Eötvös Loránd Tudományegyetem

Informatikai Kar

Valós idejű kollaboratív szerkesztés

Témavezető:
Horváth Győző

Szerző:
Mészáros Áron Attila
programtervező informatikus MSc

Budapest, 2020. 01.

EFOP-3.6.3-VEKOP-16-2017-00002.
Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein

Tartalomjegyzék

1. fejezet

Bevezetés

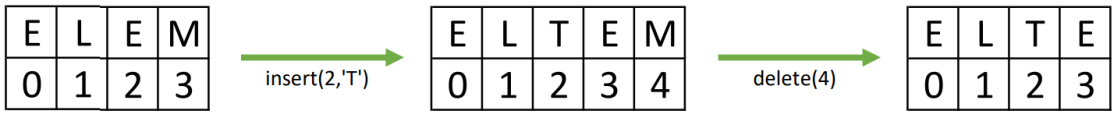
A számítógépen történő szövegszerkesztés folyamata az idők során keveset változott: általában egy dokumentumban egyszerre egy felhasználó hajthatott végre változtatásokat, majd elmenthette a létrejött tartalmat. Az internet térhódításával azonban lehetőség és igény mutatkozott arra, hogy több felhasználó egyidőben, konkurensen szerkeszthesse ugyanazt a dokumentumot. Ilyen rendszerek már az élet számos területén segítik az emberek munkáját, elég csak a Google Docs-ra gondolnunk. Ezen tanulmány célja, hogy bemutassa, mi a (közel) valós idejű szerkesztés nehézsége, és, hogy ezekre milyen megoldási lehetőségek vannak.

1.1. Alapprobléma

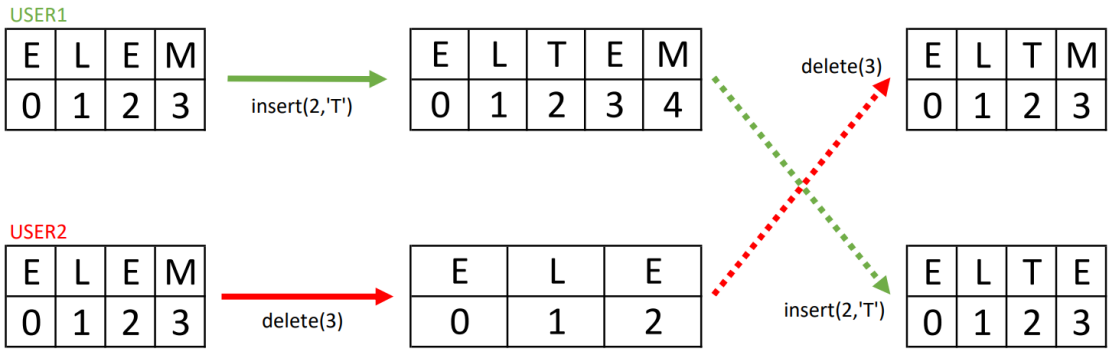
Szerkesztéskor a legalapvetőbb változtatás új karakterek beszúrása és meglévőek törlése, de természetesen ezenfelül léteznek még egyéb műveletek is (különböző formázások, felsorolások alkalmazása strukturáltság (bekezdések)). Egyelőre tekintsük a beszűrás és törlés műveletét.

1.1.1. Egy felhasználós szerkesztés

Egy felhasználó esetén a beszúrás és törlés egyszerűen megvalósítható, ha a dokumentum minden karakteréhez egy pozíciót rendelünk (indexeljük őket). Ez történhet például úgy, hogy a karaktersorozatban az első karakter pozícióját 0-val jelöljük, a második karakter pozícióját 1-el és így tovább. A beszúrás és törlés a pozícióra való hivatkozással történik. Ha például a szavunk az "ELEM", akkor az 'L' után egy 'T'



1.1. ábra. Egy felhasználós szerkesztés



1.2. ábra. Két felhasználós szerkesztés

karakter beszúrását az `insert(2,'T')` művelet írja le. Érdemes megjegyezni, hogy a művelet hatására a korábbi 2-es indexű elem új indexe 3-as lesz, a korábbi 3-as indexűé 4-es stb., a létrejövő karaktersorozat pedig "ELTEM". Az utolsó karakter törlése a `delete(4)` művelettel történik. (1.1. ábra)

1.1.2. Több felhasználós szerkesztés

Mi a helyzet azonban akkor, ha több felhasználó szeretné valós időben ugyanazt a dokumentumot szerkeszteni? Ez esetben tegyük fel, hogy a kommunikáció valamilyen hálózaton keresztül zajlik, valamint a kiindulási szöveg egyezzen meg az előbbivel ("ELEM"). Az egyik felhasználó a 'T' karaktert szúrja be a második indexhez (`insert(2,'T')`), míg a másik felhasználó az 'M' karaktert törli (`delete(3)`), majd mindketten elküldik a változásokat a másiknak. Az eredmény az első felhasználó képernyőjén "ELTM", míg a második felhasználó képernyőjén "ELTE" lesz, ahogy azt a 1.2. ábra mutatja.

