

UNIVERZA V LJUBLJANI  
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Marko Bregant

# **Primerjava algoritmov za usklajevanje besedil pri skupinskem urejanju**

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE  
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: izr. prof. dr. Zoran Bosnić

Ljubljana, 2014



Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljane ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.



Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

S porastom interneta je porasel tudi nabor spletnih aplikacij, ki omogočajo hkratno skupinsko delo več uporabnikov. Primer takšne aplikacije so urejevalniki besedil v realnem času. Kandidat naj v diplomski nalogi opiše in primerja najbolj pogoste algoritme, ki se uporabljajo za usklajevanje (sinhronizacijo) besedil pri skupinskem urejanju besedil v stvarnem času (kot ga npr. ponujata Google Documents in Microsoft OneDrive). Od primerjanih pristopov naj po lastni presoji izbere najbolj primernega in ga implementira v obliki skupinskega spletnega urejevalnika.



## IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Marko Bregant, z vpisno številko **63080011**, sem avtor diplomskega dela z naslovom:

*Primerjava algoritmov za usklajevanje besedil pri skupinskem urejanju*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Zorana Bosnića,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 23. aprila 2014.

Podpis avtorja:





*Za vso pomoč in nasvete pri izdelavi diplomske naloge se zahvaljujem mentorju, izr. prof. dr. Zoranu Bosniću. Zahvala gre tudi vsem sošolcem in prijateljem, ki so me tekom študija spodbujali. Še posebej pa bi se za podporo in bodrenje zahvalil družinskim članom in dragi Tini.*



# Kazalo

Seznam uporabljenih kratic

Povzetek

Abstract

<b>1</b>	<b>Uvod</b>	<b>1</b>
<b>2</b>	<b>Sodelovanje v realnem času</b>	<b>3</b>
2.1	Sočasnost in konsistentnost . . . . .	4
<b>3</b>	<b>Protokoli in algoritmi</b>	<b>7</b>
3.1	Diferenčna sinhronizacija . . . . .	7
3.1.1	Osnovna topologija . . . . .	8
3.1.2	Metoda dveh senc . . . . .	9
3.1.3	Metoda zagotovljene dostave . . . . .	11
3.2	Operativna transformacija . . . . .	16
3.2.1	Osnovno delovanje . . . . .	16
3.2.2	Protokol za sodelovanje . . . . .	19
3.3	Brez operativne transformacije . . . . .	26
3.3.1	Podatkovni model . . . . .	26
3.3.2	Operacije . . . . .	27
3.3.3	Odjemalec-odjemalec sodelovanje . . . . .	29
<b>4</b>	<b>Primerjava protokolov za skupinsko urejanje</b>	<b>33</b>
4.1	Hranjenje dokumenta . . . . .	33
4.2	Struktura porazdelitve . . . . .	35
4.3	Počasna povezava . . . . .	36
4.4	Zaznavanje sprememb . . . . .	37

4.5	Problem sočasnosti . . . . .	39
4.6	Zahtevnost protokola . . . . .	41
<b>5</b>	<b>Iskanje razlik</b>	<b>43</b>
5.1	Najdaljše skupno zaporedje . . . . .	44
5.2	Najmanjša razdalja urejanja . . . . .	47
5.3	Scenarij najkrajšega urejanja . . . . .	50
<b>6</b>	<b>Implementacija urejevalnika</b>	<b>57</b>
6.1	Zasnova in prijava . . . . .	57
6.2	Iskanje razlik . . . . .	58
6.3	Operativna transformacija . . . . .	59
6.4	Uporabniški vmesnik aplikacije . . . . .	66
<b>7</b>	<b>Sklepne ugotovitve</b>	<b>69</b>
	<b>Literatura</b>	<b>70</b>

# Seznam uporabljenih kratic

**DS** (Differential Synchronization)

Diferenčna sinhronizacija, več v Poglavju 3.1.

**OT** (Operational Transformation)

Operativna transformacija, več v Poglavju 3.2.

**WOOT** (WithOut Operational Transformation)

Brez operativne transformacije, več v Poglavju 3.3.

**AJAX** (Asynchronous JavaScript and XML)

Asinhroni JavaScript in XML.

**API** (Application Programming Interface)

Vmesnik za programiranje aplikacij.

**JS** (JavaScript)

Programski jezik, ki omogoča izdelavo in prikaz dinamičnih spletnih strani. Z nastankom platforme **node.js** se ga vedno več uporablja tudi na strežniškem delu.

**XML** (Extensible Markup Language)

Razširljiv označevalni jezik. Njegovo popularnost izpodriva JSON.

**JSON** (JavaScript Object Notation)

Oblika zapisa podatkov, ki se večinoma uporablja za pošiljanje podatkov med odjemalcem in strežnikom.

**SIGCE** (The Special Interest Group on Collaborative Computing)

Skupina, ki promovira raziskovalce na področju skupinskega urejanja.

**CE** (Collaborative Editing)

Sodelovalno urejanje ali skupinsko urejanje.



# Povzetek

Le malokdo si je v začetku razvoja interneta predstavljal, da bomo ljudje na oddaljenih lokacijah komunicirali takojšnje. Danes obstaja na spletu mnogo različnih orodij, ki omogočajo sodelovanje uporabnikov. Primer je urejanje golega besedila v realnem času. V diplomski nalogi smo analizirali in primerjali tri algoritme, na katerih temeljijo omenjena orodja, to so: Diferenčna sinhronizacija, Operativna transformacija in pristop Brez operativne transformacije. Pri primerjavi njihovih lastnosti se je Operativna transformacija izkazala za najboljšo. V praktičnem delu diplomske naloge smo tako implementirali enostaven spletni urejevalnik na osnovi Operativne transformacije. Urejevalnik je močno odvisen od majhnih sprememb, ki jih naredijo uporabniki in se razpošljejo drugim uporabnikom. Zaradi tega smo morali implementirati tudi algoritem za iskanje razlik v besedilu. Končni produkt je dosegljiv na <http://diploma.marek.si/>.

**Ključne besede:** sodelovanje v realnem času, sočasnost, iskanje razlik v besedilu, Operativna transformacija





# Abstract

Hardly anyone in the early development of the internet imagined that people on remote locations will communicate instantly. Today there are various online tools that provide users collaboration. An example is editing plain text in real time. In this thesis we analyzed and compared three algorithms underlying these tools: Differential Synchronization, Operational Transformation and WithOut Operational Transformation. After comparing the characteristics the Operational Transformation has proved to be one of the most efficient. In the practical part of the thesis we implemented a simple web editor based on the Operational Transformation. Editor considers the small changes made by a user and sent to other users. Because of this reason we have needed to implement the algorithm used to find differences in the text. The final product is available on <http://diploma.marek.si/>.

**Keywords:** collaboration in the real time, concurrency, finding differences in text, Operational Transformation



# Poglavje 1

## Uvod

Še ne dolgo nazaj se nam je zdel internet počasen. Vse skupaj je delovalo zelo statično. Lahko bi rekli, da smo dve desetletji nazaj splet uporabljali le za brskanje. Nato so se počasi pojavile spletne strani, ki so omogočale nekaj malega interaktivnosti. To so bile funkcionalnosti, kot so vpisovanje komentarjev, iskalniki, spletne galerije, forumi, ... Danes se nam zdijo te funkcionalnosti že skoraj integrirane v spletnih aplikacijah. Lahko se ozremo nazaj in rečemo, da so bili to prvi zametki, ki so uporabnikom omogočili, da ustvarjajo splet, kot ga poznamo danes. Z nadgrajevanjem internetne infrastrukture so se povečale hitrosti prenosa podatkov. Z razvojem spletnih tehnologij se je izboljšala celotna uporabniška izkušnja. V zadnjih dvajsetih letih smo bili priča napredku tako na programski kot tudi na strojni opremi. Na tem mestu se je smiselno vprašati, kaj je bilo tisto bistveno, ki je celotno zadevo izboljšalo, in kako se bo izboljševala v prihodnosti. Komunikacija med odjemalcem (angl. *client*) na eni strani in strežnikom (angl. *server*) na drugi strani še vedno deluje po istem principu kot dvajset let nazaj. Odjemalec vzpostavi komunikacijo s strežnikom. Od njega zahteva neko akcijo in slednji jo izvrši. Razlika je v mediju po katerem poteka komunikacija. Po optičnih vlaknih je le-ta omejena s svetlobno hitrostjo. Govorimo o prenosu podatkov v realnem času.

V okviru diplomske naloge bi radi preučili algoritme in raziskali celoten sistem, ki na spletu omogoča urejanje golega besedila v realnem času. Tega se bomo lotili tako, da bomo pregledali raziskave, ki so bile objavljene v akademski sferi. Leta 1989 so se pojavili prvi zapisi [1] o zagotavljanju vmesnika za deljno okolje (angl. *shared environment*). V naslednjih nekaj letih so bile na tej podlagi predlagane nekatere izboljšave. Kasneje je pri raziskavah precej pripomogla tudi ustanovitev SIGCE [3], ki promovira raziskovalce na tem področju. Skratka, preleteli bomo nekaj raziskav s tega področja. Po pridobljenem teoretičnem znanju bomo poiskali članke, ki so bili

napisani s strani industrije. Radi bi izvedeli, kaj od teorije se lahko uporabi v praksi. Nekateri algoritmi, ki so bili na akademskem področju uspešni, so se implementirali v končnih ali v pol produktih. Največ uporabnih informacij bomo dobili od protokola Google Wave [4]. Ne smemo pa pozabiti, da obstaja mnogo drugih uporabnih orodij, od katerih se lahko naučimo praktičnega dela. Za zgled lahko vzamemo `share.js` [6]. Cilj diplomske naloge je, da zasnujemo enega izmed algoritmov na platformi **node.js** [7]. Ker hočemo doseči, da je algoritem na strežniškem delu kar se da neodvisen od odjemalcev, mora delovati kot API. Kot smo omenili, je uporabnost take izvedbe ravno v tem, da je neodvisen od odjemalcev, ki se nanj povezujejo, naj si bo spletna aplikacija ali mobilna naprava. Poleg tega tak način izvedbe spodbuja nadaljnji razvoj celotnega sistema na različnih odjemalcih.

Po uvodu bomo v drugem poglavju najprej predstavili problem in izzive sodelovanja v realnem času. V tretjem poglavju bomo preučili protokole in algoritme za sodelovanje v realnem času. V četrtem poglavju jih bomo primerjali po njihovih lastnostih. V petem poglavju bomo poiskali algoritem za iskanje razlik v besedilu. V šestem poglavju bomo opisali našo implementacijo zasnove urejevalnikov v realnem času. Na koncu, v sedmem poglavju, sledijo še sklepne ugotovitve.

## Poglavje 2

# Sodelovanje v realnem času

Leta 2005 se je začela uveljavljati tehnologija AJAX. Njen glavni namen je, da odjemalcu omogoča pošiljanje asinhronih zahtev na strežnik. V praksi je bila razlika vidna v osveževanju spletnih strani. V preteklosti je brskalnik osvežil celotno spletno stran za vsako zahtevo, ki jo je naredil na strežnik. Uporaba AJAX-a pa omogoča, da so zahteve na strežnik manjše, bolj dinamične in najpomembnejše, asinhrone. Namesto celotne strani lahko osvežimo le manjši del. Od strežnika lahko zahtevamo ali mu pošljemo le nekaj podatkov. Kako pogosto delamo asinhrone zahtevke, ni pomembno - naj si bo za vsako uporabnikovo interakcijo ali ponavljajoče z metodo pozivanja (angl. *polling*) [10] ali dolgega pozivanja (angl. *long polling*).

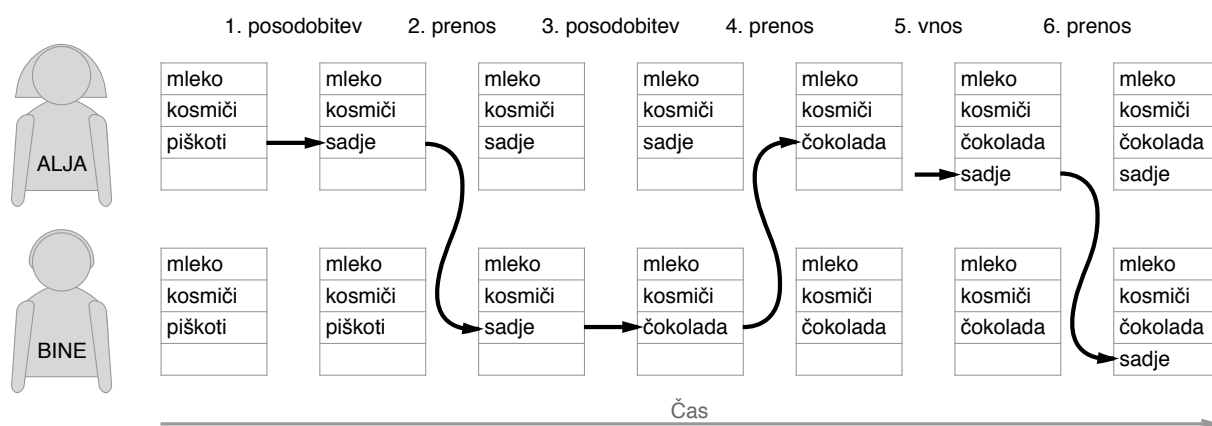
Naslednja pomembna stvar na spletu so spletni vtiči (angl. *WebSockets*), ki so trenutno še v povojih. Danes naj bi jih podpirali že vsi najnovejši brskalniki, vendar jih uporabljajo le redke spletne aplikacije. Sprememba, ki jih prinašajo spletni vtiči, je način komunikacije med odjemalcem in strežnikom. Omogočajo dvosmerno komunikacijo [5]. Po novem lahko tudi strežnik pošilja zahtevo odjemalcu.

Ti dve tehnologiji omenjamo zato, ker sta in bosta največ prispevali k izboljšanju uporabniške izkušnje na spletu. Interakcija med odjemalcem in strežnikom je postala bolj tekoča, kot je bila v preteklosti. Z njo nam je bila dana možnost za razvoj orodij za sodelovanje v realnem času (angl. *real-time collaboration tools*). Obstaja že mnogo orodij, ki preko sodelovanja (angl. *collaboration*) rešujejo nek specifičen problem. Na podoben problem smo naleteli tudi sami, in sicer urejanje besedila v realnem času. Z urejanjem besedila nimamo v mislih označevanje besedila s krepko, ležečo, podčrtano pisavo in tako naprej, ampak za urejanje golega besedila kot takega. Uporabnik lahko doda ali pobriše črko, se pravi operira z manjšimi enotami (črke, besede), ki se združujejo v večje enote (stavki, povedi, sporočilo, besedilo, dokument). Sistem mora skrbeti za izmenjevanje nastalega besedila med udeleženci [12]. Na tak način delujejo spletna kle-

petalnica. Udeleženci v pogovoru si izmenjuje sporočila, naj bodo to večja ali manjše enote. Sistem mora le skrbeti, da se le-te pravilno prenesejo med udeležence pogovora. Vendar zadeva ni niti približno tako enostavna. Bistvena razlika urejevalnika v realnem času v primerjavi s spletno klepetalnico je v obliki hranjenja in operiranja nastalih podatkov. Pri spletnih klepetalnicah se vsako posamezno sporočilo na strežniku hrani kot samostojno enoto, ki se jo v celoti razpošlje med udeležence. Pri urejevalnikih besedila v realnem času pa je besedilo, ki ga udeleženci urejajo, enotno za vse udeležence hkrati. Zaradi razumljivosti bomo v nadaljevanju besedilo, ki ga urejajo uporabniki, poimenovali dokument. Lahko bi rekli, da vsak udeleženec ureja svoj lokalni dokument, preko katerega nastaja skupni dokument. Posamezne manjše enote besedila, ki se izmenjujejo med udeleženci, so le koščki celotnega dokumenta. Pred tako nastalim dokumentom mora delovati algoritem, ki zna te manjša enote besedila združevati v dokument [11].

## 2.1 Sočasnost in konsistentnost

Da bo problem razumljiv tudi najmanj večjemu poznavalcu tehnologij, bomo problem predstavili na konkretnem primeru [8] med dvema udeležencema (angl. *participants*) oziroma uporabnikoma. Predstavljajmo si uporabnika Aljo in Bineta, ki za nakupovanje pripravljata skupni nakupovalni listek, kot je prikazano na Sliki 2.1.

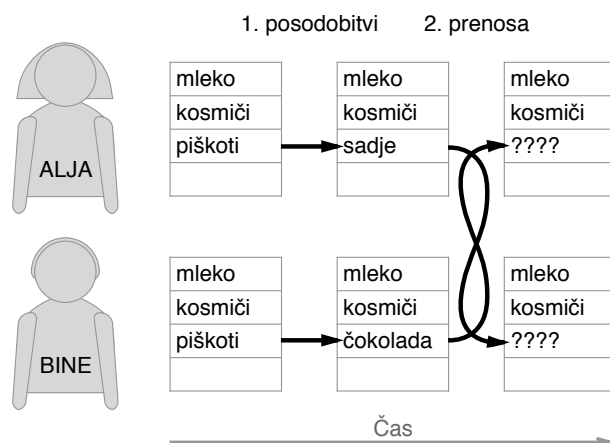


Slika 2.1: Diagram sočasnega urejanja nakupovalnega listka, pri čemer mora Alja dvakrat vpisati svojo željo po sadju. Njena prva sprememba ni dosegla pravilnega učinka ali povedano drugače, namen prve spremembe (angl. *intention preservation*) ni bil ohranjen [3].

Trenutno imata na seznamu mleko, kosmiče in piškote. To so stvari, ki jih običajano kupita vsako soboto. Alja se odloči, da bo ta vikend namesto piškotov kupila sadje,

zato uredi seznam tako, da piškote zamenja s sadjem. Aplikacija spremembe pošlje tudi Binetu, ki nato vidi piškote zamenjane s sadjem. Bine se odloči, da bi tokrat raje kupil čokolado, zato temu primerno spremeni seznam. Aplikacija pošlje Binetove spremembe Alji, ki sedaj vidi sadje, zamenjano s čokolado. Nezadovoljna Alja se z Binetom dogovori za kompromis in doda sadje na seznam, tako da Binetova čokolada še vedno ostane na seznamu. Alja kot Bine imata sedaj na seznamu mleko, kosmiče, čokolado in sadje, kot je prikazano na koncu Slike 2.1.

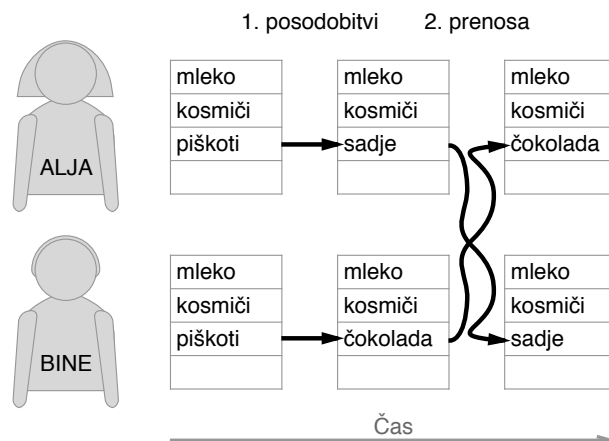
Pri tako fleksibilni interakciji lahko nastane konflikt. Alja bi lahko spremenila piškote v sadje sočasno, kot bi Bine spremenil piškote v čokolado. V primeru optimističnega nazdora sočasnosti (angl. *optimistic concurrency control*) bi oba svoje spremembe videla takoj [2]. Vendar Aljine spremembe potrebujejo nekaj časa, da pridejo do Bineta. Tudi Binetove spremembe potrebujejo nekaj časa, da pridejo do Alje. Ta zamuda lahko spremeni vrstni red Aljine in Binetove spremembe, kar povzroči nakonsistentnost, kot je to prikazano na Sliki 2.2.



Slika 2.2: Diagram sočasnega urejanja nakupovalnega listka. Vprašaji v zadnjem koraku pomenijo, da je seznam v konfliktu in mora biti razrešen.

Predstavljajmo si enostavno rešitev za posodabljanja nakupovalnega listka, ki Alji in Binetu vedno pokaže zadnjo spremembo, ki je bila narejena na seznamu. Alja bi piškote spremenila v sadje, nato pa bi prišla Binetova sprememba v čokolado. Na drugi strani je Bine piškote spremenil v čokolado, nato pa bi sprejel Aljino sadje. V tem primeru bi imela Alja in Bine na koncu dva različna seznama, česar sploh ne bi opazila. Primer slabega reševanja konfliktov je prikazan na Sliki 2.3.

Osnovna ideja vzdrževanja konsistentnosti je v konvergenци (angl. *convergence*), ki zagotavlja, da so replicirane kopije dokumentov identične na vseh lokacijah, ko so v mirovanju [9].



Slika 2.3: Diagram sočasnega urejanja nakupovalnega listka. Konflikt v zadnjem koraku je rešen na način, da Alja in Bine sprejmeta zadnjo narejeno spremembo. Rešitev je slaba, saj ne zagotavlja konsistentnosti.

Med drugim sta značilnosti sistemov za skupinsko delo v realnem času [1] tudi nestanovitnost (angl. *volatile*) in visoka stopnja interaktivnosti (angl. *highly interactive*). Prvo pomeni, da se lahko uporabnik urejanju kadarkoli pridruži ali ga zapusti. Druga značilnost pa narekuje kratke odzivne čase. Še večji problem tako nastane pri sodelovanju večih uporabnikov in s še večjo zamudo pri dostavi sprememb na nakupovalnem listku. Recimo, da se Alji in Binetu pridruži še Cene. Bine ima počasen internet. Alja in Cene na seznam dodata in odstranita deset novih artiklov, še preden Bine dobi eno spremembo. Medtem ko Bine ureja svoj seznam, so spremembe Alje in Ceneta še na poti k njemu. Za zagotovitev konsistentnosti bi morala aplikacija upoštevati zakasnjene oddaljene spremembe na osnovi prvotne verzije nakupovalnega listka.



# Poglavje 3

## Protokoli in algoritmi

Kako zagotoviti konsistentnost med oddaljenimi uporabniki pri sočasnem urejanju, je eden izmed glavnih izzivov naše diplomske naloge. V tem poglavju bomo raziskali protokole, ki omogočajo sodelovanje uporabnikov v realnem času in teoretično rešujejo omenjena problema. Najbolj razširjeni so Diferenčna sinhronizacija (angl. *Differential Synchronization*), Operativna transformacija (angl. *Operational Transformation*) in protokol Brez operativne transformacije (angl. *WithOut Operational Transformation*), bolj znan pod kratico WOOT.

### 3.1 Diferenčna sinhronizacija

Diferenčna sinhronizacija (v nadaljevanju DS) je metoda, s katero ohranjamo dokumente sinhronizirane. Z njo se je ukvarjal Neil Fraser [10]. Kot opisuje, so konceptualno enostavni načini sinhronizacije zaklepanje (angl. *locking*), prenašanje dogodkov (angl. *event passing*) in trosmerno združevanje (angl. *three way merge*).

**Zaklepanje** je najenostavnejši način. Ko uporabnik odpre skupni dokument, se mu dodeli pravica lastnika (angl. *ownership*). Le on ga lahko ureja. Vsi ostali uporabniki lahko v tistem času dokument gledajo (angl. *read-only access*). Cilj je delno dosežen. Vsi uporabniki imajo sinhroniziran dokument, vendar se njegovo urejanje ne izvaja v realnem času. Izboljšava zaklepanja je delno zaklepenje, pri katerem se uporabniku dodeli pravica lastnika le za majhen del dokumenta. Ta način ni najboljši, kadar imajo uporabniki slabo povezavo. Paket z informacijo o zaklepanju in odklepanju se lahko izgubi. V dokumentu lahko nastanejo deli, nad katerimi ima lahko več uporabnikov pravico lastnika. Pojavijo se tudi druge anomalije.

**Prenašanje dogodkov** temelji na zaznavanju vseh interakcij, ki jih naredi uporabnik z urejevalnikom, ter njihovo pošiljanje drugim uporabnikom. V teoriji je to enostavno, saj vsak sistem omogoča zaznavanje tipkanja. V praksi je težko izvedljivo. Poleg tipkanja lahko uporabnik naredi tudi operacije, kot so: izreži besedilo, prilepi besedilo, povleci in spusti besedilo, zamenjava besedila, ... Operaciji, kot sta samodokončanje in pravopisni popravek, lahko naredi tudi sam urejevalnik. Kaj narediti v teh primerih? Težava je tudi v tem, da lahko pride zaradi neke napake ali zamude v komunikaciji do dveh popolnoma različnih dokumenetov, ki jih ne moremo poenotiti. Takega način sinhronizacije ne moremo implementirati brez algoritma, ki bi reševal težavo pri zamudi prenašanja dogodkov.

**Trosmerno združevanje** uporabljajo programerji pri delu na skupnem projektu. Je zelo robusten sistem, sestavljen iz treh korakov. Uporabnik najprej svojo vsebino pošlje na strežnik, ki izvzame narejene spremembe in jih združi s spremembami drugih uporabnikov. Nova kopija dokumenta se sinhronizira vsem uporabnikom. Sistem trosmernega združevanja ima nekaj slabosti. Če naredi uporabnik spremembo na dokumentu, medtem ko je sinhronizacija v teku, mora zavreči vse na novo narejene spremembe. Slabost je, da uporabnik ne dobi nobene povratne informacije (angl. *feedback*) med tipkanjem. Način trosmernega združevanja bi pri urejanju v realnem času deloval le v dveh primerih. Če bi si lahko privoščili, da bi uporabnika blokirali in sinhronizirali z novo verzijo na strežniku ali pa v primeru, da bi uporabnika sinhronizirali, ko neha tipkati. Ampak nobena od teh dveh variant ni urejanje dokumenta v realnem času.

