Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Informatica

L'integrazione di un algoritmo di Operational Transformation in un editor strutturato

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Fabrizio Meniconi

Sessione III
Anno Accademico 2018/2019

Indice

Introduzione

Operational Transformation è una tecnologia che permette la risoluzione di conflitti in ambienti di collaborazione real-time per l'editing di documenti testuali.
Il primo sistema basato su Operational Transformation 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.
Nel corso della storia il protocollo OT, è stato oggetto di studi che hanno permesso la nascita di modelli di coerenza più efficienti e algoritmi OT-based sempre più performanti. Ad oggi questa tecnologià è alla base di sistemi di editing molto comuni come Google Docs e Google Wave.
Nei sistemi di editing collaborativo Real-Time più utenti si trovano ad apportare modifiche ad uno stesso documento contemporaneamente. Queste modifiche, a volte, possono riguardare la stessa porzione di testo ed entrare in conflitto tra di loro. La gestione di questi conflitti tra operazioni concorrenti rappresenta un problema di cruciale importanza in quanto, se affrontato inadeguatamente, può portare a stati incoerenti del modello di dati condiviso tra gli utenti connessi al sistema.
Operational Transformation propone una serie di metodologie e proprietà che permettono di risolvere queste problematiche in maniera efficiente senza appesantire troppo il sistema.
Un ulteriore grado di complessità si aggiunge quando si parla di editor strut-

turati. In questi sistemi, infatti, le operazioni non riguardano solo il contenuto testuale, ma anche elementi di layout, dimensioni dei caratteri e metadati. Più modifiche possono essere apportate contemporaneamente dallo stesso sistema, assumendo un significato semanticamente più complesso di una semplice operazione di cancellazione o di inserimento di testo.
Per questo motivo, dopo l'approfondimento degli aspetti teorici di OT, mi sono cimentato nella realizzazione di un algoritmo di Operational Transformation in un editor strutturato.
L'editor che ho scelto è SSE (Smart Structured Editor), una piattaforma web sviluppata all'interno del laboratorio DASPLab della mia facoltà, che si occupa della creazione e gestione di documenti strutturati scritti in ADF, un linguaggio di markup che è stato creato appositamente per far fronte alle esigenze dell'azienda ALSTOM.
Il mio obiettivo consiste nel verificare se gli aspetti teorici che ho approfondito siano applicabili in una realtà già esistente come SSE.
Per poter presentare questo progetto procederò, inizialmente, con una analisi approfondita di OT. Verranno mostrati i modelli di convergenza che sono stati proposti nel corso del tempo, per poi passare alla funzione di trasformazione, che descrive le metodologie per la risoluzione dei conflitti tra due operazioni, prendendo in analisi le sue proprietà di convergenza e inverse. Queste proprietà verranno anche applicate ad un caso complesso.
Chiarita la motivazione della scelta di OT rispetto ad altre alternative come Conflictfree Replicated Data Types e Differential Synchronization, sono andato a vedere come è possible approcciarsi teoricamente ad Operational Transformation su un editor strutturato.
Successivamente approfondirò le caratteristiche degli editor Real-Time mettendoli a confronto con sistemi di editing sincrono. Prenderò anche in esame le principali tecnologie Javascript presenti oggi, che ne permettono la realizzazione: Ajax Polling, XMLHttpRequest Long-Polling, SSE (Server-Sent Events) e Web Sockets.

La seconda parte di questa dissertazione sarà dedicata all'aspetto implementativo del mio progetto. Inizialmente presenterò SSE, l'editor strutturato dove si è concentrato il mio lavoro, e le principali caratteristiche della documentazione ALSTOM, il fine per cui è stata creata, quali obiettivi vuole soddisfare e la sua struttura base. Farò anche accenno all'editor per documenti strutturati TinyMCE mostrando le sue strutture dati e metodi che più mi sono stati utili.
In seguito passerò alla presentazione effettiva del mio algoritmo: l'idea generale che c'è dietro e quali soluzioni ho adottato per risolvere le principali problematiche in cui mi sono imbattuto, come l'estrazione delle operazioni, la memorizzazione di queste ultime da parte del server e infine la gestione dei conflitti per arrivare ad un documento convergente.
Nella parte finale procederò alla valutazione qualitativa/quantitativa della mia tesi verificando se l'algoritmo che ho prodotto rispetta tutti i dettami di Operational Tranformation. Successivamente esporrò come lavorare in un editor strutturato abbia influenzato alcune mie scelte di implementazione, per concludere poi con l'analisi di alcuni aspetti dove il progetto può migliorare.

Capitolo 1

Operational Transformation

Essendomi posto l'obbiettivo di realizzare un sistema di editing collaborativo Real-Time, mi sono imbattuto nei classici problemi di questo tema, ovvero: l'intercettazione delle modifiche apportate localmente dai singoli utenti nel documento, la scelta dell'approccio e delle strutture dati da utilizzare per rappresentare queste modifiche nel mio algoritmo e le modalità da seguire per far sì che ogni modifica apportata da ogni singolo utente si intersechi seguendo un modello coerente con quelle degli altri dando poi un risultato finale identico per tutti.
Ho deciso di utilizzare Operational Transformation come protocollo di gestione della correttezza e della coerenza del documento finale. In questo capitolo ne descriverò le caratteristiche e i punti di forza, spiegando il perchè è stato preferito ad altri approcci.
Nel Capitolo 4, invece, esporrò in quale modo ho incorporato i principi fondamentali di OT nel mio progetto.

1.1 Operational Transformation: Background

Nei sistemi di editing collaborativo spesso il documento condiviso viene replicato per ognuno dei client connessi. Si avranno, quindi, tante copie differenti del documento quante il numero di utenti che stanno lavorando sul 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.

Nell'approccio pessimistico si cerca di dare l'impressione ai client che esista un'unica copia del documento all'interno del sistema: la copia di un solo utente è modificabile, mentre quelle degli altri sono accessibili solo in lettura.

A differenza del meccanismo pessimistico, negli approcci ottimistici si tollera la divergenza momentanea delle copie del documento condiviso tra i client per far convergere ognuna di esse a uno stato finale entro un determinato intervallo di tempo; per questo motivo sono più adeguati all'editing collaborativo. Operational Transformation (OT), sviluppato dall'idea descritta nel paper di Ellis e Gibbs "Concurrency control in groupware systems" del 1989, è il meccanismo più studiato e più utilizzato nel mondo dello sviluppo software di editor collaborativi per il mantenimento della sincronia di documenti testuali.

In OT ogni modifica al documento viene vista come un'operazione di inserimento o di concellazione; il sistema estrae e inserisce in una struttura dati apposita, solitamente in formato JSON, il tipo di modifica, in quale posizione questa avviene e quale porzione di testo interessa.

L'applicazione della funzione di trasformazione porta il documento in un nuovo stato. 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 che possa gestire tali operazioni senza aumentare vertiginosamente la complessità del sistema.

Permette, inoltre, di mantenere un'unica copia del documento nel server, il quale diventa il punto di riferimento da cui recuperare il testo comprensivo di tutte le operzioni 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.

1.2 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 modelli di coerenza alternativi tra loro.
Ai fini della loro presentazione verrà utilizzata la notazione $\longrightarrow$ che rappresenta una relazione tra due modifiche dove l'operazione sinistra viene considerata eseguita prima dell'operazione destra.

Causality Convergence (CC)

- Casualità: per ogni coppia di operazioni $\boldsymbol{opA}$ e $\boldsymbol{opB}$ se vale la relazione $\boldsymbol{opA} \longrightarrow \boldsymbol{opB}$ allora $\boldsymbol{opA}$ viene causalmente prima di $\boldsymbol{opB}$, cioè è stata eseguita prima di essa.

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

L'esecuzione parallela delle operazioni porterà ogni replica a uno stato divergente dato che in generale non può esserci commutatività. Ellis e Gibb proposero l'introduzione di uno state vector per definire la precedenza tra un'operazione e l'altra [5].

