Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Operational Transformation su documenti strutturati

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Davide Montanari

Sessione II
Anno Accademico 2017/2018

Indice

Introduzione

Nei sistemi di collaborazione real-time, il problema di decidere come risolvere e quali tecniche adottare per la risoluzione di conflitti tra operazioni concorrenti non è banale, poiché se affrontato inadeguatamente può portare a stati incoerenti del modello di dati condiviso tra gli utenti connessi al sistema.
Questa problematica può essere ulteriormente resa complessa da altri fattori, quali ad esempio la latenza della rete o il ritardo dovuto alla computazione delle operazioni, siccome si tratta quasi sempre di un ambiente distribuito in cui i vari client possono essere anche a distanza di migliaia di kilometri. In questa tesi verrà dunque analizzata una tecnologia che permette la risoluzione di conflitti in ambienti di collaborazione real-time per l'editing di documenti testuali denominata Operational Transformation.
Tale tecnologia gode di un ricco insieme di possibilità ed è stata usata per supportare un vasto range di applicazioni per la collaborazione online di text-editing, grafica, rich-text, tool per la creazione di presentazioni, digital media deisgn, etc.
Il primo sistema OT venne proposto nel 1989 nell'articolo scientifico "Concurrency Control in Groupware Systems" di Ellis e Gibbs, per supportare appunto, il controllo della concorrenza nell'editing collaborativo in real-time di documenti di testo non strutturati.
Negli anni successivi alla pubblicazione di tale articolo furono individuati alcuni errori riguardanti la correttezza dei sistemi OT implementati sulla base delle teorie sviluppate fino a quel momento. In letteratura vennero proposti svariati approcci per la risoluzione dei problemi incontrati e iniziarono a sorgere i primi impieghi di OT in applicazioni specifiche.
Il primissimo sistema ad aver scelto di affidare il proprio controllo della concorrenza ad

OT fu Jupiter, reso noto dopo la pubblicazione del paper "High-Latency Low-Bandwidth Windowing in the Jupiter Collaboration System" nel 1993. Jupiter oltre ad essere stato uno dei primissimi sistemi OT, ha anche apportato un contributo massiccio alla teoria di Operational Transformation, introducendo l'utilizzo di un server centrale che processa tutte le richieste di edit del documento, forzando così la comunicazione dei client e limitandone la gestione dei conflitti a due siti per volta.
Proseguendo con l'excursus storico di OT, nel 2009 si arrivò all'annuncio da parte di Google di un nuovo sistema di collaborazione real-time che integrava l'invio di e-mail, servizi di messaggistica istantanea, di blogging, wiki e di social networking, che però fallì miseramente e fu accantonato nel giro di un paio di anni.
Nel 2012 viene lanciato Google Drive e al Google I/O del 2013 viene presentato Google Docs, un web-based word processor che introduce la possibilità di editing collaborativo sfruttando appunto Operational Transformation e riciclando parte della progettazione e dell'engine adoperate in precedenza per la realizzazione di Google Wave.
Google Docs stesso fa uso di un server centralizzato basato su acknowledgments delle mutazioni, o operazioni, del documento; inoltre introduce l'uso di snapshot (insieme minimale di mutazioni) per il recupero dei document states e implementa l'utilizzo di una coda per il buffering delle operazioni locali client side. Maggiori delucidazioni riguardo i meccanismi di Jupiter, Google Wave e Google Docs sono contenute nella sezione 6 del capitolo 2.
Prima di analizzare le caratteristiche e le proprietà di Operational Transformation però è necessario esaminare le tecnologie che permettono di realizzare applicazioni real-time in Javascript, il linguaggio client-side più utilizzato nel web developing.
Javascript mette a disposizione tre modalità fondamentali di comunicazione real-time col server: Ajax Long-Polling, che consiste nell'interrogazione del server a intervalli regolari, ricevendo risposta quando il server produce nuovi dati; Server Sent Events, che implicano una comunicazione monodirezionale dal server al client e che permettono al server di pushare dati al client in qualunque momento; WebSockets, che permettono l'apertura di una sessione di comunicazione interattiva tra il browser del client e il serve.

Permettendo una comunicazione bidirezionale, dunque, i WebSockets sono la tecnologia più indicata per la gestione di applicazioni real-time in Javascript. Viene fornita una descrizione completa dell'argomento nel capitolo 1.
Ora sulla base delle tecnologie esistenti per la realizzazione di applicativi di questo genere è necessario entrare nel merito del paradigma che permette la collaborazione di più utenti nella gestione e modifica di dati all'interno essi.
Operational Transformation si fonda sull'impiego di funzioni di trasformazione, che prendono in input le operazioni applicate dai client sul data model condiviso e risolvono situazioni conflittuali che porterebbero il documento a raggiungere stati incoerenti.
Dunque il problema più rilevante rimane quello della coerenza del documento finale, e appunto per questo motivo sono stati proposti più modelli di coerenza.
Il modello che elenca i requisiti base che ogni sistema OT deve avere è il modello CC(Causality Convergence), che richiede la preservazione della causalità (ordinamento cronologico) e la convergenza del documento finale (al termine dell'applicazione delle operazioni, tutte le copie del documento devono essere identiche).
Un secondo modello molto importante è il modello CCI(Causality Convergence Intention Preservation), che richiede i requisiti di CC, ma aggiunge la preservazione delle intenzioni, ovvero che per ogni operazione vengano rispettate le sue intenzioni, o meglio, quelle del client. Un altro modello proposto è stati CSM (Causality Single Operation Multiple Operations), che richiede la causalità come in CC e che l'effetto di una singola operazione in qualunque stato abbia lo stesso effetto che eseguirla nel suo stato di generazione; richiede inoltre che la relazione che intercorre tra due operazioni rimanga invariata a prescindere dallo stato di esecuzione. L'ultimo modello proposto è CA(Convergence Admissibility), che introduce il concetto di ammissibilità di un'operazione, il quale richiede che ogni operazione è ammissibile nel proprio stato di esecuzione, prima di eseguirla. Maggiori delucidazioni a riguardo si trovano nella sezione 2 del capitolo 2.
Chiariti i requisiti di coerenza, si necessita di definire quali tipologie di funzioni di trasformazione esistano e quali proprietà debbano rispettare.
Vi sono due tipi di funzione di trasformazione: quelle inclusive, che includono l'impatto

dell'operazione contro cui si va a trasformare, e quelle esclusive, che al contrario lo escludono. Perché una funzione di trasformazione converga deve rispettare le due proprietà di trasformazione TP1 e TP2. La prima richiede che il risultato della trasformazione di due operazioni sia lo stesso nel caso in cui sia possibile eseguirle in ordini differenti, mentre la seconda richiede l'equivalenza della trasformazione inclusiva di due operazioni eseguibili in due stati differenti del documento. La sezione 3 del capitolo 2 tratta l'argomento in maniera più approfondita con l'ausilio di immagini.

Formalizzate le caratteristiche fondamentali della trasformazione si può passare a una breve elencazione degli algoritmi, proposti in ambito accademico, più noti e realizzabili. Il primo è dOPT (distributed OPerational Transformation) che indica un'operazione come una quadrupla contenente l'id, il vettore di stato del client, l'operazione stessa e la sua priorità, basa il suo sistema su una matrice di trasformazione in cui ogni entry è una funzione che trasforma due operazioni.

Poi vi sono GOT e la sua versione ottimizzata GOTO, i quali introducono l'Undo e la Redo di un'operazione, facendo uso di un History Buffer che tiene traccia della storia di tutte le operazioni applicate al documento. Altro algoritmo interessante dal punto di vista teorico è COT (Context-based Operational Transformation) che invece del History Buffer di GOT, sfrutta un context vector che si basa sul contesto del documento in cui ogni operazione viene eseguita, formalizzandone il concetto con sei proprietà. L'analisi completa di questi algoritmi viene effettuata nella sezione 5 del capitolo 2.

Questi elencati però sono solo una parte degli algoritmi teorici proposti in letteratura, e tra le applicazioni basate su OT la più nota è stata Google Wave, ispirata a Jupiter, riciclata poi in Google Docs, come detto sopra nell'excursus storico. OT è un protocollo che è stato usato anche in contesti specifici come l'ambito dei documenti strutturati, come file XML. Un documento semi-strutturato è un documento che contiene informazioni aggiuntive rispetto sulla propria struttura e sul proprio contenuto.

Un documento XML (Extensible Markup Language), è un documento semi-strutturato con una struttura gerarchica ad albero con contenuto misto, in cui ogni nodo può avere svariati figli, che possono essere a loro volta nodi contenenti altri nodi, o nodi di testo.

Una struttura di questo tipo permette di formulare un insieme di operazioni ben più vasto di quello che si potrebbe applicare a un semplice documento di testo libero (Insert/Delete/Retain).
Operazioni complesse possono essere lo spostamento di un nodo da una posizione a un'altra del documento (move), l'unione di due nodi consecutivi dello stesso tipo (join), la scissione di un nodo in due nodi fratelli (split), l'inserimento di un nodo all'interno di un altro appena creato (wrap) o viceversa l'estrazione di un nodo da un altro (unwrap). Inoltre siccome ogni nodo all'interno di un documento XML ha degli attributi che lo definiscono si può gestire anche la modifica di un attributo.
Allo stato attuale l'unico modo di accedere alle funzionalità di rich-text editing di un documento XML è l'attributo contentEditable, che nel momento in cui viene settato a true su un elemento, genera un area all'interno della quale sono abilitate azioni quali drag n drop, copy and paste, navigazione da tastiera ecc. La maggior parte degli editor che implementano operazioni complesse fanno uso di contentEditable, ma il problema è che questo attributo non è standardizzato su tutti i browser e il suo comportamento non sempre è prevedibile. Maggiori chiarimenti sono contenuti nella sezione 2 del capitolo 3.
Per questo motivo ho deciso di andare alla ricerca di un editor collaborativo che non facesse uso di contentEditable e mi sono imbattuto in CKEditor5, che implementa il set di operazioni complesse sopraelencato, includendo però la move nell'insieme delle operazioni semplici con insert e delete, e definendo una sorta di protocollo di risoluzione di conflitti tra operazioni base e operazioni complesse, raffigurate con l'astrazione di delta (array di operazioni semplici). CKEditor è scritto in Javascript e la parte che dovrebbe essere implementata server-side è in NodeJS. Al momento gli implementatori hanno semplicemente definito il set di operazioni/delta e le funzioni di trasformazione (ispirate all'algoritmo dOPT che si basa sulle priorità delle operazioni), senza fornire un'implementazione open source del server. L'analisi del codice di CKEditor e maggiori dettagli sono forniti nelle sezioni 3, 4 e 5 del capitolo 3. Il loro modulo sulla conflict resolution e la loro funzione di trasformazione, non mi sono sembrati sufficientemente esaustivi.
Per questo motivo ho deciso di impegnarmi nella definizione di un modello di risoluzione

dei conflitti che potesse essere sufficientemente esaustivo, andando ad analizzare tutti, o quasi, i casi limite in cui ci si possa imbattere nell'esecuzione di operazioni complesse in maniera concorrente all'interno di un ambiente di editing collaborativo real-time.
Il modello che ho definito è un modello ad alto livello rappresentato da una matrice dei conflitti di dimensione nxn, dove n è il numero di operazioni (nel mio caso sette), in cui ogni ogni entry rappresenta una funzione di trasformazione. Ognuna delle funzioni di trasformazione prende in input due operazioni e restituisce la prima operazione invariata e la seconda operazione trasformata. In ogni entry sono stati inseriti esempi per ognuno dei casi limite considerati, i quali rappresentano gli scenari in cui avviene la trasformazione. Oltre alla matrice dei conflitti ho deciso di scrivere anche le funzioni relative alle entries, in uno pseudo Javascript molto vicino al codice reale, che sono ancora in fase di elaborazione e che per questo motivo non verranno inserite nella trattazione. Il capitolo 4 spiega nel dettaglio la realizzazione della matrice e fornisce anche alcuni esempi delle sue entries.
Il design della matrice e l'inizio della progettazione algoritmica delle funzioni di trasformazione sono la base per un progetto che aspira a diventare un'implementazione reale. Dunque il sistema che si potrebbe realizzare, potrebbe sfruttare un algoritmo di tipo dOPT molto basilare con ordinamento delle operazione prioritario, facendo uso di un History Buffer, per tenere traccia delle operazioni eseguite precedentemente. Un'altra feature fondamentale per un sistema di questo tipo, dal mio punto di vista, può essere l'implementazione di un protocollo client/server come quello attualmente usato da Google Docs, con acknowledgement delle operazioni, in modo da mantenere ogni client in linea col server e per far sì che vi sia un'unica copia veritiera del documento condiviso che stia, appunto, sul server.