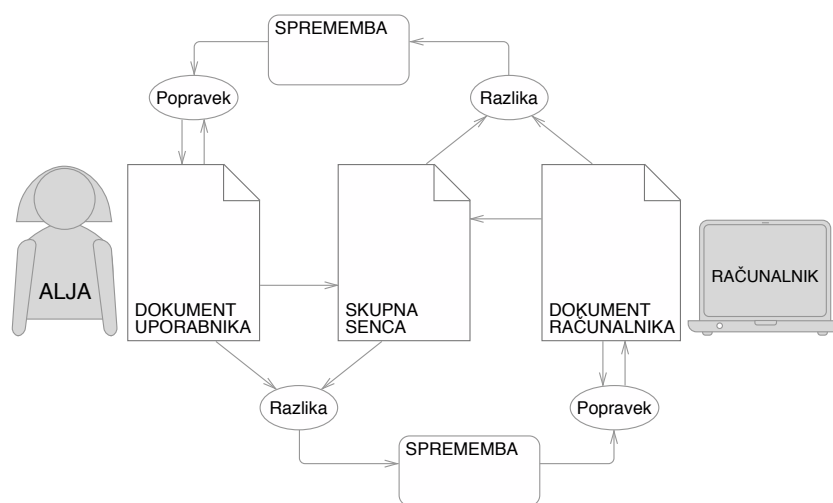
Ker nobena izmed naštetih rešitev sinhronizacije ni zadovoljiva, si oglejmo DS bolj podrobno. DS je simetričen algoritem, ki uporablja neskončno ciklov razlik na dokumentih in z njimi popravlja ostale dokumente v ciklu. Najprej bomo preučili osnovno topologijo DS, ki je teoretična podlaga za izboljšave z metodo dveh senc in z metodo zagotovljene dostave.

### 3.1.1 Osnovna topologija

Na Sliki 3.1 je z diagramom prikazana DS. V osnovni topologiji obstajata dokument uporabnika in dokument računalnika. Oba sta locirana na istem sistemu brez internetne povezave. Med njima se nahaja dokument, imenovan skupna senca (angl. *common shadow*). Predvidimo, da imajo na začetku vsi trije dokumenti isto vsebino. Cilj je, da so dokumenti vseskozi čim bolj enaki. Ostale entitete so še razlika, sprememba in

popravek. Razlika je signal, ki pove, da se dokument uporabnika razlikuje od skupne sence. Na ta način vemo, da je uporabnik naredil spremembo na svojem dokumentu. Ni nam potrebno spremljati vsake uporabnikove operacije. Ko se ve, kakšne spremembe so bile narejene, se mora dokument prekopirati v senco. Sprememba se pošlje v smeri računalnika. Na dokumentu računalnika se naredi popravek. Proces se ponovi še v smeri računalnika proti uporabniku. Vsi trije dokumenti so sinhronizirani.

Sistem je zanesljiv. Težava je, da taka zasnova ne omogoča sodelovanja oddaljenih uporabnikov v realnem času, saj vse skupaj poteka na enem sistemu.



Slika 3.1: Osnovna topologija Diferenčne sinhronizacije.

### 3.1.2 Metoda dveh senc

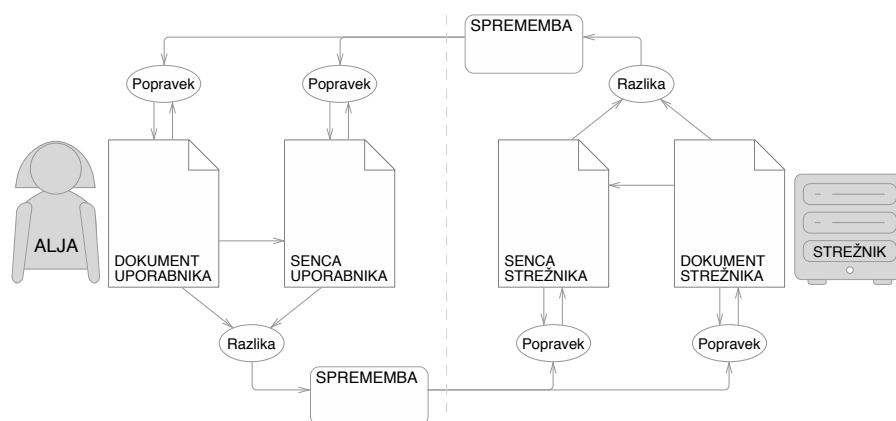
Konceptualno še vedno ostajamo pri istem algoritmu. Razlika je v topologiji. Namesto skupne sence uporabimo senco uporabnika in senco strežnika, ki se s popravki posodabljata ločeno. Taka zasnova DS omogoča sodelovanje oddaljenih uporabnikov. Vsak uporabnik je na svojem sistemu. Uporabnike povezuje centralni strežnik. Zaradi enostavnosti razlage je na Sliki 3.2 prikazan le en uporabnik in strežnik, ki sta ločena s črtkano črto. Po vsakem ciklu morata biti dokument uporabnika in dokument strežnika identična.

Predvidimo, da so na začetku vsi dokumenti v konsistentnem stanju. Uporabnik začne tipkati. Vključi se signal, da je med dokumentom in senco uporabnika razlika. Ko je znano, kakšne spremembe so bile narejene, se mora dokument prekopirati v senco. Sprememba se pošlje strežniku. Na strežniku se naredita dva popravka, na senci in na dokumentu. Pomembno je, da se popravek na senci izvede brez problemov.

Na dokumentu se naredi nejasen (angl. *fuzzy*) popravek, ki izvira iz sodelovanja med uporabniki. Če naredi eden izmed uporabnikov korenito spremembo na dokumentu, povzroči, da se popravek prvega uporabnika ne umesti v dokument tako, kot bi si on želel. Nepredvideno stanje se reši v naslednji polovici cikla. Senca in dokument strežnika sta v tem trenutku različna. Ko je znano, kakšne so spremembe med senco in nepredvidenim dokumentom strežnika, se mora dokument prekopirati v senco. Sprememba se pošlje uporabniku. Ker je bila sprememba dokumenta strežnika narejena na podlagi sence strežnika, ki je bila ista kot senca in dokument uporabnika, se brez težav naredi popravek na senci in na dokumentu uporabnika. Dokumenti so sinhronizirani.

Kot vidimo, se med uporabnikom in strežnikom pošiljajo le spremembe. Vprašamo se lahko, zakaj je tako. Zakaj se sploh uporablja senca kot vmesni korak in zakaj se dokument uporabnika direktno ne pošilja na strežnik? Razlog je v velikosti podatkov za prenos. Če bi uporabnik urejal zelo velik dokument in bi se moral ta po vsaki nastali razliki v dokumentu v celoti poslati na strežnik, bi bilo to neučinkovito. Z uporabo sence se poskrbi, da se ne glede na velikost dokumenta strežniku pošiljajo le majhni paketki. Enaka razlaga velja tudi v obratni smeri.

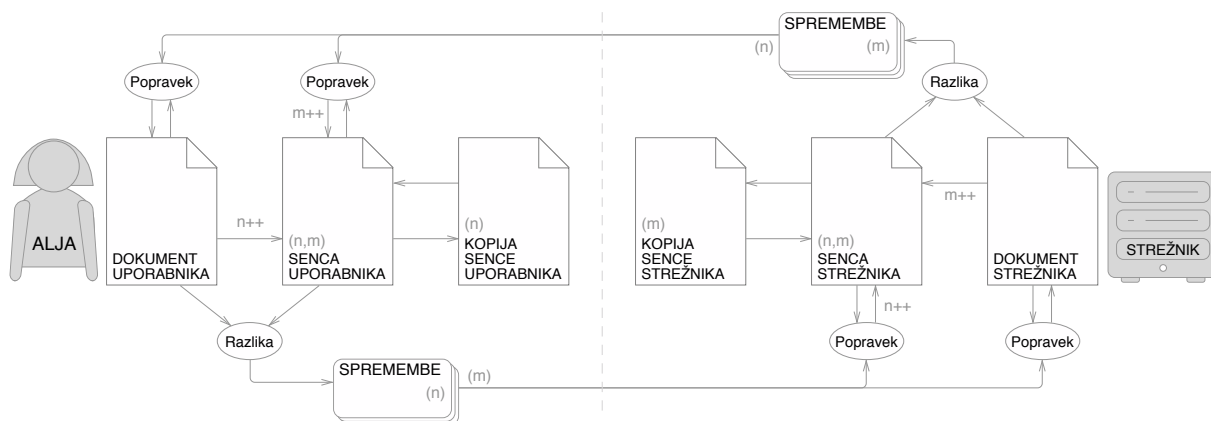
Sistem omogoča sodelovanje oddaljenih uporabnikov, vendar ni zanesljiv. Lahko se zgodi, da uporabnik na nezanesljivi povezavi od strežnika ne dobi nobenega odgovora. Do izgube spremembe lahko pride v prvem ali v drugem delu cikla. Senca uporabnika in senca strežnika nista več sinhronizirani. Edina rešitev za povrnitev nastalega stanje je, da povozimo uporabnikove spremembe. Tega si nihče ne želi.



Slika 3.2: Diferenčna sinhronizacija z metodo dveh senc.

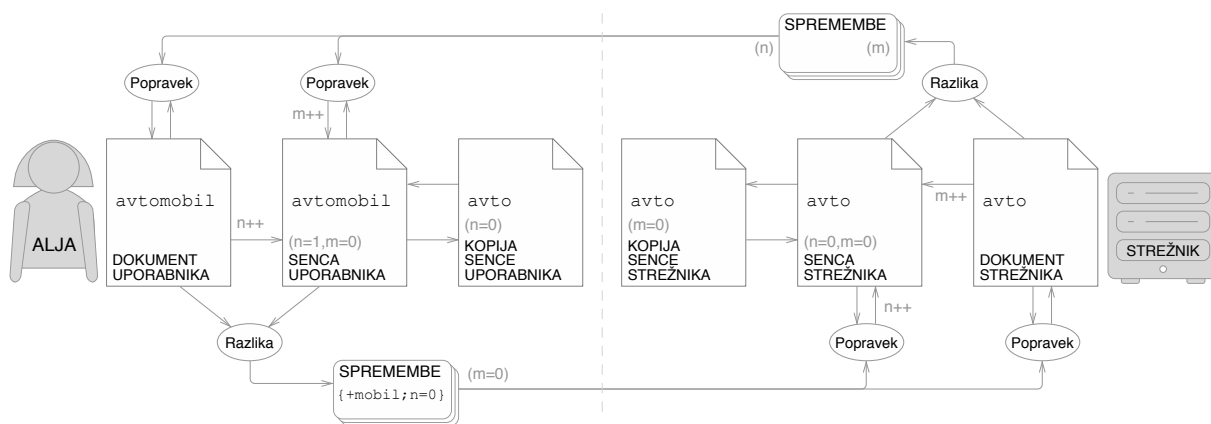
### 3.1.3 Metoda zagotovljene dostave

DS z metodo zagotovljene dostave je nadgradnja metode dveh senc. Iz Slike 3.3 lahko razberemo, da se je topologija razširila še z varnostnima kopijama sence uporabnika in sence strežnika. Tudi tokrat mora biti vseh šest dokumentov v konsistentnem stanju. Namesto ene same spremembe se lahko pošlje več sprememb hkrati. V topologiji so nove tudi številke verzij.



Slika 3.3: Diferenčna sinhronizacija z metodo zagotovljene dostave. S črko  $n$  je označena številka verzije uporabnika, s črko  $m$  pa številka verzije strežnika. Na začetku sta obe številki verzij enaki 0, tekom algoritma pa se konstatno povečujeta. Z njimi na enostaven način označimo trenutno stanje dokumentov. Uporabljajo se, ko se na dokumentih delajo popravki, saj se preko številke verzije identificira stanje dokumentov.

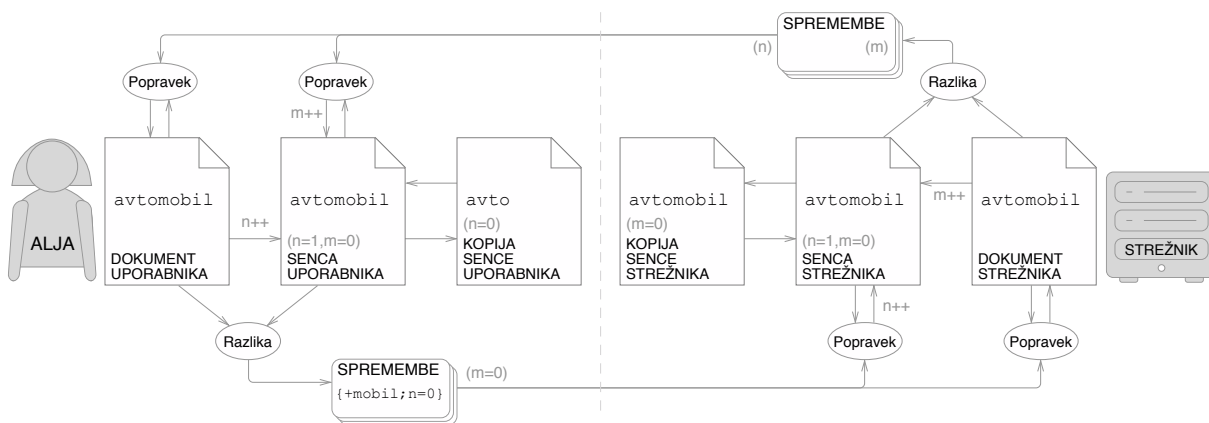
Delovanje DS z metodo zagotovljene dostave bomo razložili na primeru. V vseh dokumentih se nahaja beseda "avto". Alja v svoj dokument dopiše "mobil". Posledica razlike dokumenta uporabnika od sence uporabnika je sprememba "mobil".



Slika 3.4: Alja v svojem dokumentu vidi besedo avtomobil.

K sami spremembi se shrani številka verzije uporabnika, na podlagi katere je bila sprememba osnovana. V seznam sprememb se torej shrani  $\{ +mobil;n=0 \}$ . V tem trenutku se Aljin dokument prekopira v njeno senco, poveča se številka verzije uporabnika na  $n=1$ . Sprememba se pošlje strežniku. Poleg spremembe se strežniku pošlje tudi številka zadnje sinhronizirane verzije strežnika  $m=0$ . Stanje je vidno na Sliki 3.4.

Strežnik primerja številki verzije uporabnika in strežnika spremembe s številkami verzije uporabnika in strežnika v senci strežnika (glej Sliko 3.4). Številki  $n$  in  $m$  sta v obeh primerih  $0$ , kar dovoljuje, da se naredi popravek na senci strežnika. Številka verzije uporabnika v senci strežnika se poveča na  $n=1$ . Senca strežnika in senca uporabnika imata enako besedilo in enaki številki  $n$  in  $m$ , kar pomeni, da sta sinhronizirani. To se vidi tudi na Sliki 3.5.

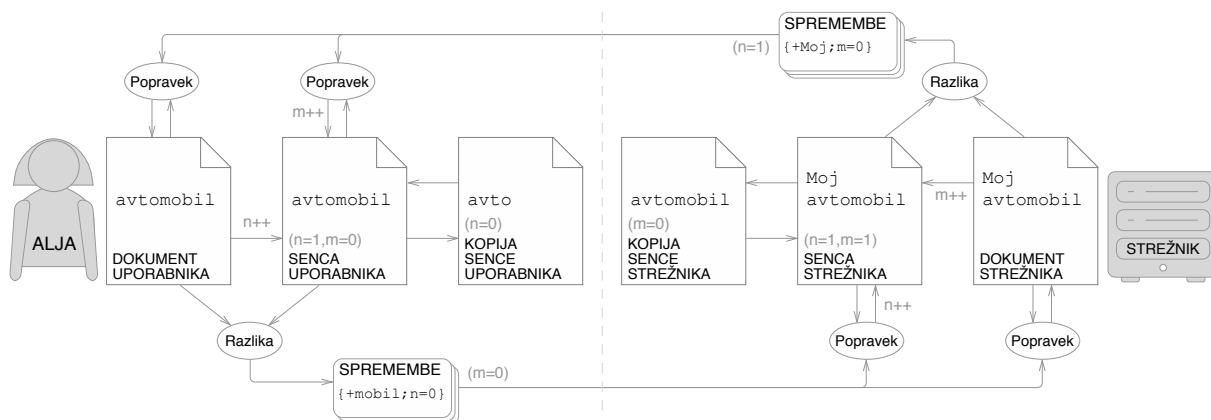


Slika 3.5: Polovica cikla je končanega. Senci sta konsistentni.

Popravek se naredi tudi na dokumentu strežnika, kar povzroči, da se senca strežnika prekopira v varnostno kopijo sence strežnika. Zakaj je to potrebno, bomo videli v nadaljevanju.

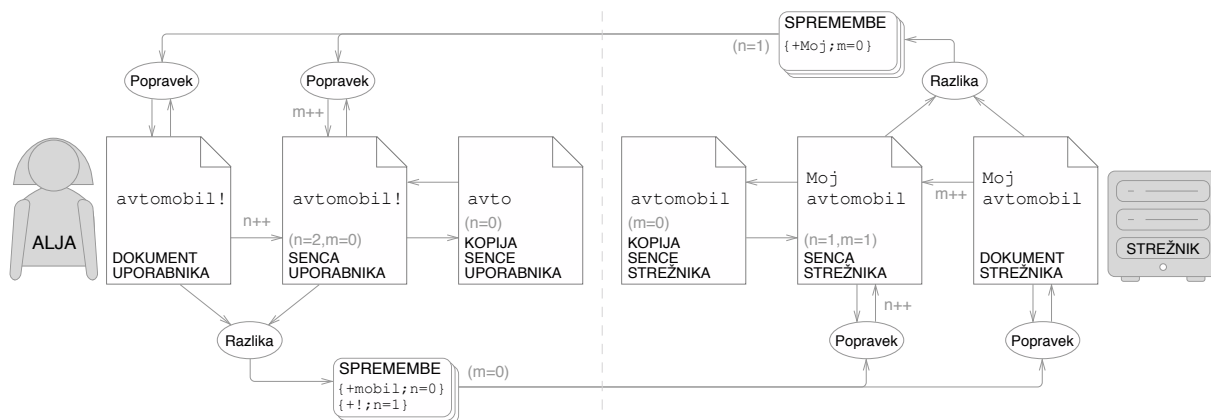
Recimo, da uporabnik Bine na začetek dokumenta vpiše besedo "Moj". Na isti način kot Aljina sprememba se tudi njegova sprememba sinhronizira v dokument strežnika. Vsebina dokumenta strežnika je tako "Moj avtomobil".

Razlika med dokumentom strežnika in senco strežnika je sprememba "Moj" osnovana na številki verzije strežnika  $0$ . Sprememba  $\{ +Moj;m=0 \}$  se mora poslati Alji. Poleg nje se mora poslati tudi  $n=1$ , s katerim strežnik potrjuje, da je sprejel zadnjo Aljino spremembo. Pred tem je potrebno dokument strežnika prekopirati v senco strežnika. Številka verzije strežnika  $m$  v senci strežnika se poveča na  $1$ , Slika 3.6.



Slika 3.6: Strežnik prejme Binetovo spremembo, ki jo mora poslati Alji.

Sedaj lahko nadaljujemo z drugim delom cikla iz smeri strežnika proti Alji. Zaradi slabe povezave Alja ne prejme spremembe  $\{ +Moj;m=0 \}$  iz strežnika. Tega niti ne opazi, ampak tipka naprej. V svoj dokument doda klicaj. Ponovi se postopek, podoben prvemu ciklu. K seznamu sprememb se doda  $\{ +!;n=1 \}$ . Aljin dokument se prekopira v njeno senco,  $n$  se poveča na 2. Obe spremembi se pošljeta strežniku. Spomnimo se, da za prvo spremembo Alja ni dobila nobenega odgovora. Številka zadnje sinhronizirane verzije strežnika je še vedno  $m=0$ . Slika 3.7 prikazuje nastalo situacijo.

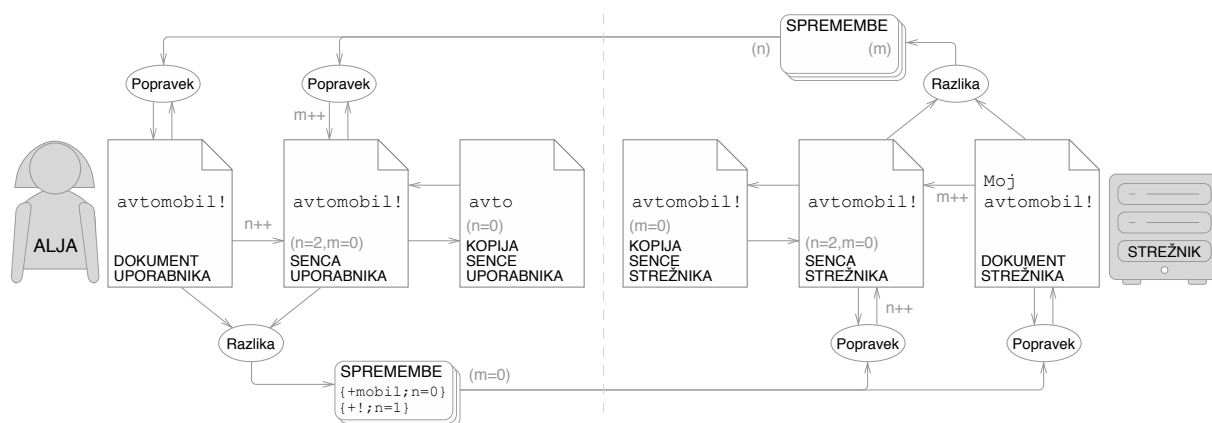


Slika 3.7: Alja ni sprejela strežnikovih sprememb. V svojem dokumentu je naredila novo spremembo.

Strežnik sprejme Aljini spremembi. Najprej poskuša procesirati prvo spremembo  $\{ +mobil;n=0 \}$ . Ker se številka zadnje sinhronizirane verzije strežnika  $m=0$  ne ujema s številko verzije strežnika v senci strežnika, se senca strežnika povrne iz kopije sence strežnika (s številko  $m=0$ ), v katerem se nahaja "avtomobil". Strežnik poskusi ponovno procesirati prvo spremembo. Številki verzije strežnika  $m=0$  se ujemata. Številka verzije uporabnika je v prvi spremembi  $n=0$ , v senci strežnika pa  $n=1$ . To pomeni, da je strežnik

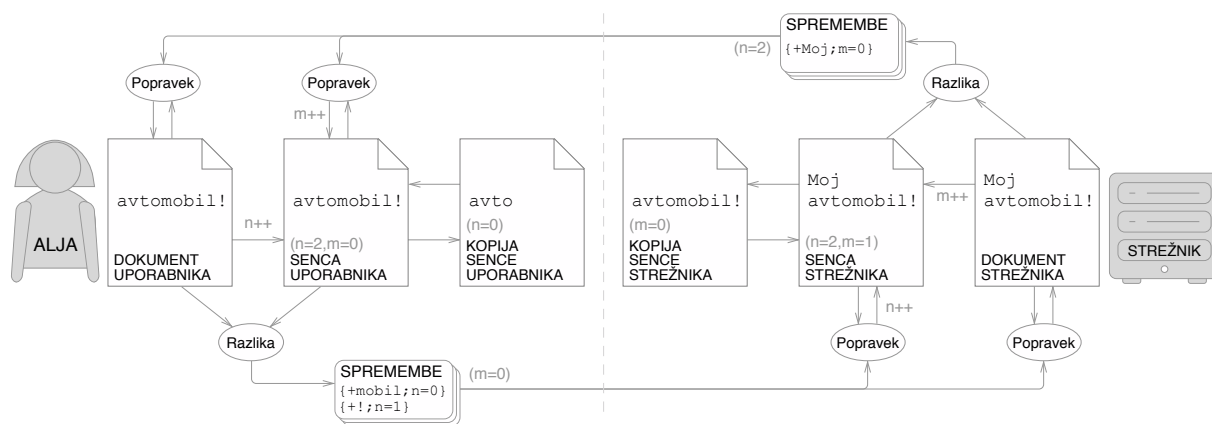
v enem izmed prejšnjih ciklov že obdelal to spremembo, zato jo lahko ignorira. Strežnik nato poskusi sprocesirati drugo spremembo  $\{ +! ; n=1 \}$ . Številki  $n=1$  in  $m=0$  se tokrat ujemata. Strežnik lahko naredi popravek na senci strežnika. Številka  $n$  se poveča za 1 na  $n=2$ . Popravek se nato naredi tudi na dokumentu strežnika, naredi pa se tudi kopija sence strežnika. Senca uporabnika in senca strežnika sta sinhronizirani.

Opomba: Ko se je naredila povrnitev kopije sence strežnika, se je seznam strežniških sprememb pobrisal. Spremembe v dokumentih in številkah  $n$  in  $m$  vidimo na Sliki 3.8.



Slika 3.8: Alja je strežniku poslala že dve spremembi.

Dokument strežnika in senca strežnika se na Sliki 3.8 razlikujeta. Strežnik pogleda, kakšne so spremembe na dokumentu. Tako kot v prvem ciklu najde spremembo  $\{ +Moj ; m=0 \}$ . Sprememba se ponovno pošlje Alji. Tokrat je številka verzije uporabnika v senci dnevnika enaka 2, zato se Alji poleg spremembe pošlje tudi  $n=2$ . Še prej se dokument prekopira v senco in v senci strežnika se poveča številka verzije strežnika iz  $m=0$  nazaj na  $m=1$ , kot je prikazano na Sliki 3.9.