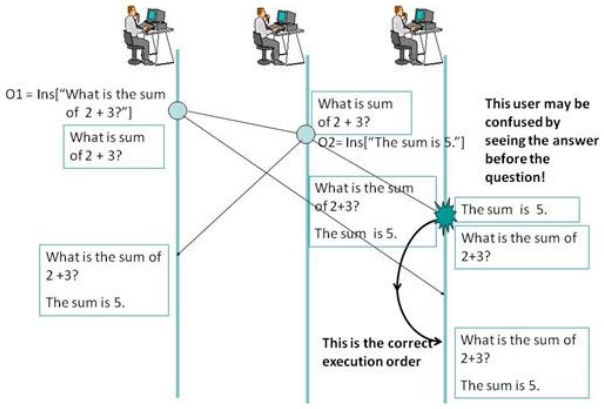


Figura 1.1: Un esempio di rispetto della casualità

Causality Convergence Intention Preservation (CCI)

- Casualità: per ogni coppia di operazioni ***opA*** e ***opB*** se vale la relazione $\boldsymbol{opA} \longrightarrow \boldsymbol{opB}$ allora ***opA*** viene causalmente prima di ***opB***, cioè è stata eseguita prima di essa.

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

- Conservazione delle Intenzioni: per ogni operazione ***op*** gli effetti della sua esecuzione su tutti i siti devono rispettare le intenzioni di ***op***.

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 [6].
La conservazione delle intenzioni del modello CCI risulta impossibile da formalizzare, per questo motivo è stato proposto il modello CSM [7].

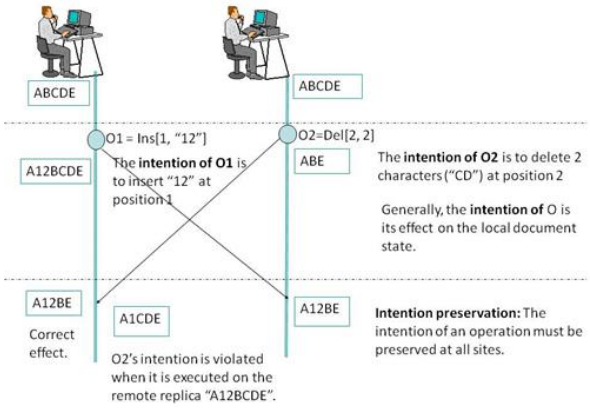


Figura 1.2: Un esempio di conservazione delle intenzioni tra due utenti

Causality Single Multiple (CSM)

- Casualità: per ogni coppia di operazioni $\boldsymbol{opA}$ e $\boldsymbol{opB}$ se vale la relazione $\boldsymbol{opA} \longrightarrow \boldsymbol{opB}$ allora $\boldsymbol{opA}$ viene causalmente prima di $\boldsymbol{opB}$, cioè è stata eseguita prima di essa.

- Effetto di una singola operazione: eseguire l'operazione in qualunque stato di esecuzione ha lo stesso effetto di eseguirla nel suo stato di generazione.

- Effetto di operazioni multiple: la relazione che intercorre tra ogni coppia di operazioni resta tale in qualunque stato esse vengano eseguite

Convergence Admissibility (CA)

- Casualità: per ogni coppia di operazioni $\boldsymbol{opA}$ e $\boldsymbol{opB}$ se vale la relazione $\boldsymbol{opA} \longrightarrow \boldsymbol{opB}$ allora $\boldsymbol{opA}$ viene causalmente prima di $\boldsymbol{opB}$, cioè è stata eseguita prima di essa.

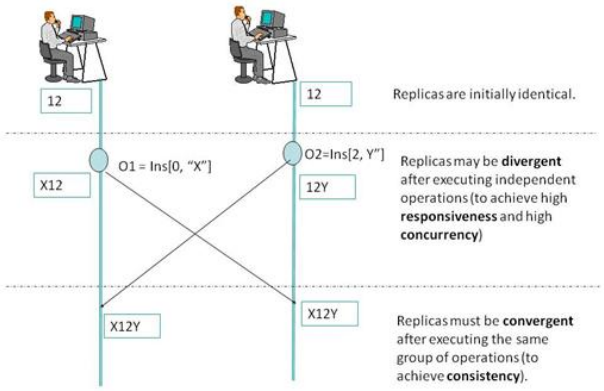


Figura 1.3: Un esempio di convergenza del documento

- Ammissibilità: 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 [8].

1.3 Proprietà della trasformazione

Per gestire le operazioni concorrenti la funzione di trasformazione prende in input due operazioni che sono state applicate al documento nello stesso stato 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 $\boldsymbol{IT(a, b)}$, trasforma $\boldsymbol{O_a}$ in rapporto a $\boldsymbol{O_b}$ in modo che l'impatto di $\boldsymbol{O_b}$ venga incluso.

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

Le funzioni di trasformazione composta possono fare uso sia delle funzionalità inclusive che esclusive [9].

1.3.1 Convergenza

Qualsiasi funzione di trasformazione necessita di due proprietà, o precondizioni, fondamentali per garantire la propria convergenza, **TP1** e **TP2** [6].

TP1 Date due operazioni $\boldsymbol{O_a}$ e $\boldsymbol{O_b}$, la funzione di trasformazione $\boldsymbol{T}$ soddisfa $\boldsymbol{TP1} \iff \boldsymbol{O_a \circ T(O_a, O_b) \equiv O_b \circ T(O_a, O_b)}$ dove $\circ$ denota una sequenza di operazioni contenente $\boldsymbol{O_i}$ seguita da $\boldsymbol{O_j}$ 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 $\boldsymbol{O_a}$,$\boldsymbol{O_b}$ e $\boldsymbol{O_c}$, la funzione di trasformazione $\boldsymbol{T}$ soddisfa $\boldsymbol{TP2} \iff \boldsymbol{T(O_c, O_a \circ T(O_b, O_a)) \equiv T(O_c, O_b \circ T(O_a, O_b))}$, 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.

1.3.2 Proprietà Inverse

OT supporta anche l'annullamento, undo, di una o più operazioni, rispettando, però, alcuni prerequisiti detti proprietà inverse della funzione di trasformazione: $\boldsymbol{IP1}$, $\boldsymbol{IP2}$ e $\boldsymbol{IP3}$.

IP1 A partire da un document state $\boldsymbol{S}$, l'esecuzione dell'operazione $\boldsymbol{O}$ e della sua inversa $\boldsymbol{O'}$ risulta in $\boldsymbol{S \circ O \circ O' = S}$, ovvero $\boldsymbol{O \circ \overline{O}}$ equivale

a un'identità rispetto all'effetto sullo stato del documento. Questa proprietà deve essere rispettata se e solo se il sistema OT ammette operazioni inverse per il raggiungimento dell'undo.

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

IP3 Date le operazioni O_a e O_b, se $O'_a = IT(O_a, O_b)$, $O'_b = IT(O_b, O_a)$ e $O_{a\rfloor} = IT(\overline{O_a}, O'_b)$, allora $(\overline{O_a})' = \overline{O_a}'$ [10] .

1.4 Il problema del diamante

In questa sezione prenderò in esame il problema del Diamante, ovvero come vengono gestite, attraverso le proprietà di trasformazione descritte nella sezione precedente, due o più operazioni avvenute nello stesso istante in maniera concorrente per ottenere un risultanto finale convergente.
Immaginiamo di avere due utenti A e B che modificano contemporaneamente lo stesso documento: avremo due o più operazioni inviate al server che dovrà mantenere l'atomicità di ogni modifica (per evitare condizioni di race-condition), quindi una di queste operazioni verrà applicata per prima.
Tuttavia, non appena viene applicata una di queste operazioni, la conservazione per le altre non sarà più valida. [11]

1.4.1 Caso base

Considerando la prima immagine: l'operazione a è eseguita dal client A e la modifica b è svolta dal client B.

1. Vengono inviate contemporaneamente le operazioni a dall'utente A e b dall'utente B al server.

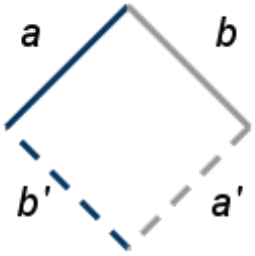
2. Il server sceglie il "vincitore" della race-condition (presupponiamo che sia l'utente A).

3. Viene applicata la funzione T dal server sulle due operazioni che produrrà in output *a'* e *b'*.

$$T(a,b) = (a',b') \text{ dove } a \circ b' \equiv b \circ a'$$

4. *a'* viene inviato ad *A* e *a'* viene inviato a *B*.

