 \quad \Rightarrow \quad o_1 \leq_c o_3 \text{ (transitivity)} \quad (8)$$
$$o_1 \leq_c o_2 \quad \vee \quad o_2 \leq_c o_1 \text{ (totality)} \quad (9)$$

PROOF ANTISYMMETRY. Let o_1, and o_2 be insertions, with $o_1 \leq_c o_2 \wedge o_2 \leq_c o_1$.

Case 1: ($origin_1 \equiv origin_2$):

$$\begin{aligned}
& o_1 \leq_c o_2 \wedge o_2 \leq_c o_1 \\
\overset{6}{\Rightarrow} \quad & (o_1 <_c o_2 \wedge o_2 <_c o_1) \vee o_1 \equiv o_2 \\
\overset{\text{Rule 3}}{\Rightarrow} \quad & (creator_1 < creator_2 \wedge creator_2 < creator_1) \vee o_1 \equiv o_2 \\
\Leftrightarrow \quad & o_1 \equiv o_2
\end{aligned}$$

The reasoning here is that the user id has a total ordering.

Case 2: ($origin_1 < origin_2$):

$$\begin{aligned}
& o_1 \leq_c o_2 \wedge o_2 \leq_c o_1 \\
\Leftrightarrow \quad & (o_1 \leq o_2 \wedge o_2 \leq o_1) \wedge (o_1 \leq_c o_2 \wedge o_2 \leq_c o_1) \\
\Leftrightarrow \quad & (o_1 \leq o_2 \wedge o_2 \leq o_1) \wedge (o_1 \equiv o_2 \vee (o_1 <_c o_2 \wedge o_2 <_c o_1)) \\
\overset{\text{Rule 1}}{\Rightarrow} \quad & o_1 \leq o_2 \wedge o_2 \leq o_1 \\
\wedge \quad & o_1 \equiv o_2 \vee ((o_1 < origin_2 \vee \underbrace{origin_2 \leq origin_1}_{false}) \\
& \wedge (o_2 < origin_1 \vee \underbrace{origin_1 \leq origin_2}_{true})) \\
\Rightarrow \quad & o_1 \leq o_2 \wedge o_2 \leq o_1 \wedge (o_1 \equiv o_2 \vee o_1 < origin_2) \\
\Rightarrow \quad & (o_1 \leq o_2 \wedge o_2 \leq o_1 < origin_2) \vee o_1 \equiv o_2 \\
\overset{origin_2 \overset{!}{<} o_2}{\Rightarrow} \quad & o_1 \equiv o_2
\end{aligned}$$

Case 3: ($origin_2 < origin_1$): Similar to Case 2.

PROOF TRANSITIVITY. Let o_1, o_2, and o_3 be insertions, with $o_1 \leq_c o_2 \wedge o_2 \leq_c o_3$, o_3 conflicts with o_1, and w.l.o.g. $o_1 \neq o_2 \neq o_3$.

$$\begin{aligned}
& o_1 <_c o_2 \wedge o_2 <_c o_3 \\
\overset{\text{Rule 2}}{\Rightarrow} \quad & (\forall o : o_2 <_c o \rightarrow o_1 \leq o) \wedge (o_2 <_c o_3) \\
\Rightarrow \quad & (o_2 <_c o_3 \rightarrow o_1 \leq o_3) \wedge (o_2 <_c o_3) \\
\Rightarrow \quad & o_1 < o_3 \\
\Leftrightarrow \quad & o_1 <_c o_3
\end{aligned}$$

PROOF TOTALITY. Let o_1, o_2 be insertions, with $o_1 \neq o_2$. Totality is fulfilled if the following statement holds:

$$o_1 <_c o_2 \vee o_2 <_c o_1 \quad \Leftarrow \quad true$$
$$\text{if and only if}$$
$$\neg(o_1 <_c o_2 \vee o_2 <_c o_1) \quad \Rightarrow \quad false$$

When we apply the ordering relation (6) we get the formula in Figure 3 and show for each case that totality is fulfilled.

Case 1 ($origin_1 \equiv origin_2$ and $creator_1 < creator_2$):

$$\begin{aligned}
& \text{First steps are depicted in Figure 3} \\
\Rightarrow \quad & \exists o_2 < o < o_1 \\
\Rightarrow \quad & o_2 < o_1 \\
\overset{\text{Rule 3}}{\Rightarrow} \quad & false
\end{aligned}$$

Case 2 ($origin_1 \equiv origin_2$ and $creator_2 > creator_1$): Similar to case 1.