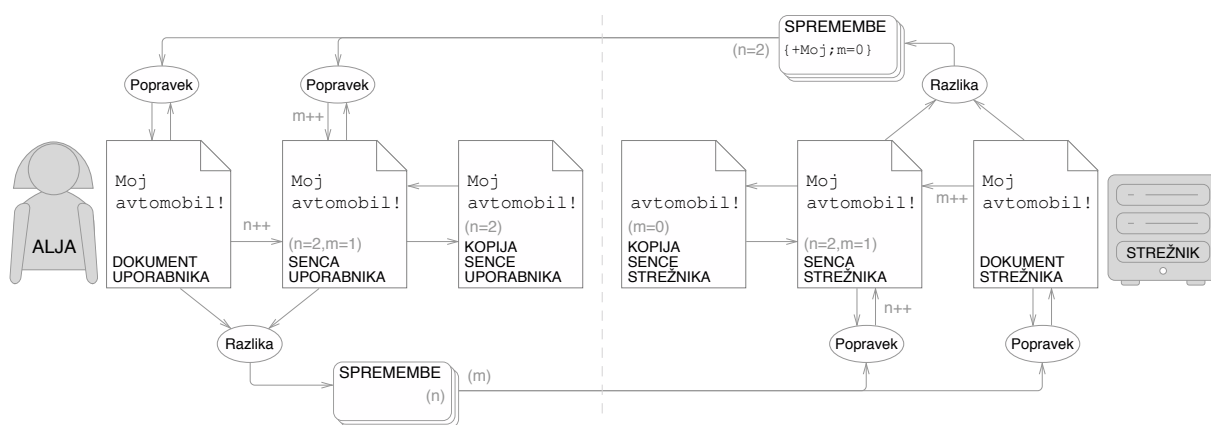


Slika 3.9: Strežnik pošilja Alji potrditev njenih sprememb in spremembo Bineta.



Če bi bil strežnik spet neuspešen pri pošiljanju svojih sprememb, bi se seznam Aljinih sprememb povečeval, vse dokler ne bi dobila odgovora od strežnika. Recimo, da tokrat strežniku uspe in Alja prejme spremembo { +Moj;m=0 }.

Tokrat Alja primerja številke verzij. Številka verzije uporabnika v spremembi in v senci uporabnika je  $n=2$ . Številka verzije strežnika v spremembi in v senci je  $m=0$ . Sprememba je primerna za popravek na senci uporabnika. Naredijo se naslednje akcije. Na senci uporabnika se naredi popravek. Številka verzije zadnje sinhronizacije s strežnikom se v senci uporabnika poveča na  $m=1$ . Popravek se naredi na dokumentu strežnika. Naredi se kopija sence uporabnika. Številka verzije uporabnika v kopiji sence uporabnika se poveča na  $n=2$ . Ker je Alja uspešno sprejela potrditev strežnika, da je sprejel vse Aljine spremembe do številke verzije uporabnika  $n=2$ , se s seznama Aljinih sprememb pobrišejo vse spremembe, ki imajo številko verzije uporabnika manjšo od 2 ( $n < 2$ ).



Slika 3.10: Konsistentno stanje dokumentov.

Kjub temu da je med sodelovanjem uporabnikov in strežnika prišlo do izpada v povezavi, so se dokumenti posinhronizirali. Na Sliki 3.10 vidimo, da v njih na koncu piše "Moj avtomobil!". Razen v varnostni kopiji sence strežnika, ki je vedno en korak za sinhronizacijo, kar je tudi njen namen, piše "avtomobil!".

Pri razlagi postopka DS nismo omenjali konkretnih lokacij sprememb, ampak le kakšne so bile spremembe. Zavedati se moramo, da so pri implementaciji algoritma tudi te pomembne. O spremembah in iskanju razlik v besedilu bomo govorili v Poglavju 5.

## 3.2 Operativna transformacija

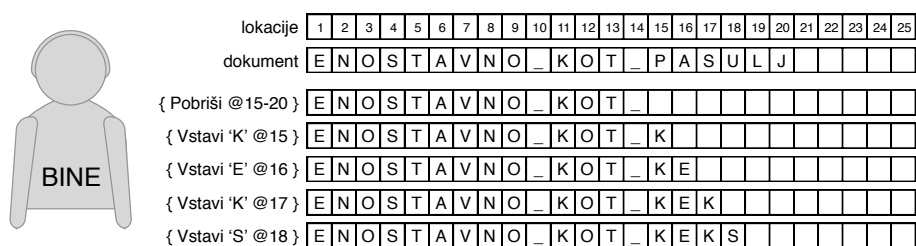
Operativna transformacija (v nadaljevanju OT) se je prvič omenjala v članku Concurrency Control in Groupware Systems [1]. Pri Googlu so Operativno transformacijo vzeli za osnovno pri načrtovanju protokola Wave [4], ki se uporablja v Google Docs-ih, kar nakazuje na njeno uporabnost.

### 3.2.1 Osnovno delovanje

Pri OT je dokument shranjen kot vrsta kronoloških sprememb (včasih poimenovanih tudi operacij), narejenih na dokumentu. Primer spremembe je { Vstavi 'M' @11 }, ki pomeni "v dokumentu na lokacijo 11 vstavi črko M" ali { Pobriši @3-7 }, ki pomeni "v dokumentu pobriši vse znake med lokacijo 3 in 7". Obstajajo še druge vrste sprememb, kot so oblikovanje besedila, zaklep odstavka, razveljevitve spremembe, ... Zaradi enostavnosti razlage se bomo osredotočili le na omenjena dva tipa sprememb. Ko uporabnik ureja besedilo, se njegove spremembe shranjujejo v revizijski dnevnik. Seveda hranimo tudi dokument kot zaključeno celoto znakov, vendar so pomembne spremembe, ki so bile narejene na tem dokumentu. Če se urejanju skupnega dokumenta pridruži nov uporabnik, mu iz revizijskega dnevnika ponovimo vse (od prve do zadnje) spremembe in že lahko sodeluje pri urejanju tako kot ostali uporabniki.

Glede na to, da poznamo revizijo vseh sprememb, narejenih na dokumentu, lahko preverimo, kaj je uporabnik imel v svojem urejevalniku, preden je naredil novo spremembo. Na ta način njegovo spremembo pravilno umestimo v skupno besedilo skupaj z ostalimi spremembami, ki so bile narejene medtem. Algoritem, ki skrbi za umeščanje ali združevanje (angl. *merging*) sprememb, se imenuje OT.

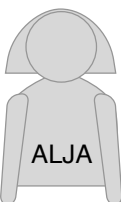
Poglejmo delovanje OT na primeru [11]. Predpostavimo, da imamo dokument, v katerem se trenutno nahaja stavek "ENOSTAVNO KOT PASULJ", ki zasede dvajset lokacij (ali mest). Urejanju tega dokumenta se pridružita Alja in Bine.



Slika 3.11: Bine naredi pet sprememb.

Če želi Bine spremeniti stavek v “ENOSTAVNO KOT KEKS”, mora za to narediti pet sprememb, ki jih prikazuje Slika 3.11. Naj pripomnimo, da bi Bine lahko stavek spremenil tudi samo z dvema spremembama, in sicer { Pobriši @15-20 } in { Vstavi ‘KEKS’ @15 }. Lahko bi pobrisal tudi vsako črko posebej. Tako bi bilo sprememb še več, kot jih je prikazanih na Sliki 3.11. Več o spremembah in iskanju razlik v besedilu bomo govorili v Poglavju 5. Zaradi enostavnosti razlage recimo, da je Bine naredil pet sprememb.

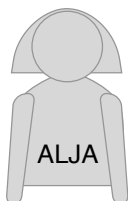
Predpostavimo, da, medtem ko Bine tipka, začne spreminjati stavek tudi Alja, in sicer v “TAKO ENOSTAVNO KOT PASULJ”. Tudi Alja je naredila pet sprememb.



lokacije	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
dokument	E	N	O	S	T	A	V	N	O	_	K	O	T	_	P	A	S	U	L	J						
{ Vstavi 'T' @1 }	T	E	N	O	S	T	A	V	N	O	_	K	O	T	_	P	A	S	U	L	J					
{ Vstavi 'A' @2 }	T	A	E	N	O	S	T	A	V	N	O	_	K	O	T	_	P	A	S	U	L	J				
{ Vstavi 'K' @3 }	T	A	K	E	N	O	S	T	A	V	N	O	_	K	O	T	_	P	A	S	U	L	J			
{ Vstavi 'O' @4 }	T	A	K	O	E	N	O	S	T	A	V	N	O	_	K	O	T	_	P	A	S	U	L	J		
{ Vstavi '_' @5 }	T	A	K	O	_	E	N	O	S	T	A	V	N	O	_	K	O	T	_	P	A	S	U	L	J	

Slika 3.12: Alja naredi pet sprememb.

Če bi Alja v naslednjem koraku naivno sprejela in izvršila Binetovo prvo spremembo, bi v stavku pobrisala napačne črke, kot je prikazano na Sliki 3.13.

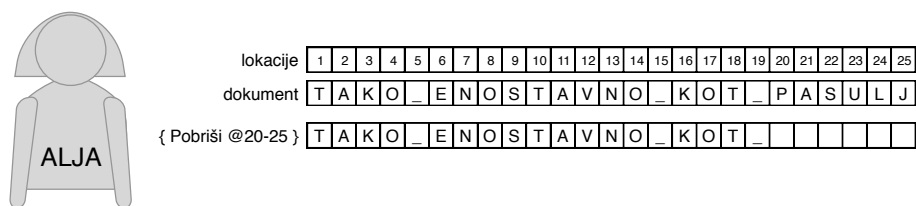


lokacije	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
dokument	T	A	K	O	_	E	N	O	S	T	A	V	N	O	_	K	O	T	_	P	A	S	U	L	J
{ Pobriši @15-20 }	T	A	K	O	_	E	N	O	S	T	A	V	N	O	A	S	U	L	J						

Slika 3.13: Brez transformacije pride do nekonsistentnosti.

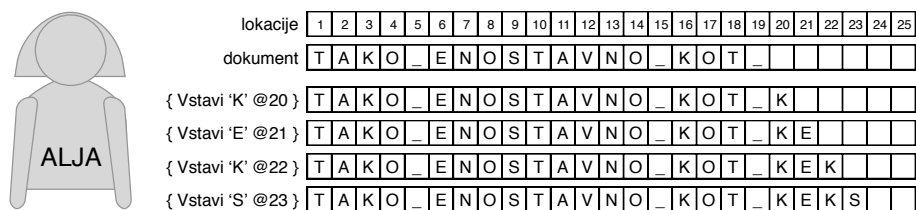
Alja je na začetku stavka dopisala pet znakov, o katerih Bine še ni bil seznanjen. Lokacija Binetove spremembe je zato napačna glede na Aljino verzijo dokumenta. Da bi se izognili temu problemu, mora narediti Alja uveljavitev Binetovih sprememb relativno na svoj lokalni dokument. V našem primeru mora Alja, ko sprejme Binetove spremembe, zamakniti lokacijo spremembe za pet znakov, kolikor jih je vpisala na začetku stavka. Sprememba { Pobriši @15-20 } se preoblikuje v { Pobriši @20-25 }, kot je pravilno prikazano na sliki 3.14.

Ko naredi Alja transformacijo in izvrši Binetovo prvo spremembo, dobi pravilen stavek.



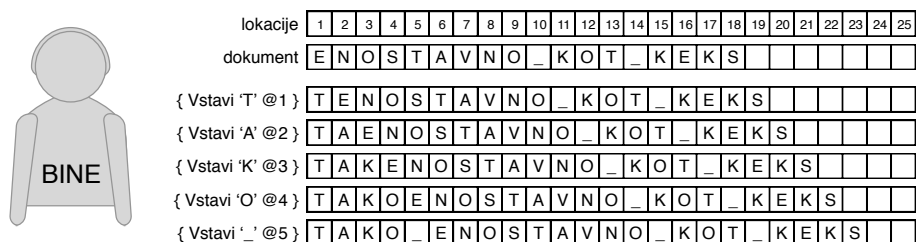
Slika 3.14: Z uporabo OT dobimo pravilen rezultat.

Ko transformira in izvede še ostale štiri spremembe, dobi končno verzijo dokumenta.



Slika 3.15: Končna verzija dokumenta, ki ga vidi Alja.

Včasih spremembe ne povzročajo konfliktov in ni potrebe po transformaciji. Ko prejme Bine Aljine spremembe, ni potrebe po zamikanju lokacij. Bine mora izvesti Aljine spremembe točno tako, kot jih je ona izvedla na svojem lokalnem dokumentu.



Slika 3.16: Končna verzija dokumenta, ki ga vidi Bine.

Tako Alja kot Bine v svojem lokalnem dokumentu na koncu vidita stavek "TAKO ENOSTAVNO KOT KEKS". To ne bi bilo mogoče, če ne bi uporabili algoritma za zamikanje sprememb. Pravilno implementiran algoritem OT nam zagotavlja, da imajo vsi uporabniki, ko prejmejo vse spremembe, isto vsebino dokumenta.

### 3.2.2 Protokol za sodelovanje

Z uporabo OT smo se naučili, kako z zamikanjem lokacij sprememb dopustiti več uporabnikom urejanje istega dokumenta brez konfliktov. Še vedno pa obstaja težava, kako vsako spremembo pravilno združiti z drugimi spremembami, če se le-te zgodijo istočasno. Zagotoviti moramo, da vsak oddaljeni uporabnik ve, da obstajajo spremembe, ki morajo biti združene. Za to skrbi Protokol za sodelovanje (angl. *Collaboration protocol*). Tehnologiji OT in Protokol za sodelovanje skupaj, znak za znakom, skrbita za sodelovanja v realnem času [12].

Zavedati se moramo, da za urejanje dokumenta v realnem času skrbijo tako strežnik kot odjemalci. Običajno so to spletni brskalniki uporabnikov. Pri urejanju dokumenta mora odjemalec sprocesirati vse spremembe, ki jih naredi uporabnik, in jih poslati na strežnik. Sprocesirati mora tudi vse spremembe drugih uporabnikov, ki mu jih pošlje strežnik. Seveda brez sodelovanja strežnika ne gre. Oglejmo si, katere podatke si morajo beležiti odjemalci in katere strežnik.

Vsak odjemalec si mora beležiti naslednje podatke:

**Številko zadnje sinhronizirane revizije** smo na Sliki 3.17 označili s sivim krogcem in številko v njem.

**Čakajoče spremembe** so spremembe, ki so bile narejene v odjemalcu in niso še bile poslane na strežnik.

**Poslane spremembe** so spremembe, ki so bile poslane na strežnik, vendar jih strežnik še ni potrdil.

**Trenutno stanje dokumenta** kot ga vidi uporabnik.

Na strežniku se shranjujejo tri stvari:

**Čakajoče spremembe** so spremembe, ki jih je strežnik sprejel, a jih še ni sprocesiral.

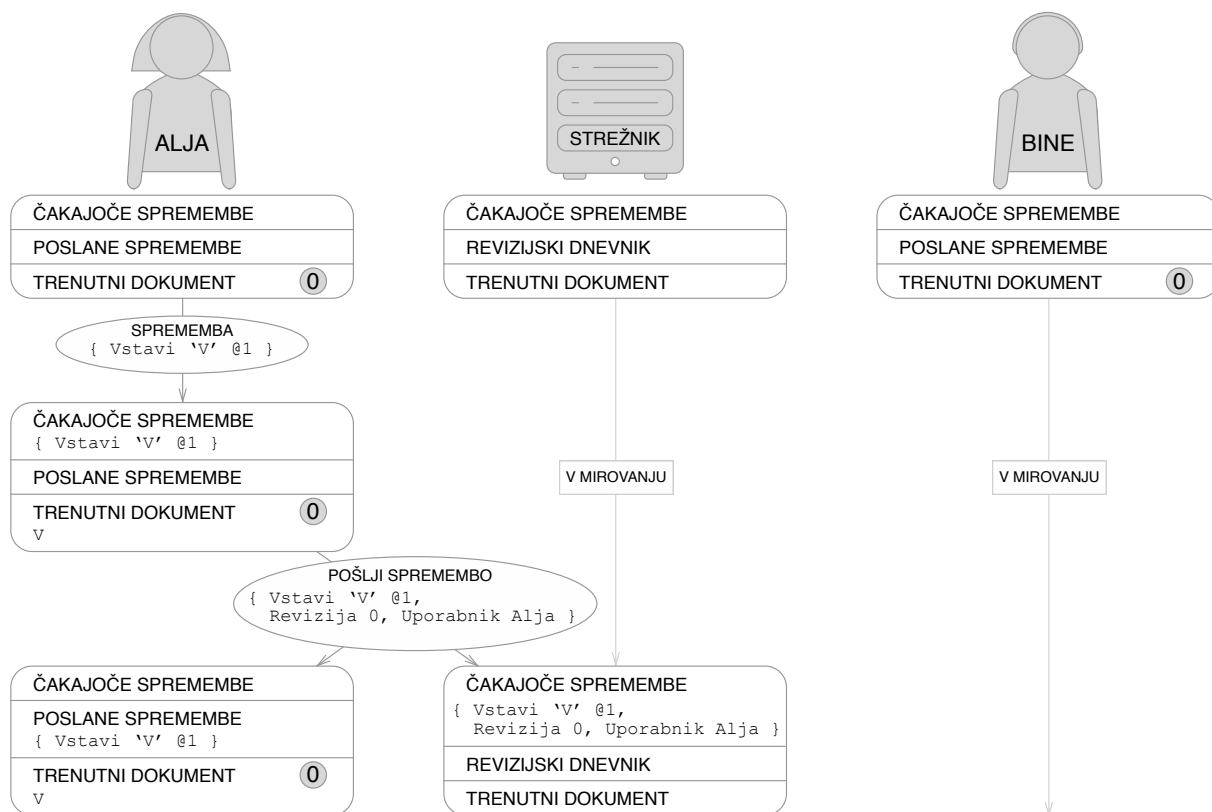
**Revizijski dnevnik** je dnevnik, v katerem je celotna zgodovina sprememb.

**Trenutno stanje dokumenta** kot bi ga morali v najkrajšem možnem času videti vsi uporabniki.

S hranjenjem in uporabo teh podatkov je mogoče zasnovati komunikacijo med strežnikom in odjemalci tako, da so oddaljeni urejevalniki sposobni drug od drugega naglo sprocesirati spremembe.

Na vzorčnem dokumentu bomo predstavili, kako je poskrbljeno za komunikacijo med strežnikom in odjemalcem. Na Sliki 3.17 predstavljata zunanja stolpce uporabnika Aljo in Bineta, ki urejata skupni dokument. Srednji stolpec je strežnik. Spremembe, ki jih naredita Alja in Bine in se pošljejo na strežnik, so označene v ovalni obliki. Transformacije, ki smo jih že spoznali, bodo na naslednjih slikah označene s peterokotnikom.

Alja začne tipkati "V" na začetku dokumenta.



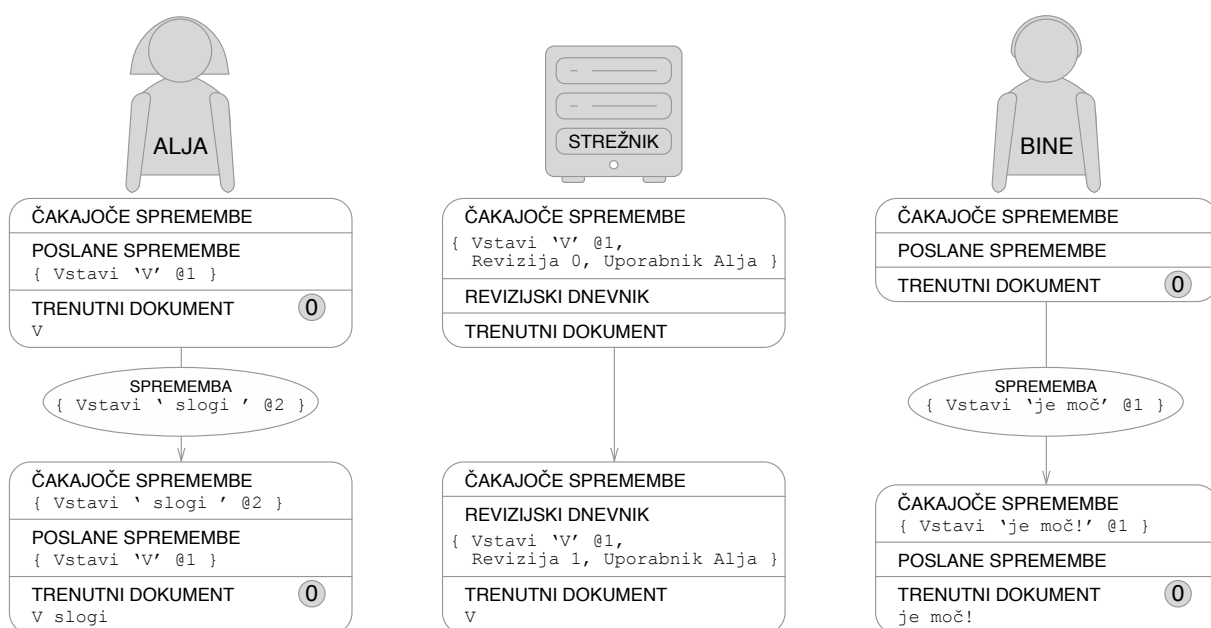
Slika 3.17: Alja začne tipkati. Bine je v mirovanju.

Aljin urejevalnik si spremembo `{ Vstavi 'V' @1 }` shrani med čakajoče spremembe. V naslednjem trenutku se ta pošlje na strežnik ter se prestavi na seznam poslanih sprememb. Poleg same spremembe se strežniku pošlje tudi številka revizije in avtorja spremembe (uporabnika). Podatek o uporabniku je pomemben z vidika avtentikacije. O številki revizije bomo več povedali v naslednjih korakih. Na Sliki 3.17 vidimo, kako strežnik sprejme Aljino prvo spremembo in jo doda med čakajoče spremembe.

Odjemalec lahko strežniku v enem pošiljanju pošlje tudi več črk za vstavljanje v dokument. Koliko črk hkrati bo poslanih, je odvisno od implementacije urejevalnika in algoritma za zaznavanje sprememb. Več o tem v poglavju 5.

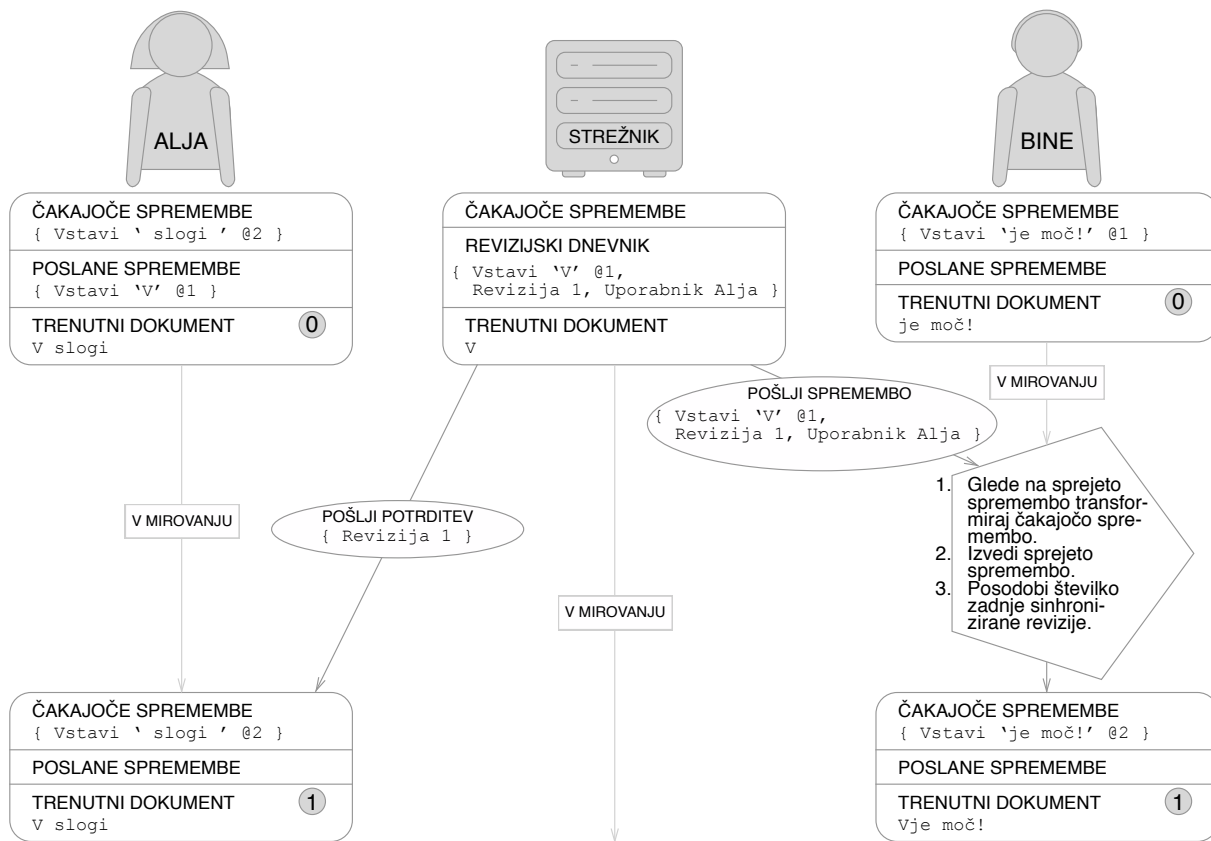
Na Sliki 3.18 vidimo, da Alja nadaljuje s tipkanjem in doda " slogi " v svoj urejevalnik. Alja v svojem trenutnem dokumentu vidi "V slogi ". V istem času Bine vpiše "je moč!" v svoj prazen dokument. Ne pozabimo, da Bine še vedno ni prejel Aljinih sprememb.