5. *A* e *B* applicano le risultanti della funzione T e ottengono entrambe lo stesso testo risultante.

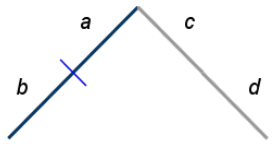


1.4.2 Caso complesso

Partendo dal caso mostrato nella figura

1. *A* produce *a* e *b*, *B* produce *c* e *d*.

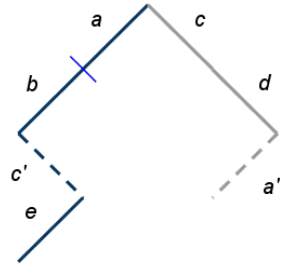
2. *A* invia solamente *a* al server e bufferizza *b* e *B* invia *c* e *d*.



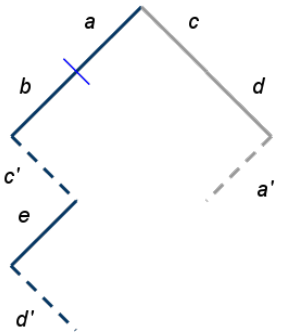
3. il server compone c con d e trasforma a con il risultato di tale composizione, ottenendo a' e propagandola al client B, imponendo un ordinamento.

4. A sua volta il client A riceve c e la trasforma con il risultato della composizione di a e b, producendo c.

5. 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 l'operazione e al proprio state space. Siccome il client ha momentaneamente inviato solo a al server, si necessita di inserire 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.

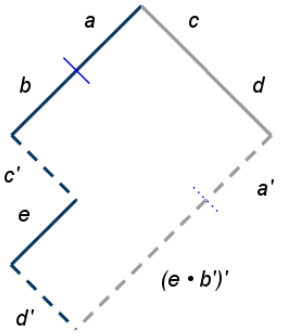


6. Da qui in poi si compone e con il ponte, il client riceve d dal server e come con c, la trasforma sfruttando il ponte, derivando. d', ora e potrà essere bufferizzata.



7. infine b ed e verranno inviate al server che le rigirerà a B.

8. Verrà applicata in b' in B per poi combinarlo con e ottenendo così la coerenza in tutti e due i documenti.



1.5 Alternative ad OT

Due alternative possibili ad Operational Transformation sono Conflict-free Replicated Data Types e Differential Synchronization.

1.5.1 Conflictfree Replicated Data Types

In Conflict-free Replicated Data Type il documento condiviso è visto come un insieme di oggetti che possono essere aggiornati senza grossi sforzi di sincronizzazione, poiché ogni utente può eseguire un'operazione senza doversi sincronizzare a priori con tutte le altre.
Tale operazione viene inviata in maniera asincrona e ogni replica applica tutti gli aggiornamenti, possibilmente in ordini differenti. La gestione degli eventuali conflitti è affidata al server CRDT che, mediante l'uso di un algoritmo, 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.

1.5.2 Differential Synchronization

Differential synchronization è un algoritmo simmetrico per stabilire la coerenza tra i dati da una sorgente a una memoria di dati target e viceversa: questa è garantita mediante la continua sincronizzazione dei dati nel tempo. In questo modello ogni utente mantiene salvate in memoria due versioni del documento: nella copia principale è contenuto il testo con tutte le modifiche recenti, nella copia ombra viene mantenuta l'ultima versione aggiornata con il server.
Quest'ultimo mantiene, invece, tre copie: una copia di testo, una shadow e una di backup.
L'algoritmo in questione consiste in un ciclo infinito di operazioni Diff e Patch. Quando viene apportata una modifica da parte del client, questo applica l'operazione di differenza (Diff) tra le due versioni del testo che possiede per poi inviare il risultato al server.
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 che come risultato dovrebbe convergere a tre copie identiche del documento. Il secondo ruolo è quello più complesso da eseguire e consiste nel tenere aggiornato il documento del Server seguendo le modifiche che vengono effettuate al documento del Client; il file 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. Infatti Patch deve esaminare due variabili (potenzialmente in conflitto) quando si tenta di trovare la posizione corretta per effettuare un inserimento. Il primo è trovare il testo con il minimo numero di operazioni elementari di differenza tra il testo atteso (basato sul contesto della patch) e il testo effettivo. Il secondo è trovare una posizione ragionevolmente vicina alla posizione prevista della patch per poi applicarla. [12]

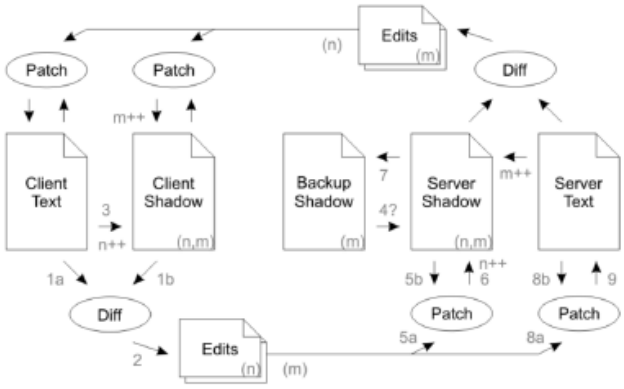


Figura 1.4: Differential Synchronization

1.5.3 La scelta di OT

E' stato preferito studiare e sfruttare Operational Transformation come protocollo di gestione della concorrenza e della coerenza del documento finale poiché, se implementato nella maniera adeguata, risulta più flessibile e alleggerisce di molto il carico di lavoro del server, assicurando la convergenza in maniera più efficace degli altri due approcci.
Differential Synchronization appesantisce troppo server e client considerato che richiede di mantenere in memoria rispettivamente tre e due copie del documento; CRDT, invece, richiede l'invio degli update a tutte le repliche comprendendo anche l'intero stato del documento, risultando troppo onerosi a livello computazionale e di occupazione di memoria.

1.6 OT su 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.
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 elemento, o nodo, può contenere un riferimento ad altri nodi oltre che un attributo che ne denota il valore.
Seguendo questa organizzazione XML (e anche HTML) fornisce una rappresentazione lineare attraverso l'inserimento di tag di apertura e di chiusura, rispettivamente all'inizio e alla fine di ogni nodo.
OT può essere utilizzato anche su questo tipo di documenti. L'utente può così modificare la struttura del documento e non solamente il testo.

In questo tipo di editor si possono distinguere tre livelli di operazioni: quelle di basso livello (meccaniche), quelle di alto livello (strutturali) e quelle semantiche. [13] [14]

1.6.1 Operazioni di basso livello o meccaniche

In memoria vengono rappresentate come un array di dati in formato JSON che presenta la seguente struttura:

```
{
    "id": "edit-00003",
    "op": "operazione meccanica",
    "pos": 100,
    "content": "testo e/o markup"
}
```

Dove:

- id: è l'identificativo dell'operazione meccanica.
- op: corrisponde all'operazione.
- pos: la posizione dell'inserimento/eliminazione viene contata sulla stringa dei caratteri del testo, considerando anche il markup e contandolo.
- content: cosa viene inserito/eliminato dall'operazione.

Le operazioni di basso livello possono essere di due tipi:

INS corrisponde all'inserimento di testo e markup.

DEL corrisponde all'eliminazione di testo e markup.

Sono scritte con tre lettere per distinguerle da quelle di alto livello. Queste due operazioni agiscono esclusivamente a livello di stringa di testo, quindi comprendendo anche il markup.

1.6.2 Operazioni di alto livello o strutturali

Le operazioni strutturali sono delle sequenze di uno o più operazioni meccaniche che hanno un senso proprio sul documento anche se vengono svolte in più fasi o in locazioni diverse. Queste possono risolversi all'interno di un unico nodo di testo (e parleremo allora di operazioni sul testo) o su strutture di markup complesse (e parleremo allora di operazioni sulla struttura). Le operazioni strutturali vengono svolte atomicamente.
In memoria vengono rappresentate come un array di dati in formato JSON che presenta la seguente struttura:

```
{
    "id": "structural-00026",
    "op": "operazione strutturale",
    "by": "autore",
    "timestamp": "2018-03-10T07:25:23.891Z",
    "items": [{
       "id": "edit-00066",
       "op": "operazione meccanica 1",
       "pos": 100,
       "content": "testo e/o markup"
    },
    {
       "id": "edit-00066",
       "op": "operazione meccanica 2",
       "pos": 100,
       "content": "testo e/o markup"
     }]
}
```

Dove:

- id: è l'identificativo dell'operazione strutturale.
- op: è l'operazione strutturale.
- by: indica l'autore della modifica.
- times stamp: indica quando è stata fatta l'operazione.
- items: è l'insieme di una o più operazioni meccaniche che compongono l'operazione strutturale.