Capitolo 1

Real-Time Collaborative Editing Systems

I sistemi di editing collaborativo real-time sono software che permettono agli utenti di editare lo stesso documento su siti diversi nello stesso istante. L'esperienza che deve provare ognuno degli utenti connessi al sistema deve essere tale da permettergli di modificare e di vedere le modifiche degli altri utenti in real-time. In base al contesto lavorativo, gli utenti possono collaborare in maniera sincrona o asincrona.
La collaborazione sincrona è anche definita real-time editing, poiché nel momento in cui un utente effettua modifiche al documento, queste modifiche sono immediatamente propagate a tutti gli altri utenti connessi al server senza ritardi.
Al contrario, nel caso asincrono gli utenti non hanno modo di collaborare in tempo reale; lavorano per conto proprio e nel momento in cui vanno a modificare il documento lo fanno a scapito delle nuove modifiche effettuate dagli altri utenti. Ogni singolo utente vedrà una sequenza di eventi differente, anche se in minima parte, rispetto a quella degli altri utenti. Si necessita quindi di arrivare a una versione convergente del documento che non dipenda dall'ordine o dalla tempistica in cui le operazioni sono state eseguite su di esso. Il protocollo che è stato scelto per la gestione della concorrenza delle operazioni nella trattazione di questa tesi è Operational Transformation.

Ulteriori informazioni a riguardo si trovano nel capitolo 2 in cui analizzo proprietà, algoritmi ed applicazioni di questo protocollo.

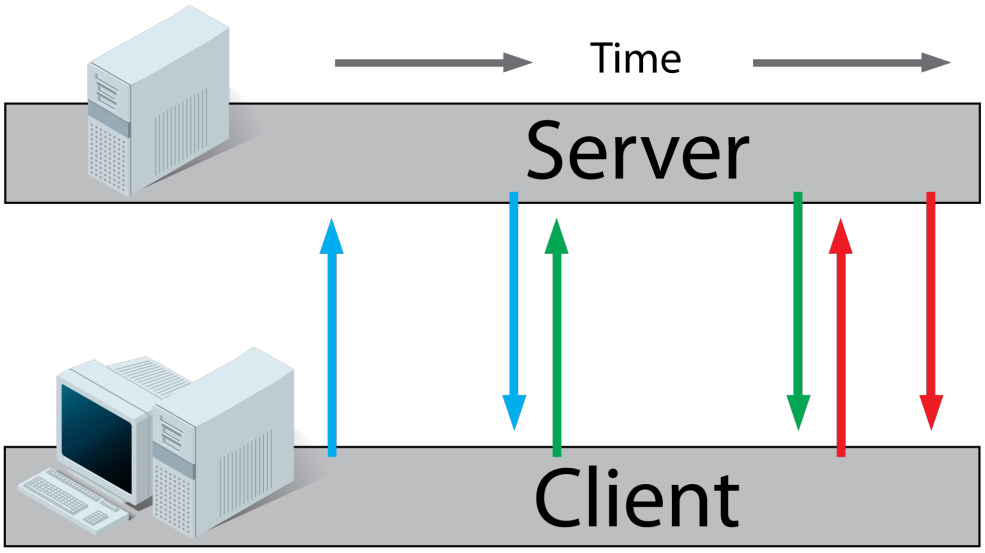
1.1 Problemi legati alla concorrenza

La grande sfida nell'implementazione di sistemi per editing collaborativo sta nel controllo della concorrenza, poiché le modifiche che vengono effettuate contemporaneamente al documento possono generare conflitti.
Queste devono essere causalmente ordinate prima della loro applicazione, tramite l'undo oppure la trasformazione, al fine di farle sembrare commutative.

1.2 Tecnologie real-time in Javascript

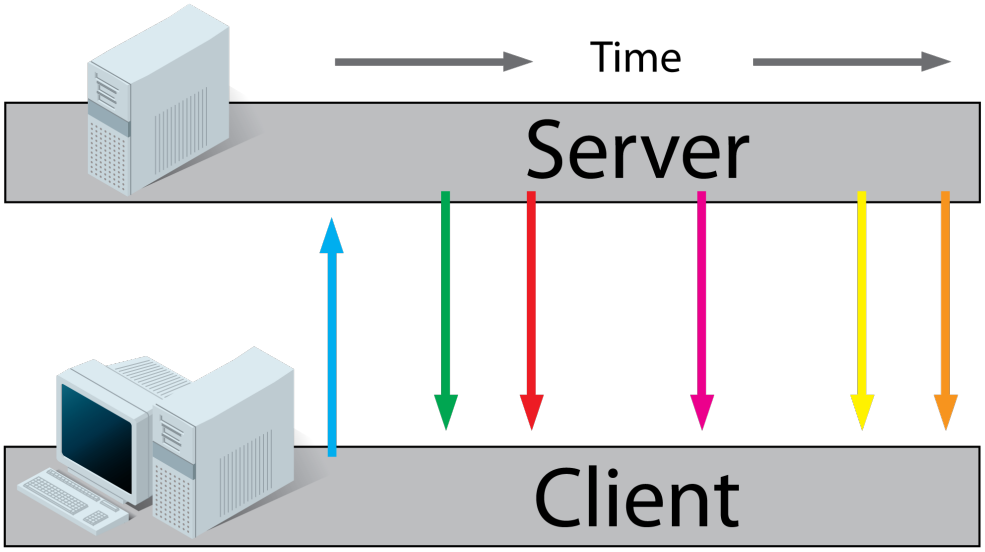
Javascript fornisce la possibilità di inviare e ricevere stream di dati in tempo reale attraverso l'utilizzo di chiamate AJAX (Asynchronous Javascript And XML, formato ormai sostituito da JSON). Il problema della collaborazione real time però richiede l'implementazione di un canale bidirezionale tra client e server in modo da permettere l'editing in parallelo. In seguito vengono presentate tre tecnologie: AJAX Long-Polling, tecnica base di request-response, SSE e WebSockets, le due più impiegate al momento nello sviluppo di applicazioni real time. [1]

1.2.1 Ajax Long-Polling



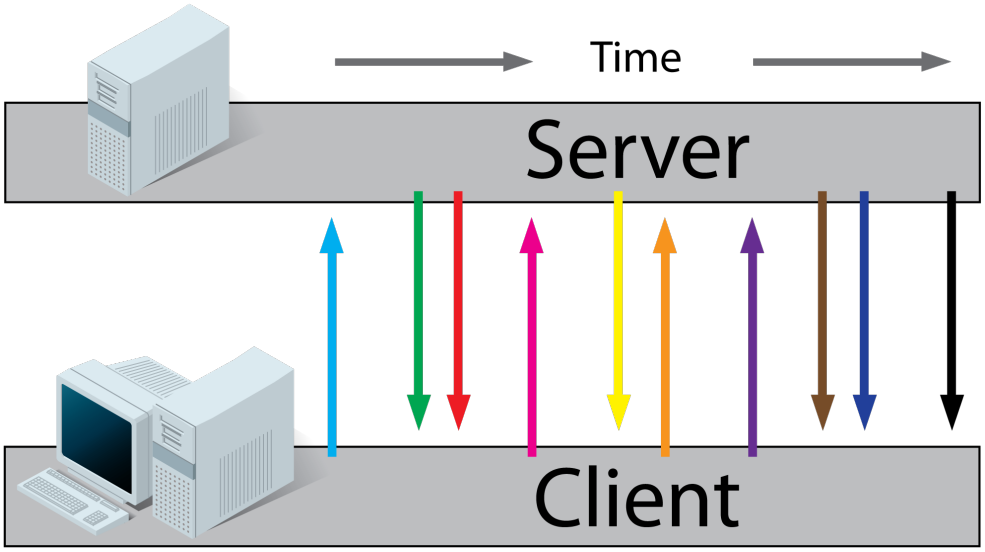
1. Client effettua la richiesta di una pagina al server attraverso HTTP;
2. La pagina esegue il codice Javascript che effettua la richiesta al server;
3. Il server non risponde immediatamente, ma attende che ci siano dati disponibili da inviare;
4. Quando ci sono nuovi dati disponibili, il server risponde al client inviandoglieli.
5. Il client riceve i nuovi dati e inviando successivamente una nuova richiesta al server, fa ricominciare il processo.

1.2.2 Server Sent Events



1. Client effettua la richiesta di una pagina al server attraverso HTTP;
2. La pagina esegue il codice Javascript che apre una connessione con server;
3. Il server risponde con un evento non appena ci sono nuovi dati disponibili.
    - Traffico real time tra server e client
    - Non permette la connessione a un server proveniente da un altro dominio

1.2.3 WebSockets



1. Client effettua la richiesta di una pagina al server attraverso HTTP;
2. La pagina esegue il codice Javascript che apre una connessione con server;
3. Server e Client possono inviarsi messaggi a vicenda quando ci sono nuovi dati disponibili (da entrambi gli estremi della comunicazione).
    - Traffico real-time da server a client e da client a server
    - Con i WebSockets è permessa la connessione a un server proveniente da un altro dominio.

1.3 Alternative ad OT

Le alternative che sono state proposte ad Operational Transformation sono Conflict-free Replicated Data Types e Differential Synchronization.

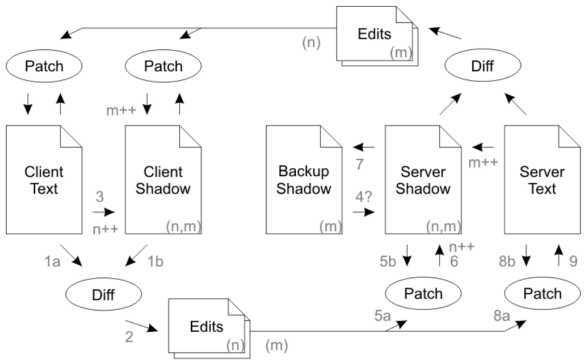
1.3.1 CRDT (Conflict-free Replicated Data Type)

I Conflict-free Replicated Data Type sono oggetti che possono essere aggiornati senza grossi sforzi di sincronizzazione, poiché ogni replica può eseguire un'operazione senza doversi sincronizzare a priori con tutte le altre. Tale operazione viene inviata asincronamente e ogni replica applica tutti gli aggiornamenti, possibilmente in ordini differenti. CRDT fanno uso di un algoritmo che stabilisce la convergenza di tutti gli aggiornamenti conflittuali in background.
Garantiscono la convergenza eventuale se tutti gli aggiornamenti concorrenti sono commutativi e se sono eseguiti da ogni replica eventualmente. I CRDT sono stati impiegati nei sistemi di chat online (League of Legends [2]), nei sistemi di scommesse (Bet365) e per l'implementazione di database NoSQL(SoundCloud [3], Facebook [4]). [5]

1.3.2 Differential Synchronization

Differential Synchronization è un algoritmo simmetrico che fa uso di un ciclo infinito di operazioni Diff e Patch in background. L'idea è di avere due versioni del documento, una client side che viene editata, e una sul server, che mantiene l'ultima versione prima dell'edit. Infatti la differenza tra queste due copie viene inviata al server successivamente a ogni modifica effettuata dal client. Il server ha tre versioni del documento: quella corrente, una copia ombra e una copia di backup. L'operazione Diff ricopre due ruoli completamente differenti all'interno del ciclo di sincronizzazione.
Il primo è di tenere aggiornata l'ombra del Server con il contenuto corrente del documento del Client e dell'ombra del Client. Il risultato dovrebbe convergere a tre copie identiche del documento. La seconda operazione è quella più complessa da eseguire e consiste nel tenere aggiornato il documento del Server seguendo le modifiche che vengono effettuate

al documento del Client. Il documento sul Server potrebbe essere stato modificato nel frattempo, quindi la Diff deve aggiornarlo in maniera semanticamente corretta.
L'operazione Patch è critica tanto quanto le operazioni del sistema. Patch deve controllare due variabili potenzialmente in conflitto che cercano di inserire testo o nodi nella stessa posizione. A sua volta la Patch deve svolgere due compiti. Il primo è di trovare la stringa con la minor distanza di Levenshtein tra la stringa attesa e quella attuale. Il secondo è di trovare una locazione ragionevolmente vicina a quella attesa dalla Patch. [6]



1.4 Operational Transformation

Si è scelto di studiare e di sfruttare Operational Transformation come protocollo di gestione della concorrenza e della coerenza del documento finale poiché: CRDT risultano troppo onerosi a livello computazionale e di occupazione di memoria, perché richiedono l'invio degli update a tutte le repliche, corredati dell'intero stato del documento.
Differential Synchronization appesantisce troppo client e server, perché devono mantenere rispettivamente due e tre copie del documento ciascuno.
OT a differenza di questi due protocolli, se implementata nella maniera adeguata, risulta più flessibile e alleggerisce di molto il carico di lavoro del server, assicurando la convergenza in maniera più efficace.
Una trattazione approfondita di OT è contenuta nel capitolo seguente.