Aljin { Vstavi ' slogi ' @2 } je bil dodan med čakajoče spremembe in še ni poslan na strežnik. Pravilo je, da strežniku nikoli ne pošljamo več kot ene čakajoče spremembe naenkrat. Dokler Alja od strežnika ne dobi potrditve prve spremembe, bo njen urejevalnik vse nove spremembe hranil med čakajočimi spremembami. Na Sliki 3.18 lahko opazimo, da si je strežnik Aljino prvo spremembo že shranil v revizijski dnevnik.



Slika 3.18: Alja nadaljuje s tipkanjem. Bine na drugi strani tudi začne pisati na začetku svojega dokumenta. Strežnik o tem še ni obveščen.

V naslednjem koraku, kot je to prikazano na Sliki 3.19, pošlje strežnik Binetu Aljino prvo spremembo ter Alji odgovori s potrditvijo, da si je zabeležil njeno spremembo v revizijski dnevnik.



Slika 3.19: Strežnik procesira Aljino prvo spremembo.

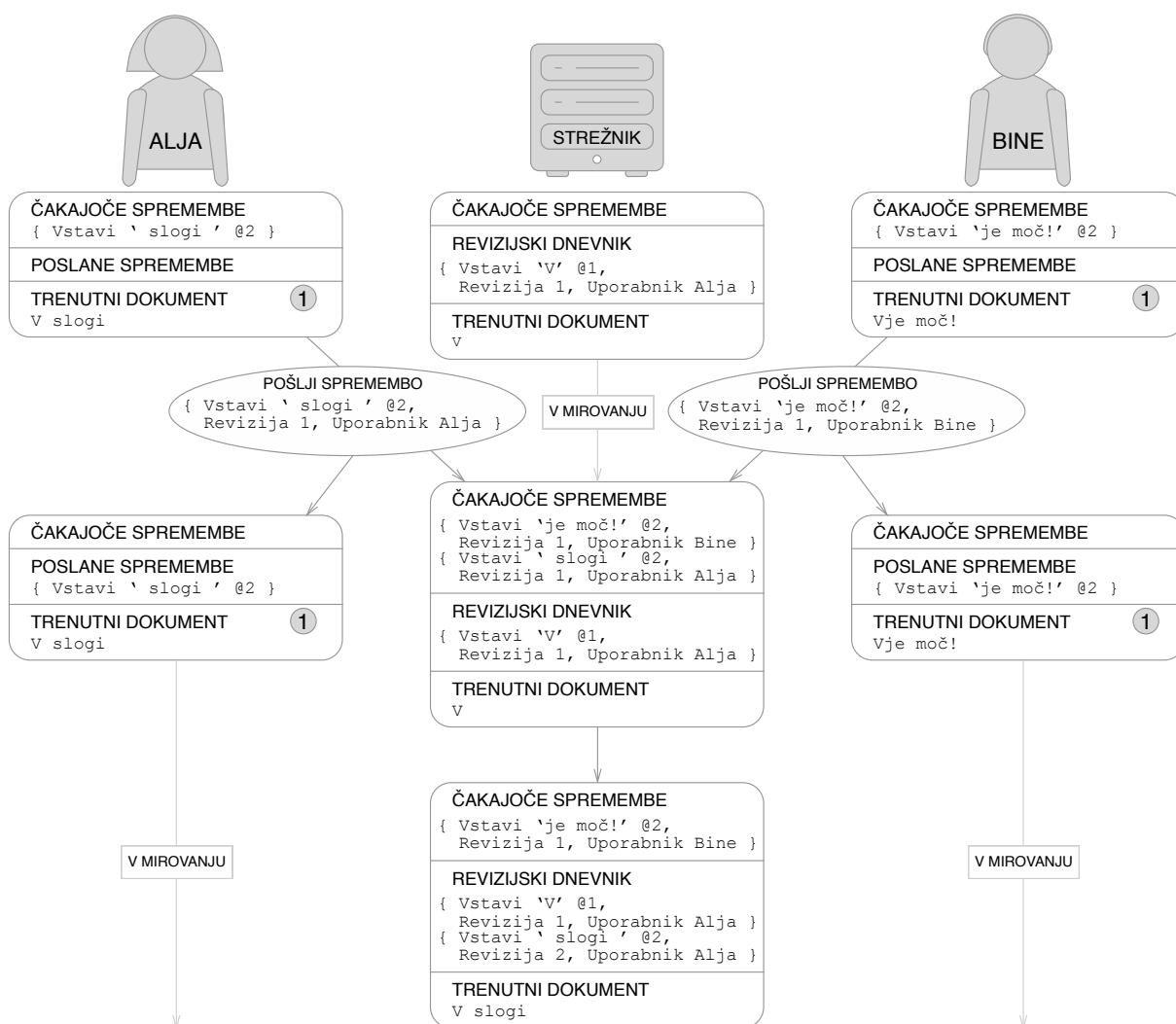
Bine sprejme Aljino spremembo od strežnika. Z uporabo OT mora transformirati svojo čakajočo spremembo { Vstavi 'je moč!' @1 }. Ker je Alja na začetek dokumenta že vpisala "V", Binetov urejevalnik zamakne njegovo spremembo za eno lokacijo. Po tem procesu vstavi Binetov urejevalnik v trenutni dokument Aljino spremembo "V" in posodobi številko zadnje sinhronizirane revizije na 1. Alja je medtem sprejela potrditev iz strežnika. Tudi ona posodobi številko zadnje sinhronizirane revizije na 1. Svojo spremembo odstrani s seznama poslanih sprememb.

V Aljinem trenutnem dokumentu se nahaja "V slogi ", v Binetovem pa "Vje moč!", kar nakazuje, da še ni prejel vseh sprememb. Sledi pošiljanje čakajočih sprememb.

Na Sliki 3.20 se vidi, da oba uporabnika hkrati pošljeta spremembo, vendar sprejem strežnik Aljino pred Binetovo in jo zato tudi prej sprocesira. Njeno spremembo s seznama čakajočih sprememb prestavi v revizijski dnevnik. Pri tem mora le popra-



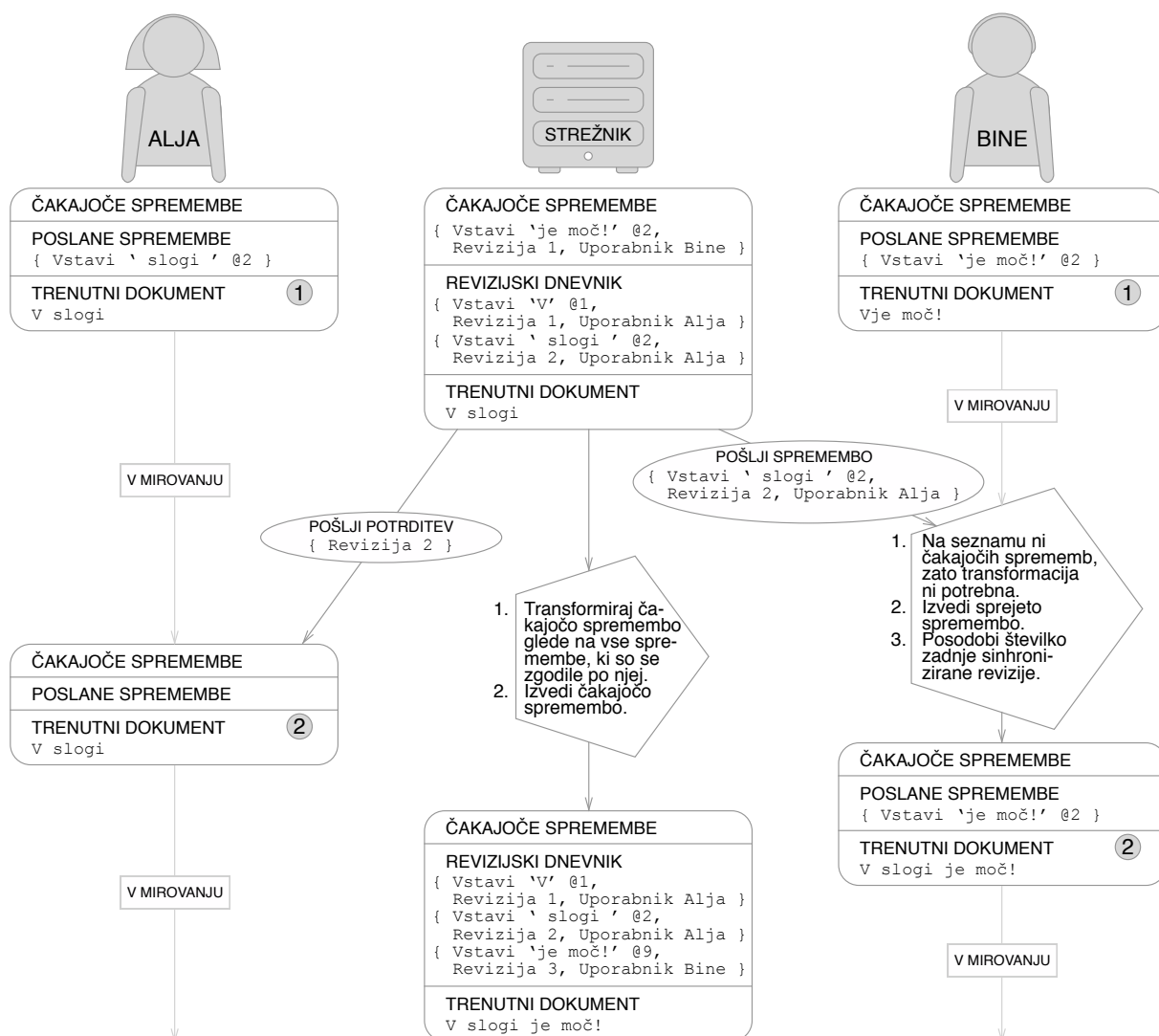
viti številko revizije, ki enolično označuje spremembo. Spomnimo se, da je Alja pri pošiljanju svoje spremembe poslala številko zadnje sinhronizirane revizije, številko 1. Strežnik na ta način ve, da je njena sprememba narejena na podlagi prve revizije. Ker se v revizijskem dnevniku pričakuje sprememba, ki je naslednja po vrsti, lahko njeno spremembo brez težav umesti v trenutni dokument.



Slika 3.20: Istočasno sodelovanje Aljinega in Binetovega urejevalnika preko strežnika.

Sledi potrditev spremembe Alji s številko revizije 2. Binetu se pošlje novo spremembo. Kot na Sliki 3.19 se začne tudi v primeru na Sliki 3.21 izvajati OT v Binetovem urejevalniku. Ker nima nobene čakajoče spremembe, ni potrebe po uporabi OT. Aljina sprememba se vstavi v njegov dokument brez transformacije. Številka zadnje sinhronizirane revizije se Binetu poveča na 2. Ker strežnik Alji odgovori s potrditvijo, se tudi njej poveča številka zadnje sinhronizirane revizije na 2.

OT se ne dogaja samo pri odjemalcih, ampak je nujna tudi na strežniku. Zakaj je temu tako, bomo ugotovi kmalu. Medtem ko se odjemalca ukvarjata z zahtevami strežnika (potrditev pri Alji in sprememba pri Binetu), je strežnik začel procesirati Binetovo čakajočo spremembo { Vstavi 'je moč!' @2, Revizija 1, Uporabnik Bine }. Bine je v času pošiljanja spremembe (Slika 3.20) verjel, da bo njegova sprememba nosila zapredno številko revizije 2. Vendar je strežnik že obdelal Aljino spremembo, ki jo je zapisal v revizijski dnevnik kot drugo spremembo. Strežnik mora z uporabo OT transformirati Binetovo spremembo, da jo bo lahko shranil kot revizijo 3.

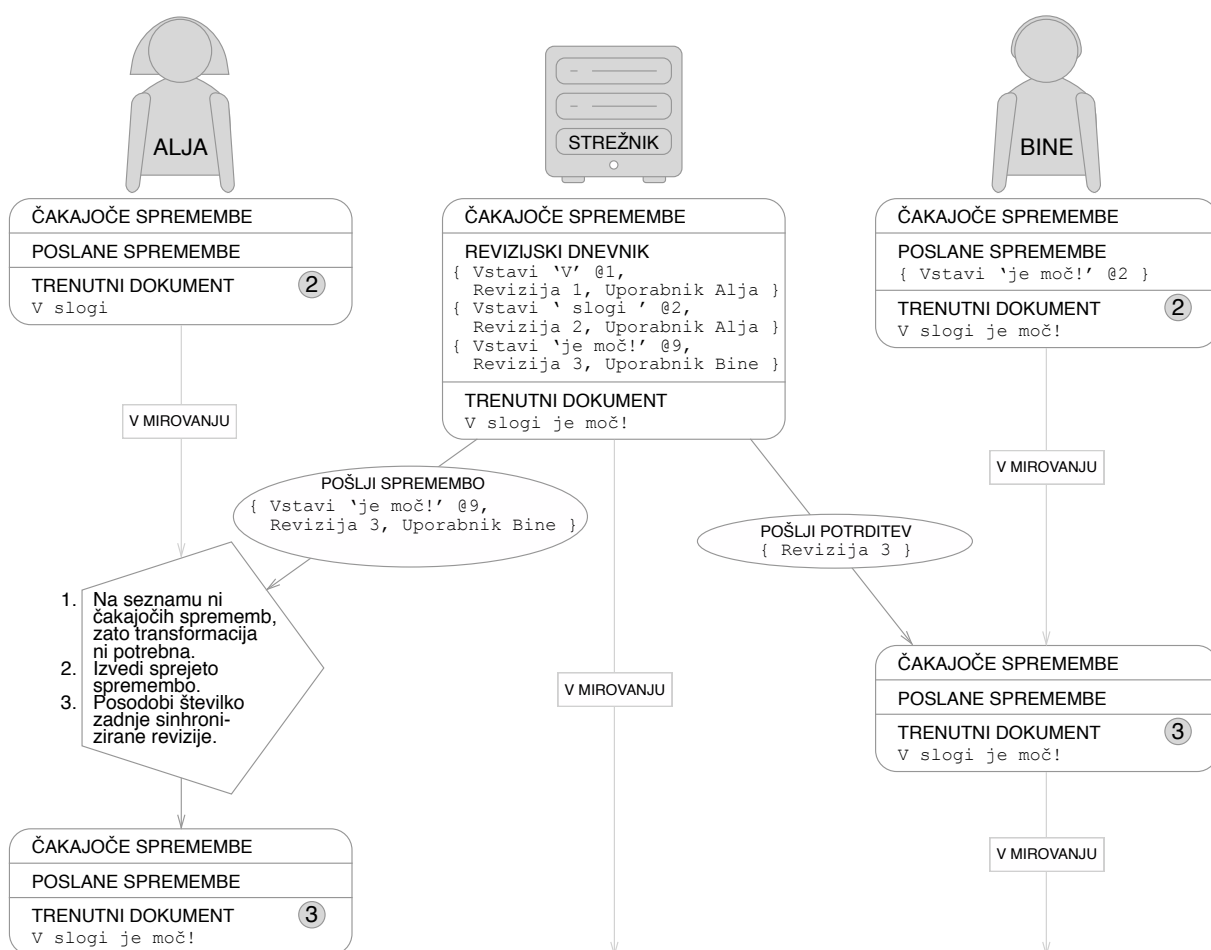


Slika 3.21: Strežnik uporabi OT pri obdelovanju Binetove spremembe.

Binetovo spremembo transformira glede na spremembe, ki so bile storjene, odkar je Bine zadnjič naredil sinhronizacijo s strežnikom. V našem primeru je bila narejena le Aljina sprememba { Vstavi ' slogi ' @2 }, ki je povzročila zamik Binetove spre-

membe za 7 lokacij. Končna sprememba v revizijskem dnevniku izgleda kot { Vstavi 'je moč!' @9, Revizija 3, Uporabnik Bine }.

Na koncu dobi Bine potrditev svoje spremembe in Alja prejme Binetovo spremembo. Številka zadnje sinhronizirane revizije se obema poveča na 3. V revizijskem dnevniku so 3 revizije. V tem trenutku imajo strežnik in oba urejevalnika isti dokument z vsebino "V slogi je moč!". Glede na to, da uporabnika nimata več čakajočih sprememb je to tudi končna verzija dokumenta, ki sta ga skupaj uredila uporabnika Alja in Bine.



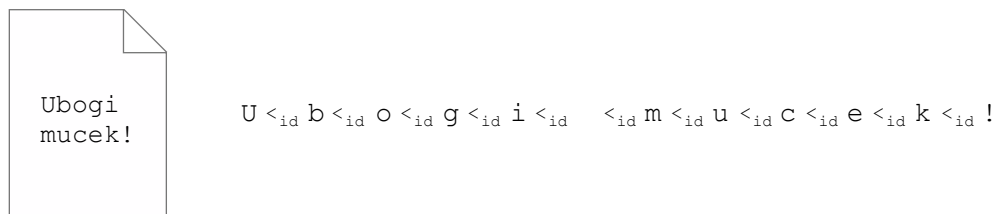
Slika 3.22: Uporabnika imata na koncu dokument z isto vsebino.

### 3.3 Brez operativne transformacije

V tem poglavju bomo opisali protokol Brez operativne transformacije [13], bolj znan pod kratico WOOT. Bil je narejen kot odgovor na kompleksnost OT. Rekli smo, da se pri OT med uporabnike pošiljajo spremembe in lokacija sprememb, primer { Vstavi 'M' @11 }. OT skrbi za pravilno transformacijo lokacij sprememb. Pri stop WOOT je drugačen in ga je lažje razumeti. WOOT skrbi za pravilno razvrščanje sprememb oziroma operacij, kot se jih običajno poimenuje. Glavna razlika je v podajanju informacij. Lahko si predstavljamo, da kaže na vsak znak v dokumentu unikatni kazalec, to je identifikacijska številka. Med oddaljene uporabnike, ki sodelujejo pri urejanju dokumenta, se pošiljajo informacije, med katerima znakoma oziroma na kateremu znaku je bila narejena sprememba. Posebnost protokola WOOT je, da se znakov v dokumentu nikoli ne briše, le označi se jih kot nevidne.

#### 3.3.1 Podatkovni model

Dokument pri protokolu WOOT je shranjen kot zaporedje znakov, predstavljenih z identifikacijskimi številkami (angl. *identifier order*). Zaporedje znakov besedila "Ubogi mucek!" se prikaže kot na Sliki 3.23.

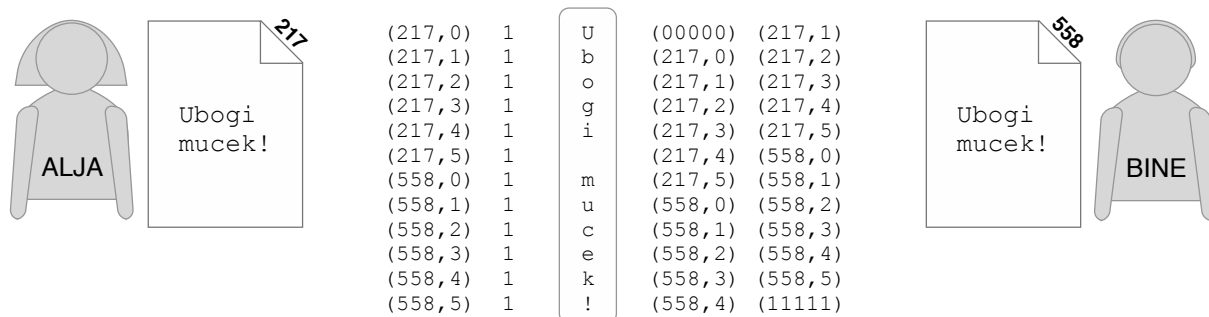


Slika 3.23: Prikaz zaporedja znakov v dokumentu.

Za vsak znak v dokumentu hranimo peterko podatkov: identifikacijsko številko (angl. *ID*), vidnost, vsebino znaka, prejšnji znak in naslednji znak. Identifikacijska številka je kazalec na znak in je sestavljena iz unikatne oznake dokumenta in lokalne ure oziroma števca. Primer identifikacijske številke je (558, 3). Unikatna oznaka dokumenta je številka, ki predstavlja uporabnika. Ko uporabnik začne prvič urejati dokument, se mu ta številka dodeli ali si jo izbere sam na podlagi predhodnega preverjanja unikatnosti. Lokalna ura je števec, ki se uporabniku povečuje za vsak vpisan znak. Na ta način je vsak znak v dokumentu označen z unikatnim kazalcem. Vidnost nam pove, ali uporabnik vidi določen znak. Pri brisanju se vidnost postavi na 0, sicer pa je vidnost pozitivna. Vsebina znaka je črka ali številka, ki jo znak predstavlja. Prejšnji

znak je identifikacijska številka levega soseda, naslednji pa identifikacijska številka desnega.

Poznamo tudi dva posebna znaka, ki označujeta konec in začetek dokumenta. Uporabljata se zato, da lahko prvemu znaku nastavimo prejšnjega in zadnjemu znaku naslednjega soseda. Alja in Bine na Sliki 3.24 urejata dokument, v katerem se trenutno nahaja besedilo "Ubogi mucek!". Dokument bi bil pri Alji in Binetu shranjen kot zaporedje znakov od U do klicaja.



Slika 3.24: Dokument, shranjen po protokolu WOOT.

Znaka (00000) in (11111) označujeta začetek oziroma konec dokumenta. Za vse ostale znake se v stolpcih od leve proti desni hranijo identifikacijska številka, vidnost, vsebina, prejšnji ter naslednji sosed. Iz identifikacijskih številk vidimo, da je prvo besedo skupaj s presledkom napisala Alja z unikatno številko 217. Drugo besedo skupaj s klicajem je napisal Bine. Njegov dokument je označen z unikatno številko 558. Vsak uporabnik bi si lahko shranjeval, katera unikatna številka predstavlja katerega uporabnika, vendar ni potrebno. Vse črke so vidne. Nobena črka do sedaj še ni bila pobrisana. Glede na oznake sosedov so črke pravilno razvrščene.

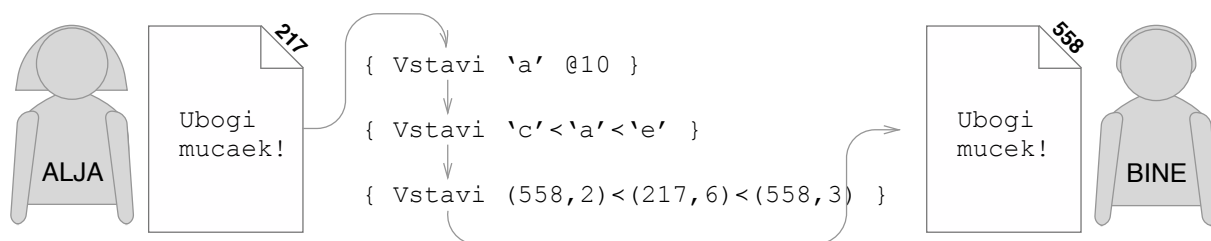
### 3.3.2 Operacije

Osredotočili se bomo na operaciji "Vstavi" in "Pobriši". Ko uporabnik generira operacijo, se operacija najprej integrira lokalno, nato se razpošlje vsem oddaljenim uporabnikom, ki sodelujejo pri urejanju. Ko jo sprejmejo, se integrira tudi njim v dokument.

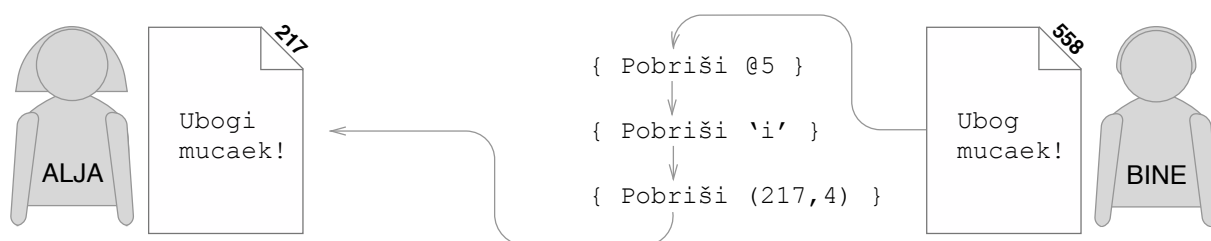
Potreba po vnaprejšnji lokalni integracija izvira iz razvrščanja znakov. Na primer, ko uporabnik generira spremembo { Vstavi 'a' @10 }, se le-ta v uporabniškem vmesniku prikaže kot črka a na lokaciji 10. Sprememba se mora pretvoriti v { Vstavi 'c' < 'a' < 'e' }, ki pomeni, vstavi črko a med črko c in e. Lokacija spremembe je tako definirana s svojima sosedoma in ne s konkretno številko lokacije.