Látható, hogy kontrollálatlan esetben nagyon egyszerű olyan helyzetet előidézni, melyben a különböző felhasználók különböző állapotot látnak. Egy kollaboratív rendszer helyes, ha kielégíti a CCI feltételeket [1]:

- Causality: Az összes Lamport $happened - before$ relációja szerint rendezett

művelet ugyanabban a sorrenben hajtódik végre minden másolaton.

- Convergence: A rendszer konvergál, ha minden másolat megegyezik, amikor a rendszer tétlen.
- Intention: A művelet elvárt eredménye figyelhető meg minden másolaton. Széles körben elfogadott definíció szövegszerkesztés esetén:
  - `delete` Egy karaktert akkor és csak akkor kell eltávolítani a dokumentumból, ha valamelyik felhasználó törölte.
  - `insert` Egy felhasználó által beszúrt karakternek meg kell jelennie minden felhasználónál. A dokumentum karakterei és az újonnan beszúrt karakterek sorrendjét meg kell tartani minden felhasználónál (ameddig ezek a karakterek léteznek).

A naív megközelítés tehát nem elég, a feltételek biztosításához kifinomultabb megközelítésekre van szükség. A következőkben bemutatom, melyek ezek.

2. fejezet

Operational Transformation

Az első módszer, melyet kidolgoztak valós idejű kollaboratív szerkesztéshez az Operational Transformation (OT) néven ismert. Első változata 1989-ben jelent meg [2] , azóta különböző változatait számos helyen használják, az egyik legismertebb ilyen például a Google Docs.

A módszer lényegében a korábban bemutatott naív megközelítésen alapszik, azaz továbbra is műveleteket használ, viszont próbálja annak hibáit kijavítani. Korábbi példánkban ez az volt, hogy míg a beszúrás és törlés a dokumentum életének szempontjából azonos időben történt, a két felhasználó két különböző sorrendben hajtotta őket végre, változtatások nélkül. Az OT alapötlete, hogy a műveleteken különböző transzformációk végrehajtásával feloldja ezeket a konfliktusokat. A példában az első felhasználó számára a `delete(3) -> delete(4)` transformáció egy megoldás.

Az eljárás két konkurens művelet esetében még egyszerűnek tűnik, azonban nagyon hamar bonyolult algoritmusok kifejlesztése válik szükségessé. Sajnos sokukról bebizonyosodott, hogy egyes esetekben hibásan működnek [3], míg bizonyosak központi szerver jelenlétét igénylik. [4] [5]. A mindennapokban elterjedt alkalmazások többnyire ilyet használnak, mivel ezek egyszerűbbek, hibabiztosabbak. [6] (Google docs [7], Etherpad [8]). A P2P algoritmusok hátránya, hogy vector-clockokat használnak, melyek küldése sok kliens esetén költséges lehet. [9] [10] [11] A felsorolt nehézségek miatt a közelmúltban megjelent egy másik megközelítés a valós idejű szerkesztés megvalósítására.

3. fejezet

Conflict-free replicated data types

A másik módszer, mely használata elterjedt valós idejű kollaboratív szerkesztéshez az úgynevezett Conflict-free replicated data type-ok - vagy röviden CRDT-k - használata. Ezek, ahogy a nevük is mutatja, adatszerkezetek, segítségükkel pedig nem csak karaktersorozatok, hanem egyéb struktúrák is leírhatóak. Az ismert CRDT-k közé tartoznak regiszterek, számlálók, mapek, halmazok, [12] [13], de JSON CRDT is létezik [14]. Ezeket számos rendszer használja, például a Riak [15].

Egy CRDT objektum az elosztott rendszerekben jól ismert replikáció módszerét használja. Replikáció esetén több számítógép tárolja egy adat másolatait, ezek az úgynevezett replikák. A cél, hogy ezen replikákon a változtatások legyenek egymástól függetlenül, konkurensen végrehajthatóak, majd ezek idővel a többi replikán is megjelenjenek.