Capitolo 2

Operational Transformation

Nei sistemi di editing collaborativo spesso il documento condiviso viene replicato per ognuno degli N client connessi. Quindi si avranno N copie differenti del documento, le quali necessiteranno di un meccanismo di sincronizzazione per salvaguardare l'integrità del documento finale.
I meccanismi di mantenimento della coerenza vengono classificati in due categorie: ottimistici e pessimistici. L'approccio pessimistico cerca di dare l'impressione ai client che esista un'unica copia del documento all'interno del sistema. Una sola copia sovrascrivibile e N-1 copie accessibili in lettura.
In questo caso il metodo che verrebbe in mente ai più potrebbe essere un semplice meccanismo di *locking*, molto simile a quello delle transazioni nei database, ma col passare degli anni e dopo la pubblicazione massiccia di letteratura accademica riguardo gli editing collaborativi si è arrivati alla conclusione che il suddetto metodo è sconveniente, poiché non permette aggiornamenti concorrenti e non supporta il lavoro off-line. Inoltre il ritardo nell'acquisizione del lock dovuto alla latenza della rete rende questo approccio impraticabile per l'editing real-time.
Al contrario, gli approcci ottimistici sono più adeguati all'editing collaborativo, poiché tollerano la divergenza momentanea delle copie del documento condiviso tra i client per far convergere ognuna di esse a uno stato finale dopo un determinato intervallo di tempo.
In questo capitolo approfondirò il meccanismo più studiato e più utilizzato, nel mondo

dello sviluppo software di editor collaborativi: Operational Transformation.

2.1 Background

In genere gli algoritmi **OT** permettono all'utente di modificare localmente la propria copia del documento senza alcun ritardo di trasmissione e, successivamente, le modifiche effettuate da tale utente vengono propagate agli altri N-1 utenti connessi nel sistema in maniera sincrona o asincrona. Infine gli aggiornamenti vengono eseguiti sulle copie remote del documento.
In **OT** ogni modifica al documento viene vista come un'*operazione*.
L'applicazione della funzione di trasformazione porta il documento in un nuovo document state. Tale funzione viene usata per gestire le operazioni tra i vari client in sistemi P2P che fanno uso di OT (proposti quasi esclusivamente in letteratura) oppure per costruire un protocollo client-server (come in Google Wave) che possa gestire tali operazioni senza aumentare vertiginosamente la complessità del sistema.
Permette, inoltre, di mantenere un'unica copia del documento sul server, il quale diventa la fonte di verità da cui recuperare il documento con facilità, nel caso in cui i client vadano in crash o restino off-line per un lungo periodo.
Questa logica obbliga i client ad attendere che il server riconosca le operazioni inviategli, ciò significa che il client sarà sempre sul path del server. Così si mantiene un'unica cronologia delle operazioni server-side senza doverla replicare su ogni singolo client connesso.
Dalla pubblicazione del paper di Ellis e Gibbs ("*Concurrency control in groupware systems*", 1989) che gettò le fondamenta teoriche di Operational Transformation, sono stati proposti diversi protocolli in letteratura e varie applicazioni.

2.2 Modelli di coerenza

La caratteristica più importante di Operational Transformation è la salvaguardia della coerenza del documento finale. Nel corso degli anni sono stati proposti vari modelli di coerenza.

CC (Causality Convergence):

- ***C**ausalità*: per ogni coppia di operazioni O_a e O_b, se vale la relazione $O_a \rightarrow O_b$, allora O_a viene causalmente prima di O_b, cioè è stata eseguita prima di essa.

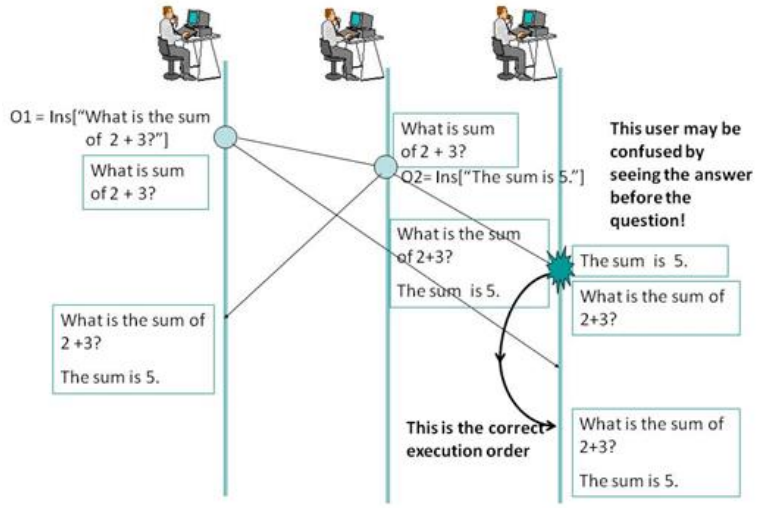


Figura 2.1: Esempio di rispetto della causalità nell'invio di operazioni tra tre utenti

- ***C**onvergenza*: quando lo stesso insieme di operazioni viene eseguito da tutti i siti, tutte le copie del documento condiviso sono identiche.

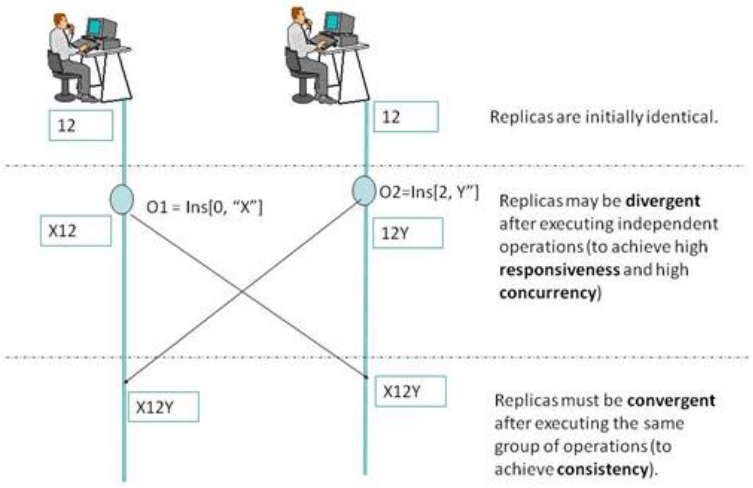


Figura 2.2: Esempio di convergenza del documento

L'esecuzione parallela delle operazioni porterà ogni replica a uno stato divergente, siccome in generale non può esserci commutatività. Ellis e Gibbs proposero l'introduzione di uno *state vector* per definire la precedenza tra un'operazione e l'altra.[7]

CCI (Causality Convergence Intention Preservation):

- ***C**ausalità*: è la stessa del modello CC.
- ***C**onvergenza*: è la stessa del modello CC.
- *Conservazione delle **I**ntenzioni*: per ogni operazione O, gli effetti della sua esecuzione su tutti i siti devono rispettare le intenzioni di O.

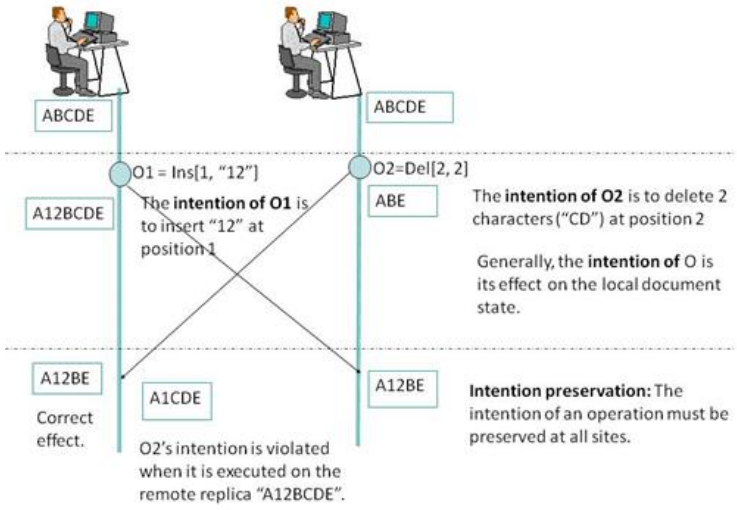


Figura 2.3: Esempio di conservazione delle intenzioni tra due utenti

Questo secondo modello introduce il concetto di conservazione delle intenzioni, che differisce da quello di convergenza. La prima è sempre raggiungibile con una semplice serializzazione delle operazioni, mentre la seconda no. Raggiungere la conservazione di intenzioni non serializzabili è la grande sfida che impegna ogni implementatore di meccanismi che adottano OT. [8] [9]

CSM (Causality Single Multiple):

- ***C**ausalità*: è la stessa del modello CC e CCI.
- *Effetto di una **S**ingola operazione*: eseguire l'operazione in qualunque stato di esecuzione ha lo stesso effetto di eseguirla nel suo stato di generazione.
- *Effetto di operazioni **M**ultiple*: la relazione che intercorre tra ogni coppia di operazioni resta tale in qualunque stato esse vengano eseguite.

La conservazione delle intenzioni del modello CCI non poteva essere formalizzata, per questo motivo è stato proposto il modello CSM. [10] [11]

CA (Convergence Admissibility):

- ***C****ausalità*: è la stessa del modello CC, CCI e CSM.
- ***A****mmissibilità*: ogni operazione è ammessa nel proprio stato di esecuzione.

Queste due condizioni implicano convergenza mantenendo un ordinamento basato sull'effetto della generazione delle operazioni. Vi sono ulteriori vincoli imposti da esse, ma ciò le rende più forti della sola convergenza. [12]

2.3 Proprietà della trasformazione

Per gestire le operazioni concorrenti la funzione di trasformazione prende in input due operazioni che sono state applicate allo stesso document state da client differenti e restituisce in output una nuova operazione che può essere applicata dopo la seconda, preservando le intenzioni della prima.

Esistono due tipologie di funzione di trasformazione:

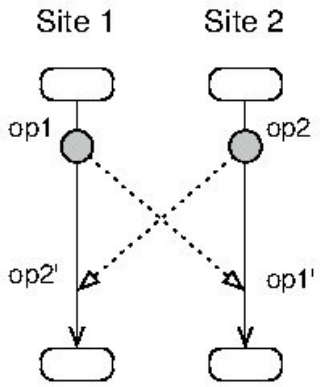
- **IT - Trasformazione Inclusiva**: definita come **IT(a,b)**, trasforma $\mathbf{O_a}$ in rapporto a $\mathbf{O_b}$ in modo che l'impatto di $\mathbf{O_b}$ venga incluso.
- **ET - Trasformazione Esclusiva**: definita come **ET(a,b)**, trasforma $\mathbf{O_a}$ in rapporto a $\mathbf{O_b}$ in modo che l'impatto di $\mathbf{O_b}$ venga escluso.

Le funzioni di trasformazione composta possono fare uso sia delle funzionalità inclusive che esclusive. [13]

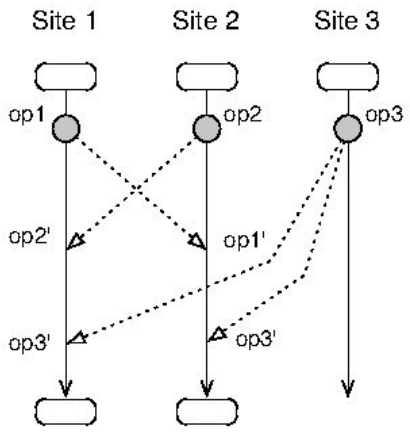
2.3.1 Convergenza

Qualsiasi funzione di trasformazione necessita di due proprietà, o precondizioni, fondamentali per garantire la propria convergenza, **TP1** e **TP2**.

TP1: Date due operazioni $\mathbf{O_a}$ e $\mathbf{O_b}$, la funzione di trasformazione **T** soddisfa **TP1** sse $\mathbf{O_a} \circ \mathbf{T(O_a, O_b)} \equiv \mathbf{O_b} \circ \mathbf{T(O_a, O_b)}$ dove $\circ$ denota una sequenza di operazioni contenente $\mathbf{O_i}$ seguita da $\mathbf{O_j}$ e $\equiv$ denota l'equivalenza di due operazioni.
Questa proprietà è necessaria solo nel caso in cui il sistema **OT** ammetta la possibilità di eseguire due operazioni in ordini differenti.