Pretvorba je prikazana poenostavljeno. V dejanskem algoritmu bi sosednja dva znaka označil z njunima identifikacijskima številka. Podobno kot vstavljanje se mora tudi brisanje pretvoriti WOOT protokolu primerno. Sprememba { Pobriši @5 } pobriše znak na lokaciji 5. V primeru Alje in Bine imamo na lokaciji 5 črko i, zato moramo spremembo pretvoriti v { Pobriši 'i' }. Seveda je tudi ta primer poenostavljen, saj je tudi operacija brisanja vezana na identifikacijsko številko in ne na konkretno črko ali številko. Pomembno je poudariti, da znakov pri protokolu WOOT nikoli ne brišemo, ampak jih le skrivamo. Dokument je shranjen kot zaporedje uporabniku vidnih in uporabniku nevidnih znakov. Na ta način je razvrščanje spremembe vedno pravilno, saj je odvisno tudi od nevidnih znakov.

Prikažimo delovanje na primeru. Alja in Bine želita stavek "Ubogi mucek!" urediti v "Uboga muca!". Alja začne urejati drugo besedo in vstavi črko a. Bine začne urejati prvo besedo in pobriše črko i. Operaciji sta prikazani na Slikah 3.25 in 3.26.



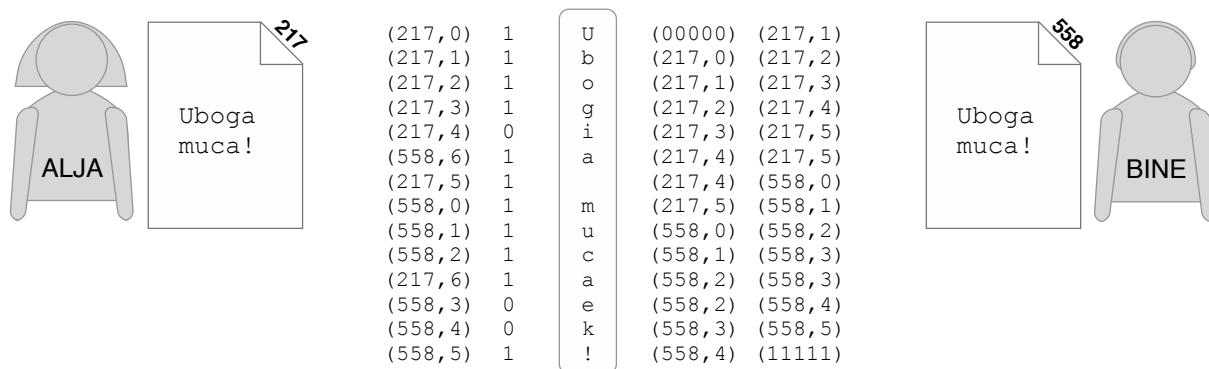
Slika 3.25: Generiranje in integracija vstavljanja črke. Zadnji znak (presledek), ki ga je Alja vstavila (na Sliki 3.24), ima identifikacijska številka (217, 5). Črka a zatorej dobi številko (217, 6). Vstaviti jo želimo na mesto med črko c in e, ki imata identifikacijski številki (558, 2) in (558, 3).



Slika 3.26: Generiranje in integracija brisanja črke. Na Sliki 3.24 vidimo, da je identifikacijska številka črke i enaka (217, 4). Operacija se pošlje Alji.

V naslednjem koraku Bine vstavi črko a z identifikacijsko številko (558,6) na podoben način, kot je to naredila Alja na Sliki 3.25. Alja pa pobriše črki e (558,3) in k (558,4) v drugi besedi. Operacija je podobna kot na Sliki 3.26, ko je to storil Bine. Končni stavek v dokumentu lahko vidimo na Sliki 3.27.

Čeprav so v dokumentu shranjene vse črke, ki sta jih napisala Alja in Bine, se jima v urejevalniku kažejo le črke, ki so označene kot vidne. Na Sliki 3.27 opazimo še eno zanimivost. Črka c (558,2) ima kot naslednjega soseda shranjeno črko e (558,3). Pravi naslednji сосед pa je v bistvu črka a (217,6). Ta pojav je običajen, ni napaka, je le posledica integracije vstavljanja.



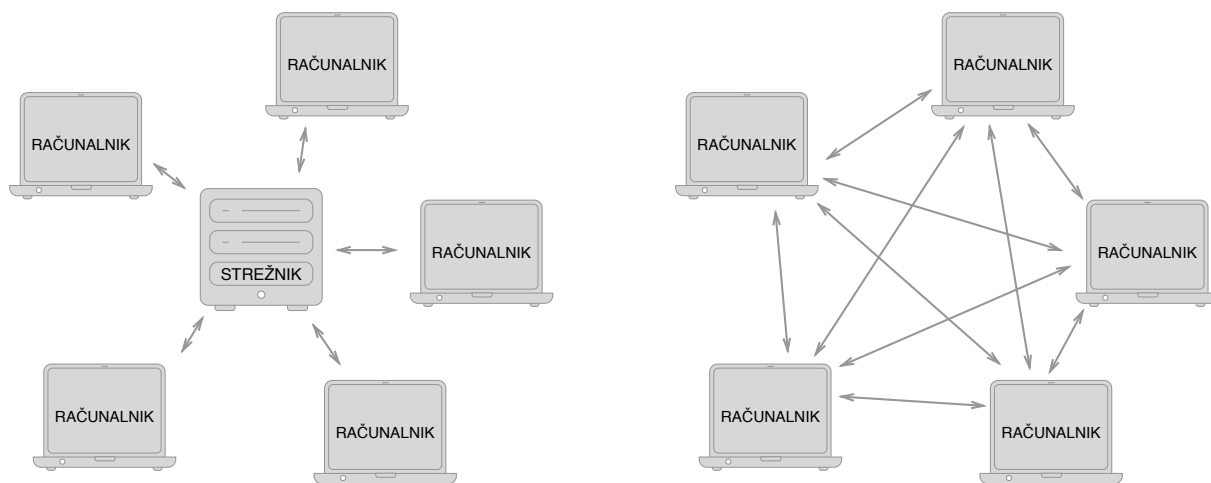
Slika 3.27: Nekaj črk v dokumentu je skritih in jih Alja in Bine ne vidita.

V našem enostvanem primeru nismo omenjali čakalne vrste. Ko uporabnik sprejme operacijo, mora iti ta najprej preko preverjanje predpogojev za izvršitev. Če predpogoj ni izpolnjen, se integracija operacije začasno prestavi nazaj med čakajoče operacije. Primer, da pogoj ni izpolnjen, je, ko uporabnik sprejme operacijo { Pobriši 'M' }, črka M pa v njegovem dokumentu še ne obstaja. Operacija brisanja črke M je prehitela vstavljanje črke M. Operacija brisanja bo integrirana v naslednji iteraciji.

### 3.3.3 Odjemalec-odjemalec sodelovanje

Protokola DS in OT, omenjena v prejšnjih podpoglavjih, temeljita na arhitekturi odjemalec-strežnik in sta od njih odvisna. Za distribucijo vsebine je bolj učinkovita arhitektura odjemalec-odjemalec (angl. *peer-to-peer*, kratica P2P). Ta potencial želimo izkoristiti tudi pri sodelovanju urejanja vsebine. Protokol WOOT je lahko implementiran popolnoma decentralizirano. Na primeru bomo videli, da ni odvisen od centralnih strežnikov. Razlika v arhitekturi omrežja je prikazana na Sliki 3.28.

Integracija brisanja pri oddaljenih uporabnikih je enostavna operacija. Pri njej se le postavlja vidnost na 0 ali 1. Pri integraciji vstavljanja pri oddaljenih uporabnikih lahko pride do problemov. Znak, ki se ga integrira, se mora postaviti direktno med dva sosednja znaka. Če so se na to mesto pred sprejemom operacije vrinili tudi znaki drugih uporabnikov, so potrebne primerjave z vrinjenimi znaki. Pomembno je, da se vedno izvede enaka strategija, ki zagotovi konsistentnost med uporabniki.



Slika 3.28: Na levi je model odjemalec-strežnik, na desni je model odjemalec-odjemalec.

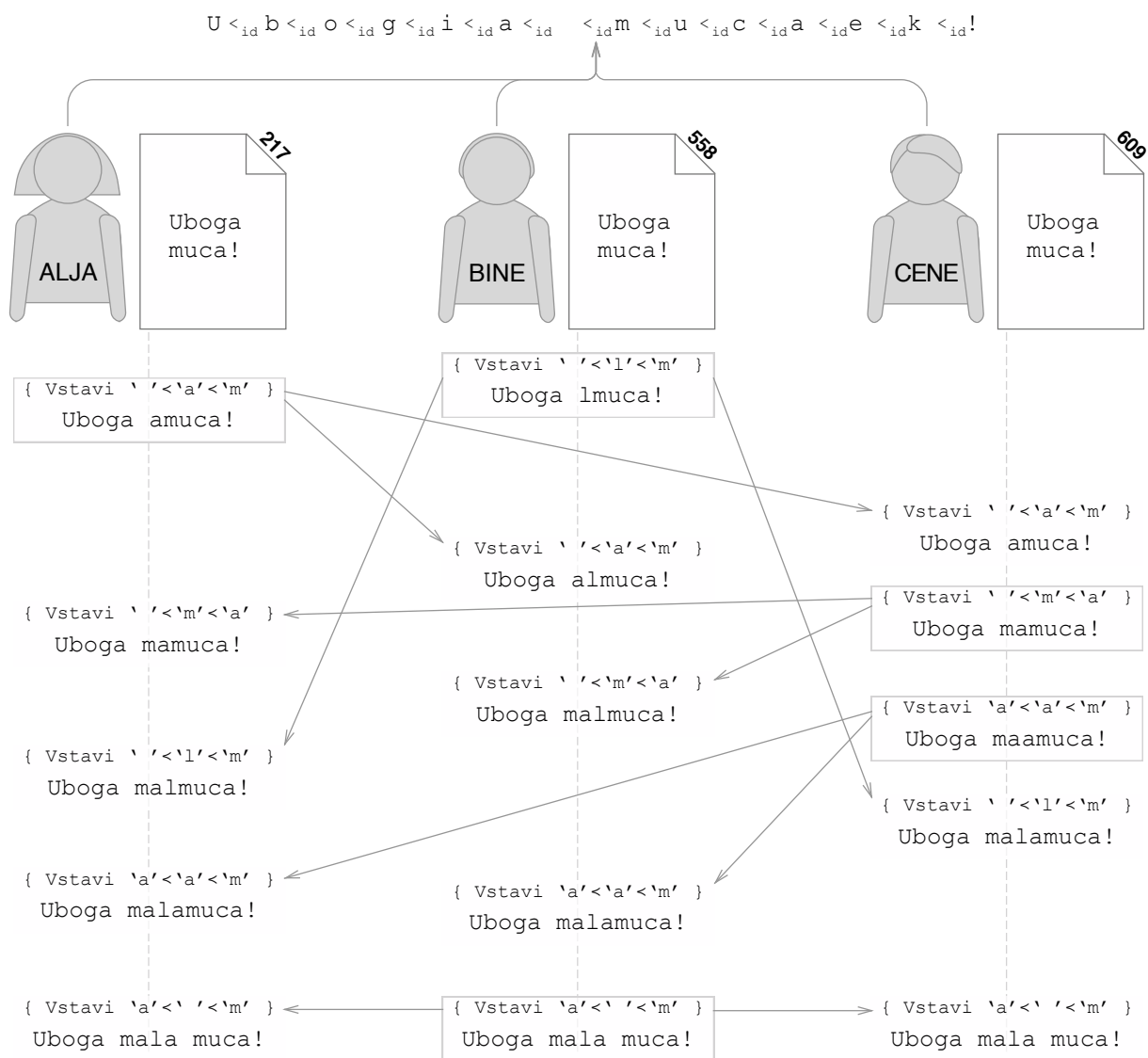
Našima dvema uporabnikoma se je pridružil uporabnik Cene. Vsi trije imajo v svojem urejevalniku dokument, v katerem piše "Uboga muca!". Vmes (za presledkom) bodo dopisali besedo "mala". Ker se želijo pri urejanju dokumenta tudi zabavati, se dogovorijo sledeče. Uporabnik Cene, ki se je urejanju pridružil zadnji, lahko vpiše dve črki, Alja in Bine pa vsak po eno črko. Vprašanje je, ali lahko na koncu dobimo pravilno oblikovan stavek "Uboga mala muca!".

Vsi hkrati začnejo urejati. Alja se odloči, da bo vpisala črko a, ker je to začetnica njenega imena. Bine je skoraj prepričan, da bo Alja vpisala črko a, zato sam raje vpiše črko l. Cene zase ve, da bo vpisal črko a, saj sta v besedi dve in je manj verjetnosti, da bo zamočil. Ko prejme Aljino črko a, se odloči, da bo pred njo vpisal črko m v upanju, da je ni vpisal že Bine. Na konec dopiše še črko a, za katero je bil že prej odločen, da jo bo vpisal.

Poglejmo na Sliko 3.29, kaj se je zgodilo. Alja in Bine sta hkrati vpisala črki a in l pred začetek besede "muca". Nato se je aktiviral Cene in dodal še njegov prispevek s črkami m in a. Da je stavek brez pravopisnih napak, je na koncu Bine vpisal še presledek. Ker zadnja sprememba ni nič posebnega, jo pri razmišljanju odmislimo. Operacije, označene v pravokotnikih, so se integrirale najprej lokalno, nato pa so bile poslone naprej po omrežju. Puščice prikazujejo pošiljanje operacije še drugima dvema uporabnikoma. Uporabniki so skupno naredili štiri vstavljanja črk m, a, l in a. Kljub temu, da se pri nobenem uporabniku črke niso vstavljale v tem zaporedju, vsi na koncu vidijo isti pravilen rezultat.

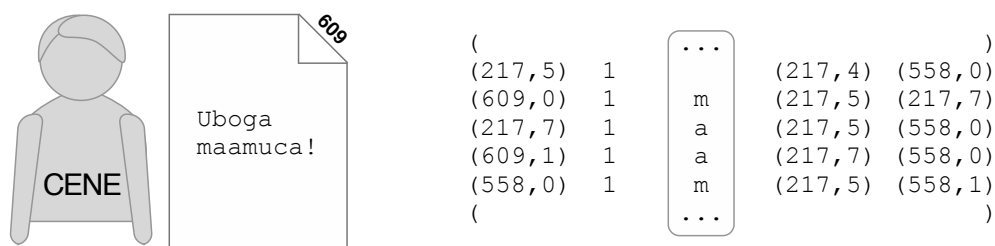
Za študijo primera vzemimo Ceneta. Najprej je od Alje prejel, da mora med presledek in črko m vpisati črko a. Ker sam ni še ničesar urejal, je to operacijo integriral brez





Slika 3.29: Prikaz sodelovanja s protokolom WOOT preko omrežja odjemalec-odjemalec. Vsak od uporabnikov je vpisal svojo izbrano črko oziroma črki.

težav. Nato je naredil dve lokalni integraciji. Pred prej od Alje vpisani a je vpisal črko m, za njim pa črko a. Ker je šlo za lokalno integracijo, jo je lahko izvedel brez težav. Cene je imel v tem trenutku v svojem dokumentu nesmiseln stavek "Uboga maamuca!". Nato je prejel Binetovo spremembo { Vstavi ' ' < '1' < 'm' }. Zmeden bralec bi lahko mislil, da je pravilna integracija črke l na lokacijo 7. Nastal bi stavek "Uboga lmaamuca!". Res je, da Cene sprejme informacijo, da mora locirati črko l med presledek in m, vendar je ta informacija podana z identifikacijskimi številkami in ne z alfanumeričnimi znaki, kot smo že omenili. Po premisleku nam je jasno, da mora biti črka l integrirana med presledek in drugi m v stavku, torej { Vstavi (217,5)<(558,7)<(558,0) }. Potemtakem so kar štiri možnosti, kam se lahko integrira l. Nastanejo lahko variante: "Uboga lmaamuca!", "Uboga mlaamuca!", "Uboga malamuca!" ali "Uboga maalmuca!". Kako urejevalnik ve, kam ga mora postaviti? Da bomo lažje razrešili ta problem, si pogledjmo Sliko 3.30.



Slika 3.30: Del Cenetovega dokumenta pred integracijo črke l.

Prikazane so konfliktne črke v Cenetovem dokumentu tik pred integracijo črke l (558, 7). Postopek je sledeč. Vzamemo seznam črk, ki so znotraj meje, med katere bi se moral integrirati l. Dobimo črke m (609, 0), a (217, 7) in a (609, 1). Iz tega seznama odstranimo črke, ki imajo prejšnjega ali naslednjega sosedo znotraj meje, med katere bi se moral integrirati l. Ostane nam samo črka a (217, 7). Primerjamo identifikacijski številki črke a in črke l. Ker je 217 manjše od 558, se bo črka l (558, 7) postavila desno od črke a (217, 7). Primerjavo moramo nadaljevati z vsemi črkami do meje. Naslednja po vrsti je črka a (609, 1). Ker je 558 manjše od 609, se bo črka l (558, 7) postavila levo od črke a (609, 1). Primerjanje je zaključeno. Pravilna oblika stavka je "Uboga malamuca!".

Ko se integrira še Binetov presledek, dobimo "Uboga mala muca!". Po podobnem postopku dobita tudi ostala dva uporabnika dokument z isto vsebino.

## Poglavje 4

# Primerjava protokolov za skupinsko urejanje

V Poglavju 3 smo spoznali glavne tri protokole in njihove algoritme za sodelovanje pri urejanju besedila v realnem času. Pozoren bralec je lahko ugotovil, da smo nekatere pomanjkljivosti določenega algoritma namenoma izpustili. Na drugi strani pa bi lahko nekatere prednosti bolje poudarili. Namen tega poglavja je, da jih primerjamo po njihovih lastnostih in predstavimo tako njihovo negativno kot pozitivno plat.

### 4.1 Hranjenje dokumenta

V vseh treh primerih vsak uporabnik ureja svoj lokalni dokument. Vendar je ta dokument skupen vsem uporabnikom. Ko uporabniki obmirujejo (nehajo tipkati ali kako drugače delati spremembe na dokumentu), bi se morali vsi dokumenti v najkrajšem možnem času poenotiti. Najsibo to preko centralnega strežnika v primeru DS in OT ali preko ostalih uporabnikov pri WOOT. Vsi uporabniki bi morali videti isti dokument, ne glede na to, kateri algoritem za sodelovanje v realnem času se uporablja. Obstaja pa bistvena razlika, kako je ta dokument v osnovi shranjen. Vprašamo se lahko, kaj se zgodi, ko se urejanju pridruži nov uporabnik. Na kakšen način se dokument, shranjen v svoji primarni obliki, prikaže v uporabniku berljivi (angl. *human readable*) obliki?

Pri DS se posamezen dokument za vse uporabnike hrani kot dokument strežnika. Za vsakega uporabnika, posebej pa na strežniku, obstajata še senca dokumenta strežnika in varnostna kopija sence strežnika. Na strani uporabnika imamo podobne tri dokumente, vendar so to lokalni dokumenti, ki se nahajajo pri uporabniku. Zanimivo je, da je dokument strežnika že v uporabniku berljivi obliki. Ko se urejanju pridruži

nov uporabnik, mu v njegovem grafičnem vmesniku takoj prikažemo vsebino dokumenta. V ozadju je potrebno poskrbeti, da se pravilno inicializirata senca in kopija sence strežnika. Pri uporabniku pa se morajo postaviti vsi trije dokumenti.

Pri OT smo v Poglavju 3.2.2 zapisali, da morata za posamezen dokument tako strežnik kot uporabnik hraniti trenutno stanje dokumenta. Kar je shranjeno v trenutnem stanju dokumenta v odjemalcu, je to, kar uporabnik dejansko vidi. Vendar je pri OT poudarek na spremembah. Najpomembnejša stvar na strežniku je revizijski dnevnik dokumenta in je enoten za vse uporabnike. V njem so kronološko shranjene vse spremembe, ki so jih naredili uporabniki. Da dokument prikažemo novemu uporabniku, le ponovimo revizijski dnevnik od začetka do konca [11]. Seveda se mora v uporabnikovem odjemalcu pripraviti, da si bo beležil spremembe in številko zadnje sinhronizirane revizije.

Pri primerjavi DS in OT lahko ugotovimo, da je hranjenje dokumenta v obeh primerih z nekaj modifikacijami precej podobno. Če bi dopolnili algoritem DS, da bi beležil spremembe, ki jih naredijo uporabniki, bi imeli tudi pri DS revizijski dnevnik dokumenta. Tudi na OT bi lahko naredili nekaj modifikacij. Novemu uporabniku bi kar takoj prikazali trenutno stanje dokumenta, kot ga hrani strežnik. Številko zadnje sinhronizirane revizije bi odjemalcu poslali posebej. Z omenjenima modifikacijama bi bilo hranjene dokumenta precej podobno.

WOOT deluje v arhitekturi odjemalec-odjemalec in ne uporablja centralnega strežnika. Da bi se urejanju pridružil nov uporabnik, mora biti za to poskrbljeno preko kakega drugega sistema ali preko souporabnika v obliki povabila (angl. *request*). Posebnost WOOT je tudi to, kako se uporabniku prikaže dokument. Kot smo omenili v Poglavju 3.3.1, za vsak znak v dokumentu hranimo peterko podatkov. Ko želimo novemu uporabniku prikazati dokument v uporabniku berljivi obliki, se naredi ena iteracija preko vseh znakov. Pobrisanih znakov (označenih kot skritih) se uporabniku ne prikaže. Vse ostale znake se prikaže glede na to, katero črko ali številko predstavljajo. V bistvu se ta iteracija naredi vedno, ko uporabnik naredi ali prejme spremembo v dokumentu. Le tako je lahko dokument vedno v pravi obliki.

Slabost WOOT v primerjavi z DS in OT je, da se znaki v dokumentu nikoli ne brišejo, le skrivajo se. V primeru, da se pri urejanju dokumenta med uporabniki veliko tudi briše, velikost dokumenta hitro raste.

## 4.2 Struktura porazdelitve

Ko govorimo o strukturi porazdelitve, govorimo o arhitekturi omrežja. Poznamo model odjemalec-strežnik (angl. *client-server*) in model odjemalec-odjemalec (angl. *P2P*). Pri modelu odjemalec-strežnik vsak posamezen odjemalec (uporabnik) centralnemu strežniku pošilja zahteve, medtem ko si pri modelu odjemalec-odjemalec med seboj povezane točke (uporabniki) izmenjujejo vire brez uporabe centraliziranega strežnika [14, 15]. Njuna aritektura je prikazana na Sliki 3.28.

Oba modela smo že omenili, ko smo opisovali izbrane tri protokole ter algoritme. Avtorji algoritma WOOT so se naslanjali na dejstvo, da je model odjemalec-odjemalec boljši za porazdeljevanje vsebine. Ta potencial so želeli izkoristiti in tako WOOT v osnovi deluje na modelu odjemalec-odjemalec, čeprav bi ga bilo možno realizirati tudi na modelu odjemalec-strežnik. Za DS in OT smo rekli, da temeljita na modelu odjemalec-strežnik. Pri OT to ni čisto res. Hipotezo ovrže prvi članek o OT [1], saj pravi: "Za sisteme za skupinsko delo v realnem času je značilno, da so porazdeljeni (angl. *distributed*). Na splošno ni mogoče domnevati, da so vsi udeleženci povezani z istim strojem ali da so v istem lokalnem omrežju." Glede na potrebe po rešitvi zapletenih stanj, ki lahko nastanejo, če poleg "Vstavi" in "Pobriši" dodamo še bolj zapletene spremembe, so se v naslednjih raziskavah OT [2] začele pojavljati rešitve, zasnovane na modelu odjemalec-strežnik. Izkazale so se za boljše, saj zmanjšujejo verjetnost napak pri zapletenih spremembah na dokumentu.

Torej imamo DS na modelu odjemalec-strežnik, OT na modelu odjemalec-strežnik pri zapletenih spremembah oziroma na modelu odjemalec-odjemalec pri enostavnih spremembah ter WOOT na modelu odjemalec-odjemalec ali na modelu odjemalec-strežnik. Vprašanje je, kaj je boljše. Od česa je sploh odvisno, kateri model je boljši? Če med sabo primerjamo modela po zanesljivosti (angl. *reliability*) in skalabilnosti (angl. *scalability*), je model odjemalec-odjemalec zmagovalec. Pri skalabilnosti je neomejen. Zanesljivost pa je zelo visoka. Če en souporabnik odpove, so na voljo še vsi ostali souporabniki, na katere se uporabnik lahko priklopi. Pri modelu odjemalec-strežnik je zgodba drugačna. Skalabilnosti je slaba, saj je število povezav na en strežnik omejeno. Tudi z zanesljivostjo ni drugače. Ko odpove strežnik, odpovejo vsi souporabniki.

Ker sta DS in OT močno odvisni od modela odjemalec-strežnik, obstajajo raziskave [10] na temo izboljšanja zanesljivosti in skalabilnosti. Ideja je, da med sabo povežemo več strežnikov, ki si izmenjujejo spremembe v dokumentu ravno tako, kot si jih izmenjujejo odjemalec in strežnik. Tako hkrati izboljšamo skalabilnost in zanesljivost. Več uporabnikov se lahko pridruži urejanju. V primeru izpada enega strežnika, se lahko

prevežejo na drugega.

Končna ugotovitev je, da so DS, OT in WOOT enakovredne po strukturi porazdelitve, saj sta model odjemalec-strežnik in model odjemalec-odjemalec primerljiva po zanesljivosti in skalabilnosti.