Case 3 ($origin_1 < origin_2$):

$$\begin{aligned}
& \text{Figure 3} \\
\Rightarrow \quad & (\neg(o_1 < origin_2) \vee (\exists o : o_2 < o < o_1)) \\
& \wedge (\exists o : o_1 < o < o_2) \\
\Leftrightarrow \quad & (\neg(o_1 < origin_2) \wedge (\exists o : o_1 < o < o_2)) \\
& \vee (\exists o : o_2 < o < o_1 \wedge \exists o : o_1 < o < o_2) \\
\Rightarrow \quad & (\neg(o_1 < origin_2) \wedge (\exists o : o_1 < o < o_2)) \\
& \vee (o_2 < o_1 \wedge o_1 < o_2) \\
\overset{\text{antisymmetry}}{\Rightarrow} \quad & \neg(o_1 < origin_2) \wedge (\exists o : o_1 < o < o_2) \\
\Rightarrow \quad & origin_2 \leq o_1 \wedge o_1 < o_2 \\
\overset{\text{assumption}}{\Rightarrow} \quad & origin_1 < origin_2 \leq o_1 \wedge o_1 < o_2 \\
\Rightarrow \quad & origin_1 < origin_2 \leq o_1 < o_2 \\
\overset{o_1 \text{ and } o_2 \text{ conflict}}{\Leftrightarrow} \quad & origin_1 < origin_2 < o_1 < o_2 \\
\overset{\text{Rule 1}}{\Rightarrow} \quad & false
\end{aligned}$$

Case 4 ($origin_2 < origin_1$): Similar to Case 3.

$$\begin{aligned}
& \neg(o_1 <_c o_2) \wedge \neg(o_2 <_c o_1) \\
\Leftrightarrow \quad & \neg((o_1 < origin_2 \vee origin_2 \leq origin_1) \wedge (\nexists o : o_2 <_c o < o_1) \wedge (origin_1 \equiv origin_2 \rightarrow creator_1 < creator_2)) \\
\wedge \quad & \neg((o_2 < origin_1 \vee origin_1 \leq origin_2) \wedge (\nexists o : o_1 <_c o < o_2) \wedge (origin_2 \equiv origin_1 \rightarrow creator_2 < creator_1)) \\
\Leftrightarrow \quad & (\neg(o_1 < origin_2) \wedge \neg(origin_2 \leq origin_1)) \vee (\exists o_2 <_c o < o_1) \vee (origin_1 \equiv origin_2 \wedge \neg(creator_1 < creator_2)) \\
\wedge \quad & (\neg(o_2 < origin_1) \wedge \neg(origin_1 \leq origin_2)) \vee (\exists o_1 <_c o < o_2) \vee (origin_2 \equiv origin_1 \wedge \neg(creator_2 < creator_1)) \\
\Leftrightarrow \quad & (\neg(o_1 < origin_2) \wedge \neg(origin_2 \leq origin_1)) \vee (\exists o_2 < o < o_1) \vee (origin_1 \equiv origin_2 \wedge \neg(creator_1 < creator_2)) \\
\wedge \quad & (\neg(o_2 < origin_1) \wedge \neg(origin_1 \leq origin_2)) \vee (\exists o_1 < o < o_2) \vee (origin_2 \equiv origin_1 \wedge \neg(creator_2 < creator_1))
\end{aligned}$$

Figure 3: Derivation of totality on (6).

3.4 Insert Algorithm

Previously, we proved that there exists a total order relation on conflicting insertions. In this section we show how we can compute the new position for an insertion, when we already have an ordered list of insertions.

Listing 3.4 shows how the conflicting insertion can be solved algorithmically. The algorithm exploits property (3) (no line crossing) as a breaking condition. Therefore, we stop computing when origin connections definitely will cross.