Le operazioni di alto livello sono:

Insert inserimento di testo ed eventualmente di nuovi nodi nel documento.

Delete eliminazione di testo ed eventualmente di nuovi nodi nel documento.

Join unione di due nodi fratelli e dello stesso tipo. Il nodo unito prende le caratteristiche del primo dei due nodi, e quelle del secondo nodo vengono perse.

Split separazione di un nodo in due nodi fratelli dello stesso tipo. Entrambi i nodi prendono le caratteristiche del nodo unito.

Wrap annidamento di uno o più nodi con un nuovo nodo. Il contenuto non cambia, ma scende di un livello.

Unwrap rimozione di un nodo con promozione del suo contenuto al livello del nodo rimosso. Il contenuto non cambia, e sale di un livello.

1.6.3 Operazioni semantiche

Sono operazioni semantiche tutte quelle (sequenze di) operazioni a cui è semplice dare un significato immediatamente comprensibile ad un essere umano. La distinzione è intuitiva, intrinsecamente imprecisa, per cui è soggetta ad interpretazione e può essere discutibile da persona a persona.

Inoltre non è facile determinare un elenco definitivo e preciso di categorie utilizzabili.
Molte operazioni strutturali hanno un significato proprio semplice e ben descritto, per cui non ha senso raggrupparle in sovrastrutture.
Per tale motivo la caratterizzazione semantica viene usata alcune volte per raggruppare in sovrastrutture le operazioni strutturali, mentre negli altri casi viene considerata come un'ulteriore modalità di descrizione di quella che è, e rimane, un'operazione di alto livello.
Data la natura soggettiva ed interpretativa delle operazioni semantiche, lo scopo ultimo della caratterizzazione semantica delle operazioni strutturali è quello di fornire un ulteriore livello interpretativo, non quello di fornire una nuova realtà indiscutibile e definitiva, sopra le operazioni strutturali, che rimangono il livello massimo di caratterizzazione oggettiva delle operazioni che avvengono su un documento di testo. [14]
In memoria vengono rappresentate come un array di dati in formato JSON che presenta la seguente struttura:


```
{
    "id": "semantic-00055",
    "op": "operazione semantica",
    "items": [{
        "id": "structural-00026",
        "op": "operazione strutturale 1",
        "by": "autore",
        "timestamp": "2018-03-10T07:25:23.891Z",
        "items": [...],
    },
    {
        "id": "structural-00036",
        "op": "operazione strutturale 2",
```

```
        "by": "autore",
        "timestamp": "2018-03-10T07:25:23.891Z",
        "items": [...],
      }]
}
```

Dove:

- id: è l'identificativo dell'operazione semantica.
- op: è l'operazione semantica in questione.
- items: è l'insieme di una o più operazioni strutturali che compongono l'operazione semantica.

Le operazioni semantiche sono:

Fix Operazione che identifica la correzione di un errore grammaticale o strutturale.

Style Sono cambi stilistici quelle operazioni strutturali che cambiano aspetto e/o contenuto del documento senza cambiarne il significato.

Meaning operazioni strutturali che cambiano completamente il significato di un testo.

Edit War Una edit war è in continuo aggiornamento. Un'eventuale ulteriore modifica, anche molto successiva alle precedenti, che viene apportata da uno o più eventi nella stessa porzione di testo.

Edit Make Una edit wake (scia di modifiche) è una modifica tipicamente "vera" (cioè semantica e contenutistica) a cui segue una scia di piccole modifiche causate dalla prima per ristrutturare e riorganizzare la frase in seguito ai problemi, anche banalmente grammaticali, causati dalla prima modifica.

Edit Chain Una edit chain (catena di modifiche) è una serie di modifiche uguali o molto simili attivate spesso in maniera automatica, come un cerca e sostituisci globale o un cambiamento strutturale importante (una serie di paragrafi trasformati in lista).

Capitolo 2

Tecnologie Javascript per sistemi di editing Real-Time

Il mio obiettivo consiste nel realizzare in SSE, una piattaforma di editor di collaborativo asincrono di documenti strutturati, un algoritmo di Operational Tranformation per renderlo un sistema collaborativo Real-Time. Pertanto è opportuno definire questi sistemi e quali sono le tecnologie JavaScript principali che servono a realizzarli.
In base alle esigenze del contesto lavorativo più utenti possono avere la necessità di modificare uno stesso documento. Queste modifiche possono avvenire in maniera asincrona o sincrona.

2.1 Comunicazione sincrona e asincrona

Nei sistemi di editing asincroni gli utenti non hanno modo di collaborare in tempo reale, lavorando individualmente, anche in postazioni molto distanti tra loro e possono eseguire operazioni sul documento anche a discapito di modifiche apportate da un altro utente nello stesso momento.
Una soluzione possibile, adottata da SSE, è consentire le modifiche ad un

solo soggetto alla volta, bloccando agli altri l'accesso fino alla chiusura del testo o al superamento di una certa soglia di tempo. Questo sistema è più sicuro ma può causare rallentamenti nelle attività lavorative.
Nei sistemi di editing sincroni, detti anche sistemi Real-Time, nel momento in cui un utente effettua modifiche al documento, queste sono immediatamente propagate a tutti gli altri utenti connessi al server senza ritardi.
E' necessario, quindi, 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.
Va da sé che questi sistemi sono molto più potenti ma anche più difficili da implementare.

2.2 Tecnologie Javascript per il Real-Time

Javascript, attraverso l'utilizzo di chiamate AJAX (Asynchronous Javascript And XML, formato ormai sostituito da JSON), fornisce la possibilità di inviare e ricevere dati in tempo reale. 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.
Qui di seguito vengono presentate quattro tecnologie volte a risolvere questo problema:

- Ajax Polling
- XMLHttpRequest Long-Polling
- SSE (Server-Sent Events)
- Web Sockets

2.2.1 Ajax Polling

Una volta stabilita la connessione HTTP, il client continua a mandare richieste al server ad intervalli di tempo regolari (solitamente 0,5 secondi).

Il server calcola ogni risposta e la rimanda al client. Nel caso non ci sia nessun messaggio effettivo da inviare, non invia nulla [1].

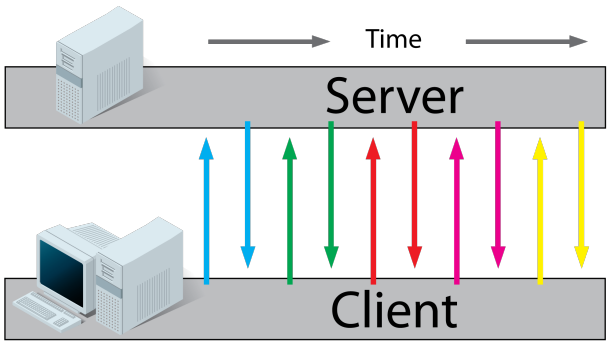


Figura 2.1: Ajax Polling

2.2.2 XMLHttpRequest Long-Polling

XMLHttpRequest ci fa subito capire che le chiamate vengono effettuate come se fossero delle banali chiamate Ajax. Il concetto di Long-Polling invece rappresenta la vera novità che rende possibile la gestione di eventi server-side.
Il client, appena avviata l'applicazione, effettua una chiamata verso il server. Quest'ultimo, dato che non ha ancora eventi da comunicare al client, mantiene aperta la connessione e la imposta in uno stato "pending" in attesa che succeda qualcosa.
Al verificarsi di un evento (per esempio un altro utente invia un messaggio) il server può rispondere immediatamente al client che aveva stabilito con lui una connessione precedentemente dato che possedeva già una sua richiesta disponibile.
Il client appena ricevuta la risposta, oltre a dare feedback all'utente, ricontatterà il server per eventuali altri eventi futuri.
Grazie a questa tecnica in qualsiasi momento esiste un canale diretto tra il client e il server da utilizzare per comunicare in maniera reattiva l'evento

occorso.
I browser moderni permettono di aprire almeno due richieste contemporanee. Ciò consente di avere un flusso dati reciproco. Una connessione verrà aperta e mantenuta in stato pending dal server, l'altra invece verrà utilizzata quando il client avrà realmente qualcosa da dire al server [2].