### 4.3 Počasna povezava

Vsi trije algoritmi zagotavljajo sodelovanje v realnem času. Nismo pa omenili, kaj se dogaja pri visoki zakasnitvi prenosa (angl. *latency*). Na primer, ko imajo uporabniki počasno povezavo ali ko je oddaljenost med uporabniki ekstremno velika.

DS je omejena na največ en sinhronizacijski paket v enem sinhronizacijskem ciklu v danem trenutku [10]. Uporabnik lahko v sinhronizacijskem paketu pošlje več sprememb. Vendar to problema ne reši. V primeru, da se uporabnik nahaja na Marsu, strežnik pa na Zemlji, je čas enega sinhronizacijskega cikla pol ure. To pomeni, da uporabnik pošlje svoje spremembe, potem pa kar pol ure ne dobi odgovora od strežnika. V tem času sicer lahko tipka v svoj dokument, vendar od strežnika ne dobi ne povratne informacije o uspešnem prejemu sprememb ne novih sprememb od drugih uporabnikov, kar je problem. Če naredijo ostali uporabniki korenite spremembe na skupnem dokumentu, se lahko zgodi, da se spremembe uporabnika na Marsu v zadnje pol ure pobrišejo brez možnosti povrnitve.

Podobno kot pri DS je tudi pri OT pravilo, da se na strežnik ne pošlje več kot ene čakajoče spremembe. Na seznamu poslanih sprememb je največ ena sprememba, ki še ni potrjena s strani strežnika. Algoritem za pošiljanje sprememb bi sicer lahko prilagodili, da bi poslal več sprememb v paketu, tako kot je to predvideno pri DS s sinhronizacijskim paketom. Prilagoditev bi bilo potrebno narediti tudi na strežniku, vendar ne reši problema pri počasni povezavi. Uporabnik na Marsu, ki bi pošiljal spremembe na Zemljo vsake pol ure, bi lahko dokument izmaličil za vse uporabnike. Kljub transformacijam se lahko v pol ure naredi preveč sprememb. Če uporabnik na Marsu naredi spremembo "Vstavi" na isti lokaciji kot drug uporabnik na Zemlji, se bo želena lokacija spremembe transformirala v nezaželeno. Nastalo napako uporabnika hitro rešita, vendar uporabnik na Marsu potrebuje nadaljnje pol ure za popravek. Ponovno lahko pride do napake, če želita napako popraviti oba uporabnika. Skratka, pri počasni povezavi, bi bilo preveč napak.

Če sta DS in OT zelo odvisni od potrditev sprememb s strani strežnika, je pri WOOT precej drugače. Že v osnovi WOOT ne temelji na centralnem strežniku in se

potrditev ne predvideva. Komunikacija je rešena kot stalen tok posodobitev med vsemi uporabniki. Poleg tega so spremembe vezane na identifikacijske številke, integracija sprememb pri oddaljenih uporabnikih pa ni odvisna od časa dostave. Če pa se že zgodi, da brisanje znaka prehití vstavljanje znaka, potem se sprememba "Pobriši" postavi v čakalno vrsto. Integrira se v enem od naslednjih integracij. V primeru enega uporabnika na Marsu in drugega uporabnika na Zemlji bo protokol vse spremembe brez težav pravilno integriral.

Pri visoki zakasnitvi prenosa je torej protokol WOOT boljši kot DS in OT. Po drugi strani pa se lahko vprašamo v čem je smisel implementirati sodelovanje v realnem času med uporabnikom na Marsu in uporabnikom na Zemlji. Zaradi velike oddaljenosti lahko dostava in odgovor paketa trajata pol ure. Nekako se zdi, da je nesmiselno uporabljati urejevalnik v realnem času, če je uporabnik na Marsu pol ure v zaostanku v primerjavi z ostalimi uporabniki na Zemlji. Teoretično je WOOT res boljši kot DS in OT, vendar je na Marsu ravno tako neuporaben kot ostala dva. Če bi že hoteli izboljšati uporabniško izkušnjo na Marsu, bi morali iskati rešitev v izboljšanju infrastrukture, ne pa v izboljšanju algoritma.

Na Zemlji so vsi trije algoritmi nekako enakovredni. Bolj problematičen od počasne povezave je izpad povezave, kar se ne zgodi redko. Uporabnik je o nedostavljenih spremembah obveščen preko časovne omejitve (angl. *timeout*). Kako to zgleda v praksi? Pri DS in OT se spremembe, ki po določenem času niso potrjene s strani strežnika, obravnavajo kot nedostavljene in se jih pošlje še enkrat. Pri WOOT pa je nekoliko drugače. Ker temelji na omrežju odjemalec-odjemalec, potrjevanje ni potrebno. Če se zgodi, da sprememba ni dostavljena, ni težav, sej korenito ne vpliva na vse nadaljnje spremembe. Vsi uporabniki imajo še vedno podoben dokument, le da nekomu kakšen del manjka. To pa se lahko reši s posebnim preverjanjem, ki preveri, ali je uporabnikov dokument konsistenten z dokumentom drugega uporabnika.

## 4.4 Zaznavanje sprememb

Pri razlagi protokolov za urejanje besedila v realnem času smo zapisali, da delajo uporabniki spremembe v besedilu. Osredotočili smo se na spremembi "Vstavi" in "Pobriši", vendar obstajajo še druge, ki jih v diplomski nalogi nismo zajeli. Dejstvo pa je, da pojma "sprememba" nismo nikoli natančno definirali. Govorili smo le o tem, da uporabnik naredi spremembo. Razlog je, da uporabljajo različni akademski članki različno terminologijo za poimenovanje uporabnikove interakcije z urejevalnikom be-

sedila. Druga težava pa je, da predvidevajo različni protokoli različno zaznavanje sprememb in jih tudi različno obravnavajo.

Ko uporabnik ureja besedilo v svojem dokumentu, sta dve možnosti, ali zaznavamo vse njegove operacije preko poslušalcev dogodka (angl. *event listeners*) ali pa iščemo v besedilu nastale spremembe. Pri primitivnih implementacijah so bili poslušalci dogodkov precej popularni. Zaznavanje pisanja in brisanja golega besedila je precej enostavno na katerikoli platformi. Poleg tipkanja lahko uporabnik naredi tudi operacije, kot so: izreži besedilo, prilepi besedilo, povleci in spusti besedilo, zamenjava besedila, povrni besedilo, ... Operaciji, kot sta samodokončanje in pravopisni popravek, lahko naredi tudi sam urejevalnik. Ko dodamo osnovnima dvema operacijama še bolj zapletene, postanejo poslušalci dogodkov preveč zapleteni. Včasih se jih niti ne da implementirati. Tako so se vedno bolj začeli uveljavljati algoritmi za iskanje razlik med besediloma. Kako delujejo, bomo opisali v Poglavju 5. Na tem mestu povejmo le to, da je njihov glavni namen iskanje sprememb v besedilu. Spremembe, ki jih algoritmi najdejo, so posledica uporabnikove interakcije z urejevalnikom. Njihova dobra lastnost je, da so v osnovi implementirani povsem neodvisno od uporabnika. Vseeno pa najdejo vse spremembe, ki nastanejo ob uporabnikovih operacijah v besedilu, in tudi vse spremembe, ki nastanejo ob operacijah samega urejevalnika.

Poglejmo si, kako je z obravnavanimi protokoli. Pri DS smo že v osnovni topologiji na Sliki 3.1 omenili tri entitete, in sicer razliko, spremembo in popravek. To nakazuje, da predvideva DS uporabo algoritma za iskanje razlik v besedilu. Najdene spremembe se kot popravki uveljavijo na oddaljenem računalniku. Pri OT o nastanku sprememb nismo kaj dosti govorili, smo pa poudarili, da so spremembe njen pomemben del, saj je dokument shranjen kot revizijski dnevnik sprememb. Kako implementirati zaznavanje spremembe v praksi, je programerjeva odločitev. Potrebno je biti le pazljiv, da se že poslanih sprememb ne pošlje še enkrat. Pri WOOT se spremembe običajno imenujejo operacije, ker to v bistvu tudi so. Če se uporabnikove operacije zaznava preko poslušalcev dogodkov, je posamezna operacija tudi že sprememba. Na tak način pa naj bi WOOT tudi delovale. Urejevalnik mora biti sposoben zaznavati uporabnikove operacije kot spremembe, jih integrirati v lokalni dokument in jih poslati ostalim uporabnikom za integracijo v njihovih dokumentih.

Seveda bi se dalo tudi WOOT implementirati z algoritmom za iskanje razlik v besedilu. Vendar obstaja bistvena razlika v primerjavi z DS in OT. Pri DS in OT so spremembe, ki vključujejo več znakov skupaj, nekaj povsem običajnega. Pri WOOT pa je vsaka sprememba vezana na posamezen znak. Večznakovne spremembe bi vodile v napake pri delovanju. Recimo, da želimo besedo "pot" spremeniti v besedo "pilot".



Pri DS in OT bi strežniku poslali le eno spremembo, in sicer { Vstavi 'il' @2 }. Pri WOOT bi morali ostalim uporabnikom poslati dve spremembi, in sicer { Vstavi 'p' < 'i' < 'o' } in nato { Vstavi 'i' < 'l' < 'o' }. Drobljenje sprememb še dodatno zakomplicira implementacijo urejevalnika, zato se zdi, da je za WOOT bolj primerno uporabiti poslušalce dogodkov.

Ponovno ni konkretnega zmagovalca med protokoli glede zaznavanja sprememb. Po našem mnenju je pri DS in OT najbolj primerno uporabiti algoritme za iskanje razlik med besedilom, pri WOOT pa je bolj primerno uporabiti poslušalce dogodkov. Katerega je bolj smiselno implementirati, je odvisno od platforme in od predvidene dolžine besedila.

## 4.5 Problem sočasnosti

O problemu sočasnosti smo govorili na začetku naše diplomske naloge v Poglavju 2.1. Do težave pride, ko več uporabnikov v istem času na isti lokaciji naredi neko spremembo. Za vzdrževanje konsistentnosti potrebujemo algoritme za nadzor sočasnosti, kot so DS, OT in WOOT. Algoritme za nadzor sočasnosti lahko klasificiramo kot pesimistične ali optimistične [2]. Pesimistični algoritem potrebuje vnaprej komunikacijo s centralnim strežnikom ali s souporabnikom. Ko uporabnik naredi spremembo, mora počakati na povratno informacijo ali je s spremembo vse v redu ali ni. Če ni težav, se uveljavi nova sprememba na dokumentu. Sicer mora uporabnik popraviti svojo spremembo in poskusiti ponovno. Optimistični algoritem deluje ravno nasprotno. V tem primeru uporabnik naredi spremembo in jo pošlje na strežnik ali souporabniku. Naloga strežnika oziroma souporabnika je, da to spremembo umesti v dokument na način, da se med souporabniki ohrani konsistentnost. Pesimistični nadzor pri urejanju golega besedila v realnem času ne pride v poštev. V praksi ga po navadi najdemo v kombinaciji z zaklepanjem dokumenta. Vsi trije naši protokoli temeljijo na optimističnem nadzoru sočasnosti. Kljub učinkovitosti protokolov se vseeno najde kakšna nezaželjena zadeva.

Poglejmo si specifičen primer sočasnega brisanja istega znaka v dokumentu pri uporabi OT. Recimo, da imamo dokument z nekaj deset znaki. Urejata ga uporabnika Alja in Bine. Oba naenkrat pobrišeta četrti znak in v istem času odpošljeta svoji spremembi { Pobriši @4 }. Na strežnik bosta obe spremembi prišli hkrati, vendar se bosta sprocesirali ena za drugo. Recimo, da se bo najprej sprocesirala Aljina sprememba. Pobrisala bo četrti znak. Alji bo nazaj poslana potrditev o uspešnem izbrisu,

Binetu pa zahteva za brisanje četrtega znaka. Nato začne strežnik procesirati Binetovo spremembo, v kateri piše, naj se pobriše četrti znak. Res se pobriše trenutno četrti znak, ampak to ni isti znak, ki ga je izbrisal Bine v svojem lokalnem dokumentu. Ne strežniku se dejansko pobriše znak, ki ga ima Bine v svojem lokalnem dokumentu na petem mestu. Potrditev o uspešnem izbrisu se pošlje Binetu, Alji pa zahteva za brisanje četrtega znaka. Podobno kot na strežniku se tudi tako v Aljinem kot Binetovem lokalnem dokumentu še enkrat pobriše četrti znak. Končni rezultat v vseh dokumentih sta pobrisana četrti in peti znak, čeprav sta Alja in Bine želela pobrisati le četrti znak.

Še en primer ne najboljše rešitve algoritma je pri protkolu WOOT. Predstavljajmo si uporabnike Aljo (217), Bineta (558) in Ceneta (609), ki urejajo dokument. Številka v oklepaju predstavlja posameznega uporabnika. Alja in Bine istočasno na isto mesto vpišeta vsak svojo začetnico imena. Alja naredi spremembo { Vstavi 't' < 'a' < 'd' }, Bine pa { Vstavi 't' < 'b' < 'd' }. Ne pozabimo, da nosi sprememba informacijo, med katerima dvema znakoma se mora vstaviti uporabnikov znak. Kot vidimo, se vstavljanje integrira med črki "t" in "d". Vsak pošlje svojo spremembo Cenetu. On obe spremembi sprejme. Najprej začne procesirati Binetovo spremembo, vstavljanje črke "b". Le-ta se integrira brez težav. Nato začne procesirati Aljino spremembo, vstavljanje črke "a". Vstaviti se mora med črki "t" in "d", med katerima dvema se že nahaja "b". Črka "a" bi se lahko vstavila pred "b" ali za "b". Ker je Aljina številka 217 manjša od Binetove 557, se črka "a" vstavi pred črko "b". Rešitev je sicer enostavna, vendar daje ta način vedno prednost uporabnikom z nižjo številko pred tistim z višjo. Privilegiranost enega uporabnika pred drugimi bi lahko rešili z uro oziroma s časovnimi žigi. Pri integraciji sprememb pri oddaljenem uporabiku bi lahko istoležeče spremembe razvrščali glede na vrstni red procesiranja. Kasneje sprocesirani znak bi moral ležati bolj desno. To je sicer rešitev z enakopravnim razvrščanjem sprememb, vendar nam podre konsistentnost pri drugih uporabnikih, zato je rešitev neuporabna. Druga rešitev je, da uporabnik sam pošlje, kdaj je naredil spremembo. Skupaj s spremembo se torej pošlje tudi časovni žig spremembe. Istoležeče spremembe se bi tako razvrščale glede na časovni žig. To zahteva natančno usklajenost ure pri vseh oddaljenih uporabnikih. Že to je precej težko doseči, poleg tega pa lahko uporabnik tudi goljufa, kdaj je naredil spremembo. Tako si ustvari privilegiran položaj.

Kot vidimo, ima OT težave s sočasnostjo istoležečih sprememb pri kombinaciji "Pobriši-Pobriši", WOOT pa pri kombinaciji "Vstavi-Vstavi". DS nima izrazitih težav.

## 4.6 Zahtevnost protokola

Zahtevnost protokola oziroma algoritma je širok pojem. Objektivno gledano bi lahko ta pojem zajel, koliko procesorske moči porabi posamezen algoritem. Eden izmed načinov merjenja zahtevnosti je, koliko časa bi potrebovali za njegovo implementacijo. Veliko bolj subjektivna ocena je, kako lahko oziroma kako težko je algoritem razumeti. Ljudje imamo različno predznanje. Nek algoritem je za nekoga lažje razumljiv, medtem ko je isti algoritem za drugega težje razumljiv. Če pa imamo statistične podatke dovolj velikega vzorca anketirancev, lahko iz njega povzamemo objektivne podatke. Naredil sem manjšo raziskavo med prijatelji, ki imajo podobno računalniško znanje, kot ga imam jaz. Navodila so bila sledeča: "V tretjem poglavju moje diplomske naloge na primeru opisujem protokole oziroma algortime za sodelovanje v realnem času. To so Diferenčna sinhronizacija, Operativna transformacija in Brez operativne transformacije. Potreboval bi tvojo pomoč. Cilj je, da si prebereš tretje poglavje in na koncu primerjaš protokole po zahtevnosti (razumljivosti). Najtežjemu daj 3 točke, srednje težkemu 2 točki in najlažjemu 1 točko." Dobil sem 14 razporeditev 14 ljudi. Točke, ki so jih prejeli posamezni protokoli, si sledijo: DS 37 točk, OT 29 točk in WOOT 18 točk. To nam pove, da je algoritem DS najbolj zahteven za razumevanje, OT lažji in WOOT najlažje razumljiv. Upoštevati moramo, da so dobljene točke močno odvisne od kakovosti besedila v Poglavju 3. Tudi vzorec 14 anketirancev je res zelo majhen, verjetno statistično še neznačilen, vendar kaže razlike glede zahtevnosti. Tudi moje subjektivno mnenje je, da je vrstni red zahtevnosti protokolov isti, kot sem ga dobil v moji raziskavi.



# Poglavje 5

## Iskanje razlik

Algoritmi za iskanje razlik v besedilu se uporabljajo za zaznavanje sprememb, ki nastanejo, ko uporabnik ureja besedilo. Preprost primer razlike med dvema besediloma je prikazan na Sliki 5.1.

Izvirno besedilo:	Ledolomilec
Spremenjeno besedilo:	Letalonosilka
Razlika:	<del>L</del> <del>e</del> <del>d</del> <del>e</del> <u>t</u> <u>a</u> <u>l</u> <u>o</u> <u>n</u> <u>o</u> <u>s</u> <u>i</u> <u>l</u> <u>e</u> <u>e</u> <u>k</u> <u>a</u>

Slika 5.1: Primer razlike. Prečrtane črke so črke, ki so bile v izvirnem besedilu izbrisane, podčrtane črke pa dodane.

Raziskave na tem področju so se začele že pred pojavom sodelovanja v realnem času na spletu in so jih obširno proučevali [16, 17, 18] (kasneje [19]). Namen vseh raziskav je bil, da se izboljša časovna zahtevnost algoritma za iskanje razlik. V čem je problem? Medtem ko uporabnik tipka, se morajo spremembe v najkrajšem možnem času poslati na strežnik. Če bi bil algoritem počasen in bi zamujal več sekund ali celo minut, bi bila uporabniška izkušnja zelo slaba. Pri večminutni zamudi že težko govorimo o sodelovanju uporabnikov v realnem času.

Besedilo, ki je skupno tako izvirnemu kot tudi spremenjenemu besedilu, imenujemo najdaljše skupno zaporedje (angl. *the longest common subsequence*). V našem primeru to zaporedje sestavljajo črke "Leloil". Na Sliki 5.1 so to črke v razliki, ki niso ne prečrtane ne podčrtane. Med izvirnim in spremenjenim besedilom lahko izračunamo tudi najmanjšo razdaljo urejanja (angl. *minimum edit distance*), poimenovano tudi Levenshteinova razdalja (angl. *Levenshtein distance*). To je številka, ki pove, koliko sprememb bi morali narediti na izvirnem besedilu, da dobimo spremenjeno besedilo. Najmanjša razdalja urejanja našega primera je 12. Na Sliki 5.1 lahko vidimo pet prečrtanih in sedem podčrtanih črk. Skupaj torej 12 sprememb, petkrat brisanje in sedemkrat vsta-

vljanje. Na podlagi najdaljšega skupnega zaporedja in najmanjše razdalje urejanja lahko predvidimo scenarij najkrajšega urejanja (angl. *shortest edit script*). To bi bili { Pobriši 'do' }, { Vstavi 'ta' }, { Pobriši 'm' }, { Vstavi 'nos' }, { Pobriši 'ec' }, { Vstavi 'ka' }. Ali so to res operacije, kot jih je naredil uporabnik, ne vemo. Dejstvo pa je, da so to spremembe v besedilu. Uporabnik bi lahko v celoti pobrisal besedo "Ledolomilec" in na novo napisal "Letalonosilka". Problem, ki ga rešujemo, je zagotavljanje konsistentnosti med uporabniki. Ne zanima nas, katere operacije so se zgodile v urejevalniku. Ni pomembno, kako je uporabnik uredil besedilo, pomembna je njegova vsebina. Zato je dovolj, da ugotavljamo le, kakšne so spremembe v besedilu in ne, kako smo prišli do njih.

V nadaljevanju bomo na primeru prikazali, kako poisakati **nadaljše skupno zaporedje**, kako izračunati **najmanjšo razdaljo urejanja** in kako priti do **scenarija najkrajšega urejanja**.

## 5.1 Najdaljše skupno zaporedje

Iskanje najdaljšega skupnega zaporedja bomo prikazali tabelarično [20], kot je prikazano na Sliki 5.2.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		L	e	t	a	l	o	n	o	s	i	l	k	a
1	L	0												
2	e	0												
3	d	0												
4	o	0												
5	l	0												
6	o	0												
7	m	0												
8	i	0												
9	l	0												
10	e	0												
11	c	0												

Slika 5.2: Priprava na iskanje najdaljšega skupnega zaporedja.

Horizontalno na vrh tabele vpišemo daljšega od besedil, ki ju primerjamo. V našem primeru je to spremenjeno besedilo "Letalonosilka". Ima  $n=13$  znakov. Vsak znak je označen s številko od 1 do 13. Vertikalno ob tabeli vpišemo krajšega od besedil, to je izvirno besedilo "Ledolomilec". Ima  $m=11$  znakov. Vsak znak je označen s številko od 1

do 11. Zgornjo ničto vrstico in levi ničti stolpec zapolnimo z ničlami. To lahko naredimo zaradi logičnega razloga. Če katerokoli besedilo primerjamo s praznim besedilom, ti dve besedili nimata skupnega zaporedja.

Postopek za iskanje najdaljšega skupnega zaporedja se prične v zgornjem levem polju  $(1, 1)$  in se nadaljuje do konca vrstice. Nato gre v naslednjo vrstico, se ponovi za vse vrstice in se konča v spodnjem desnem polju  $(11, 13)$ . V vsakem polju v tabeli se primerja črki, ki pokrivata to polje. Če sta črki enaki, se v polje vpiše za eno večje število, kot je v zgornje-levo ležečem polju. Med ta dva polja se vpiše še tako imenovani mostiček (angl. *bridge*). Če sta črki različni, se v polje vpiše večjega od zgornje ali levo ležečega polja.

Vsa polja  $(i, j)$  v tabeli bomo zapolnili s številkami od 0 do števila znakov, ki se ujemajo v obeh besedilih. Začnimo v polju  $(1, 1)$ . Primerjamo črki "L" in "L". Ker sta črki enaki in je v zgornje-levo ležečem polju  $(0, 0)$  ničla, se v polje  $(1, 1)$  vpiše 1. Med polja  $(0, 0)$  in  $(1, 1)$  vrišemo mostiček. Nadaljujemo s poljem  $(1, 2)$ . Črki "L" in "e" sta različni. Vrednost levo ležečega polja  $(1, 1)$  je 1, kar je več od zgornje ležečega polja  $(0, 2)$ , ki je 0. V polje  $(1, 2)$  se torej vpiše vrednost 1. Slika 5.3 prikazuje izpolnjeni prvi dve polji.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		L	e	t	a	l	o	n	o	s	i	l	k	a
1	L	0	1	1										
2	e	0												
3	d	0												
4	o	0												
5	l	0												
6	o	0												
7	m	0												
8	i	0												
9	l	0												
10	e	0												
11	c	0												

Slika 5.3: Izgled tabele po prvih dveh korakih.

Ker se črka "L" v besedi "Letalonosilka" ne ponovi več, je v vseh poljih v prvi vrstici vrednost 1. Nadaljujemo v drugi vrstici. Pričnemo v polju  $(2, 1)$ . Ker sta črki "e" in "L" različni, se v polje  $(2, 1)$  vpiše večja od vrednosti levo ležečega polja  $(2, 0)$  0 in zgornje ležečega polja  $(1, 1)$  1, torej 1. Postopek se nadaljuje v polju  $(2, 2)$ . Črki "e" in "e" sta enaki. Vrednost zgornje-levo ležečega polja  $(1, 1)$  je 1. V polje  $(2, 2)$  se

vpiše za 1 večja vrednost, torej 2. Doriše se mostiček. Glej Sliko 5.4.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		L	e	t	a	l	o	n	o	s	i	l	k	a
0		0	0	0	0	0	0	0	0	0	0	0	0	0
1	L	0	1	1	1	1	1	1	1	1	1	1	1	1
2	e	0	1	2										
3	d	0												
4	o	0												
5	l	0												
6	o	0												
7	m	0												
8	i	0												
9	l	0												
10	e	0												
11	c	0												

Slika 5.4: Pri primerjavi besedil smo našli že dve skupni črki.

Po opisanem postopku nadaljujemo čez vse vrstice. Pazljivi moramo biti pri črkah, ki se v besedi ponovijo dvakrat, kot recimo črka "o". Izpolnjena tabela je na koncu taka, kot je prikazano na Sliki 5.5.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		L	e	t	a	l	o	n	o	s	i	l	k	a
0		0	0	0	0	0	0	0	0	0	0	0	0	0
1	L	0	1	1	1	1	1	1	1	1	1	1	1	1
2	e	0	1	2	2	2	2	2	2	2	2	2	2	2
3	d	0	1	2	2	2	2	2	2	2	2	2	2	2
4	o	0	1	2	2	2	3	3	3	3	3	3	3	3
5	l	0	1	2	2	3	3	3	3	3	3	4	4	4
6	o	0	1	2	2	3	4	4	4	4	4	4	4	4
7	m	0	1	2	2	3	4	4	4	4	4	4	4	4
8	i	0	1	2	2	3	4	4	4	4	5	5	5	5
9	l	0	1	2	2	3	4	4	4	4	5	6	6	6
10	e	0	1	2	2	3	4	4	4	4	5	6	6	6
11	c	0	1	2	2	3	4	4	4	4	5	6	6	6

Slika 5.5: Izpolnjena tabela po postopku za iskanje najdaljšega skupnega zaporedja.