The worst case time complexity of the algorithm is $O(|C|^2)$ where $|C|$ is the number of conflicting operations. In the case that the breaking condition is reached in the first iteration, no positions are compared. This is why the best case time complexity is $O(1)$. A complexity analysis is presented in Section 3.7.

```
// Insert 'i' in a list of
// conflicting operations 'ops'.
insert(i, ops){
  i.position = ops[0].position
  for o in ops do
    // Rule 2:
    // Search for the last operation
    // that is to the left of i.
    if (o < i.origin
        OR i.origin <= o.origin)
      AND (o.origin != i.origin
        OR o.creator < i.creator) do
      // rule 1 and 3:
      // If this formula is fulfilled,
      //  i is a successor of o.
      i.position = o.position + 1
    else do
      if i.origin > o.origin do
        // Breaking condition,
        // Rule 1 is no longer satisfied
        // since otherwise origin
        // connections would cross.
        break
}
```

3.5 Garbage Collection

In the literature, garbage collection has been also proposed in [21] where "cold" areas of a document are identified or in Logoot [31], which uses a graveyard for removed operations. Conceptually, an insertion marked for deletion can be garbage collected when all sites received the remove operation and have in their internal representation the operation that is to be garbage collected. However, it is hard to determine if all collaborators know simultaneously that a content was deleted. Ideally, using classic methods such as state vectors, a mechanism where YATA ensures that all sites have applied a removal can be a candidate solution. As a downside, such a mechanism would require more network resources and leads to a decrease in performance, especially in P2P settings. An optimal solution to this issue is still being considered.

In the current approach, the problem is simplified by assuming that all users retrieved a certain remove operation after a fixed time period t which can be set according to the expected protocol and network characteristics (e.g., 30 s). In practice, YATA uses two buffers for garbage collection, to ensure that list elements are not directly removed. As such, once o_k can be garbage collected, it will be moved into the first buffer. If nothing changes, after t seconds it will be copied into the second array and from here will be removed by the garbage collector (i.e., can be safely removed from the list and the buffer). From our practical experiences and the use in production, such a delay is sufficient to ensure that content will be removed safely, without any losses that may lead to inconsistencies. This is in line with experiments performed for assessing the NRT criteria and measuring the time in which operations are being applied. These experiments (cf. Figure 9, Section 6) show that the average time for receiving and applying a remove operation using text (with a length under 10^3 characters) at a remote site is approx. 12 ms. Under the same conditions, receiving and applying a single remove operation with a length of 10^5 characters at a remote site is done within an average time of 39.3 ms (SD 2.45 ms), from a pool of ten measurements.

As a consequence of YATA's rules, in some cases it is not possible to remove insert operations. The reason is that for an operation that is inserted between two undeleted insert-type operations, this could lead to a deleted predecessor or successor (cf. Figure 4).

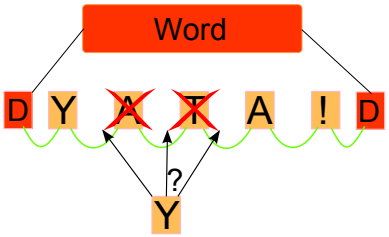

Figure 4: Insertion between deleted characters.

In order to ensure consistency, YATA demands that a new insertion is always inserted between the most left non-deleted character and its direct successor. Only then, the garbage collector can remove all operations that are to the right of the first deleted insertion.

Furthermore, due to its design, the garbage collector in YATA may break late join mechanisms. This is because when a user is offline for a period longer than t seconds, it will still hold references to deleted operations, while online users who already performed certain removals do not. Therefore, YATA does not support garbage collection for a site while it is offline.

3.6 Ofline Editing Support

YATA supports offline editing using the internal data representation which is maintained at each client. Once clients are online, YATA performs a check for diverged states of the shared data and synchronizes it.

Every site holds a state vector. It saves the next expected operation id per user. As an example, consider *user1* with *userid* 1 is in a session with *user2* with *userid* 2. Both users created two operations. As we explained above, the operation id is defined as a tuple of *userid* and *operationcounter*. Therefore, the state vector is expressed as:
$[(1, 2), (2, 2)]$ (assuming we start counting with 0).

For synchronization, the state vector is not sent with each operation, but it is sent only once to all clients. A user that

receives a state vector compares it with the local state vector and sends all remaining operations to the synchronizing client. In order to make operations integrable on the remote instances, operations are sent in the order and the form in which they were created. Our YATA's implementation can transform integrated operations to their original form.

3.7 Complexity Analysis

The complexity of YATA depends on the retrieval efficiency of insertions. As such, the retrieval of a specific operation can be efficiently implemented using a balanced tree, which exposes time complexities of $O(log(H))$ for insertion and retrieval. Here, H denotes the number of all applied operations in the balanced tree implementation. The best case time complexity for applying an operation is $O(log(H))$ - i.e. there are no conflicting operations. The worst case time complexity is $O(log(H)+C^2)$ and only depends on the amount of conflicting operations (C). A comparison with other CRDTs performance, adapted from [1] is depicted in Table 1. In this comparison, H the total number of operations that affect a shared document.