TP2: Date tre operazioni $\mathbf{O_a}$, $\mathbf{O_b}$ e $\mathbf{O_c}$, la funzione di trasformazione **T** soddisfa **TP2** sse $\mathbf{T(O_c, O_a \circ T(O_b, O_a)) \equiv T(O_c, O_b \circ T(O_a, O_b))}$, dove $\circ$ e $\equiv$ assumono lo stesso significato definito sopra.
Questa proprietà è richiesta solo nel caso in cui il sistema **OT** ammetta la possibilità che due operazioni vengano inclusivamente trasformate in due document state differenti. [9]



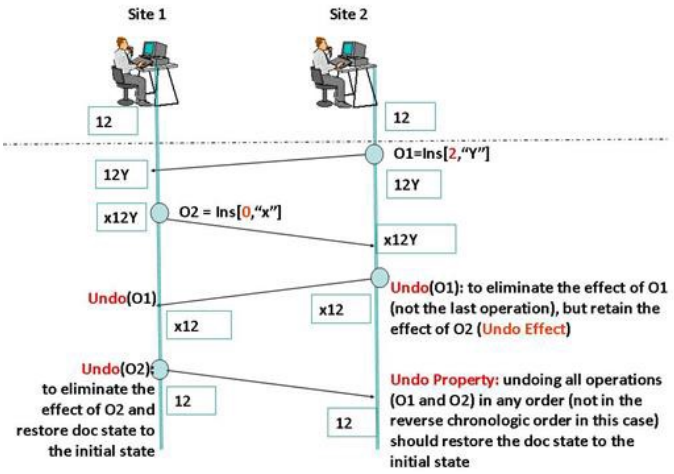
2.3.2 Proprietà Inverse

OT supporta anche l'annullamento, *undo*, di una o più operazioni, rispettando però alcuni prerequisiti. Il raggiungimento di tale azione sul documento deve rispettare le proprietà: **IP1**, **IP2** e **IP3**.

IP1: A partire da un document state S, l'esecuzione dell'operazione **O** e della sua inversa **O'** risulta in $\mathbf{S} \circ \mathbf{O} \circ \mathbf{O'} = \mathbf{S}$, ovvero $\mathbf{O} \circ \overline{\mathbf{O}}$ equivale a un'identità rispetto all'effetto sul document state. Questa proprietà deve essere rispettata sse il sistema **OT** ammette operazioni inverse per il raggiungimento dell'*undo*.

IP2: Data un'operazione $\mathbf{O_a}$ e una coppia $(\mathbf{O_b}, \overline{\mathbf{O_b}})$, appartenenti allo stesso document state vale $\mathbf{IT(IT(O_a, O_b), \overline{O_b}) = IT(O_a, I) = O_a}$, dove **I** è l'identità data da $\mathbf{O_b} \circ \mathbf{O_b'}$.

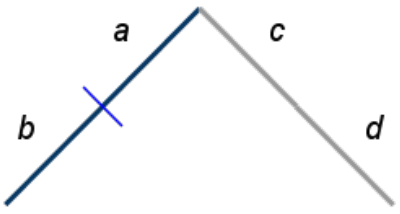
IP3: Date le operazioni $\mathbf{O_a}$ e $\mathbf{O_b}$, se $\mathbf{O_a' = IT(O_a, O_b)}$, $\mathbf{O_b' = IT(O_b, O_a)}$ e $\mathbf{O_a'' = IT(\overline{O_a}, O_b')}$, allora $(\overline{\mathbf{O_a}})' = \overline{\mathbf{O_a'}}$. [8]



2.4 Un caso complesso

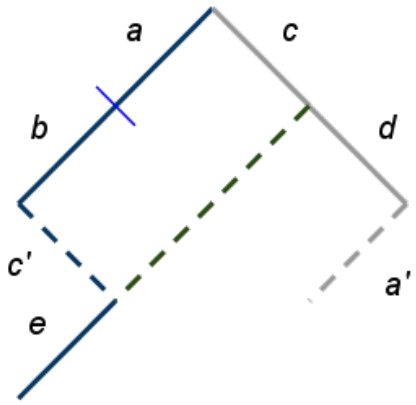
In questa sezione analizzerò la casistica più complessa che si possa prospettare nell'implementazione di un sistema basato su OT con protocollo client/server come in Google Wave. La gestione di un problema di questo genere garantisce l'efficienza del sistema a prescindere dal numero di client che deve supportare e dal numero di operazioni entranti che il server deve gestire.

Il client inizia eseguendo due operazioni a e b, inviando immediatamente la prima, poiché imparentata con un'operazione che sta nello state space del server, e bufferizzando la seconda (la linea chiara indica che da b in poi le operazioni vengono bufferizzate). Intanto il server ha applicato c e d provenienti da altri clients.



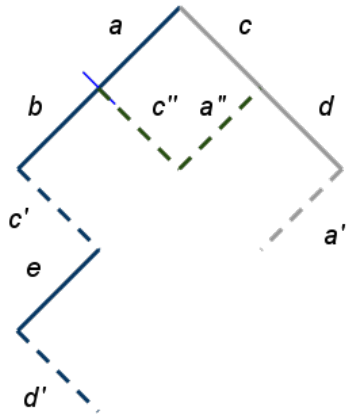
Successivamente il server compone c con d e trasforma a con il risultato di tale composizione, ottenendo a' e propagandola a tutti gli altri client, imponendo un ordinamento.

A sua volta il client riceve c e la trasforma con il risultato della composizione di a e b, producendo c'. Essendo un text editor in fase di modifica di un documento realmente condiviso, si può presupporre che il client prosegua nell'esecuzione di nuove operazioni locali e aggiunga e al proprio state space.
Siccome il client ha momentaneamente inviato solo a al server, si necessita di inferire un "ponte" tra lo state space del server e quello del client al fine di poter trasformare d e mantenere un percorso assoluto tra l'ultimo punto nel client state space e l'ultimo punto nel server state space.

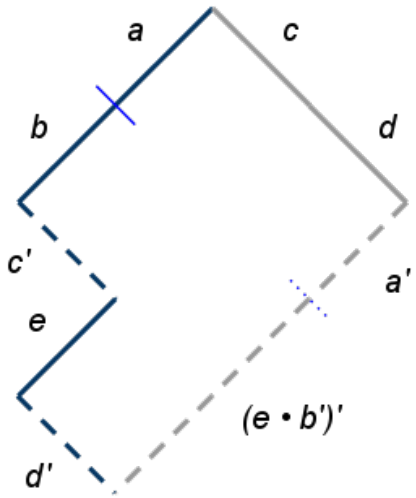


Il ponte può essere visto come la trasformazione di c con il risultato della composizione di a e b. Da qui in poi si compone e con il ponte, il client riceve d dal server e come con c, la trasforma sfruttando il ponte, derivando d'. Con d' a disposizione si può tornare ad a'. Ricevuta a', si necessita di inviare le operazioni bufferizzate al server, ma tali operazioni non sono imparentate con alcun punto del server state space. Il buffer verrà aggiustato di conseguenza in qualsiasi momento in cui client e server non generano operazioni sul documento. Inoltre dovrà essere sempre imparentato con l'ultima operazione che porta il server ad essere perfettamente sincronizzato col client. La sua costruzione è abbastanza complessa.
Ricapitolando, dopo l'invio di a al server, viene eseguita b, prima operazione bufferizzata, poi tocca a c che non è imparentata, siccome il buffer contiene solo b al momento. Dunque bisogna trasformare c con a per derivare c'' e di conseguenza si deriva anche a'', versione

trasformata di *a*, come si può notare in figura. Quando il server produrrà *a'*, questa dovrà essere equivalente ad *a"*.



Ora, *e* verrà bufferizzata perché imparentata con *c'*, ottenuta dalla trasformazione di *c* col risultato della composizione di *a* con *b*. Questo stato è lo stesso in cui ci si è trovati sopra applicando *a"*. Ma componendo *a"* col buffer equivale a inferire il "ponte" sopraccitato.
Il buffer conterrà la composizione di *b* ed *e* trasformate; la composizione del buffer con *a"* serve da "ponte" tra l'ultimo punto nel server state space e l'ultimo punto nel client space. Infine sapendo che *a"* è equivalente ad *a'* basta inviare il buffer al server sapendo che è imparentato con un punto nel server state space. Il server dunque invierà il suo acknowledgement e client e server giungeranno a uno stato convergente del documento condiviso. [14]



2.5 Algoritmi OT

In questa sezione analizzo in maniera abbastanza approfondita buona parte degli algoritmi OT disponibili in letteratura che siano degni di nota, o per motivi storici o perché hanno apportato un contributo più o meno importante alla teoria di Operational Transformation.

2.5.1 dOPT - distributed OPerational Transformation / adOPTed

Basato su una matrice di trasformazione m x m, dove m è il numero di operazioni supportate dal sistema e ogni entrata è una funzione che trasforma m_i con m_j.

Ogni operazione è una quadrupla $<i, s, o, p>$, dove:

- i è l'id del client che invia la richiesta;
- s è il vettore di stato del client;
- o è l'operazione da applicare al documento;

- p è la priorità di o;

Se lo stato del vettore al momento della richiesta è equivalente a quello della ricezione della richiesta, allora questa viene processata immediatamente, se invece lo stato del sender è successivo a quello del receiver la richiesta viene incodata, altrimenti viene trasformata e successivamente eseguita. Sfruttando strutture dati semplici è di facile implementazione, inoltre assicura l'ordine di precedenza delle operazioni e garantisce il rispetto della prima delle due proprietà delle funzioni di trasformazione, TP1, ma non la seconda. Non fa uso di un server centrale.
adOPTed è stato introdotto come miglioramento di dOPT, poiché preserva anche la seconda proprietà delle funzioni di trasformazione, TP2, che come detto sopra assicura che la trasformazione di un'operazione lungo percorsi diversi porti allo stesso risultato. Non garantisce, però, la salvaguardia delle intenzioni. [7] [15] [16]

2.5.2 GOT / GOTO

GOT oltre ad avere un ottimo meccanismo per garantire la convergenza, introduce le operazioni di *Undo*, *Do*, *Redo*, grazie all'utilizzo di un *History Buffer* per ogni client, nel quale vengono mantenute tutte le operazioni eseguite e annullate.
TP2 non è condizione necessaria. Lo sarebbe se il sistema che fa uso di GOT permettesse trasformazioni tra operazioni trasformate in due stati differenti del documento.
Supporta solo operazioni base su stringhe come inserimento ed eliminazione. GOTO è la versione ottimizzata di GOT poiché riduce il numero di trasformazioni inclusive ed esclusive sfruttando TP1 e TP2, aggiungendo le proprietà di adOPTed. Fa uso di una funzione denominata *convert2HC()* che traspone la storia delle operazioni per ottenere raggiungere la concorrenza quando va ad integrare operazioni remote. Rispetta convergenza, precedenza, TP1, TP2 e undo. [17] [9]

2.5.3 COT - Context-Based Operational Transformation

COT sfrutta il context vector, una struttura dati differenti dall'history buffer di GOT, che si basa sullo stato/contesto del documento in cui ogni operazione viene eseguita.
A stati diversi corrispondono operazioni diverse e ogni operazione ha un id e un sequence number, assegnati quando viene generata.
Ogni operazione ha anche un context vector che contiene le operazioni che l'hanno preceduta, che a loro volta avranno il loro id e sequence number.
Se un'operazione viene eseguita nello stesso stato da ogni client, il risultato non può divergere.
Il context vector può essere visto come una struttura dati di questo tipo:

$$\mathbf{CV(O) = [(ns^0, ic^0), \ldots ,(ns^{n-1}, ic^{n-1})]}$$

con ns^i che indica l'operazione e ic^i che indica il vettore delle coppie contenenti ogni singola operazione e la relativa inversa in^i, per poter effettuare l'*undo* di ognuna di esse.
Per far sì che vi sia una corretta composizione e trasformazione delle operazioni questo algoritmo deve rispettare sei requisiti essenziali definiti come *Context-Based Conditions* (CC):

- **CC1**: data un'operazione non appartenente allo stato attuale del documento ($O \notin DS$), essa può essere trasformata per essere eseguita sse il suo contesto è incluso nello stato del documento ($C(O) \subseteq DS$).
- **CC2**: data un'operazione per cui vale CC1 (($O \notin DS$) && ($C(O) \subseteq DS$)), l'insieme delle operazioni con cui deve essere trasformata prima di essere eseguita sono $DS \backslash C(O)$.
- **CC3**: un'operazione può essere eseguita sse il suo contesto è uguale allo stato del documento ($C(O) \equiv DS$)
- **CC4**: data un'operazione non trasformata e un operazione di qualunque tipo, dove la prima non sta nel contesto della seconda ($O_i \notin C(O_j)$), la prima può essere

trasformata sse il contesto della prima è incluso nel contesto della seconda ($C(O_i) \subseteq C(O_j)$).

- **CC5**: date due operazioni per cui vale CC4 (($O_i \notin C(O_j)$) && ($C(O_i) \subseteq C(O_j)$)), l'insieme delle operazioni con cui deve essere trasformata inclusivamente la prima delle due è ($C(O_j) \backslash C(O_i)$)

- **CC6**: due operazioni possono essere trasformate inclusivamente l'una con l'altra, sse hanno lo stesso contesto ($C(O_i) \equiv C(O_j)$).