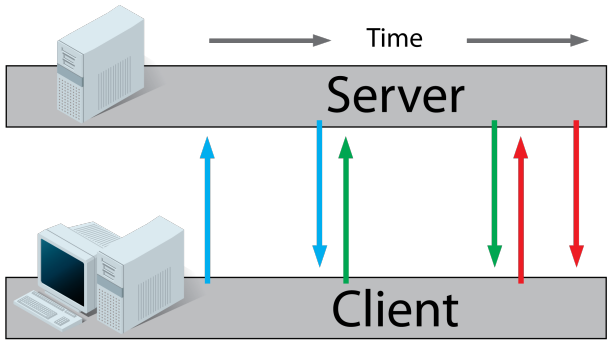


Figura 2.2: XMLHttpRequest Long-Polling

2.2.3 SSE (Server-Sent Events)

Con l'avvento dell'HTML5, e in particolare della libreria API Server-Sent Events, è possibile per un server, inviare dei dati a una pagina web "in qualsiasi momento", tramite dei messaggi push, senza che la pagina in questione ne faccia esplicita richiesta. I messaggi in arrivo possono essere trattati come una coppia <eventi, dati> all'interno della pagina web.
Grazie all'estrema semplicità di utilizzo risulta un'ottima alternativa ai classici meccanismi di polling sviluppati attorno alle tecniche AJAX. Nonostante il loro supporto non abbracci ancora l'intero insieme dei browser di ultima generazione esistono ottime librerie capaci di emulare in modo soddisfacente questa interessante feature [3] [4].

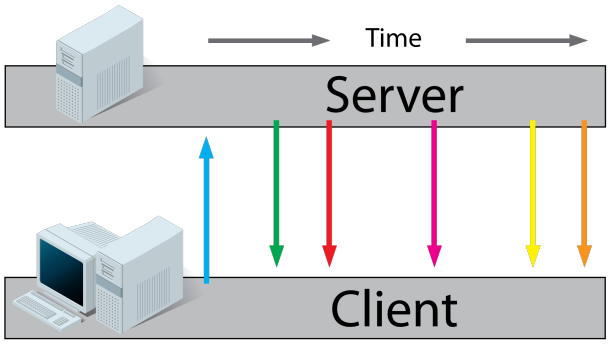


Figura 2.3: SSE (Server-Sent Events)

2.2.4 Web Sockets

WebSocket API, offerto oggi dai maggiori browser, rende non più necessario ricorrere alle chiamate AJAX per lo sviluppo e editor Real-Time.
Con le WebSocket è possibile aprire un canale di comunicazione bilaterale costantemente attivo. Ciò significa non solo che è possibile implementare il polling senza ricorrere ad AJAX, ma anche che meccanismi molto più efficaci ed apparentemente complessi (quali le notifiche push o la messaggistica istantanea) sono ora realizzabili con estrema facilità [1].

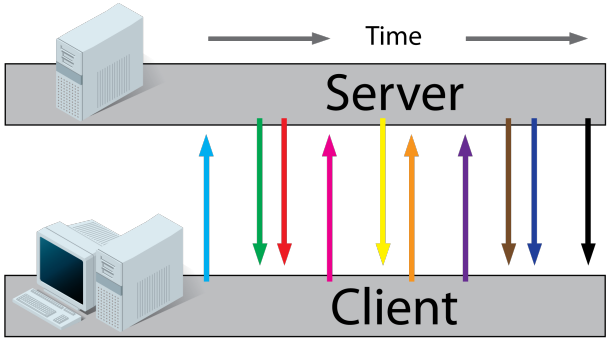


Figura 2.4: Web Sockets

Capitolo 3

Ambiente di sviluppo: SSE, TinyMCE e ADF

Prima di mostrare i dettagli del funzionamento del mio algoritmo è importante chiarire in che contesto informatico è stato inserito e quali sono state le tecnologie utilizzate e la piattaforma dove è avvenuta la sua realizzazione.
Il progetto nasce come upgrade della piattaforma web SSE (Smart Structured Editor), al fine di implementare un algoritmo di Operational Tranformation per gestire la modifica dei documenti presenti al suo interno con un sistema Real-Time.
SSE fa uso dell'editor di testo TinyMCE per la creazione, modifica e visualizzazione dei propri documenti.
Tutte le modifiche al testo dei documenti salvati all'interno di SSE vengono trasformate dall'algoritmo della piattaforma per renderle conformi a ADF, un linguaggio di Markup specifico per documentazione ALSTOM. Tutti i documenti presenti in SSE sono scritti in ADF.

3.1 Smart Structured Editor (SSE)

SSE (Smart Structured Editor) è una piattaforma web creata dagli sviluppatori del DASPLab del Dipartimento di Informatica dell'Università di Bologna per la creazione e gestione di documenti ADF.
SSE nasce come sistema di editing collaborativo asincrono che consente agli utenti di produrre facilmente contenuti strutturati "arricchiti" con alcuni metadati per facilitarne il tracciamento del documento e la sua visualizzazione; si pone anche l'obiettivo di agevolare la condivisione di frammenti di testo presenti nei suoi documenti tramite templating.
Gli utenti di SSE non hanno a che fare con gli aspetti della tipografia e della presentazione, ma devono solo concentrarsi sul contenuto del testo, mentre tutta la formattazione, controllo di qualità e generazione di tabelle e degli indici sono richiesti al sistema [15].

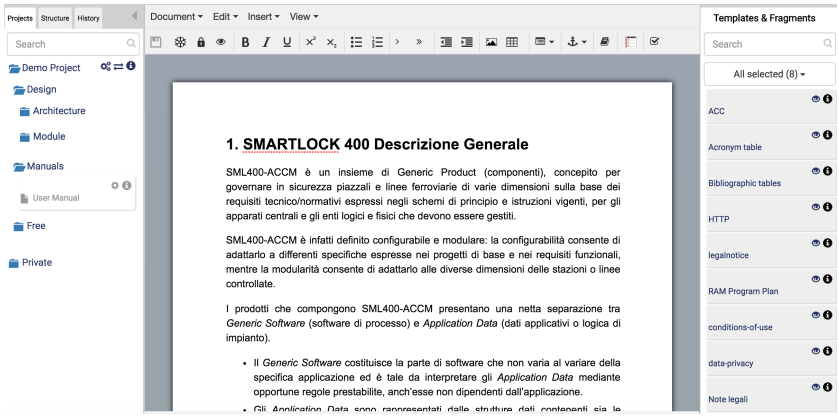


Figura 3.1: Un documento aperto in SSE

3.2 ADF

ADF è un vocabolario per i documenti ALSTOM generati dall'editor SSE. E' stato creato dal DASPLab del Dipartimento di Informatica del-

l'Università di Bologna per le esigenze del team di Ingegneri dell'azienda ALSTOM.
In ADF è definita la struttura generale del documento, nonché un formato limitato a elenco chiuso di strutture e funzionalità linguistiche supportate.
Fornisce inoltre le indicazioni su come esprimere questi elementi in HTML5 attraverso l'uso di un dialetto specifico chiamato RASH, sottoinsieme di 32 tag HTML usato soprattutto per documentazione scientifica, e l'uso di JSON-LD e RDF-a per la specifica di parti semanticamente rilevanti.
ADF è stato pensato con lo scopo di aiutare i disegnatori a scrivere rapidamente documenti tecnici di alto livello, nonchè facilitarne la valutazione mediante dei requisiti tecnici di qualità formale e strutturale.
Per raggiungere questi obiettivi, il vocabolario del markup deve implementare nella propria stuttura le seguenti caratteristiche:

- Deve avere un'architettura strutturale e semantica tale da rendere ogni elemento descrittivo di un aspetto del documento, secondo i pattern e le regole definite a priori dagli sviluppatori.
- La struttura deve essere riutilizzabile, frammenti di testo piccoli e grandi di testo devono poter essere facilmente identificati in documenti già archiviati, estratti e riutilizzati in quelli più recenti, in modo da generare una libreria completa di modelli pronti per l'uso e il riutilizzo.
- Ogni parte del documento, dai metadati ai singoli frammenti, è chiaramente e inequivocabilmente associato alla data e ora della sua creazione o modifica e all'utente che lo ha generato, con una storia completa di tutte le sue modifiche nel tempo.