CRDT	LOCAL		REMOTE	
	INS	DEL	INS	DEL
WooT	$O(H^3)$	$O(H)$	$O(H^3)$	$O(H)$
WooTO	$O(H^2)$	$O(H)$	$O(H^2)$	$O(H)$
Logoot	$O(H)$	$O(1)$	$O(H \cdot log(H))$	$O(H \cdot log(H))$
RGA	$O(H)$	$O(H)$	$O(H)$	$O(log(H))$
YATA	$O(log(H))$	$O(log(H))$	$O(H^2)$	$O(log(H))$

Table 1: Worst case time-complexity analysis, adapted from [1].

Discussion As it can be observed, compared to classical CRDT tombstone approaches (such as WooT or WooTO), YATA has a better time complexity. As expected, non-tombstone approaches (such as Logoot of RGA) output a better time complexity for some operation types. Regardless of the complexity, YATA offers an approach for client-side conflict resolution, which is not common among existing CRDTs. It proposes a simple data type that can be used to compose more complex ones (cf. Section 4). In Section 6, we show that YATA's implementation fulfills NRT scenarios and give an overview on its usage performance.

A downside of the YATA approach is that when each character is represented as an operation, the space complexity is $O(|D|)$, where $|D|$ is the size of the shared document. In OT for example, it is possible to have an empty history buffer, when garbage collection is enabled.

However, YATA outputs the following advantages over OT approaches:

- Time-complexity: YATA reduces time to synchronize, when late join is supported.
- The size of propagated messages is small: there are no state vectors that need to be propagated with each operation. OT approaches for handling tree-like data require the user to send a path-vector with each operation. This usually has the length of the depth of the changed element, which is not necessary in YATA.
- It is possible to define many data types.

4. EXTENDABLE TYPES

This section describes some of the basic operation types and the general data structures which YATA supports. Building on top of the data structures, one can implement certain abstract data types and thus enable collaboration on common data formats such as JSON and XML. The supported types currently include linear data types (e.g., arrays, linked lists, sorted arrays, bitmaps), trees, graphs and associative arrays.

4.1 List Manager Operation

A List Manager (cf. Figure 5) is an abstract operation type that manages insert operations. It basically handles two delimiters that denote the beginning and the end of the list, as exemplified in the algorithm's description. Hence, new insertions are placed somewhere between these delimiters according to YATA's rules.

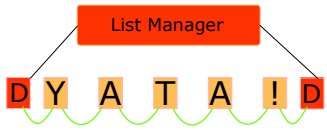

Figure 5: Illustration of the List operation type.

The List Manager operation also handles how to address the elements in the associative list and how to transform it to a certain data type (e.g. String). It represents linear data structures such as lists and arrays, but it can also be used in order to represent tree-like data structures. In this case, the trees are achieved by allowing the content of insertions to contain in their turn List Managers.

4.2 Replace Manager Operation

YATA supports only insert and delete operations. However, when dealing with more complex types, update operations are also required in order to ease the development. As such, YATA supports updates of existing content by offering a dedicated type which enables content replacement. A Replace Manager (cf. Figure6 handles the replace functionality. As a basic example, consider the case where two users (with user ids 1 and 2) concurrently replace the number 0 in a text with their respective user id. In order to keep consistency, each site should reflect the replace operation and reach the same content, i.e. either 1 or 2 will replace the old number 0. YATA solves this problem by transforming it into an already solved problem, using data types which ensure consistency.

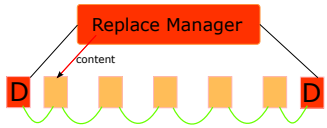

Figure 6: Illustration of the Replace Manager operation type.

The Replace Manager inherits its functionality from the List Manager. Using this data representation, the first insertion (which must not be a Delimiter) of a Replace Manager denotes the actual content. In consequence, in order to replace this content once a user performs a new insertion, the new content will be added as the first insertion of the Replace Manager. In Figure 6 the red line references the current content of the Replace Manager. Since YATA ensures that all sites will have the same content in the end, all sites eventually reference the same insertion as the content of the Replace Manager. Moreover, in order to free up memory all other insertions can be implicitly deleted.

4.3 Map Manager Operation

A map is also known as a dictionary, or an associative array. It is an abstract data type that maps *keys* to *values*.