In sostanza la prima e la quarta formalizzano il corretto ordinamento delle operazioni e trasformazioni, la seconda e la quinta l'insieme delle operazioni, o contesto, con cui deve essere trasformata un'operazione, la terza e la sesta la corretta esecuzione e trasformazione. [18]

2.6 Applicazione basate su OT

2.6.1 Jupiter

Nel sistema **Jupiter**, i client sono connessi a un server centrale via *TCP* in una topologia a stella. Un client può comunicare direttamente solo con il server; il server trasforma e propaga a tutti i client le operazioni performate da tale client.

Client

Quando $\mathbf{O_x}$ viene generata da un client, viene immediatamente eseguita in locale e propagata al server.

Le operazioni locali vengono propagate in maniera sequenziale e salvate nello state-space locale.

Server

Il server mantiene uno state-space per ogni client.
Uno state-space locale è relativo alle operazioni del client e globalmente alle operazioni di tutti i siti.
Il server gestisce $\mathbf{O_x}$ come segue:

1. Localizza lo spazio relativo al client che gli ha inviato $\mathbf{O_x}$;
2. cerca il match tra spazio e stato per contestualizzare $\mathbf{O_x}$, salva $\mathbf{O_x}$ in questo stato lungo la dimensione locale;
3. trasforma $\mathbf{O_x}$ con una serie di operazioni $\mathbf{L_1}$, composta di operazioni lungo la dimensione globale dallo stato trovato fino allo stato finale.
   Questa trasformazione può essere espressa come:

$$\mathbf{SLT}(\mathbf{O_x},\mathbf{L_1}) = (\mathbf{O_x}\{\mathbf{L_1}\}, \mathbf{L_1}\{\mathbf{O_x}\});$$

4. propaga $\mathbf{O_x}\{\mathbf{L_1}\}$ agli altri clients;
5. salva $\mathbf{O_x}\{\mathbf{L_1}\}$ al termine della dimensione globale di ogni state-space.

Remoto

Ogni client mantiene uno state-space simile a quello del server. Jupiter gestisce $\mathbf{O_x}\{\mathbf{L_1}\}$ da remoto come segue:

1. cerca lo spazio in cui trovare uno stato che matchi con $\mathbf{O_x}\{\mathbf{L_1}\}$, e la salva in questo stato lungo la dimensione globale;
2. trasforma $\mathbf{O_x}\{\mathbf{L_1}\}$ con una serie di operazioni $\mathbf{L_2}$, composta di operazioni lungo la dimensione globale dallo stato trovato fino allo stato finale.

   Questa trasformazione può essere espressa come:

$$\mathbf{SLT}(\mathbf{O_x}\{\mathbf{L_1}\},\mathbf{L_2}) = (\mathbf{O_x}\{\mathbf{L_1}, \mathbf{L_2}\},\mathbf{L_2}\{\mathbf{O_x}\})$$

3. esegue $\mathbf{O_x}\{\mathbf{L_1}, \mathbf{L_2}\}$. [19] [20]

2.6.2 Google Wave

Il sistema di Google prese spunto dal sistema Jupiter descritto sopra e proprio da questi, Wave assorbì l'idea di basare il proprio sistema OT su un protocollo client/server con ankowledgement.
Wave introduce un insieme massiccio di nuove componenti per la gestione del proprio sistema. Una Wave viene definita come una collezione di Wavelet.
Un Wavelet è a sua volta una collezione di documenti e ogni documento consiste di codice XML misto ad annotazioni sui nodi che compongono il suo albero.
Il Wavelet è l'ambiente in cui avvengono le modifiche al documento che entrano in conflitto e su viene applicata effettivamente Operational Transformation. Il meccanismo OT di Wave è un surrogato di quello di Jupiter, poiché sfrutta la stessa architettura a stella con un unico server centrale che trasmette le operazioni trasformate e tutti i client, ma vi aggiunge un nuovo restrizione talmente innovativa da diventare il più illustre meccanismo di OT non sviluppato in ambito accademico, aggiungendo un protocollo stop-and-wait tra client e server. Il risultato è l'eliminazione degli state-space bidimensionali di Jupiter e l'introduzione di un unico buffer monodimensionale sul server.

Client

Quando $\mathbf{O_x}$ è generata viene eseguita e salvata subito in locale nello spazio bidimensionale del client, ma non viene propagata al server finché il client non ha ricevuto l'acknowledgement per $\mathbf{O_{x\text{-}1}}$. Successivamente deve essere trasformata simmetricamente con una sequenza di operazioni remote $\mathbf{L_1}$ propagate dal server con ordinamento totale prima di $\mathbf{O_{x\text{-}1}}$. Tale trasformazione può essere espressa come:

$$\mathbf{SLT}(\mathbf{O_x}, \mathbf{L_1}) = (\mathbf{O_x}\{\mathbf{L_1}\}, \mathbf{L_1}\{\mathbf{O_x}\})$$

$\mathbf{O_x\{L_1\}}$ viene propagata al server.

Server

Il server possiede un buffer monodimensionale per tenere traccia delle operazioni di tutti i client con un ordinamento totale e tenendo traccia anche del contesto in cui vengono eseguite tali operazioni.
Il server gestisce $\mathbf{O_x\{L_1\}}$:

1. scansiona il buffer monodimensionale per trovare lo stato che corrisponda al contesto di $\mathbf{O_x\{L_1\}}$;
2. trasforma $\mathbf{O_x\{L_1\}}$ contro una sequenza di operazioni definita come $\mathbf{O_x\{L_2\}}$, che va dallo stato corrispondente al termine del buffer; trasformazione esprimibile come

$$\mathbf{LT(O_x\{L_1\}, L_2) = O_x\{L_1, L_2\}};$$

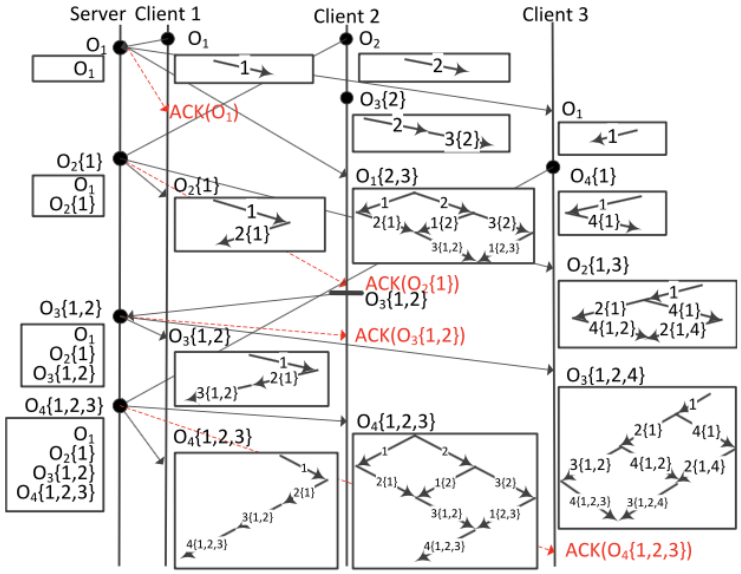
3. appende $\mathbf{O_x\{L_1, L_2\}}$ al termine del buffer;
4. propaga $\mathbf{O_x\{L_1, L_2\}}$ a tutti i client, includendo $\mathbf{O_x\{L_1\}}$ come acknowledgement;

Remoto

Ogni client possiede uno state-space bidimensionale simile a quello di Jupiter. La gestione di $\mathbf{O_x\{L_1, L_2\}}$ da remoto è simile a quella di $\mathbf{O_x\{L_1\}}$ in Jupiter:
$\mathbf{O_x\{L_1, L_2\}}$ viene trasformata simmetricamente con una sequenza di operazioni $\mathbf{L_3}$, costituita di operazioni locali a partire dallo stato che corrisponde al contesto fino a quello finale dello state-space. Questa trasformazione può essere espressa come:

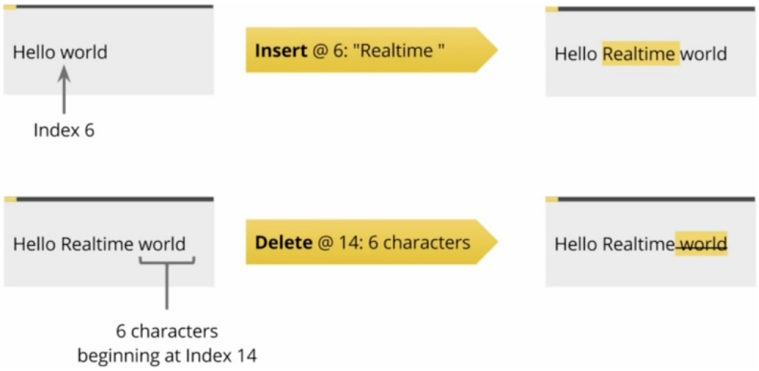
$$\mathbf{SLT(O_x\{L_1, L_2\}, L_3) = (O_x\{L_1, L_2, L_3\}, L_3\{O_x\}).}$$

Infine, $\mathbf{O_x\{L_1, L_2, L_3\}}$ viene eseguita. [21] [20]



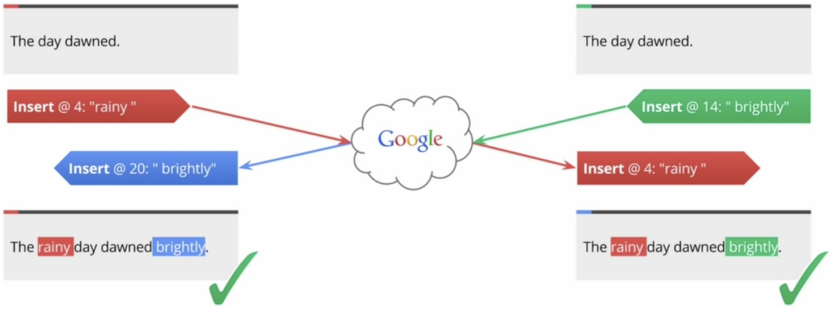
2.6.3 Google Docs

Successivamente all'abbandono del progetto di Google Wave, avvenuto nel 2010, a un anno di distanza dalla presentazione al Google I/O del 2009, Google ha deciso di riciclare parte dell'engine e della logica architetturale di Wave per l'implementazione di un nuovo sistema collaborativo che tutti conosciamo come Google Docs. Il meccanismo OT sottostante scompone ogni modifica al data model in una singola *mutazione*, cioè una semplice operazione applicata ad esso.

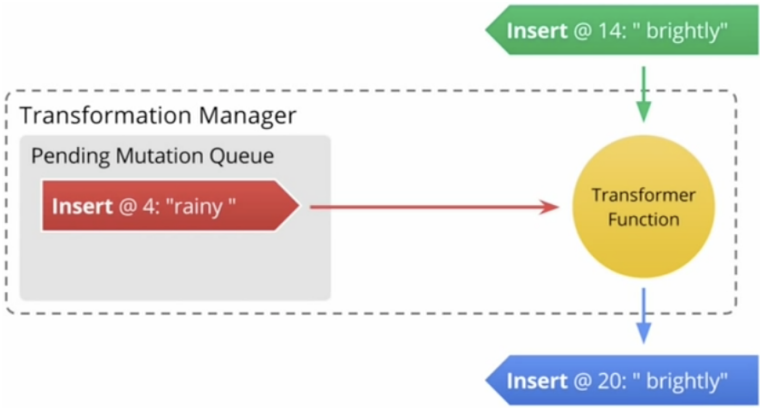


Ogni modifica viene salvata all'interno di uno snapshot dello stato del documento in quell'istante. Viene tenuta in memoria l'intera cronologia delle mutazioni applicate,

così da avere la possibilità di tornare a qualunque stato del documento precedente a quello attuale. Siccome questo log delle mutazioni diventerebbe troppo lungo da inviare ogni volta dal server al client, si fa uso del concetto di snapshot introdotto sopra, che permette di ricostruire il data model senza fare uso dell'intero insieme di mutazioni, ma solo di un *minimo insieme* di esse. Come in Jupiter e in Wave, le mutazioni locali vengono applicate immediatamente, ma siccome queste possono diventare obsolete si fa uso di Operational Transformation, per far sì che le operazioni restino sincronizzate, preservando le intenzioni del singolo utente.



In Docs, così come in Wave, viene fatto uso di un server centralizzato che riceve le operazioni e le processa. Ciononostante applicare le trasformazioni server side può non bastare, poiché il server potrebbe non essere a conoscenza delle mutazioni di uno dei client finché non gli invia le mutazioni degli altri client. Per risolvere situazioni di questo tipo Google ha deciso di applicare la stessa logica del server ad ogni client, mantenendo una coda delle mutazioni inviate e trasformando le mutazioni entranti con le mutazioni incodate. Così come in Wave il server, detto *Transformation Manager*, deve inviare un acknowledgement delle mutazioni al client, in modo che possa andarle a rimuovere dalla propria coda locale con le mutazioni pendenti.