Un documento ALSTOM è composto da tre parti: i metadati che forniscono informazioni relative al documento, al suo contesto e ai parametri predefiniti utilizzati; il frontespizio inclusa la prima pagina con le tabelle iniziali e il corpo del documento con il contenuto effettivo [15].

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" vocab="http://www.alstom.com/sse/"
prefix="doc: http://www.alstom.com/sse/ tpl: http://www.alstom.com/sse/templates/">
   <head>
      <link ... />
   <script ...> </script>
      ...
   <script type="application/ld+json" id="doc-metadata">
      <-- Metadata about the document -->
   </script>
   </head>
   <body>
      <section id="frontmatter" data-pdx-role="doc-frontmatter" class="doc-ro">
         <!-- Content of the front matter -->
      </section>
      <section id="content" data-pdx-role="doc-content">
         <-- Content of the document -->
      </section>
      <section id="endnotes" data-pdx-role="doc-endnotes">
         <!-- Footnotes -->
      </section>
   </body>
</html>
```

Figura 3.2: Un esempio di Struttura base di un documento ALSTOM

3.3 TinyMCE

TinyMCE è un rich-text editor, scritto in Javascript, creato per essere implementato nei siti web, si tratta di una piattaforma open source sotto LGPL (GNU Lesser General Public License), supportato da tutti i principali Browser. E' progettato per integrarsi facilmente con le librerie Javascript come React, Vue.js e AngularJS, nonché con i sistemi di gestione dei contenuti come Joomla! E WordPress.
La sua principale caratteristica consiste nel convertire campi textarea HTML o altri elementi HTML in istanze dell'editor.
I metodi e le strutture dati fornite da tinyMCE, presenti nella documentazione [16], sono stati utili alla realizzazione del mio progetto. Di seguito ne verranno descritti alcuni.

3.3.1 Undo Manager

L'Undo Manager consiste in un insieme di strutture e metodi messi a disposizione da TinyMCE per la gestione della cronologia delle modifiche. I dati dell'Undo Manager sono tutti memorizzati in un array di stati del documento, disposti in ordine cronologico all'interno di esso in base alle modifiche apportate dall'utente.
In tinyMCE.activeEditor.undoManager.data[0] verrà memorizzato l'intero testo originale del documento, in tinyMCE.activeEditor.undoManager.data[1] il testo risultante dopo la prima modifica e così via fino all'ultima cella che contiene il contenuto attuale visualizzabile anche da schermo.
Il contenuto di tinyMCE.activeEditor.undoManager.data è stato usato per ricavare le modifiche apportate da ogni singolo utente mediante un procedimento realizzato da me che verrà descritto nel capitolo successivo.
Attraverso il metodo add() è possibile aggiungere manualmente strati all'Undo Manager senza necessariamente apportare modifiche manuali al documento; si tratta di una funzione molto utile per memorizzare nel buffer le operazioni che sono state apportate dagli altri utenti del sistema di editing Real-Time.

3.3.2 Editor Events

Tramite editor.on("event") è possibile associare una funzione al verificarsi di un determinato evento. Si tratta di una funzionalità indispensabile per catturare tutte quelle operazioni che vanno a cambiare lo stato del documento; in particolare, gli eventi più utili alla nostra causa sono:

- Init: Viene attivato in seguito al caricamento del documento. Può essere sfruttato per inviare al server, attraverso chiamate AJAX, un insieme di dati in formato JSON contenente il nome dell'utente e il percorso assoluto in memoria documento che è stato aperto al fine di inserire queste informazioni nella struttura dati principale del server.

- Change: Si attiva quando lo stato dell'Undo Manager cambia, ovvero quando vengono apportate delle modifiche al testo dall'utente o al codice, oppure quando viene effettuata un'operazione di undo o redo.
  E' essenziale per poter ricavare le operazioni da inviare al server ogniqualvolta ne venga effettuata una.

- Undo: Si attiva quando viene eseguita un'operazione di Undo, va sfruttata in algoritmo OT per poter ricavare le operazioni inverse alle ultime bufferizzate.

- Redo: Si attiva quando viene eseguita un'operazione di Redo, va sfruttata in algoritmo OT per poter ricavare e ripristinare le ultime operazioni cancellate.

Capitolo 4

Un algoritmo di Operational Transformation in SSE

Come già accennato, l'obiettivo che mi sono posto è quello di realizzare un algoritmo di Operational Transformation nell'editor strutturato SSE.
In questa parte della mia dissertazione, andrò a presentare le scelte che ho intrapreso per cercare di rendere il mio programma a tutti gli effetti conforme ai dettami di OT spiegati nel capitolo 1.
Il linguaggio di programmazione scelto per la stesura di questo progetto è il javascript, mentre il lato server è implementato mediante Node.js.
La tecnologia Javascript utilizzata per creare il mio sistema di editing Real-Time è l'Ajax Polling (capitolo 1) che permette di stabilire una connessione tra server e client dove avviene lo scambio di strutture dati, usate per registrare le operazioni, in formato JSON.
Dopo aver descritto il funzionamento generale del mio algoritmo, dividerò questa presentazione in tre parti andando ad analizzare i tre problemi principali in cui mi sono imbattuto durante la stesura del codice:

- Come estrarre le modifiche apportate al testo nel modo più efficiente e funzionale possibile e in quale struttura dati inserirle.

- Come il server deve registrare e disporre le operazioni che riceve da ogni utente.

- Come manipolare le operazioni memorizzate in modo da poter inviare ad ogni client il giusto insieme di strutture JSON da elaborare per poter ottenere un documento convergente.

4.1 L'idea generale

Una volta effettuato il login a SSE e aver scelto di aprire un documento facente parte di un progetto pubblico, il sistema si attiva tramite l'evento document.on("init"), inviando al server un array contenente al suo interno [nome utente, documento], che ne memorizzerà le informazioni e restituirà al client lo stato attuale del documento.
Dopo aver sostituito il contenuto del testo con quello aggiornato, il client si mette in ascolto mandando una richiesta una volta ogni 0,5 secondi per ricevere eventuali modifiche apportate dagli altri utenti.
Ogni modifica apportata al documento viene intercettata e inserita in una struttura JSON e spedita al server. L'operazione viene rappresentata mediante questa struttura:

$$
\begin{aligned}
&\{\\
&\quad \text{"}id\text{"} : \text{"}NumberID\text{"},\\
&\quad \text{"}op\text{"} : \text{"}DEL/INS\text{"},\\
&\quad \text{"}doc\text{"} : theDocContent.currentDocHref,\\
&\quad \text{"}by\text{"} : Application.loggedinUser,\\
&\quad \text{"}timestamp\text{"} : time,\\
&\quad \text{"}pos\text{"} : opStart,\\
&\quad \text{"}leng\text{"} : opStart + opLeng,\\
&\quad \text{"}content\text{"} : document.substring(opStart, opLeng + 1),\\
&\}
\end{aligned}
$$

dove *op* è il tipo, *doc* è il percorso assoluto del file di testo, *by* è l'autore dell'operazione, *pos* è la posizione del primo carattere in cui inizia l'operazione, *leng* è la posizione dell'ultimo carattere che interessa l'operazione e *content* è il suo contenuto.
Il server registra ogni utente dopo la connessione e si mette all'ascolto aspettando di ricevere le operazioni; quando ne riceve più di una dalla stessa persona applica la funzione *Optimize* per alleggerire il sistema cercando di eliminare operazioni inutili.
Una volta ogni 5 secondi, il server manda ad ogni client connesso l'insieme di tutte le operazioni che serviranno a ricreare la stesso stato del documento per ognuno di loro; questo insieme di operazioni sarà diverso per ogni client e sarà l'output della funzione *Conflict*.