Figure 7 depicts YATA's representation of a Map Manager operation. In order to support concurrent actions on a shared map, we assign each *key* to a Replace Manager. Therefore, the values addressed by keys will converge upon concurrent changes performed by multiple users i.e., users collaboratively edit the content addressed by a specific key. We can use any map data structure to map *keys* to Replace Managers. The values can contain primitive data types or YATA own types. This construct is suitable for easily enabling collaboration on name/value pairs (e.g., objects, dictionaries, etc.). The current *value* of a *key* is replaced and retrieved by accessing the respective Replace Manager.

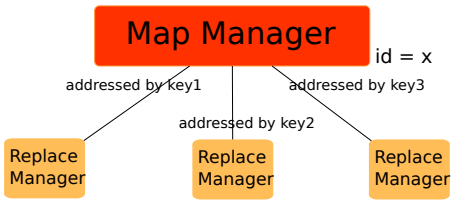

Figure 7: Illustration of the Map Manager operation type.

4.4 Representation of Specific Data Formats

Using YATA, by combining the simple types explained above, data formats such as JSON or XML can be realized as shared data types.

The JSON data format is built on collections of name/-value pairs and ordered lists of values. Hence, with YATA JSON can be easily created by using a Map Manager, in combination with other data types which can be used for the attributes, such as further Map Manager operations or List operations (for attributes implementing strings or arrays).

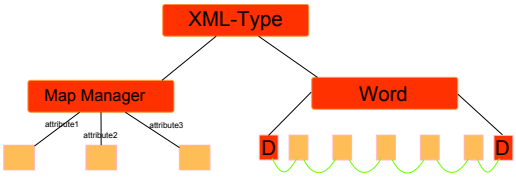

Figure 8: Illustration of XML as an operation type.

YATA can also enable NRT collaboration on XML (cf. Figure 8). An XML DOM Element[2] has XML attributes and an ordered list of children (XML elements or XML text). Therefore, we compose an XML-Type in YATA as a Map Manager (which handles the XML attributes) and a List Manager (which handles the children). However, since the List Manager and the Map Manager support to insert any value, we have to put some restrictions to the structure of the XML-Type:

- The Map Manager, which handles the XML attributes, must only map from a non-empty string to a non-empty string
- The List Manager, which handles the children, must only contain other XML types, or strings which represent XML text elements

5. YJS: THE P2P SHARED EDITING FRAMEWORK

As already introduced, Yjs [19] is the open-source implementation of the YATA approach. In contrast to similar OT, CRDT and shared editing implementations that support only a very limited number of document structures or represent them differently, the Yjs framework encourages developers to build custom data types. A custom type can use existing implemented types (as previously explained) in order to give meaning to the actions on the data and to fire custom events. At the time of writing this paper, Yjs has implemented support for *list*, *associative arrays*, *XML*,*text*, and *rich text* types. Yjs works on modern Web browsers, including mobile (on Android devices) and offers a quick and easy way to embed collaboration in Web applications.

In order to maintain modularity and to be able to employ Yjs in various Web engineering settings, the communication protocols and the shared data type formats support are implemented as dedicated interchangeable modules. This greatly simplifies the process of integrating the framework into an existing project, since such projects typically use diverse communication protocols (e.g., WebRTC, Web Sockets, XMPP), and collaborate on various data types and data formats (e.g., XML, JSON, graphs). The support for the different communication protocols is implemented in connector modules. The collection of connector modules and type modules are available as open-source JavaScript libraries on GitHub[3]. Currently, connectors are implemented for WebRTC, Web Sockets and XMPP.

6. EVALUATION

6.1 Performance

Apart from the correctness reasoning explained above, we have evaluated the approach and the Yjs library in terms of responsiveness - how quickly an operation is integrated into each user's local structure and scalability - what is the impact of the number of users in this setting [19].

In order to evaluate the speed and correctness of YATA we performed multiple automatic tests simulating many users working exhaustively on a single shared document. The simulation introduced certain constraints: generation of random operations, different arrival orders of operations at different sites and network delay with operations which were

[2]http://www.w3.org/TR/REC-xml/
[3]https://github.com/y-js

not ready to be applied because they were dependent on other pending operations. As our implementation is meant for client-side applications and runs on the Web, we could not use existing benchmarking tools described in the literature [1]. Moreover, using collaboration logs from online text editing systems such as Wikipedia would have lead to very few conflicts on the shared document. Instead, we chose an alternative approach with the goal to measure the fulfillment of the NRT scenario in distributed settings. We first used the testing environment with a test connector to measure the scaling capability with the number of operations by measuring how many concurrent operations YATA can apply locally. Second, we measured the time performance for applying insert and delete operations between two Yjs instances, including the network delay using the Web Sockets connector. Similar to [1], we consider that the framework performs in NRT if the response time for the insert operation is in average under 50ms.