Ciò fa pensare che il meccanismo OT sottostante a Google Docs non sia molto differente da quello di Google Wave. [22]

2.7 Limiti degli algoritmi OT analizzati

Gli algoritmi provenienti da paper accademici precedentemente analizzati sono utili solo in parte allo sviluppo della mia tesi, poiché come descritto sopra, tali algoritmi si occupano di gestire un insieme di operazioni ristretto rispetto a quello che realmente servirebbe per agire in maniera efficace su documenti con una struttura come i file XML.

OT control/integration algorithms (systems)	Required transformation function types	Support OT-based Do?	Support OT-based Undo?	Transformation properties supported by control algorithm	Transformation properties supported by transformation functions	Transformation ordering and propagation constraints	Timestamp
dOPT[1] (GROVE)	T (IT)	Yes	No	None	CP1/TP1, CP2/TP2	Causal order	State vector
selective-undo[10] (DistEdit)	Transpose (IT and ET)	No	Selective Undo	NA	CP1/TP1, CP2/TP2, RP, IP1, IP2, IP3	Causal order	??
adOPTed[2][26] (JOINT EMACS)	LTransformation (IT)	Yes	Chronological Undo	IP2, IP3	CP1/TP1, CP2/TP2, IP1	Causal order	State vector
Jupiter[4]	xform (IT)	Yes	No	CP2/TP2	CP1/TP1	Causal order + Central transformation server	Scalar
Google Wave OT[17]	transform and composition(IT)	Yes	No	CP2/TP2	CP1/TP1	Causal order + Central transformation server + stop'n'wait propagation protocol	Scalar
GOT[3] (REDUCE)	IT and ET	Yes	No	CP1/TP1, CP2/TP2	None	Causal order + Discontinuous total order	State vector
GOTO[5] (REDUCE, CoWord, CoPPT, CoMaya)	IT and ET	Yes	No	None	CP1/TP1, CP2/TP2	Causal order	State vector
AnyUndo[6] (REDUCE, CoWord, CoPPT, CoMaya)	IT and ET	No	Undo any operation	IP2, IP3, RP	IP1, CP1/TP1, CP2/TP2	Causal order	State vector
SCOP[23] (NICE)	IT	Yes	No	CP2/TP2	CP1/TP1	Causal order + Central transformation server	Scalar
COT [24] (REDUCE, CoWord, CoPPT, CoMaya)	IT	Yes	Undo any operation	CP2/TP2, IP2, IP3	CP1/TP1, (no ET therefore no IP1 necessary)	Causal order + Discontinuous total order	Context vector

Figura 2.4: Tabella riassuntiva degli algoritmi OT

Capitolo 3

OT su documenti strutturati

3.1 I documenti strutturati

Un documento strutturato contiene informazioni addizionali riguardo la propria struttura e su come è aggregato il proprio contenuto, non solo sul proprio layout.
La struttura gerarchica descrive quali operazioni possono essere effettuate su di esso, dove e in quale ordine. A partire dal 1986 con la nascita di **SGML** (Standard Generalized Markup Language), sono stati derivati **HTML** (Hypertext Markup Language) e **XML** (Extensible Markup Language), rispettivamente nel 1993 e nel 1998, come metodi di marcatura di documenti e sequenze di caratteri per definirne le caratteristiche strutturali e di layout.
Nello specifico XML, così come HTML, ha una peculiare struttura gerarchica ad albero. Ogni nodo è un elemento che può essere decorato definendo degli attributi su di esso e può avere svariati figli, ognuno dei quali è ancora un nodo che può avere figli, o un nodo di testo, da qui la definizione di struttura ad albero con contenuto misto.
Data questa struttura ad albero, XML fornisce una rappresentazione lineare attraverso l'inserimento di tag di apertura e di chiusura, rispettivamente all'inizio e alla fine di ogni nodo.

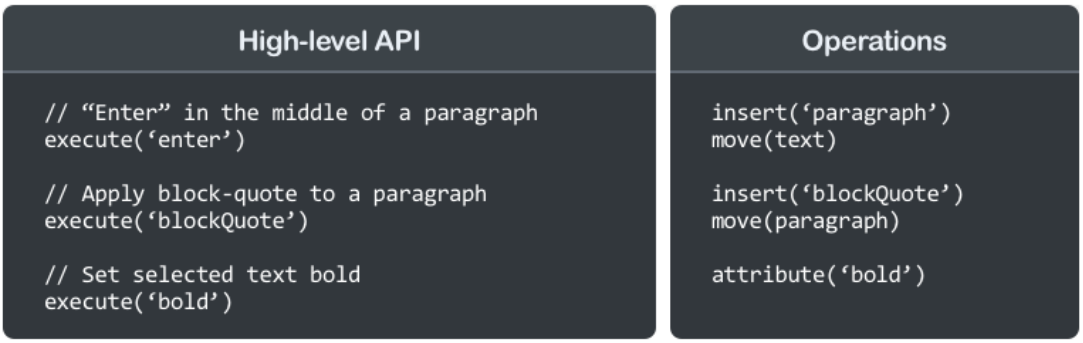
3.2 HTMLElement.contentEditable

Allo stato attuale l'unico modo per accedere alle funzionalità più complesse per la manipolazione di una porzione di un documento HTML/XML è l'utilizzo dell'attributo **contentEditable**. Questa proprietà indica se un elemento sia editabile o meno.
Tale attributo può assumere tre valori: true, se contenteditable, false se non può essere editato o inherit se eredita tale status dal parent.
Con la semplice aggiunta e attivazione di tale attributo è possibile abilitare la navigazione da tastiera, il correttore ortografico, il drag and drop, copy and paste e l'undo.
Inoltre, avere un editor che gira all'interno del browser significa che il contenuto che crea è in qualche modo limitato dal HTML che è in grado di rappresentare.
Si è scelto di creare l'attributo ContentEditable perché sembrava la soluzione più ovvia per abilitare l'editing complesso di contenuti sfruttando l'infrastruttura del browser.
Semplicemente wrappando del HTML all'interno di un <div *contenteditable*="true">si riesce a generare un editor all'interno di una webpage.
La maggior parte degli editor che implementano operazioni complesse come quelle precedentemente elencate si basano su ContentEditable, ma il problema principale di questo attributo è che non è stato ancora standardizzato su tutti i browser e il suo comportamento per gli utenti finali non sempre è prevedibile.

3.3 Gestione di operazioni complesse su DOM in CKEditor5

Nella versione 5 di CKEditor gli sviluppatori hanno deciso di innovare il loro approccio al data model su cui lavorare. Piuttosto che richiedere al DOM di memorizzarlo, hanno implementato un modello personalizzato in Javascript, il quale ignora i vincoli dettati dalle implementazioni del DOM all'interno dei vari browser con un sistema più semplice.
L'indipendenza dall'implementazione DOM del browser utilizzato fa sì che l'avere un'i-

stanza attiva dell'editor gestita in NodeJS permette la manipolazione del modello alla stregua di come farebbe un qualsiasi utente finale.
Chiaramente ho deciso di prendere in considerazione l'analisi delle feature di questo particolare editor, poiché sfrutta il meccanismo di Operational Transformation per la manipolazione del proprio data model.
Ad alto livello CKEditor5 fornisce un'API in linea con l'argomentazione della mia tesi. A basso livello tale API ha il tradeoff della traduzione delle proprie operazioni complesse in un insieme ristretto di operazioni standard come "insert", "delete"e "move".
Ciò significa che a basso livello tali operazioni figureranno semplicemente come insiemi di operazioni standard dal punto di vista del processing interno di CKEditor5. [23]



3.4 Low-Level API

3.4.1 Insert di nodi

```
1 class InsertOperation extends Operation{
2   _execute(){
3     const originalNodes = this.nodes;
4     this.nodes = new NodeList([...originalNodes].map(node => node.clone
          (true)));
5     const range = insert(this.position, originalNodes);
6     return {range};
7   }
8 }
```

Listing 3.1: insertoperation.js from CKEditor5 Model

La funzione *insert* richiamata in riga 5, importata da writer.js, normalizza i nodi contenuti in *originalNodes*, quantifica l'offset prima di inserirli, salva la posizione del parent in una variabile e splitta il nodo di testo in posizione *this.position* con una funzione ausiliaria.

Successivamente richiama sul parent la funzione *insertChildren* cui passa l'indice da dove partire e i nodi da inserire. Infine richiama una funzione per fare il merge dei nodi di testo, in caso siano stati toccati nodi "vecchi".

L'insieme originale di nodi viene passato all'utente che sta scrivendo in modo da inserirlo nel model. Il metodo *_execute* ritorna il range ritornatogli dalla insert.

3.4.2 Move

```
1 class MoveOperation extends Operation{
2   _execute(){
3     const sourceElement = this.sourcePosition.parent;
4     const targetElement = this.targetPosition.parent;
5     const sourceOffset = this.sourcePosition.offset;
6     const targetOffset = this.targetPosition.offset;
```

```
7       const range = writer.move(Range.createFromPositionAndShift(this.
            sourcePosition, this.howMany), this.targetPosition);
8       return { sourcePosition: this.sourcePosition, range};
9     }
10  }
```

Listing 3.2: classe MoveOperation in moveoperation.js

La funzione *move* richiamata in riga 8, importata da writer.js, prende in input il range di partenza che viene creato col metodo *createFromPositionAndShift* dell'oggetto Range, definito in range.js, e *this.targetPosition*, che sarebbe la posizione in cui andare a spostare i nodi. Tale funzione a sua volta richiama la remove, che fondamentalmente rimuove il range e lo ritorna al chiamante, e salva il range appena rimosso in una variabile. Poi, in una nuova variabile, salva il risultato del metodo *_getTransformedByDeletion* richiamato su *this.targetPosition*. Questo viene fatto perché il model è stato modificato dopo la *remove* e questo potrebbe aver modificato anche la *this.targetPosition*. Infine richiama la insert su *this.targetPosition* e sul range ritornato dalla *remove*, e ritorna il range.

3.4.3 Remove di nodi

```
1   function remove(range){
2     const parent = range.start.parent;
3     _splitNodeAtPosition(range.start);
4     _splitNodeAtPosition(range.end);
5     const removed = parent.removeChildren(range.start.index, range.end.
          index - range.start.index);
6     _mergeNodesAtIndex(parent, range.start.index);
7     return removed;
8   }
```

Listing 3.3: Funzione remove in writer.js

Richiamata dalle istanze di RemoveOperation, questa funzione rimuove i nodi dal range passatole come argomento. Salva l'inizio del range del parent e splitta il nodo, poi va a rimuovere tutti i figli di tale nodo. Infine fa il merge del parent coi nodi successivi al

range rimosso e ritorna i nodi rimossi.

3.4.4 Modifica di Attributi

```
1 class AttributeOperation extends Operation{
2   _execute(){
3     if (!isEqual( this.oldValue, this.newValue)){
4       writer.setAttribute(this.range, this.key, this.newValue);
5     }
6     return{range: this.range, key: this.key, oldValue: this.oldValue,
          newValue: this.newValue};
7   }
8 }
```

Listing 3.4: attributeoperation.js from CKEditor5 Model

_execute richiama il metodo *setAttribute* di riga 4, il quale estrae il range passatogli come primo argomento e nel caso in cui *this.newValue* sia diverso da null setta l'attributo del nodo al nuovo valore. In caso contrario lo rimuove usando *this.key*. Infine reinserisce il nodo richiamando la *_mergeNodesAtIndex*. Ritorna un oggetto contenente il range, la chiave, il vecchio e il nuovo valore dell'attributo.

3.5 High-Level API

La gestione di operazioni più complesse quali split/merge o wrap/unwrap di nodi su un documento, viene espresso in CKEditor5, come in tanti altri editor che dispongono di OT, mediante l'astrazione del *delta*. Un delta è un insieme di operazioni che vengono applicate al documento e rappresentano una singola modifica dal punto di vista dell'utente. A livello implementativo un delta viene rappresentato con un array di operazioni. A ogni delta vengono applicate le composizioni e le trasformazioni per mantenere la convergenza del documento e per gestire eventuali conflitti tra operazioni concorrenti.