V spodanjem desnem polju  $(m, n)$ , to je  $(11, 13)$ , se nahaja vrednost 6. To pomeni, da je v besedah, ki smo ju primerjali, najdaljše skupno zaporedje dolgo 6 znakov. Vprašanje je, kateri znaki ga sestavljajo. Opazimo nekaj zanimivega. Ustvarili smo otočke (angl. *islands*) števil od 1 do 6. Na Sliki 5.6 smo jih poudarili z debelejšo črto,



ki predstavlja mejo otkov. Ti otki so povezani v mostiki. Da bo bomo izvedeli, kateri znaki sestavljajo najdalje skupno zaporedje, bomo morali potovati od spodnjega desnega polja vse do zgornjega levega polja. Premikamo se v levo vse do leve meje otka, nato navzgor vse do zgornje meje otka. Pri mostiku prestopimo mejo in ponovimo postopek. Na ta naan pridemo vse do zgornjega levega polja. Glej Sliko 5.6.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		L	e	t	a	l	o	n	o	s	i	l	k	a
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	L	0	1	1	1	1	1	1	1	1	1	1	1	1
2	e	0	1	2	2	2	2	2	2	2	2	2	2	2
3	d	0	1	2	2	2	2	2	2	2	2	2	2	2
4	o	0	1	2	2	2	2	3	3	3	3	3	3	3
5	l	0	1	2	2	2	3	3	3	3	3	4	4	4
6	o	0	1	2	2	2	3	4	4	4	4	4	4	4
7	m	0	1	2	2	2	3	4	4	4	4	4	4	4
8	i	0	1	2	2	2	3	4	4	4	4	5	5	5
9	l	0	1	2	2	2	3	4	4	4	4	5	6	6
10	e	0	1	2	2	2	3	4	4	4	4	5	6	6
11	c	0	1	2	2	2	3	4	4	4	4	5	6	6

Slika 5.6: Oznaena pot je razlika med izvornim in spremenjenim besedilom.

Oznaena pot od polja (0,0) do (11,13) predstavlja razliko, ki smo jo omenili v Sliki 5.1. Diagonalne rrtice (mostiki) nam povedo, da se tiste rreke v besedilu niso spremenile in predstavljajo najdalje skupno zaporedje "Leloil".

## 5.2 Najmanjša razdalja urejanja

Iskanje najmanjše razdalje urejanja  $D(i, j)$  bomo prikazali po korakih. Pripravimo si tabelo, kot je prikazano na Sliki 5.7.

Horizontalno na vrh tabele vpišemo daljšega od besedil, ki ju primerjamo. Vertikalno ob tabeli vpišemo krajšega od besedil. V prvem koraku je tabela zelo podobna tabeli na Sliki 5.2. Razlika je v nihti vrstici in v nihtem stolpcu. Zgornjo nihto vrstico zapolnimo z vrednostmi od 0 do  $n$ . Vrednost  $n=13$  predstavlja daljše od obeh besedil. Levi nihti stolpec zapolnimo z vrednostmi od 1 do  $m$ . Vrednost  $m=11$  predstavlja krajše od obeh besedil. To lahko naredimo iz logičnega razloga. Če katerokoli besedilo primerjamo s praznim besedilom, je število operacij enako številu vseh vstavljanj oziroma vseh brisanj, da besedili poenotimo.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13
			L	e	t	a	l	o	n	o	s	i	l	k	a
0		0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	L	1													
2	e	2													
3	d	3													
4	o	4													
5	l	5													
6	o	6													
7	m	7													
8	i	8													
9	l	9													
10	e	10													
11	c	11													

Slika 5.7: Priprava na iskanje najmanjše razdalje urejanja.

Postopek za iskanje najmanjše razdalje urejanja se prične v zgornjem levem polju (1, 1) in se nadaljuje do konca vrstice. Nato gre v naslednjo vrstico, se ponovi za vse vrstice in se konča v spodnjem desnem polju (11, 13). Vrednost vsakega polja v tabeli je odvisna od vrednosti sosednjih treh polj. V posamezno polje se vpiše minimum od treh vrednosti. To so vrednost levo ležečega polja +1, vrednost zgornje ležečega polja +1 ali vrednost zgornje-levo ležečega polja +2, če sta polji različni, ali +0, če sta polji enaki. Matematično to zapišemo, kot je prikazano na Sliki 5.8.

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

Slika 5.8: Izračun najmanjše razdalje urejanja [21], pri čemer  $X(i)$  predstavlja  $i$ -ti znak v krajšem (izvirnem) besedilu in  $Y(j)$  predstavlja  $j$ -ti znak v daljšem (spremenjenem) besedilu. Opomba: Pri iskanju vrednosti  $D(i, j)$  nad diagonalo z oznako  $\Delta$  (glej Sliko 5.11) lahko upoštevamo le vrednosti levo ležečega polja in zgornje-levo ležečega polja. Zgornje ležeče polje bistveno ne vpliva na izračun. Pri iskanju vrednosti  $D(i, j)$  pod diagonalo z oznako  $\emptyset$  lahko upoštevamo le vrednosti zgornje ležečega polja in zgornje-levo ležečega polja. Levo ležeče polje bistveno ne vpliva na izračun.

Začnimo v polju (1,1). Primerjamo črki "L" in "l". Ker so vrednosti, ki jih pridobimo iz sosednjih treh polj, enake 2 in 0 in 2, se v polje (1,1) vpiše najmanjša od vrednosti, to je 0. Nadaljujemo s poljem (1,2). Vrednosti, ki jih dajo sosednja tri polja, so enake 1 in 3 in 3. Na mesto (1,2) vpišemo 1. Izgled tabele po dveh korakih je prikazan na Sliki 5.9.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		L	e	t	a	l	o	n	o	s	i	l	k	a
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	L	1	0	1										
2	e	2												
3	d	3												
4	o	4												
5	l	5												
6	o	6												
7	m	7												
8	i	8												
9	l	9												
10	e	10												
11	c	11												

Slika 5.9: Izgled tabele po prvih dveh korakih.

Postopek nadaljujemo do konca prve vrstice. Ker se črka "L" v besedilu "Letalonosika" ne ponovi več, je vrednost vsakega nadaljnega polja za eno večja od levo ležečega polja. Postopek ponovimo še čez vse vrstice. Slika 5.10 prikazuje v celoti izpolnjeno tabelo.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		L	e	t	a	l	o	n	o	s	i	l	k	a
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	L	1	0	1	2	3	4	5	6	7	8	9	10	11
2	e	2	1	0	1	2	3	4	5	6	7	8	9	10
3	d	3	2	1	2	3	4	5	6	7	8	9	10	11
4	o	4	3	2	3	4	5	4	5	6	7	8	9	10
5	l	5	4	3	4	5	4	5	6	7	8	9	8	9
6	o	6	5	4	5	6	5	4	5	6	7	8	9	10
7	m	7	6	5	6	7	6	5	6	7	8	9	10	11
8	i	8	7	6	7	8	7	6	7	8	9	8	9	10
9	l	9	8	7	8	9	8	7	8	9	10	9	8	9
10	e	10	9	8	9	10	9	8	9	10	11	10	9	10
11	c	11	10	9	10	11	10	9	10	11	12	11	10	11

Slika 5.10: Izgled tabele po postopku za iskanje najmanjše razdalje urejanja.



Označimo diagonalna polja  $k$  z oznako od  $-m$  do  $n$ , kot je prikazano s krogci na Sliki 5.11. Označili smo pas med diagonalo  $k=0$  in  $k=\Delta=n-m=2$ . Prva diagonalna poteka skozi izvir, druga pa skozi ponor. Če bi bili besedili identični, bi se ti dve diagonali prekrivali. Po tej diagonalni pa bi potekala tudi pot, ki smo jo našli v Sliki 5.6. Glede na to, da sta v našem primeru besedili različni, to pomeni, da bo pot potekala po označenem pasu in v njegovi bližini. Koliko se bo pot oddaljila od označenega pasa, je odvisno od vrednosti  $p$ . Ker vrednosti  $p$  v začetku algoritma ne poznamo, jo moramo računati za vsako polje sproti. Na začetku tega poglavja smo rekli, da je vrednost  $p=d/2-\Delta/2$ . To drži, vendar le za spodnje desno polje  $(n,m)$ . Posamezno polje znotraj tabele se računa na način, ki je prikazan na Sliki 5.12.

$$V(i,j) = \frac{D(i,j) - k}{2}$$

$$P(i,j) = \begin{cases} V(i,j) & ; \text{ če se } (i,j) \text{ nahaja pod diagonalo } \Delta \\ V(i,j) + (k-\Delta) & ; \text{ če se } (i,j) \text{ nahaja nad diagonalo } \Delta \end{cases}$$

Slika 5.12: Računanje vrednosti  $p$  med postopkom.

Levi ničti stolpec zapolnimo z vrednostimi od 1 do  $m$ . Vrednost  $m=11$  predstavlja krajšega od obeh besedil. Pri zgornji ničti vrstici imamo manjšo posebnost. Polja znotraj obarvanega pasu (glej Sliko 5.11) zapolnimo z vrednostimi 0. Vrednost vsakega nadaljnjega polja v ničti vrstici pa je za eno večja od levo ležečega polja. Zadnje polje v ničti vrstici mora imeti vrednost  $m=11$ .

Podobno kot v Poglavjih 5.1 in 5.2 se išče vrednosti polj v tabeli od leve proti desni in od zgoraj navzdol. Bistvena razlika je, da ne iščemo vedno do konca vrstice in ne začnemo vedno iskati na začetku vrstice. Iskanje poti bomo torej optimizirali tako, da bomo namesto vseh polj v tabeli raziskali samo tiste, ki so možni kandidati za pot (scenarij najkrajšega urejanja). Prične se v zgornjem levem polju  $(1,1)$  z vrednostjo  $p=0$ . Ko najdemo v tabeli vsa polja z vrednostjo 0, začnemo iskati polja z vrednostjo 1. Ko najdemo v tabeli vsa polja z vrednostjo 1, začnemo iskati polja z naslednjo vrednostjo. Postopek ponavljamo, dokler ne dosežemo spodnjega desnega polja  $(n,m)$ .

Smiselno se je vprašati, kako ve algoritem, da v tabeli ne obstaja nobeno polje s trenutno vrednostjo  $p$ , ne da bi preiskali celotno tabelo. Kot bomo na koncu videli, so vrednosti polj na posamezni diagonalni vedno naraščajoče. V praksi v našem primeru to pomeni (glej Sliko 5.11), da bodo imele polja na diagonalah z oznakami 0, 1 in 2 vrednosti 0 ali več. Polja na diagonalah z oznakami -1 in 3 bodo imela vrednosti 1 ali več. Polja na diagonalah z oznakami -2 in 4 pa vrednost 2 ali več itd. Na podlagi tega dejstva je algoritem pri iskanju vrednosti  $p$  omejen z robnimi diagonalami.

Začnimo torej v zgornjem levem polju. Vrednost polja  $(1,1)$  po formuli na Sliki 5.12 je  $p(1,1)=0$ . Nadaljujemo proti koncu vrstice. Vrednosti polj  $(1,2)$  in  $(1,3)$  sta tudi  $p=0$ . Vrednost naslednjega polja nas ne zanima več. Zakaj? Zadnja vrednost  $p$  na diagonali z oznako 3 je 1, kar je več od trenutno iskanih polj z vrednostjo  $p=0$ . Nadaljujemo v naslednjo (drugo) vrstico. Prvo polje  $(2,1)$  lahko izpustimo, saj je zadnja vrednost na diagonali z oznako -1 že 1, kar je več od trenutno iskanih polj z vrednostjo  $p=0$ . Nadaljujemo po vrstici. Izračunamo, da imajo naslednja tri polja  $(2,2)$ ,  $(2,3)$  in  $(2,4)$  vrednost  $p=0$ . Polje  $(2,5)$  nas trenutno ne zanima. Nadaljujemo v naslednjo (tretjo) vrstico. Ugotovimo, da moramo za polja z vrednostjo  $p=0$  iskati le na diagonalah z oznako 0, 1 in 2. V tretji vrstici na teh diagonalah ne obstaja nobeno polje več, ki bi imelo vrednost  $p=0$ . Po iskanju polj z vrednostjo  $p=0$  izgleda naša tabela, kot je prikazano na Sliki 5.13.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		L	e	t	a	l	o	n	o	s	i	l	k	a
1	0	0	0	1	2	3	4	5	6	7	8	9	10	11
2	-1	1	0	0	0									
3	-2	2		0	0	0								
4	-3	3												
5	-4	4												
6	-5	5												
7	-6	6												
8	-7	7												
9	-8	8												
10	-9	9												
11	-10	10												
12	-11	11												
13	-12	12												

Slika 5.13: V tabeli smo našli vsa polja, ki imajo vrednost  $p=0$ .

Nadaljujemo z iskanjem polj v tabeli, ki imajo vrednost  $p=1$ . Ta polja se lahko v našem primeru nahajajo le na diagonalah z oznakami od  $-1$  do  $3$ . Po iskanju polj z vrednostjo  $p=1$  izgleda naša tabela, kot je prikazano na Sliki 5.14.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	-1	L	e	t	a	l	o	n	o	s	i	l	k	a
2	1	0	0	0	1									
3	2	1	0	0	0	1								
4	3		1	1	1	1								
5	4						1							
6	5													
7	6													
8	7													
9	8													
10	9													
11	10													
12	11													
13	12													

Slika 5.14: V tabeli smo našli vsa polja, ki imajo vrednost  $p=1$ .

Nadaljujemo z iskanjem polj v tabeli, ki imajo vrednost  $p=2$ . Ta polja se lahko v našem primeru nahajajo le na diagonalah z oznakami od  $-2$  do  $4$ . Po iskanju polj z vrednostjo  $p=2$  izgleda naša tabela, kot je prikazano na Sliki 5.15.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	-1	L	e	t	a	l	o	n	o	s	i	l	k	a
2	1	0	0	0	1	2								
3	2	1	0	0	0	1	2							
4	3	2	1	1	1	1	2							
5	4		2	2	2	2	1	2						
6	5					2	2	2						
7	6						2	2	2					
8	7													
9	8													
10	9													
11	10													
12	11													
13	12													

Slika 5.15: V tabeli smo našli vsa polja, ki imajo vrednost  $p=2$ .

Nadaljujemo z iskanjem polj v tabeli, ki imajo vrednost  $p=3$ . Ta polja se lahko v našem primeru nahajajo le na diagonalah z oznakami od  $-3$  do  $5$ . Po iskanju polj z vrednostjo  $p=3$  izgleda naša tabela, kot je prikazano na Sliki 5.16.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		L	e	t	a	l	o	n	o	s	i	l	k	a
0	0	0	0	1	2	3	4	5	6	7	8	9	10	11
-1														
1	L	1	0	0	0	1	2	3						
-2														
2	e	2	1	0	0	0	1	2	3					
-3														
3	d	3	2	1	1	1	1	2	3					
-4														
4	o	4	3	2	2	2	2	1	2	3				
-5														
5	l	5		3	3	3	2	2	2	3				
-6														
6	o	6					3	2	2	2	3			
-7														
7	m	7					3	3	3	3				
-8														
8	i	8									3			
-9														
9	l	9										3		
-10														
10	e	10												
-11														
11	c	11												

Slika 5.16: V tabeli smo našli vsa polja, ki imajo vrednost  $p=3$ .

Nadaljujemo z iskanjem polj v tabeli, ki imajo vrednost  $p=4$ . Ta polja se lahko v našem primeru nahajajo le na diagonalah z oznakami od  $-4$  do  $6$ . Po iskanju polj z vrednostjo  $p=4$  izgleda naša tabela, kot je prikazano na Sliki 5.17.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		L	e	t	a	l	o	n	o	s	i	l	k	a
0	0	0	0	1	2	3	4	5	6	7	8	9	10	11
-1														
1	L	1	0	0	0	1	2	3	4					
-2														
2	e	2	1	0	0	0	1	2	3	4				
-3														
3	d	3	2	1	1	1	1	2	3	4				
-4														
4	o	4	3	2	2	2	2	1	2	3	4			
-5														
5	l	5	4	3	3	3	2	2	2	3	4			
-6														
6	o	6		4	4	4	3	2	2	2	3	4		
-7														
7	m	7					4	3	3	3	3	4		
-8														
8	i	8					4	4	4	4	3	4		
-9														
9	l	9									4	3	4	
-10														
10	e	10										4	4	
-11														
11	c	11												

Slika 5.17: V tabeli smo našli vsa polja, ki imajo vrednost  $p=4$ .



Nadaljujemo z iskanjem polj v tabeli, ki imajo vrednost  $p=5$ . Ta polja se lahko v našem primeru nahajajo le na diagonalah z oznakami od  $-5$  do  $7$ . Po iskanju polj z vrednostjo  $p=5$  izgleda naša tabela, kot je prikazano na Sliki 5.16.

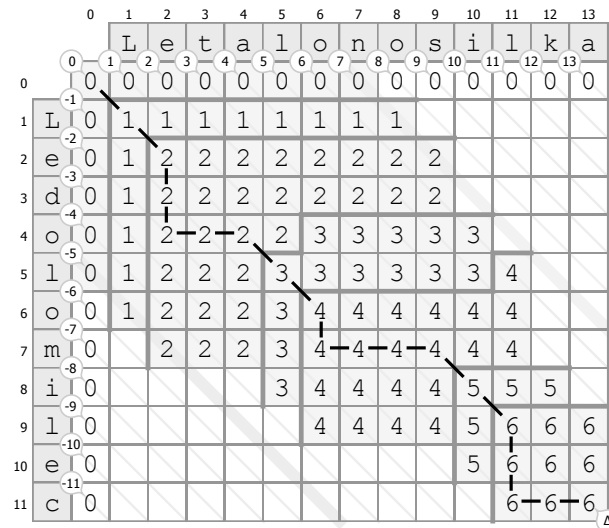
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
1	L	1	0	0	0	1	2	3	4	5					
2	e	2	1	0	0	0	1	2	3	4	5				
3	d	3	2	1	1	1	1	2	3	4	5				
4	o	4	3	2	2	2	2	1	2	3	4	5			
5	l	5	4	3	3	3	2	2	2	3	4	5	5		
6	o	6	5	4	4	4	3	2	2	2	3	4	5		
7	m	7		5	5	5	4	3	3	3	3	4	5		
8	i	8					5	4	4	4	4	3	4	5	
9	l	9						5	5	5	5	4	3	4	5
10	e	10										5	4	4	5
11	c	11											5	5	5

Slika 5.18: V tabeli smo našli vsa polja, ki imajo vrednost  $p=5$ .

Postopek se zaključi, ko najdemo v spodnjem desnem polju  $(n,m)$  vrednost  $p$ . V našem primeru je  $p(11, 13)=5$ . Če primerjamo Sliko 5.6 in Sliko 5.18, lahko opazimo, da pot iz Slike 5.6 res poteka po obarvanih poljih na Sliki 5.18. Privarčevali smo računanje vrednosti nekaterih polj v zgornjem desnem in v spodnjem levem delu tabele. V našem primeru je to le manjši del tabele. V praksi se izkaže, da je pri urejanju besedila to večji del tabele, saj se iskana pot razlike besedil vedno nahaja v pasu med diagonalama z oznako  $-p$  in  $\Delta+p$ .

Do **scenarija najkrajšega urejanja** pridemo preko najdaljšega skupnega zaporedja, ki smo ga poiskali v Poglavju 5.1 in prikazali na Sliki 5.6. V trenutnem Poglavju 5.3 smo preko vrednosti  $p$  predstavili, kako optimizirati iskanje poti. V praksi se morajo vsi ti izračuni (**najdaljše skupno zaporedje**, **najmanjša razdalja urejanja** in **vrednost  $p$** ) pri primerjavi podbesedil računati sočasno tekom algoritma, saj le na ta način najbolje optimiziramo časovno zahtevnost.

Šele na koncu, ko imamo Sliko 5.19, se sprehodimo po poti in izluščimo scenarij najkrajšega urejanja.



Slika 5.19: Optimizirano iskanje najdaljšega skupnega zaporedja.

Spremembe na besedilu so torej { Pabriši @3-4 }, { Vstavi 'ta' @5 }, { Pabriši @7-7 }, { Vstavi 'nos' @8 }, { Pabriši @10-11 }, { Vstavi 'ka' @12 }. Lokacije vseh sprememb temeljijo na izvirnem besedilu "Ledolomilec".

## Poglavje 6

# Implementacija urejevalnika

Odločili smo se, da bomo urejevalnik v realnem času zasnovali kot enostavno spletno aplikacijo. Dostop do urejevalnika smo omejili le na dodeljene uporabniške račune. Ko se uporabnik prijavi v spletno aplikacijo, ga na osrednjem delu zaslona pričaka polje za vnos besedila, ki ga lahko ureja več uporabnikov. Na desni strani zaslona pa si lahko ogleda zgodovino vseh sprememb, ki so bile narejene na besedilu.

Na strani strežnika smo uporabili platformo **node.js**. To je platforma za enostavno izdelavo hitrih in skalabilnih spletnih aplikacij. Uporablja dogodkovno-gnan model, ki je ustrezen za podatkovno intenzivne aplikacije v realnem času. Uporabniške račune hranimo v podatkovni bazi RethinkDB. Sama prijava je v brskalniku realizirana s knjižnico AngularJS. Sicer pa smo na strani odjemalca (v brskalniku) uporabili standardne tehnologije, to so HTML, CSS in JS. Za manipulacijo drevesa DOM in urejevalnika smo se posluževali še knjižnice jQuery.

Implementacija urejevalnika je potekala v treh korakih. V prvem koraku smo zasnovali aplikacijo in naredili prijavo. V drugem koraku smo se lotili iskanja razlik v besedilu. Na koncu smo realizirali še Operativno transformacijo. V opisu teh treh korakov se bomo sklicevali na nekatere datoteke. Bralca naprošamo, da si jih podrobno ogleda na GitHub-u (<https://github.com/eMarek/Revision>), kjer je dosegljiva celotna koda aplikacije.

### 6.1 Zasnova in prijava

Aplikacijo smo želeli zasnovati kot enostavno modularno ogrodje (angl. *framework*). Uporabnost modularne zasnove je, da se jo lahko enostavno razširi za uporabo sodelovanja, ki ga bomo opisali v Poglavju 6.3. Glavni funkcionalnosti, ki ju naš modul

omogoča, sta serviranje statičnih datotek ter zagotavljanje podatkov preko API-ja.

V datoteki `“server.js”` v vrstici 91, se za vsak zahtevek na strežnik najprej pogleda ime poti.<sup>1</sup> V primeru, da je prvih pet znakov enakih `“/api/”`, zadnjih pet znakov enakih `“.json”` in je metoda zahtevka `“POST”`, potem modul zagotovi gole podatke. V nasprotnem primeru poskuša odjemalcu vrniti statično datoteko iz javnega (angl. *public*) direktorija.

Za potrebo prijave smo implementirali še konfiguracijsko datoteko (angl. *config*) in krmilnik (angl. *controller*). Razlika med njima je, da se konfiguracijska datoteka požene le enkrat ob zagonu aplikacije, krmilnik pa se izvede ob vsakem zahtevku na strežniku. Iz tega razloga je konfiguracijska datoteka primerna za ustvarjanje povezave na bazo, krmilnik pa lahko uporabimo za preverjanje avtentikacije.

V konfiguracijski datoteki `“example/config.js”` se v vrstici 17 naredi povezavo na bazo. Po uspešno ustvarjeni povezavi do baze se pokliče funkcija `“starter”`, ki požene spletni strežnik. V krmilniku v datoteki `“example/controller.js”` v vrstici 34 se sejo poskuša dekriptirati in nato pretvoriti v JSON obliko. V vrstici 71 se naredi poizvedba iz podatkovne baze in preveri, ali obstaja uporabnik z ID-jem iz seje. V primeru, da je vse v redu, se na koncu pokliče še funkcija `“handler”`, ki glede na ime poti vrne podatke.

Da si lahko uporabnik sploh ustvari sejo, smo naredili rokovalnik API (angl. *handler*) v datoteki `“example/api/default.js”` v vrstici 58. Uporabnik pošlje na strežnik svoje uporabniško ime in geslo. Geslo se pretvori v zgoščeno obliko. V podatkovni bazi se preveri, ali obstaja uporabnik s prispelim uporabniškim imenom in zgoščeno obliko gesla. V primeru, da obstaja, se uporabniku vrne zakriptirana seja, v kateri je med drugim shranjen uporabnikov ID. Uporabnikov brskalnik si sejo in uporabnikov ID shrani v skladišče seje (angl. *session storage*). Glej datoteko `“example/public/js/application.js”` v vrsticah 85 in 86.

Na tak način je zasnovana naša aplikacija in realizirana prijava vanjo.

## 6.2 Iskanje razlik

Iskanje razlik v besedilu smo implementirali po algoritmu, ki smo ga že opisali v Poglavju 5. Celotna koda se nahaja v datoteki `“changes.js”`. Na tem mestu omenimo le, na kakšen način se vrnejo razlike. Funkcija za iskanje razlik vrne seznam (angl. *array*) objektov JSON.

---

<sup>1</sup>Glej priloženo programsko kodo na zgoščenki.