La funzione *Conflict* dà per scontato che le modifiche apportate dal proprio autore siano già presenti all'interno del documento e aggiunge quelle create dagli altri gestendo tutti i potenziali conflitti.
Ogni client una volta ricevuto l'aggiornamento dal server non dovrà far altro che applicare in sequenza tutte le operazioni per ottenere il documento aggiornato.

4.2 L'estrazione delle modifiche

L'attivarsi dell'evento Change si può manifestare per vari motivi: l'utente ha effettuato una modifica al testo, SSE ha cambiato la struttura del documento o l'algoritmo di Operational Transformation ha appena applicato le operazioni eseguite dagli altri utenti.
Al manifestarsi dell'ultimo caso non occorre fare nulla, in quanto non è necessario riinviare al server operazioni che sono già state processate. I primi due casi, invece, sono il segnale che è avvenuta un'operazione che necessita di essere estratta e inviata al server.
Attraverso l'uso dell'Undo Manager di TinyMCE è possibile trovare una

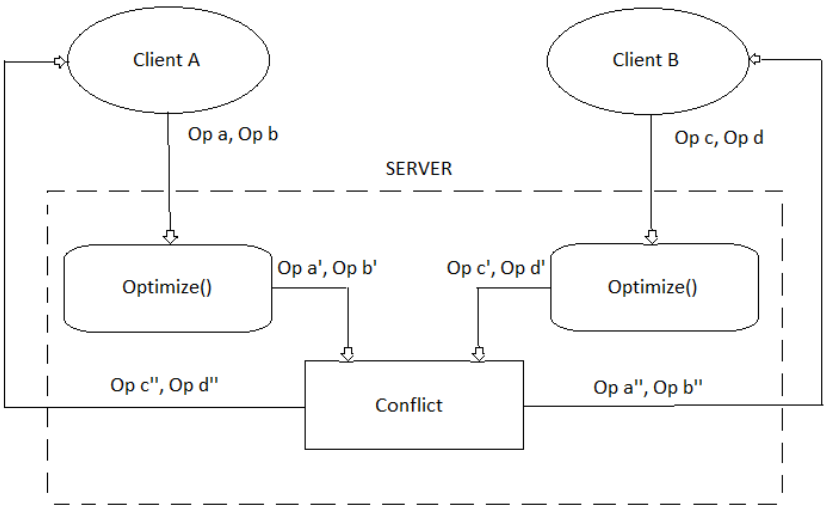


Figura 4.1: Schema funzionamento dell'algoritmo

stringa dove è salvato l'intero testo attualmente presente nel documento aperto dal client e una stringa contenente lo stato che aveva il documento prima dell'ultima modifica.
Lo stato attuale, che chiameremo now, è il valore di tinyMCE. activeEditor.undoManager.data[n], mentre lo stato appena precedente, che chiameremo past, si trova in tinyMCE.activeEditor.undoManager.data[n-1] dove n = tinyMCE.activeEditor.undoManager.data.length.
La funzione *diff* prende come input due stati del documento consecutivi e restituisce le operazioni risultanti. Essa scorre in contemporanea *now* e *past* partendo dalla loro prima posizione e individua, se presente, il primo carattere divergente tra i due; allo stesso modo fa la medesima operazione partendo dall'ultimo carattere trovando la prima differenza.
A questo punto si aprono diversi scenari:

- Se non sono state trovate lettere divergenti non invia nulla.

- Se è stata trovata una differenza e past è stato percorso interamente, allora viene inviata un'operazione di inserimento *(Ins)* che contiene la parte di now che non è stata percorsa.

- Se è stata trovata una differenza e now è stato percorso interamente, allora viene inviata un'operazione di cancellazione *(Del)* che contiene la parte di past che non è stata percorsa.

- Se è stata trovata una differenza e sia *now* che *past* non sono stati percorsi interamente, viene creata un'operazione *Del* che contiene la parte di *past* che non è stata percorsa e un'operazione *Ins* con lo stesso criterio considerando *now*.
  In questo specifico caso avviene un ulteriore controllo, se *Ins.content* è contenuto o uguale a *Del.content*, significa che è stata fatta un'operazione a livello strutturale del testo, quindi verranno inviate due operazioni *Ins1* e *Ins2*, dove *Ins1* contiene la parte iniziale di *Ins* non presente in *Del* e *Ins2* la parte finale di *Ins* non presente in *Del*.
  La stessa operazione viene effettuata specularmente nel caso opposto.

  Esempio:
  past = "....Università di Bologna..."
  now = ".... Università di Bologna ..."

  Viene applicato diff(past,now) che produce:

  Del.content = "Università di Bologna"
  Ins.content = " Università di Bologna "

  dato che $Del \subset Ins$ allora verranno inviate *Ins1* e *Ins2* dove:

  Ins1.content=" "
  Ins2.content = " "

4.3 La rielaborazione delle operazioni

Il server rimane in attesa dell'arrivo delle operazioni da parte degli utenti durante tutta la durata dell'esecuzione: ogniqualvolta ne riceve una, la inserisce in una struttura JSON, associata al nome dell'utente e al documento in cui è stata creata.
Ogni gruppo di modifiche al documento di ogni utente viene sottoposto ad un processo di ottimizzazione, mediante la funzione *optimize*, che ha lo scopo di diminuire il numero di operazioni, cancellando quelle inutili, e non memorizzare le operazioni di Del che vanno a cancellare elementi inseriti molto recentemente dal client.

Esempio
Se l'utente effettua due operazioni di inserimento e successivamente un'operazione di cancellazione come segue:

Ins1.content = "Ciao, come" alla posizione 1
Ins2.content = " stai Francesco" alla posizione 11
Del.conten = "come stai " alla posizione 5

Viene applicato *optimize(ins,del)* che produrrà due operazioni di cancellazione Ins1' e Ins2' ed eliminerà completamente Del.

$Ins1'.content =$ "$Ciao,$" alla posizione 1
$Ins2'.content =$ "$Francesco$" alla posizione 5

4.4 La gestione dei conflitti

Come accennato in precedenza, una volta ogni 5 secondi il sistema invia a tutti agli utenti tutte le operazioni per poter ricostruire la versione conver-

gente del documento.
Le modifiche apportate dai vari utenti possono entrare in conflitto tra loro a seconda di che tipologia sono. Diventa necessario gestire questi conflitti nella maniera più adeguata.
Considereremo caso per caso prendendo in esame una coppia di operazioni per volta, la prima eseguita dall'utente A e la seconda dall'utente B, tenendo presente che l'operazione di A è avvenuta per prima.

Insert - Insert

- Se $\boldsymbol{InsA.pos < InsB.pos}$ allora la posizione di *InsB* viene shiftata verso destra di un numero di caselle uguale alla lunghezza di $InsA.content$.

Insert, Delete

- Se $\boldsymbol{InsA.pos < DelB.pos}$ && $\boldsymbol{InsA.leng < DelB.pos}$, allora la posizione di *DelB* viene shiftata verso destra di un numero di caselle uguale alla lunghezza di $InsA.content$.

- Se $\boldsymbol{InsA.pos > DelB.pos}$ && $\boldsymbol{InsA.pos <= DelB.leng}$, allora *DelB* viene splittata in due operazioni differenti come segue:
  $DelA1.pos = DelA.pos$,
  $DelA1.leng = Insb.pos - 1$,
  $DelA2.pos = InsB.pos + 1$,
  $DelA2.leng = InsB.leng - DelA1.content.length$.

Delete - Insert

- Se $\boldsymbol{DelA.pos < InsB.pos}$, allora la posizione di *InsB* viene shiftata verso destra di un numero di caselle uguale alla lunghezza di $DelA.content$.

Delete - Delete

- Se ***DelA.leng < DelB.pos***, allora la posizione di *InsB* viene shiftata verso sinistra di un numero di caselle uguale alla lunghezza di *DelA.content*.

- Se ***DelA.pos < DelB.pos*** && ***DelA.leng > DelB.pos*** viene tolta a DelB la porzione di testo calcellata già da DelA ottenendo:
  DelB'.pos = DelA.pos,
  DelB'.leng = DelA.pos + (DelB.leng - DelA.leng)

- Se ***DelA.pos > DelB.pos*** && ***DelA.pos <= DelB.leng*** viene tolta a DelB la porzione di testo calcellata già da DelA ottenendo:
  DelB'.pos = DelB.pos,
  DelB'.leng = DelA.pos - 1

Capitolo 5

Valutazione

5.1 L'algoritmo è OT?

Dopo aver approfondito i principi base di Operational Transformation mi sono chiesto se fosse possibile realizzare un algoritmo OT all'interno di un editor strutturato come SSE.
Per raggiungere il mio scopo ho dovuto scegliere quale tecnologia Javascript utilizzare e la mia decisione è ricaduta su Ajax Polling, che garantisce la realizzazione di un sistema di editing real time in maniera facile e intuitiva, in quanto era necessario dare maggior enfasi agli aspetti più cruciali del sistema.
Il progetto che ho realizzato, descritto nel Capitolo 4, è sicuramente un algoritmo di Operational Transformation per i seguenti motivi:

- Ogni modifica al documento è vista come una singola operazione che viene estratta in tempo reale, rappresentata in una struttura JSON e passata al sistema.

- Le funzioni di trasformazione sono inclusive transformation (IT) functions, poichè includono il risultato dell'applicazione di OpB, contro cui si trasforma OpA.