The testing environment can be configured with respect to number of users and number of actions that are created. Furthermore, it is possible to configure the test environment in such a way that the collaboration is restricted to a specific data type (e.g. text only or text and JSON with primitive data types). The users are collaborating concurrently with each other, whereby each simulated user can perform one of the following actions: *wait* for a time period, *retrieve an operation* from a random collaborator, *go offline* (all operations that are currently sent to the users get lost. After going offline, the user is allowed to create more operations, simulating late join), *go online* (user reconnects to all users, retrieves all missing operations, and sends all offline generated operations). For text, a user can perform insert and delete actions. For JSON, the user can perform randomly the following actions: find a random child, create a new property, replace an existing property or delete a property.

When a configured amount of actions are executed, the simulated users stop generating additional actions and wait for all incoming operations to be executed. When the data types of every simulated instance converge, we consider that the test framework succeeds.

We used our test framework with the creation and execution of 10000 actions on Text and JSON with a specified number of users, ranging from one to ten. The test ran on one CPU only (Intel i7 - 3.40 GHz). All mentioned constraints were considered. We ran the test 15 times in order to get good average times. The time to create 10000 random actions and apply them to all collaborators was measured and divided by the number of users times the number of created actions: operation per millisecond $= \frac{time}{10000 \times |users|}$ The time for transforming operations did not thwart the performance noticeably. For a setting with 7 and 8 users, the test framework registered 250 operations executed in one ms, whereas for 9 and 10 users the average time was approx. 230 operations per ms. This did not even change when we restricted the test framework to work on text only, where even more conflicts should happen.

The time performance was measured using two Yjs instances using one Chrome browser on a laptop (Intel i7 - 2.8 GHz) with a wireless Eduroam connection. Using Web Sockets connector, we measured the time difference (in ms) between applying an insert or delete operation on the first instance and applying the same operation on the second instance.

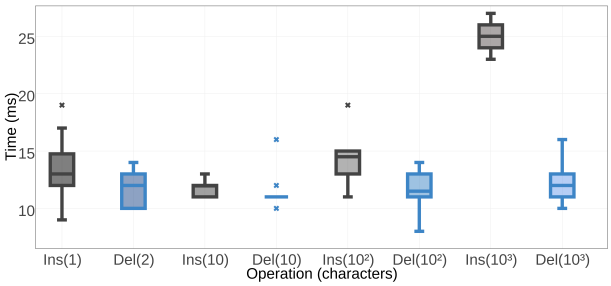

Figure 9: NRT performance for applying one remote operation with different content sizes

This shows the delay between creating an operation on a site and seeing the effects of that change at the second site, including the delay created from the message propagation across the communication channel (i.e., Web Sockets).

The experiment (cf. Figure 9) was performed with a single insert or delete operation using a text editor. Each operation was executed ten times in order to obtain a reliable estimation. We also increased the content of each operation (in characters), ranging from 1 to 10^3 characters. The results for the insert operations are represented in gray and for the delete operations in blue. As it can be observed, the average time for all operations is smaller than 25 ms. We also used the same environment to write a text of 1000 characters. Here, we obtained an average execution time for an operation of 12 ms, which is consistent with the results presented in the figure for inserting and deleting one character.

Overall, the test results fulfill our expectations for the framework's performance in NRT settings.

6.2 Applications

Overall, the Yjs library with its various connectors and types was well received by the open-source developer community. The library's Website and repository analytics show that the Website gathered over one year more than 11600 page views from over 4500 users and the library over 800 downloads.

Yjs was used to enable shared editing with the popular Quill rich text editor, which is very similar to Google Docs. The richtext component is already used by two companies in production, for NRT shared text editing on the Web, embedded in WebRTC video meeting tools.

As shown in Figure 10, Yjs is currently used for synchronizing 3D objects movement across multiple users in the Web browser[4].

Here, the Map Manager is used in order to synchronize the position properties of a Javascript 3D object. This is possible by observing changes in the x3dom 3D object representation and ensuring that the object's coordinate values are converging across all sites. The synchronization performance and accuracy is currently being evaluated with students of the Faculty of Medicine from RWTH Aachen University, in an Exploratory Teaching Spaces project that aims to support formal and informal learning through NRT collaboration.