3.5.1 Merge

```
1  function(position) {
2    const delta = new MergeDelta();
3    this.addDelta(delta);
4    const nodeBefore = position.nodeBefore;
5    const nodeAfter = position.nodeAfter;
6    const positionAfter = Position.createFromParentAndOffset(nodeAfter,0)
7    const positionBefore = Position.createFromParentAndOffset(nodeBefore,
         nodeBefore.maxOffset);
8    const move = new MoveOperation(positionAfter, nodeAfter.maxOffset,
        positionBefore, this.document.version);
9    move.isSticky = true;
10   delta.addOperation(move);
11   this.document.applyOperation(move);
12   /* ... */
13   const remove = new RemoveOperation(position, 1, gyPosition, this.
        document.version);
14   delta.addOperation(remove);
15   this.document.applyOperation(remove);
16   return this;
```

```
17 }
```

Listing 3.5: Funzione merge in mergedelta.js

Questa funzione crea un nuovo MergeDelta, si salva i due nodi da unire e le relative posizioni. Genera una move del secondo nodo all'interno del primo e aggiunge tale move al delta. Applica l'operazione al documento e poi rimuove il secondo nodo che a questo punto è vuoto poiché il suo contenuto è stato mosso all'interno del primo. Infine aggiunge la remove al delta e applica l'operazione al documento ritornando il delta al chiamante.

3.5.2 Split

```
1  function( position ) {
2    const delta = new SplitDelta();
3    this.addDelta(delta);
4    const splitElement = position.parent;
5    /* ... */
6    const insert = new InsertOperation(Position.createAfter(splitElement)
       , copy, this.document.version);
7    delta.addOperation(insert);
8    this.document.applyOperation(insert);
9    const move = new MoveOperation(position, splitElement.maxOffset -
       position.offset, Position.createFromParentAndOffset(copy, 0), this
       .document.version);
10   delta.addOperation(move);
11   this.document.applyOperation(move);
12   return this;
13 }
```

Listing 3.6: Funzione split in splitdelta.js

Questa funzione crea un nuovo SplitDelta e si salva la posizione di split. Genera una insert del contenuto del nodo dalla posizione di split fino alla fine, all'interno di un nodo fratello creato subito dopo di esso. Poi aggiunge l'operazione al delta e la applica al documento.

Successivamente genera una move, muove tale contenuto, aggiunge l'operazione al delta e la applica, ritornando il delta al chiamante.

3.5.3 Wrap

```
function(range, elementOrString){
  /*...*/
  const delta = new WrapDelta();
  this.addDelta(delta);
  const insert = new InsertOperation(range.end, element, this.document.
     version);
  delta.addOperation(insert);
  this.document.applyOperation(insert);
  const targetPosition = Position.createFromParentAndOffset(element,0);
  const move = new MoveOperation(range.start, range.end.offset - range.
     start.offset, targetPosition, this.document.version);
  delta.addOperation(move);
  this.document.applyOperation(move);
  return this;
}
```

Listing 3.7: Funzione wrap in wrapdelta.js

Questa funzione crea un nuovo WrapDelta, genera una insert del contenuto del nodo da wrappare, la aggiunge al delta e la applica al documento. Successivamente genera una move del contenuto del nodo e lo wrappa col nuovo nodo creato dalla createFromParentAndOffset. Aggiunge anche la move al delta, la applica e ritorna il delta al chiamante.

3.5.4 Unwrap

```
function(element){
  const delta = new UnwrapDelta();
  this.addDelta(delta);
  const sourcePosition = Position.createFromParentAndOffset(element,0);
  const move = new MoveOperation(sourcePosition, element.maxOffset,
     Position.createBefore(element), this.document.version);
```

```
6     delta.addOperation(move);
7     this.document.applyOperation(move);
8     /* ... */
9     const remove = new RemoveOperation(Position.createBefore(element), 1,
          gyPosition, this.document.version);
10    delta.addOperation(remove);
11    this.document.applyOperation(remove);
12    return this;
13  }
```

Listing 3.8: Funzione wrap in unwrapdelta.js

Questa funzione crea un nuovo UnwrapDelta, si salva la posizione attuale dell'elemento da unwrappare. Genera una move del contenuto del nodo, la aggiunge al delta e la applica al documento. Successivamente genera una remove del parent (nodo che wrappava il contenuto dell'elemento) che ora è vuoto, la aggiunge al delta, la applica e ritorna il delta al chiamante.

3.5.5 Transform

```
1  function defaultTransform(a, b, context){
2    const transformed = [];
3    let byOps = b.operations;
4    let newByOps = [];
5    for(const opA of a.operations){
6      const ops = [opA];
7      for (const opB of byOps){
8        for(let i=0; i<ops.length; i++){
9          const op = ops[i];
10         const results = operationTransform(op, opB, context);
11         Array.prototype.splice.apply(ops, [i, 1].concat(results));
12         i += results.length-1;
13         const reverseContext = Object.assign({}, context);
14         reverseContext.isStrong = !context.isStrong;
15         reverseContext.insertBefore = context.insertBefore !==
               undefined ? !context.insertBefore:undefined;
```

```
16          const updatedOpB = operationTransform(opB, op, reverseContext);
17          Array.prototype.push.apply(newByOps, updatedOpB);
18        }
19      }
20      byOps = newByOps;
21      newByOps = [];
22      for(const op of ops){
23        transformed.push(op);
24      }
25    }
26    return getNormalizedDeltas(a.constructor, transformed);
27  }
```

Questa funzione viene sfruttata da CKEditor come funzione di trasformazione di default per i delta che non rappresentano casi particolari di conflitto.

L'algoritmo usato è simile a dOPT (distributed OPerational Transformation), elencato tra gli algoritmi analizzati nella sezione 5 del capitolo 2.

L'array *transformed* salva le operazioni del deltaA che vengono trasformate contro quelle di deltaB. byOps una copia delle operazioni di b. newByOps salva le operazioni di b trasformate con quelle di a. Il primo for scorre tutte le operazioni di a, il secondo tutte quelle di b e applica la trasformazione alle operazioni di a sfruttando quelle di b, poi va a rimpiazzarle in deltaA. Successivamente inverte il contesto per poter applicare la trasformazione anche alle operazioni di b sfruttando le trasformate di a (caso speculare) e andando ad aggiornare deltaB (newByOps) con le operazioni trasformate. Così ogni singola operazione di a è stata trasformata da ogni singola operazione di b e viceversa. [24]

Capitolo 4

DMCM: Una matrice per la risoluzione di conflitti in OT su documenti strutturati

Dopo uno studio approfondito di CKEditor5 ho notato che gli implementatori hanno trattato solo parte delle trasformazioni possibili derivanti dal set di operazioni adottato, quindi ho deciso di modellare una mia matrice per gestire i conflitti, che ho chiamato DMCM (le mie iniziali seguite da Conflict Matrix).
Le operazioni complesse che ho deciso di prendere in considerazione sono: Move, Join/-Split e Wrap/Unwrap.

4.1 Esempi di entries della matrice dei conflitti

La matrice di risoluzione dei conflitti che ho definito si ispira a quella usata nel modello algoritmico dOPT (descritto nel capitolo 2, sezione 5), ma si prefigge l'obiettivo di dare uno schema alla risoluzione di conflitti ben più articolati di quelli cui fa riferimento il suddetto algoritmo.
La matrice ha dimensione nxn, dove n è il numero di operazioni prese in considerazione.

Nel mio caso le operazioni complesse esaminate sono Move, Join/Split e Wrap/Unwrap, che si sono andate ad aggiungere a Insert e Delete, dando così origine a una matrice dei conflitti di dimensione sette per sette.

4.1.1 Join vs. Wrap

Questo esempio di entry considera i casi di trasformazione generati dal conflitto di un' operazione di Join, operazione A, e un'operazione di Wrap, operazione B.
Il range su cui agisce A è quello compreso tra le parentesi quadre in rosso, mentre il range su cui agisce B è quello tra parentesi nere. Guardando il passaggio dei parametri alla pseudofunzione di trasformazione si suppone che la Join venga trasformata contro la Wrap, dunque che la Wrap venga applicata al documento e che la Join venga trasformata di conseguenza.
Nel primo caso limite analizzato si prende in considerazione lo scenario in cui il range che viene wrappato dall'operazione B sia uno dei due nodi fratelli di cui l'operazione A vuole fare Join. Il risultato come si può notare dalla figura 4.1 è che la Wrap del range di B venga applicata e che la Join venga annullata perché il primo (o il secondo) nodo è sceso gerarchicamente e non è più fratello del secondo <i>.
Nel secondo caso limite il range che deve essere wrappato da B contiene entrambi i nodi del Join di A, ne risulta che il range viene wrappato regolarmente dall'operazione B e l'operazione A deve ricalcolare la posizione dei due nodi da unire perché sono scesi di un livello nella gerarchia e hanno un nuovo nodo padre.
Il terzo ed ultimo caso particolare esamina l'intersezione tra il range da wrappare dall'operazione B e i nodi del Join. Come si può vedere in figura l'intersezione è data da uno dei due fratelli del Join che sta anche nel range dell'operazione B. In questo caso, come nel primo la Wrap viene applicata regolarmente e la Join viene annullata perché uno dei due nodi è sceso gerarchicamente e ha un nuovo nodo padre, diverso da quello del fratello.
Rimane il caso default che non genera conflittualità, in cui i range risultano disgiunti

e nessuno dei due contiene l'altro, quindi le operazioni vengono applicate in sequenza senza sfruttare la trasformazione.

transform(join, wrap)

caso 1 (rangeB.isOneSiblingInJoinA()):

[[Tizio]Caio]

TizioCaio

caso 2 (rangeB.contains(rangeA)):

[[Tizio Caio]]

[Tizio Caio]

Tizio Caio

caso 3 (intersection(rangeB, rangeA)):

[Tizio[Caio]Sempronio]

Tizio<u>CaioSempronio</u>

caso default:

[Tizio Caio][Sempronio]

[Tizio Caio]<u>Sempronio</u>

Tizio Caio<u>Sempronio</u>

Figura 4.1: Esempio di entry Join vs. Wrap

4.1.2 Delete vs. Split

Questo esempio di entry considera i casi di trasformazione generati dal conflitto di un'operazione di Delete, operazione A, e un'operazione di Split, operazione B.
Il range su cui agisce A è quello compreso tra le parentesi quadre in rosso, mentre la posizione di split di B è segnalata dalla frecciona nera che punta verso il basso. Si suppone che la Delete venga trasformata contro la Split, dunque che la Split venga applicata al documento e che la Join venga trasformata di conseguenza.

L'unico caso particolare analizzato per questo tipo di conflitto prende in considerazione lo scenario in cui la posizione di Split dell'operazione B sia contenuta all'interno del range che deve essere eliminato successivamente dalla Delete trasformata. L'esecuzione vedrà la Split invariata perché eseguita prima e la Delete modificata perché necessiterà di ricalcolare il numero di nodi da eliminare, che sarà aumentato di uno.
Il caso default non genera conflittualità, quindi le operazioni vengono applicate in sequenza senza sfruttare la trasformazione.

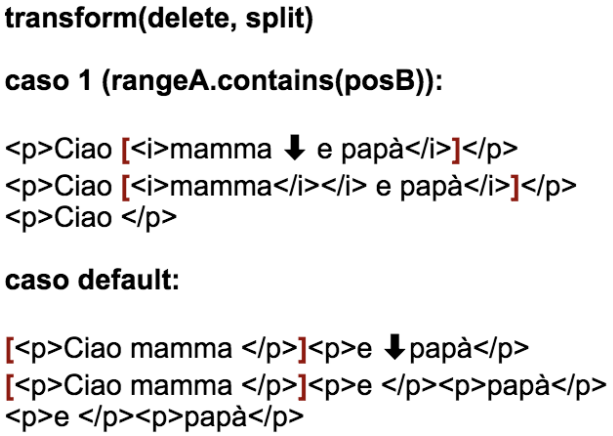


Figura 4.2: Esempio di entry Delete vs. Split

4.1.3 Unwrap vs. Insert

Questo esempio di entry considera i casi di trasformazione generati dal conflitto di un'operazione di Unwrap, operazione A, e un'operazione di Insert, operazione B.
Il range su cui agisce A è quello compreso tra le parentesi quadre in rosso, mentre la posizione di inserimento di B è segnalata dalla freccetta nera che punta verso il basso. Si suppone che l'Unwrap venga trasformata contro la Insert, dunque che la Insert venga applicata al documento e che l'Unwrap venga trasformata di conseguenza.
Il primo caso limite analizzato considera lo scenario in cui la posizione di inserimento sia interna al range della Unwrap e, in particolare, che tale posizione sia nel nodo padre da eliminare che contiene il range da unwrappare. L'esecuzione vedrà la Insert invariata,

mentre la Unwrap dovrà ricalcolare la dimensione del range da estrapolare perché sarà aumentata della dimensione del range inserito da B.
Il secondo caso limite considera lo scenario complementare, in cui la posizione di inserimento sia interna all'interno di uno dei nodi del range contenuti dal nodo padre da eliminare. L'esecuzione vedrà la Insert invariata e l'Unwrap modificata nella dimensione del range perché il sottoalbero sarà aumentato della dimensione del range di B.
Il caso default non genera conflittualità, quindi le operazioni vengono applicate in sequenza senza sfruttare la trasformazione.