Azt, hogy ez milyen megszorításokkal történik, az elosztott rendszerek tudománya különböző konzisztenciamodellekel írja le. A CRDT-k szempontjából két fontos konzisztenciamodell ismerete szükséges:

- Eventual consistency [12] [16] [17]: Teljesüléséhez három feltétel szükséges:
  - Eventual delivery: Egy korrekt replikán végrehajtott frissítés végül az összes korrekt replikán végrehajtódik
  - Convergence: Azok a replikák, melyek ugyanazokat a frissítéseket hajtották végre végül ekvivalens állapotba kerülnek
  - Termination: Az összes metódusvégrehajtás terminál

- Strong eventual consistency: Teljesül, ha az eventual consistency feltételei teljesülnek, valamint teljesül:
  - Strong Convergence: Azok a replikák, melyek ugyanazokat a frissítéseket hajtották végre ekvivalens állapotban vannak

Előbbi informálisan azt jelenti, hogy végül minden replika egyazon állapothoz konvergál. Eszerint, ha a rendszert tétlen állapotban hagyjuk, azaz nem hajtunk végre változtatásokat, akkor a replikák állapota végül meg fog egyezni. Az eventual consistency egy gyenge konzisztenciamodell, mely a CAP tétel [18] miatt fontos szerepet játszik elosztott rendszerek esetében. A probléma vele, hogy az ilyen rendszerek a felfedezett konfliktusokat sokszor csak konszenzus segítségével képesek feloldani, mely a roll-backek miatt hosszadalmas tud lenni, s eközben az egyes replikák állapotára nincs semmilyen garancia. Strong eventual consistency esetében azonban nincs szükség konszenzusra az erős konvergencia miatt.

A CRDT-k az utóbbi konzisztenciamodellnek felelnek meg, azaz az ilyen adatszerkezetek biztosítják, hogy a változtatások által okozott konfliktusok mindig helyben feloldhatóak. A CRDT megközelítés célja az OT-val ellentétben nem a műveletek manipulációja, hanem egy megfelelő az adatszerkezet kialakítása, mely biztosítja, hogy költséges szinkronizáció nélkül a replikák garantáltan egy állapothoz konvergálnak.

A CRDT-ket az irodalom két nagy csoportba sorolja, megkülönbözteti a State és Operation-based osztályokat. Shapiro bebizonyította, hogy ezek kifejezőereje ekvivalens, mivel az egyik emulálható a másikkal [19]. Mivel azonban bizonyos adatszerkezetek megvalósítására az egyik, míg más adatszerkezetekez a másik megközelítés terjedt el, érdemesnek tartom bemutatni őket.

3.1. State-based Convergent Replicated Data Type (CvRDT)

Az állapotalapú CRDT-k megadásához a Shapiro által használt specifikációt [13] fogom használni. Ez a következő komponensekből áll:

- payload: A típus állapotát leíró adatok összesége. Ez lehet például egy integer, vagy akár egy halmaz is. Fontos, hogy egy kezdeti állapot megadása kötelező, mely egy replika kezdeti értékét írja le
- query: Olyan műveletek, melyek nem módosítják az állapotot. Tipikus getter metódusok tartoznak ide.
- update: Olyan művelet, melyek módosítják az állapotot.
- merge: Két állapotot fésül össze
- compare: Két állapotot hasonlít össze

Megjegyzendő, hogy mind a query, mint az update műveletek rendelkezhetnek paraméterekkel, visszatérési értékkel, valamint opcionálisan előfeltétellel. Utóbbi megléte esetén csak teljesülő feltételkor hajtható végre a művelet.

Az utolsó két komponens további magyarázatra szorul, megértésükhöz azonban a CvRDT általános működésének ismerete szükséges. Állapotalapú esetben a forrás replika először önmagán végrehajtja a frissitést, majd a teljes payloadját elküldi a többi replikának, azaz a változások a teljes állapot továbbküldésével terjednek. Mikor egy ilyen megérkezik egy replikához, az a merge művelet segítségével frissíti a saját payloadját.

A fenti specifikációt használva elégséges feltétel adható [19] arra, hogy a CvRDT mikor felel meg a strong eventual consistency-nek. Ennek kimondásához a következő fogalmakat vezetjük be:

Legkisebb felső korlát ($\sqcup_v$) $m = x \sqcup_v y$ legkisebb felső korlátja $\{x, y\}$-nak, akkor és csak akkor, ha $x \leq_v m$ és $y \leq_v m$ és nincs $m' \leq_v m$ melyre $x \leq_v m'$ és $y \leq_v m'$.

A definícióból adódik, hogy $\sqcup_v$ kommutatív, idempotens és asszociatív.