Furthermore, Yjs was used to create a tool for achieving

[4]http://eiche.informatik.rwth-aachen.de/3dnrt

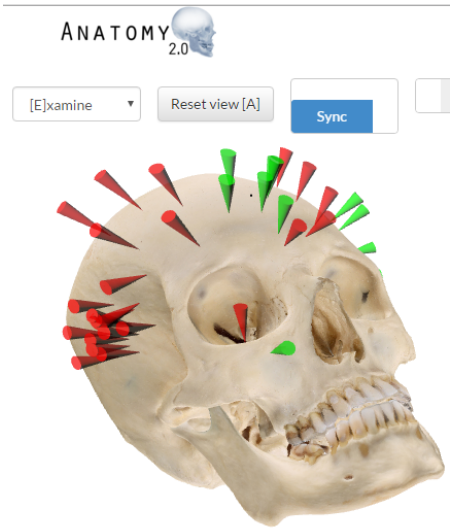

Figure 10: Illustration of the 3D objects synchronization with Yjs.

liquid Web applications [8], by providing a seamless user experience in using different Polymer Web components distributed across various devices. The evaluation of a Yjs-based collaborative drawing tool on Web videos [14] (performed using the WebRTC connector) showed that during concurrent drawings the framework proved to be reliable while keeping the NRT behavior, for all participating peers. Several other examples for manipulating text, HTML5 elements, JSON objects and additional usages in various projects are provided on the Yjs Website.

Moreover, Yjs is currently being integrated into SyncMeta [5] (cf. Figure 11), a framework for enabling NRT collaborative (meta) modeling on the Web.

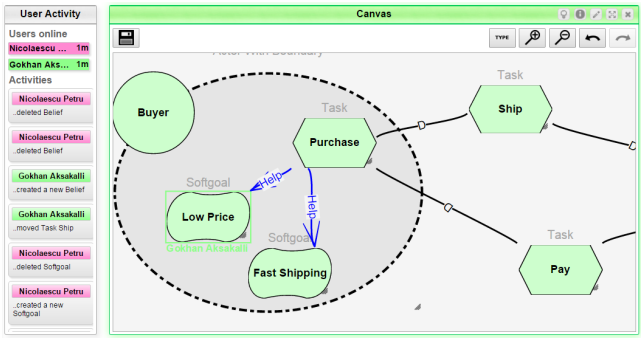

Figure 11: Illustration of graph editing with Yjs.

SyncMeta is using an OT solution for enabling shared editing on text and ensuring that the created nodes and edges are converging during the modeling process at all sites. However, this solution uses timestamps for ensuring the consistency in a P2P manner, which does not scale well with an increasing number of users. However, by synchronizing the underlying JSON model representation using YATA, Yjs is a perfect candidate for achieving scalable optimistic NRT collaborative modeling. The Yjs implementation benefits from the advantages of YATA over OT algorithms. Thus, it guarantees a faster synchronization of the graph elements and their respective properties with an increasing number of users and operation content size in P2P settings. Moreover, Yjs SyncMeta reduces the network traffic through the smaller size of messages, while keeping the reliability and hence the convergence of the shared models across collaborators.

7. CONCLUSION AND FUTURE WORK

In this work, we have described YATA, a new scalable P2P approach for maintaining consistency of shared arbitrary data types, designed for NRT settings on the Web. We proved that it ensures convergence and that it has several advantages compared to other existing algorithms for shared editing. The contributions of YATA rely on its flexibility for various shared data types, as well as in its performance regarding the necessary number of exchanged updates between collaboration sites, time to convergence and its support for offline editing (enabled by YATA's late join mechanism). Its' open-source implementation, Yjs, supports multiple NRT communication protocols and can be easily used into existing and new applications for facilitating NRT collaboration and shared editing.

In the future, we plan to change the available garbage collection mechanism in order to support operation removal while working offline. Moreover, we want to improve the Yjs framework implementation with new supported data types and persistence mechanisms for increasing the performance. Concerning persistence we are exploring efficient methods for the storage of shared data in the P2P network that can be included into the framework.

As the main motivation for developing YATA and Yjs was to enable easy NRT collaboration for online communities, we plan to use Yjs for the collaborative engineering of Web applications and synchronizing the application content and state across multiple devices [15].

8. ACKNOWLEDGMENTS

This research was funded in part by the European Commission in the projects "Layers" (FP7-318209) and "Metis" (531262-LLP-2012-ES-KA3-KA3MP).