transform (unwrap, insert)

caso 1 (rangeA.contains(posB) && posB.isInside(outerNodeA)):

<main>[Ciao mamma ↓]</main>
oppure([↓ Ciao mamma])

[<main> Ciao mamma e papà]</main>

<main> Ciao mamma e papà </main>

caso 2 (rangeA.contains(posB) && posB.isInside(innerNodeA)):

<main>[Ciao ↓mamma]</main>

<main>[Ciao ciao mamma]</main>

<main> Ciao ciao mamma </main>

caso default:

[Tizio] Caio e ↓

[Tizio] Caio e Sempronio

Tizio Caio e Sempronio

Figura 4.3: Esempio di entry Unwrap vs. Insert

4.1.4 Move vs. Join

Questo esempio di entry considera i casi di trasformazione generati dal conflitto di un'operazione di Move, operazione A, e un'operazione di Join, operazione B.
Il range sorgente su cui agisce A è quello compreso tra le parentesi quadre in rosso, la posizione di destinazione è quella indicata dalla freccetta all'interno delle parentesi quadre rosse, mentre il range su cui agisce la Join è segnalato dalle parentesi quadre nere. Si suppone che la Move venga trasformata contro la Join, dunque che la Join venga applicata al documento e che la Move venga trasformata di conseguenza.
Il primo caso limite analizzato considera lo scenario in cui il range della Move di A sia all'interno di uno dei nodi di cui viene fatto Join da B. L'esecuzione vede la Join invariata e la Move deve ricalcolare il range interno a uno dei due nodi fratelli che faceva parte del join internamente al nuovo nodo generato e lo sposta nella posizione bersaglio.
Il secondo caso limite considera lo scenario in cui i due nodi della Join di B siano interni al range sorgente della Move di A. L'esecuzione vedrà la Join invariata e la Move dovrà ricalcolare la dimensione del proprio range, perché nel sotto albero il numero di nodi sarà diminuito di uno.
Il terzo caso limite è quello dell'intersezione tra i due range. L'esecuzione vedrà la Join invariata, mentre la Move avrà il proprio range spezzato internamento al nodo generato dalla Join e esternamente, quindi dovrà andare a estrapolare la parte di range e aggiungerla a quella rimasta fuori e muovere il range nella posizione bersaglio.
Il caso default non genera conflittualità, quindi le operazioni vengono applicate in sequenza senza sfruttare la trasformazione.

transform(move, join)

caso 1 (rangeB.contains(rangeA)):

<section>[↓]</section>
[Ciao[mamma]]

<section>[↓]</section>
Ciao[mamma]

<section>mamma</section>
Ciao

caso 2 (rangeA.contains(rangeB)):

<section>[↓]</section>
[[Ciaomamma]e papà]

<section>[↓]</section>
[Ciao mammae papà]

<section>Ciao mammae papà</section>

caso 3 (intersection(rangeA, rangeB)):

<section>[↓]</section>
[Ciao[mamma]e papà]

<section>[↓]</section>
Ciao [mamma][e papà]

<section>mammae papà</section>
Ciao

Figura 4.4: Esempio di entry Move vs. Join

Questi sono solo alcuni degli esempi di risoluzione dei conflitti, la matrice completa è in un repository sul mio profilo github disponibile al link: https://github.com/davidmantegna/Conflict-Matrix.git.

Capitolo 5

Valutazione Qualitativa/Quantitativa

5.1 Una metrica di valutazione

Avendo analizzato a fondo il codice sorgente di CKEditor5 (in fase di sviluppo solo da quest'anno, dopo tre anni di studio di OT) ed avendo notato delle incuranze nell'analisi degli scenari di conflitto, ho deciso di modellare una mia matrice di gestione dei conflitti, che ho deciso di chiamare DMCM, che riassume l'insieme delle situazioni che possono portare al raggiungimento di uno stato incoerente del documento.
Ho deciso di mantenere l'idea degli implementatori di CKEditor di suddividere l'API in Low-Level e High-Level, apportandovi una sola modifica: considerare la move come operazione complessa e non più come operazione base. Le operazioni semplici costituiscono la Low-Level API e sono Insert e Delete, mentre quelle complesse, come si può dedurre, sono incluse nella High-Level API e sono: Move, Join/Split e Wrap/Unwrap.
Ho deciso, inoltre, di scrivere le funzioni di trasformazione relative ad ogni entry della matrice in pseudo Javascript, basandole su un modello a priorità come quello di dOPT. Questo algoritmo è quello che mi è sembrato più vicino alle mie necessità, siccome è quello che ha introdotto il concetto di matrice di trasformazione.

I restanti algoritmi analizzati nella sezione 5 del capitolo 2 si concentrano più su altre feature di OT (undo, context-vector, etc.), che però non sono utili alla mia causa.
Google Wave nonostante includesse l'implementazione di operazioni su rich-text, trattava un insieme piuttosto ristretto e semanticamente poco significativo di operazioni (insert element start, delete element end, replace attributes, etc.) per essere usato in questo contesto e Google Docs, invece, è un semplice word processor per documenti testuali, quindi si limita alle sole Insert e Delete.

Sempre riguardo le funzioni di trasformazione (entry della matrice):

- sono inclusive transformation (IT) functions, poiché includono il risultato dell'applicazione di OpB, contro cui si trasforma OpA;
- rispettano il principio di causalità, cioè l'ordinamento temporale delle operazioni, perché si basano su priorità come suggerito da dOPT;
- preservano la coerenza del documento finale;
- rispettano la prima proprietà di trasformazione (TP1), poiché garantiscono che l'identità relativa a questa proprietà sia rispettata

Lo pseudocodice associato alla matrice, che non includo nella trattazione, è contenuto in un repository su github, che viene linkato al termine del capitolo 4.
DMCM, dunque, vuole dare uno schema esaustivo alla risoluzione di conflitti tra operazioni complesse in OT. Essa ha dimensione nxn, dove n è il numero di operazioni prese in considerazione. La sua dimensione è di sette operazioni contro sette, quindi 49 entries.
Ogni entry analizza gli scenari di conflitto tra due operazioni che vengono passate in input alla funzione di trasformazione relativa ad essa.
Ogni funzione ha la sua complementare: nella riga i e colonna j viene valutata la trasformazione di OpA contro OpB, mentre nella riga j e colonna i avviene il contrario. In alcuni casi limite viene restituito null perché dal punto di vista dell'utente potrebbero sorgere comportamenti anomali.

Questa matrice definisce l'utilizzo di Operational Transformation in un contesto specifico, quello dei documenti strutturati, e propone una soluzione a livello teorico per un problema che anche in letteratura è stato trattato relativamente poco.

Conclusioni

In questa tesi sono state analizzate approfonditamente le fondamenta teoriche di Operational Transformation, un protocollo di gestione della concorrenza e della coerenza nei sistemi di collaborazione real-time, prendendo in esame gli algoritmi teorici e le applicazioni all'editing di documenti e sfociando nel design di un modello di gestione di conflitti di operazioni complesse applicate ai documenti strutturati.
Si è partiti da un'infarinatura generale sui sistemi di collaborazione real-time, sulle tecnologie che permettono la realizzazione del real-time in Javascript e si sono introdotte le possibili alternative ad OT, dando però una giustificazione, nella sezione 1.4, alla sua scelta.
Nel capitolo 2 sono stati evidenziati i requisiti fondamentali che devono essere rispettati dalle funzioni di trasformazione e dai sistemi OT per evitare di intaccare la coerenza del documento finale. Sempre nello stesso capitolo sono state presentate le applicazioni di OT più note nel mondo dell'industria (Jupiter, Google Wave, Google Docs), che hanno apportato un grossissimo contributo alla sua teoria, rivoluzionandola in parte. Si è preso in esame anche un caso molto complesso di trasformazione e composizione delle operazioni del client e del server per restare sullo stesso path e mantenere le copie del documento sincronizzate.
Il capitolo 3 si apre con una breve digressione sui documenti strutturati, in particolare HTML e XML, e sulla possibilità di effettuare operazioni complesse, come copy and paste, drag n drop, navigazione da tastiera, etc. attualmente messa a disposizione solo dall'attributo contentEditable. Continua con l'analisi di un caso reale di implementazione di un sistema OT per la gestione di operazioni complesse su documenti HTML. Il

sistema in questione è CKEditor5, ancora in versione alpha, che non fornisce un'implementazione vera e propria di server che gestisca le trasformazioni, ma fornisce comunque un modulo di gestione di trasformazioni base tra coppie di operazioni.
Dunque, partendo dall'incompletezza del modulo di CKEditor, nel capitolo 4 illustro la mia idea di design di una possibile matrice per la gestione di conflitti, sfruttando la loro idea di suddividere le operazioni su due livelli, una Low-Level e una High-Level. Le operazioni incluse nella matrice, a basso livello sono Insert e Delete, mentre ad alto livello sono Move, Split e Join, Wrap e Unwrap.
La suddetta matrice è il risultato di un'analisi accurata degli scenari di conflitto che possono essere generati agendo con le operazioni sopraccitate su un documento HTML condiviso. Essa fornisce una gestione ragionevole dei conflitti e può tranquillamente essere la base di partenza per l'implementazione di un sistema funzionante basato su Operational Transformation, possibilmente che usufruisca dell'algoritmo dOPT come in CKEditor, con l'uso di un History Buffer per mantenere la cronologia delle operazioni applicate al documento e che si basi su un protocollo client/server (e non P2P) con acknowledgement delle operazioni, in modo da alleggerire il server e mantenere i client sul suo stesso path.

La mia è solo una proposta algoritmica per risolvere un problema decisamente complesso. La realizzazione di un sistema OT con queste caratteristiche richiederebbe sicuramente una mole di lavoro piuttosto massiccia, ma con uno schema iniziale di questo genere, si parte da una base concettuale già avanzata.

Bibliografia

[1] Tieme van Veen. What are long-polling, websockets, server-sent events (sse) and comet? https://stackoverflow.com/questions/11077857/what-are-long-polling-websockets-server-sent-events-sse-and-comet/12855533#12855533.

[2] Todd Hoff. How League Of Legends Scaled Chat To 70 Million Players - It Takes Lots Of Minions. http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.

[3] Peter Bourgon. Roshi: a CRDT system for timestamped events. https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events.

[4] Sander Mak. Facebook Announces Apollo at QCon NY 2014. https://dzone.com/articles/facebook-announces-apollo-qcon.

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400, Grenoble, France, October 2011. Springer.

[6] Neil Fraser. Differential Synchronization. https://neil.fraser.name/writing/sync/eng047-fraser.pdf.

[7] S.J. Gibbs C.A. Ellis. Concurrency control in groupware systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(2):399–407, 1989.

[8] Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Transactions Computer-Human Interaction*, 9(4):309?361, 2002.

[9] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[10] Rui Li Du Li. Preserving operation effects relation in group editors. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, page 457?466, 2004.

[11] Du Li Rui Li. A new operational transformation framework for real-time group editors. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):307?319, 2007.

[12] Du Li Rui Li. Commutativity-based concurrency control in groupware. *Proceedings of the First IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[13] Xiaohua Jia Chengzheng Sun, David Chen. Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems. *Proceedings of The 21st Australasian Computer Science Conference*, 1998.

[14] Daniel Spiewak. Code Commit - Understanding and Applying Operational Transformation. `http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation`.

[15] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors.

Proceedings of the ACM conference on Computer supported cooperative work, pages 288–297, 1996.

[16] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. Reducing the problems of group undo. *Proceedings of the ACM Conference on Supporting Group Work*, pages 131–139, 1999.

[17] Clarence Ellis Chengzheng Sun. Operational transformation in real-time group editors: Issues, algorithms, and achievements. *Proceedings of ACM Conference on Computer Supported Cooperative Work*, pages 59–68, Nov 1998.

[18] D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, Oct 2009.

[19] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120, 1995.

[20] Yi Xu and Chengzheng Sun. Conditions and patterns for achieving convergence in ot-based co-editors. *IEEE Transactions on Parallel and Distributed Systems*, pages 695–709, 2016.

[21] Google. Google Wave Documentation. `https://incubator.apache.org/wave/documentation.html`.

[22] Google I/O 2013. The Secrets of the Drive Realtime API. `https://www.youtube.com/watch?v=hv14PTbkIs0`.

[23] Frederico Knabben. Ckeditor 5: A new era for rich text editing. `https://ckeditor.com/blog/CKEditor-5-A-new-era-for-rich-text-editing/`.

[24] Piotrek Koszuliski. Ckeditor 5 source code. https://github.com/ckeditor/ckeditor5-engine.