- Le funzioni di trasformazione rispettano il principio di causalità, cioè l'ordinamento temporale delle operazioni, perchè si basano su priorità.
- Le funzioni di trasformazione rispettano la prima proprietà di trasformazione (TP1), poichè se due client effettuano due operazioni che entrano in conflitto, entrambe, dopo l'elaborazione delle informazioni del server, ottengono una copia del documento convergente.
- Le funzioni di trasformazione preservano la coerenza del documento finale per qualsiasi numero di utenti connessi in qualsiasi documento.
- Solo il server mantiene in memoria una copia del documento aggiuntiva per permettere agli utenti che si connettono ad esecuzione già iniziata di recuperare le operazioni che sono già state eseguite.

5.2 Lavorare in un editor strutturato

Lavorare in un editor strutturato ha comportato l'adozione di alcune misure aggiuntive rispetto ad un normale documento di testo.
Innanzitutto c'è stata una particolare attenzione nel distinguere le modifiche al markup rispetto alle modifiche testuali semplici. Ogniqualvolta viene eseguita una modifica, infatti, l'algoritmo controlla se si tratta solo di un'aggiunta o una cancellazione di tag e in tal caso divide l'operazione in due parti, tralasciando la parte testuale.
Anche la scelta di far attendere il server alcuni secondi prima di spedire le modifiche è dovuta a questo fattore. Infatti viene lasciato questo spazio di tempo non solo, come già detto, per cercare di alleggerire il carico del sistema eliminando le operazioni inutili, ma anche per dar tempo eventualmente al sistema di combinare le operazioni che riceve, per dar loro un significato semanticamente più rilevante, rendendole operazioni di tipo strutturale o semantico.

5.3 Possibili miglioramenti

Questo algoritmo di Operational Transformation non è da considerarsi come un prodotto finale, ma come una base facente parte di un progetto molto più grande. Rimangono infatti alcuni problemi aperti.

- L'estrazione delle operazioni si serve di due stringhe contenenti due versioni consecutive del testo per estrarre le operazioni. Si tratta di una soluzione poco efficiente per quanto riguarda le prestazioni e porta anche a delle situazioni in cui le operazioni estratte, pur garantendo la convergenza del documento finale, non assicurano che la conservazione delle intenzioni dell'utente che ha effettuato la modifica al testo venga mantenuta.
  Questa situazione si verifica, ad esempio, quando un utente inserisce in una determinata posizione una stringa in cui il primo carattere è uguale al carattere che si trovava nella stessa posizione prima dell'inserimento. In questo caso l'algoritmo, andando a cercare la prima differenza tra now e past, non riconoscerà mai la prima lettera come parte della modifica, ma inizierà dalla lettera successiva includendo poi una lettera che già si trovava nel testo nella nuova operazione.

  Esempio:
  Se l'attuale testo del documento comprende solo la parola "di Informatica", e l'utente aggiunge la parola "dipartimento", il sistema intercetterà la modifica di tipo ins come alla posizione 3 contente "partimento di".

  Questo errore non andrà ad inficiare in nessun modo la convergenza del documento finale, ma non rappresenta pienamentele le intenzioni che aveva l'utente quando è andato ad effettuare l'operazione.
  Per ovviare a questo problema avrei dovuto intercettare le modifiche

utilizzando gli eventi keyup, keydown e keypress di tinyMCE, che reagiscono alla spinta di un pulsante della tastiera, ma questa scelta avrebbe completamente ignorato tutte le modifiche al markup che SSE apporta al testo durante l'esecuzione.
Per poterle impiegare è necessaria una profondissima conoscenza di SSE.

- Le prestazioni del sistema potrebbero migliorare impiegando una tecnologia javascript più performante, come ad esempio Web Soket.

- Le operazioni presenti nel sistema sono attualmente solo *Ins* e *Del*, sarebbe possibile implementare delle sovrastrutture di operazioni al livello strutturale e semantico.

Conclusioni

Questa tesi nasce dalla volontà di creare un algoritmo di Operational Transformation nell'editor strutturato SSE.
Per poter realizzare il mio proposito è stato necessario analizzare approfonditamente le fondamenta teoriche di questo protocollo di gestione della concorrenza e della coerenza nei sistemi di collaborazione real-time.
Sono partito analizzandone le principali caratteristiche, mostrando i modelli di coerenza e le proprietà delle funzioni di trasformazione, andando poi ad applicarle in un caso complesso attraverso il problema del diamante.
Chiarita la motivazione della scelta di questo approccio rispetto ad altre alternative come Conflictfree Replicated Data Types e Differential Synchronization, sono andato a vedere come è possible approcciarsi teoricamente ad Operational Transformation su un editor strutturato.
Successivamente sono state mostrate alcune tecnologie che Javascript mette a disposizione per la realizzazione di un sistema di editing collaborativo Real-Time, elencandone caratteristiche, pregi e difetti.
Avendo scelto SSE come piattaforma su cui lavorare, ho deciso di descriverne le caratteristiche facendo anche una panoramica su tutto il contesto informatico di cui fa parte, dalla documentazione ALSTOM a TinyMCE.
Sono passato poi a descrivere il mio progetto esponendone il funzionamento generale e ponendo l'accento sui punti cruciali chiarendone le scelte di implementazione.
Infine sono andato a valutare il mio lavoro, analizzandone pregi e difetti e

suggerendo alcune idee per migliorarlo, arrivando alla conclusione che l'algoritmo da me creato racchiude in sé tutte le linee guida e le caratteristiche del protocollo di Operational Transformation.

Bibliografia

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

[2] Alberto Bottarini. Il modello di comunicazione long-polling https://www.html.it/pag/33600/il-modello-di-comunicazione-long-polling/

[3] Sandro Paganotti. HTML5 Server-Sent Events https://www.html.it/articoli/html5-server-sent-events/

[4] Gianluca Tramontana. Server-Sent Events in HTML5 http://www.gianlucatramontana.it/blog/2013/12/server-sent-events-in-html5/

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

[6] 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 Trans". Comput.-Hum. Interact., 5(1):63-108, March 1998.

[7] 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.

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

[9] X. Jia, C. Sun, D. Chen. "Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems". Proceedings of The 21st Australasian Computer Science Conference, 1998.

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

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

[12] M. Shapiro, N. Pregui¸ca, C. Baquero and M. Zawirski, "Conflict-free Replicated Data Types". In Xavier Defago, 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.

[13] D. Montanari. "Operational Transformation su documenti strutturati" Tesi Unibo Laurea in Informatica 2017.

[14] A. Di Iorio, G. Spinaci, and F. Vitali. 2019. Multi-layered edits for meaningful interpretation of textual differences. In Proceedings of the ACM Symposium on Document Engineering 2019 (DocEng '19). ACM, New York, NY, USA, Article 22, 4 pages. DOI: https://doi.org/10.1145/3342558.33454069

[15] A. Caponi, A. Di Iorio, F. Vitali, P. Alberti and M. Scatà. Exploiting patterns and templates for technical documentation. . In Proceedings of the ACM Symposium on Document Engineering 2018 (DocEng '18). ACM New York, NY, USA, Article 30. DOI: `https://dl.acm.org/citation.cfm?id=3209537`

[16] Documentazione di TinyMCE: `https://www.tiny.cloud/docs/`