9. REFERENCES

[1] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating CRDTs for Real-time Document Editing. In *Proceedings of the 11th ACM symposium on Document engineering*, pages 103–112. ACM New York, 2011.

[2] L. Andre, S. Martin, G. Oster, and C.-L. Ignat. Supporting Adaptable Granularity of Changes for Massive-scale Collaborative Editing. In *Collaboratecom 2013*. IEEE, 2013.

[3] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[4] A. H. Davis, C. Sun, and J. Lu. Generalizing Operational Transformation to the Standard General Markup Language. In *CSCW '02*, pages 58–67. ACM, 2002.

[5] M. Derntl, P. Nicolaescu, S. Erdtmann, R. Klamma, and M. Jarke. Near Real-Time Collaborative

Conceptual Modeling on the Web. In *ER'15*, volume 9381, pages 344–357. Springer International Publishing, 2015.

[6] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *ACM SIGMOD '89*, volume 18, pages 399–407. ACM, 1989.

[7] N. Fraser. Differential synchronization. In *Proceedings of the 2009 ACM Symposium on Document Engineering*, pages 13–20. ACM, 2009.

[8] A. Gallidabino and C. Pautasso. The Liquid.js Framework for Migrating and Cloning Stateful Web Components across Multiple Devices. In *WWW'16 Companion*, pages 183–186, 2016.

[9] A. R. S. Gerlicher. A Framework for Real-time Collaborative Engineering in the Automotive Industries. In *CDVE'06*, pages 164–173. Springer-Verlag, 2006.

[10] J. Grudin. Computer-Supported Cooperative Work: History and Focus. *Computer*, 27(5):19–26, 1994.

[11] C. Ignat and M. C. Norrie. Tree-based Model Algorithm for Maintaining Consistency in Real-Time Collaborative Editing Systems. In *CSCW '02*. ACM, 2002.

[12] C.-L. Ignat and M. C. Norrie. Multi-level Editing of Hierarchical Documents. *Computer Supported Cooperative Work*, 17(5-6):423–468, 2008.

[13] I. Koren, A. Guth, and R. Klamma. Shared Editing on the Web: A Classification of Developer Support Libraries. Collaboratecom 2013, pages 468–477. IEEE, 2013.

[14] I. Koren, P. Nicolaescu, and R. Klamma. Collaborative Drawing Annotations on Web Videos. In *ICWE'15*, volume 9114. Springer, 2015.

[15] D. Kovachev, D. Renzel, P. Nicolaescu, I. Koren, and R. Klamma. DireWolf: A Framework for Widget-based Distributed User Interfaces. *Journal of Web Engineering*, 13(3&4):203–222, 2014.

[16] R. Li, Du Li, and C. Sun. A Time Interval Based Consistency Control Algorithm for Interactive Groupware Applications. In *ICPADS '04*, pages 429–436. IEEE Computer Society, 2004.

[17] Y. Liu, Y. Xu, S. J. Zhang, and C. Sun. Formal Verification of Operational Transformation. In *FM 2014: Formal Methods*, volume 8442, pages 432–448. Springer International Publishing, 2014.

[18] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *UIST '95*, pages 111–120. ACM, 1995.

[19] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *ICWE'15*, volume 9114, pages 675–678. Springer, 2015.

[20] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *CSCW '06*, pages 259–268. ACM Press.

[21] N. Preguiça, J. M. Marques, M. Shapiro, and M. Letia. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems, 2009*, pages 395–403. IEEE, 2009.

[22] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors. In *CSCW '96*, pages 288–297. ACM, 1996.

[23] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types, 2011.

[25] C. Sun. Undo As Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, 9(4):309–361, 2002.

[26] C. Sun and C. Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *CSCW '98*, pages 59–68.

[27] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.

[28] C. Sun, Y. Xu, and A. Agustina. Exhaustive Search of Puzzles in Operational Transformation. In *CSCW '14*, pages 519–529. ACM, 2014.

[29] D. Sun and Chengzheng Sun. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, 2009.

[30] D. Sun and C. Sun. Operation Context and Context-based Operational Transformation. In *CSCW '06*, pages 279–288. ACM Press, 2006.

[31] S. Weiss, P. Urso, and P. Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010.

[32] Y. Xu, C. Sun, and M. Li. Achieving Convergence in Operational Transformation: Conditions, Mechanisms and Systems. In *CSCW '14*, pages 505–518. ACM, 2014.