Vsak objekt predstavlja eno razliko v besedilu in vsak objekt ima lastnost "a", ki pomeni akcijo (angl. *action*). Akcija je lahko vrednosti "+" (dodajanje besedila) ali "-" (brisanje besedila). Vsak objekt ima še lastnosti "s", ki predstavlja niz (angl. *string*) znakov, in "l", ki predstavlja dolžino (angl. *length*) niza znakov. V primeru, da gre za brisanje besedila, ima objekt še lastnosti "f" in "t", ki pomenita od (angl. *from*) katere in do (angl. *to*) katere lokacije je potrebno pobrisati znake. V primeru, da gre za dodajanje besedila, pa ima objekt še lastnost "p", ki pomeni pozicijo (angl. *position*), na katero je potrebno dodati besedilo. Primer razlike med besedama "Ledolomilec" in "Letalonosilka" je prikazan v Primeru 6.1.

---

```
[ { a: '-', s: 'do', l: 2, f: 3, t: 4 },
  { a: '+', s: 'ta', l: 2, p: 5 },
  { a: '-', s: 'm', l: 1, f: 7, t: 7 },
  { a: '+', s: 'nos', l: 3, p: 8 },
  { a: '-', s: 'ec', l: 2, f: 10, t: 11 },
  { a: '+', s: 'ka', l: 2, p: 12 } ]
```

---

Primer 6.1: Razlike med besedama "Ledolomilec" in "Letalonosilka".

## 6.3 Operativna transformacija

Kot smo omenili že v Poglavju 3.2.2, se Operativna transformacija izvaja tako na strani strežnika kot na strani odjemalca. Koda za strežniški del Operativne transformacije se nahaja v datoteki "collaboration.js", koda Operativne transformacije pri odjemalcu pa v datoteki "example/public/js/collaboration.js". Algoritem je podoben (malenkost poenostavljen) tistemu, ki smo ga opisali v Poglavju 3.2. Ker je strežniški del enostavnejši, ga bomo opisali najprej.

Na strani strežnika si beležimo revizijski dnevnik (angl. *revision diary*), trenutni dokument (angl. *current document*) in uporabnike (angl. *users*), kot je prikazano v Kodi 6.2. Strežnik od odjemalca pričakuje, da bo vedno dobil številko zadnje potrjene revizije (req.payload.revision). V primeru, da odjemalec pošlje številko zadnje revizije -2, se celotni urejevalnik resetira. V primeru, da odjemalec pošlje številko zadnje revizije -1, ali v primeru, da uporabnik še ne obstaja, strežnik vrne odjemalcu podatke za inicializacijo urejevalnika. To so trenutni dokument, revizijski dnevnik in številka zadnje revizije.

---

```
var revisionDiary = [];
var currentDocument = "";
var users = {};
```

```

/* collaboration
----- */
module.exports = function collaboration(req, rsp, data) {

    // reset editor
    if (req.payload.revision == -2) {
        revisionDiary = [];
        currentDocument = "";
        users = {};
    }

    // forced initialization or new user
    if (req.payload.revision == -1 || !users.hasOwnProperty(data.user.id))
    {

        // remember user
        users[data.user.id] = true;

        // respond with initialization data
        rsp.send({
            "currentDocument": currentDocument,
            "revisionDiary": revisionDiary,
            "revision": revisionDiary.length
        });

    } else {

        // more code here...
        // Koda 6.3
    }
};

```

---

Koda 6.2: Strežnik odjemalcu vrne podatke za inicializacijo urejevalnika.

Ko odjemalec pošlje strežniku novo spremembo oziroma natančneje popravek spremembe (`req.payload.patch`), se mora ta pravilno umestiti v trenutni dokument. V Kodi 6.3. lahko vidimo, kako se to izvede. Naredi se zanka preko vseh sprememb v revizijskem dnevniku. Osredotočiti se moramo na spremembe, ki imajo številko revizije večje ali enako številki zadnje revizije, ki jo pošilja odjemalec. Le te spremembe lahko povzročijo potrebo po uporabi Operativne transformacije. Spremembe tipa dodajanje besedila kvečjemu povečajo odmik, medtem ko spremembe tipa brisanje besedila kvečjemu pomanjšajo odmik. Po končani zanki se izračunani odmik prišteje k popravku, ki ga pošilja odjemalec. S popravkom, ki ima pravilne pozicije, posodobimo trenutni dokument in popravek shranimo v revizijski dnevnik. Ker tudi odjemalec od strežnika pričakuje, da bo vedno dobil številko zadnje revizije v revizijskem dnevniku, se mora le-ta poslati odjemalcu. V primeru, da so v revizijskem dnevniku nove spremembe, o katerih uporabnik še ni obveščen, se morajo poslati tudi te.

---

```

// new incoming patches from current user

```

```

if (req.payload.patch) {

    // prepare bundle
    bundle = {..., "patch": req.payload.patch, ...};

    // calculate offset for incoming patch - OPERATIONAL TRANSFORMATION
    offset = 0;
    for (var rd in revisionDiary) {
        if (rd >= req.payload.revision) {

            // single revision diary patch
            patch = revisionDiary[rd].patch;

            // some characteres were added in previous revision
            if (patch.a === "+") {
                if (bundle.patch.a === "+" && bundle.patch.p >= patch.p) {
                    offset = offset + patch.s.length;
                }
                if (bundle.patch.a === "-" && bundle.patch.f >= patch.p) {
                    offset = offset + patch.s.length;
                }
            }

            // some characteres were deleted in previous revision
            if (patch.a === "-") {
                if (bundle.patch.a === "+" && bundle.patch.p >= patch.f) {
                    offset = offset - patch.s.length;
                }
                if (bundle.patch.a === "-" && bundle.patch.f >= patch.f) {
                    offset = offset - patch.s.length;
                }
            }
        }
    }

    // adding characters
    if (bundle.patch.a === "+") {

        // take offset into account
        bundle.patch.p = bundle.patch.p + offset;

        // update current document
        currentDocument = currentDocument.substr(0, bundle.patch.p - 1) +
            bundle.patch.s + currentDocument.substr(bundle.patch.p - 1);
    }

    // deleting characters
    if (bundle.patch.a === "-") {

        // take offset into account
        bundle.patch.f = bundle.patch.f + offset;
        bundle.patch.t = bundle.patch.t + offset;

        // update patch string just in case
        bundle.patch.s = currentDocument.substring(bundle.patch.f - 1,
            bundle.patch.t);

        // update current document
        currentDocument = currentDocument.substr(0, bundle.patch.f - 1) +

```

```

        currentDocument.substr(bundle.patch.t);
    }

    // push bundle to revision diary
    revisionDiary.push(bundle);
}

// do we have new patches in revision diary
if (revisionDiary.length > req.payload.revision) {
    rsp.send({
        "revisionDiary": revisionDiary.slice(req.payload.revision),
        "revision": revisionDiary.length
    });
} else {
    rsp.send({
        "revision": revisionDiary.length
    });
}

```

---

Koda 6.3: Izvajanje Operativne transformacije na strani strežnika.

Na strani odjemalca se Operativna transformacija izvaja podobno kot na strani strežnika. Razlika je le v nekaj dodatnih opravilih, ki jih mora opraviti odjemalec. Najprej mora poskrbeti, da se sodelovanje izvaja periodično na vsako desetinko. Po odgovoru strežnika pa mora poskrbeti, da se uporabnikova utripalka (angl. *caret*) po Operativni transformaciji postavi na pravo mesto. Uporabnik ne sme imeti občutka, da skače utripalka levo in desno po besedilu. Zagotoviti mu moramo tekoče tipkanje.

Sodelovanje se dejansko izvede le v primeru, da urejevalnik obstaja v drevesu DOM in ni čakajočih sprememb. Interval s preverjanjem je prikazan v Kodi 6.4.

---

```

setInterval(function() {

    // does editor exist on page
    if ($(editor).length) {

        // in case of waiting patches collaboration will call itself
        if (!waitingPatches[0]) {

            // fire collaboration
            collaboration();
        }

    } else {

        // reset collaboration
        revision = -1;
        // some more declarations here...
    }
}, 100);

```

---

Koda 6.4: Periodično izvajanja sodelovanja.

Tekom sodelovanja se v Kodi 6.5 ponovno preveri, ali obstajajo čakajoče spremembe.



Če jih ni, se poskuša s funkcijo za iskanje razlik v besedilu poiskati nove spremembe. V nadaljevanju algoritma se iz čakajočih sprememb vzame prvo in se jo pošlje na strežnik skupaj z revizijsko številko. Če ni čakajočih sprememb, se na strežnik pošlje le številko zadnje revizije, saj se tako preveri, ali je kakšno spremembo naredil kdo drug od uporabnikov. Klic na strežnik se naredi s klicem AJAX.

---

```
function collaboration() {

    // calculate waiting patches with changes function if there is non
    if (!waitingPatches[0]) {
        newDocument = $(editor).val();
        waitingPatches = changes(oldDocument, newDocument);
    }

    // sent patch to server if there is any
    if (waitingPatches[0]) {

        // take on patch from waiting patches and
        // sent it with expected revision number
        data = {
            "patch": waitingPatches.shift(),
            "revision": revision
        };

    } else {
        // sent just revision number to the server
        data = {
            "revision": revision
        };
    }

    // collaboration ajax
    xhr.collaboration = $.ajax({
        url: "api/collaboration.json",
        contentType: "application/json",
        type: "post",
        data: JSON.stringify(data),
        headers: {
            session: window.sessionStorage.session
        },
        dataType: "json",
        success: function(server) {
            // Koda 6.6
        }
    });
}
```

---

Koda 6.5: Iskanje sprememb v besedilu in AJAX klic na strežnik.

Po tem, ko odjemalec uspešno (angl. *success*) prejme strežnikov odgovor, mora sprocesirati revizijski dnevnik (`server.revisionDiary`). V njem se nahajajo le nove spremembe od zadnje revizijske številke. Razlago Operativne transformacije na strani odjemalca bomo izpustili, saj je podobna, kot smo jo opisali na strani strežnika. Nekaj

več besed pa lahko namenimo pozicioniranju utripalke. Izvedba je prikazana v Kodu 6.6. Upoštevati je potrebno, da ima lahko uporabnik v urejevalniku del besedila kar označen (angl. *highlighted*). V tem primeru sta začetek in konec označbe (angl. *selection start and end*) različna, sicer pa enaka. Algoritem naredi zanko preko vseh potrjenih sprememb. Če avtor spremembe ni uporabnik sam, je potrebno izračunati odmik utripalke.

Pri dodajanju besedila sta torej dve možnosti, pri katerih je potrebno popravljati pozicijo označenega dela. Besedilo se dodaja pred označen ali pa med označen del. V prvem primeru je potrebno popravljati tako začetek kot konec označbe, v drugem pa le konec označbe.

Pri brisanju besedila so štiri možnosti. Poleg omenjenih dveh se lahko besedilo briše ravno na mestu, kjer je začetek ali konec označbe. V primeru, da se besedilo briše na začetku označbe, se spremenita tako začetek kot konec. V primeru, da se besedilo briše na koncu označbe, se spremeni le konec.

---

```
// remember acknowledged patches prepare caret values
acknowledgedPatches = acknowledgedPatches.concat(server.revisionDiary);
caretStart = $(editor)[0].selectionStart;
caretEnd = $(editor)[0].selectionEnd;
caretOffsetStart = 0;
caretOffsetEnd = 0;

// loop
for (var lp in acknowledgedPatches) {
    patch = acknowledgedPatches[lp].patch;

    // if not my patch
    if (acknowledgedPatches[lp].author !== window.sessionStorage.userID) {
        // do OPERATIONAL TRANSFORMATION and correct waiting
        // patches position or from/to values here...

        // characteres were added in previous revision
        if (patch.a === "+") {
            if (...++++...[.....].....) {
                caretOffsetStart = caretOffsetStart + patch.s.length;
                caretOffsetEnd = caretOffsetEnd + patch.s.length;
            }
            if (...++++...[.....].....) {
                caretOffsetStart = caretOffsetStart;
                caretOffsetEnd = caretOffsetEnd + patch.s.length;
            }
        }

        // characteres were deleted in previous revision
        if (patch.a === "-") {
            if (...----...[.....].....) {
                caretOffsetStart = caretOffsetStart - patch.s.length;
                caretOffsetEnd = caretOffsetEnd - patch.s.length;
            }
        }
    }
}
```

```

        if (.....--[--.....].....) {
            caretOffsetStart = caretOffsetStart - (caretStart - patch.f
                );
            caretOffsetEnd = caretOffsetEnd - patch.s.length;
        }
        if (.....[...---...]......) {
            caretOffsetStart = caretOffsetStart;
            caretOffsetEnd = caretOffsetEnd - patch.s.length;
        }
        if (.....[.....--]--.....) {
            caretOffsetStart = caretOffsetStart;
            caretOffsetEnd = caretOffsetEnd - (caretEnd - patch.f) - 1;
        }
    }
}
}
// update current document here...
// put current document into the editor
$(editor).val(currentDocument);

// correct caret position with offset
$(editor)[0].selectionStart = caretStart + caretOffsetStart;
$(editor)[0].selectionEnd = caretEnd + caretOffsetEnd;

// revision
revision = server.revision;

```

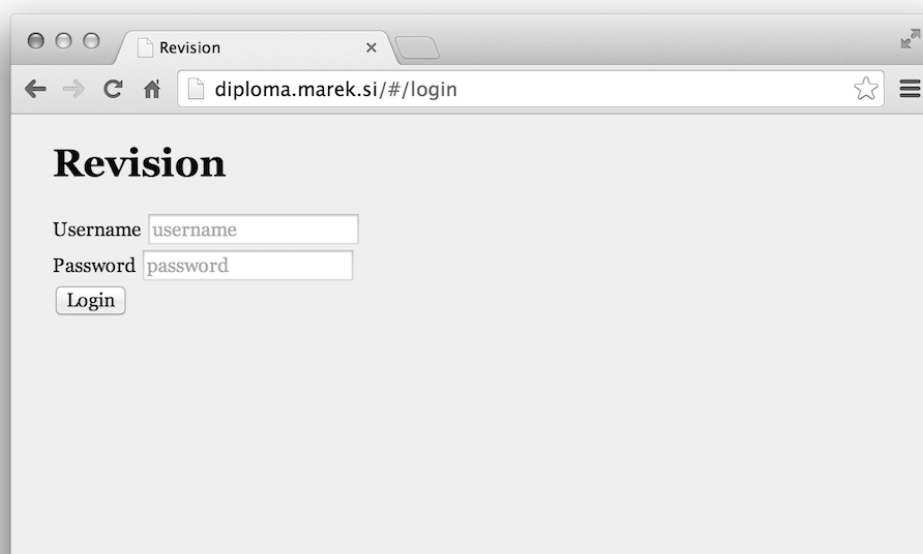
---

Koda 6.6: Popravljanje pozicije utripalke na strani odjemalca.

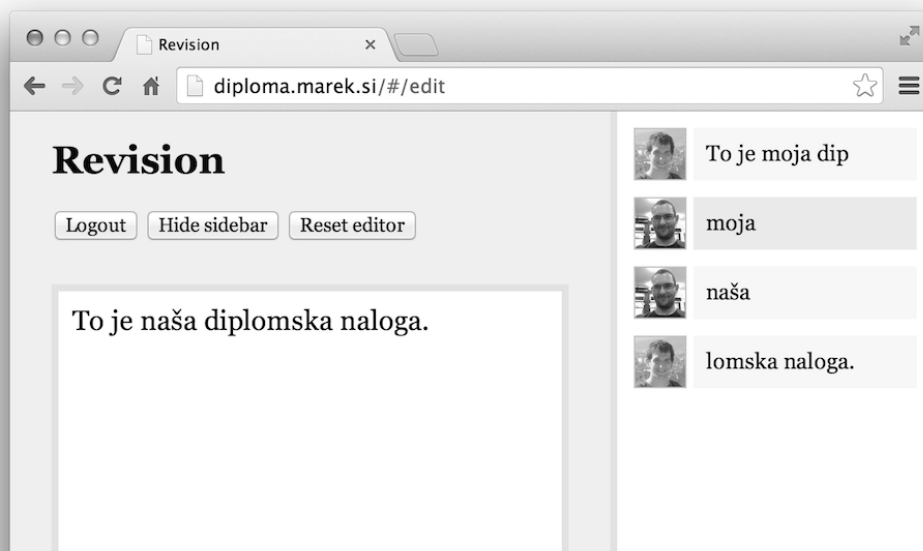
Na koncu se trenutni dokument vpiše v urejevalnik in nastavi se pozicija utripalke oziroma označbe. Popravi se še številka zadnje revizije in kolaboracija se zaključi.

## 6.4 Uporabniški vmesnik aplikacije

Cilj diplomske naloge je bil implementirati enega izmed algoritmov za urejenje besedila v realnem času in tako zasnovati urejevalnik. Na Sliki 6.1 se vidi uporabniški vmesnik za prijavo v aplikacijo, na Sliki 6.2 pa uporabniški vmesnik po prijavi. Urejevalnik je enostaven za uporabo in prikazuje, kako napredna tehnologija omogoča in spodbuja sodelovanje oddaljenih uporabnikov. Večino testiranja smo naredili v brskalniku Google Chrome verzije 34.0, vendar aplikacija zelo dobro dela tudi na pametnih mobilnih telefonih in na tabličnih računalnikih.



Slika 6.1: Prijava v aplikacijo, ki smo jo poimenovali "Revision".



Slika 6.2: Urejevalnik in prikaz urejanja dveh uporabnikov.

V primerjavi z ostalimi podobnimi orodji (share.js) je naš urejevalnik bolj prilagodljiv. Le z malo spremembami lahko zamenjamo podatkovno bazo. To pomeni, da je naše orodje enostavno priključiti na že obstoječe projekte. Prednost našega orodja je tudi v vgrajeni avtentikaciji. Vsak uporabnik se mora za uporabo urejevalnika prijaviti. Razvijalcem, ki bi želeli naše orodje uporabiti in nadgraditi v končni produkt, ne bi bilo potrebno skrbeti še za avtentikacijo uporabnikov, saj je to že narejeno. Kot lahko vidimo na Sliki 6.2, smo v aplikacijo vgradili tudi prikaz sprememb, ki jih naredijo uporabniki v besedilu.



# Poglavje 7

## Sklepne ugotovitve

V diplomski nalogi smo raziskali različne algoritme za skupinsko urejanje besedila, to so: Diferenčna sinhronizacija, Operativna transformacija in pristop Brez operativne transformacije. Pri DS smo ugotovili, da sta na osnovni topologiji zgrajeni metoda dveh senc in metoda zagotovljene dostave. Osnovnega delovanja OT ni bilo težko razumeti. Smo pa ugotovili, da je OT neuporabna brez uporabe protokola za sodelovanje. Če sta si DS in OT v nekaterih pogledih podobna, je WOOT zgodba zase tako v podatkovnem modelu, v načinu podajanja operacij, kot tudi v arhitekturi. Vsi trije pristopi se med seboj močno razlikujejo, zato smo jih podrobneje primerjali po njihovih lastnostih. Pogledali smo si, na kakšen način hranijo dokumente, arhitekturo omrežja, na katerem delujejo, kaj se dogaja pri počasni povezavi, kako zaznavajo spremembe v besedilu, kakšni problemi lahko nastanejo pri sočasnosti. Naredili smo tudi manjšo raziskavo glede zahtevnosti algoritma. Odločili smo se, da bomo uporabili OT in implementirali enostaven urejevalnik. Pred tem smo morali raziskati še algoritem za iskanje razlik v besedilu. Tu smo se spoznali s pojmi, kot so: najdaljše skupno zaporedje, najmanjša razdalja urejanja in scenarij najkrajšega urejanja. V nadaljevanju smo se lotili izvedbe aplikacije. Razdelili smo jo v tri korake. Najprej smo postavili zasnovo skupaj s prijavo. Sledil je algoritem za iskanje razlik v besedilu. Na koncu smo implementirali še OT ter protokol za sodelovanje. Nastal je produkt, ki smo ga poimenovali "Revision".

Kot cilj smo si zadali, da bomo enega izmed algoritmov implementirali. Namen in cilj sta bila dosežena. Naj omenimo, da poleg raziskanih treh pristopov za sodelovanje v realnem času obstajajo tudi še drugi pristopi oziroma različice raziskanih pristopov, ki jih nismo obravnavali. Tudi algoritmov za iskanje razlik v besedilu je mnogo. Raziskali in implementirali smo takega, ki ima trenutno znano najboljšo časovno zahtevnost. Tekom pisanja diplomske naloge pa smo naleteli tudi na slabše plati raziskave. Predvideli smo, da bomo algoritem implementirali neodvisno od odjemalcev. Na ta način bi na

enostaven način omogočili nadaljni razvoj mobilnih in namiznih aplikacij. Delno nam je uspelo, saj se lahko na API povežejo različne naprave. Težava je v protokolu za sodelovanje, ki se v veliki meri izvaja pri odjemalcu. Če platforma omogoča izvajanje kode JavaScript, je implementacija enostavna. Sicer bi bilo potrebno protokol za sodelovanje programirati še v drugih programskih jezikih.

Tako kot vsako tehnologijo je mogoče tudi našo še izboljšati. Skoraj nujna potreba po izboljšavi je uporaba spletnih vtičev (angl. *WebSockets*), ki bi nadomestili periodične klice AJAX. Privarčevali bi kar nekaj procesorske moči tako na strani odjemalca kot na strani strežnika. Danes podpirajo spletne vtiče že vsi novejši brskalniki, vendar smo zaradi želje po kompatibilnosti s starejšimi brskalniki uporabili klice AJAX. Kodo, ki smo jo razvili v diplomski nalogi, bi lahko v nadalje uporabili in preoblikovali v splošno knjižnico. Dalo bi se jo prilagoditi tudi v plaformo za grajenje spletnih aplikacij, ki omogoča prenos podatkov v realnem času. Nekaj podobnega je meteor.js, pri čemer bi bila naša platforma neodvisna od podatkovne baze.



# Literatura

- [1] C.A. Ellis, S.J. Gibbs. "Concurrency Control in Groupware Systems", 1989.  
<http://www-ihm.lri.fr/~mbl/ENS/CSCW/2012/papers/Ellis-SIGMOD89.pdf>
- [2] D. A. Nichols, P. Curtis, M. Dixon, J. Lamping. "High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System", 1995.  
<http://lively-kernel.org/repository/webwerkstatt/!svn/bc/15693/projects/Collaboration/paper/Jupiter.pdf>
- [3] The Special Interest Group on Collaborative Computing, 1998.  
<http://cooffice.ntu.edu.sg/sigce/>
- [4] D. Wang, A. Mah, S. Lassen. "Google Wave Operational Transformation", 2010.  
<http://www.waveprotocol.org/whitepapers/operational-transform>
- [5] J. Gregorio. "Google Wave Client-Server Protocol Whitepaper", 2010.  
<http://www.waveprotocol.org/whitepapers/internal-client-server-protocol>
- [6] J. Gentle. "ShareJS - Live concurrent editing in your app", 2011.  
<http://sharejs.org/>
- [7] Joyent, Inc., R. Dahl. "**node.js**", 2009.  
<http://nodejs.org/>
- [8] The Dojo Foundation. "Open Cooperative Web Framework", 2011.  
<http://opencoweb.org/ocwdocs/intro/openg.html>
- [9] Wikipedia, the free encyclopedia. "Operational transformation".  
[http://en.wikipedia.org/wiki/Operational\\_transformation](http://en.wikipedia.org/wiki/Operational_transformation)
- [10] N. Fraser. "Differential Synchronization", 2009.  
<https://neil.fraser.name/writing/sync/>

- [11] J. Day-Richter. "What's different about the new Google Docs: Conflict resolution", 2010.  
[http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs\\_22.html](http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs_22.html)
- [12] J. Day-Richter. "What's different about the new Google Docs: Making collaboration fast", 2010.  
[http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs\\_23.html](http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs_23.html)
- [13] G. Oster, P. Urso, P. Molli, A. Imine. "Real time group editors without Operational transformation", 2005.  
<http://hal.inria.fr/docs/00/07/12/40/PDF/RR-5580.pdf>
- [14] Wikipedia, the free encyclopedia. "Peer-to-peer".  
<http://en.wikipedia.org/wiki/Peer-to-peer>
- [15] Wikipedia, the free encyclopedia. "Client-server model".  
[http://en.wikipedia.org/wiki/Client-server\\_model](http://en.wikipedia.org/wiki/Client-server_model)
- [16] W. Miller, E. W. Myers. "A File Comparison Program", 1985.  
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.189.70&rep=rep1&type=pdf>
- [17] E. W. Myers. "An  $O(ND)$  Difference Algorithm and Its Variations", 1986.  
[https://neil.fraser.name/software/diff\\_match\\_patch/myers.pdf](https://neil.fraser.name/software/diff_match_patch/myers.pdf)
- [18] S. Wu, U. Manber, G. Myers, W. Miller. "An  $O(NP)$  Sequence Comparison Algorithm", 1989.  
<http://www.itu.dk/stud/speciale/bepjea/xwebtex/litt/an-onp-sequence-comparison-algorithm.pdf>
- [19] N. Fraser. "Diff Strategies", 2006.  
<https://neil.fraser.name/writing/diff/>
- [20] D. Sodkiewicz. "Longest Common Subsequence Algorithm", 2013.  
<https://www.youtube.com/watch?v=P-mMvhfJhu8>
- [21] D. Jurafsky. "Computing Minimum Edit Distance", 2012.  
[http://www.youtube.com/watch?v=z\\_CB7Gih\\_Mg](http://www.youtube.com/watch?v=z_CB7Gih_Mg)