Join Semilattice Egy rendezett $(S, \leq_v)$ halmaz Join Semilattice, akkor és csak akkor, ha $\forall x, y \in S \; \exists \; x \sqcup_v y$

A CvRDT-ktől megköveteljük, hogy payloadjuk Join Semilattice-t alkosson, ahol a merge definíció szerint megegyezik $\sqcup_v$-vel, a compare pedig $\leq_v$-vel. Továbbá megköveteljük, hogy a frissítések monoton növelnek $\leq_v$ szerint, azaz $s \leq s \circ u$. Ezen

feltételek mellett a replikák a legkisebb felső korlát felé konvergálnak, mely a legfrissebb állapotot takarja. Feltéve az Eventual delivery és Termination követelményeket, a CvRDT-k megfelelnek a strong eventual consistencynek. [19]

Összefoglalva, CvRDT-k esetén a változtatások propagálása teljes állapot-reprezentációjuk továbbküldésével történik. Az adatszerkezet biztosít egy merge metódust, mely segítségével egy replika a beérkező reprezentációt összefésüli saját reprezentációjával, így az esetleges változtatások rajta is megjelennek. Mivel minden ilyen merge művelet véglegesnek tekinthető (nincs szükség utólagos konszenzusra), a merge-nek idempontensnek, kommutatívnak és asszociatívnak kell lennie. Fontos megjegyezni, hogy ezen tulajdonságok miatt a módszer jól működik gyenge megbízhatósággal rendelkező hálózatok esetén: üzenetek elveszhetnek, felcserélődhetnek, vagy többször megérkezhetnek egészen addig, amíg az új állapot végül eléri az összes replikát. A frissítések a hálózat kiesését is tudják tolerálni, ha a kapcsolat végül helyreáll.

3.2. Op-based Commutative Replicated Data Type (CmRDT)

Hasonlóan az előbbihez, a műveletalapú CRDT-k megadásához is a Shapiro által használt specifikációt használom.

- payload: A típus állapotát leíró adatok. Ez lehet például egy integer, vagy akár egy halmaz is. Fontos, hogy egy kezdeti állapot megadása kötelező, mely egy replika kezdeti értékét írja le
- query: Olyan műveletek, melyek nem módosítják az állapotot. Tipikus getter metódusok tartoznak ide.
- update: Olyan művelet, melyek módosítják az állapotot. Két további részre osztható:
  - atSource: A frissités első fázisa, helyben a forrás replikán hajtódik végre (mellékhatásmentes, a második fázishoz esetleg szükséges adatok kiszámításához)

– downStream: A második fázis rögtön az első után hajtódik végre, először a forrás replikán, majd az összes többi replikán

Az állapotalapú megközelítéstől eltérően ebben az esetben nem a payload, hanem csupán egy függvény kerül továbbküldésre. Ennek alkalmazásával a megfelelő változtatások a többi replikán is végrehajtódnak. Ez a műveletalapú megközelítés nagyon hasonlónak látszik az Operational Transformationnél látott módszerhez, azonban fontos különbség, hogy ebben az esetben nincs szükség a műveletek transzformációjára. Ezt a CmRDT-k azzal biztosítják, hogy a konkurens frissítések kommutatívak. A kommunikációs csatornára tett egyéb megszorítások mellett ez elégséges feltétele is a SEC-nek.

3.3. Számláló CRDT példa

A koncepció alátámasztására Shapiro számos gyakorlatban működő CRDT-t kidoglozott, ebben a fejezetben egy nagyon egyszerű adatszerkezet, az increment-only számláló State és OP-based megvalósítását mutatom be ezek közül. Az adatszerkezet egy számot reprezentál, melyet egyetlen műveletével, az `increment`-tel lehet eggyel növelni.

3.3.1. Op-based

Az adatszerkezet egyszerűbben megvalósítható műveletalapú megközelítés segítségével. Ennek specifikációja megtalálható a 3.1 ábrán.

```
1 payload: integer i
2     initial 0
3 query value () : integer j
4     let j = i
5 update increment ()
6     downstream ()
7         i := i + 1
```

3.1. forráskód. Op-based increment-only számláló

A reprezentációhoz elegendő egyetlen `integer` adattag, melynek értékét a `value query` segítségével tudjuk lekérdezni. Egyetlen `update` metódussal rendelkezik, az

`increment`-tel, melynek `atSource` komponense üres, hiszen nincs semmilyen előfeldolgozásra szükség az adatokon. `Downstream` komponense a reprezentáló `integer` értékét növeli meg eggyel.

Könnyen belátható, hogy ez a művelet valóban kommutatív, így ha minden replikához eljut minden művelet, a fenti specifikációval megadott adatszerkezet valóban CmRDT.

Megjegyzendő, hogy az adatszerkezet nagyon egyszerűen kiegészíthető számlálóvá, melyet növelni és csökkenteni is lehet. Ehhez csupán egy új `update` metódusra van szükség, a `decrement`-re. Ez a reprezentáló integer értékét eggyel csökkenti. Itt is belátható, hogy a két update művelet kommutatív.

3.3.2. State-based

Állapotalapú megközelítés segítségével némileg bonyolultabb megvalósítani az adatszerkezetet. Ennek specifikációja a 3.2 ábrán látható.

```
1  payload integer[n] P
2      initial [0, 0, ..., 0]
3  query value () : integer v
4      let v = ∑_i P[i]
5  update increment ()
6      let g = myID()
7      P[g] := P[g] + 1
8  compare (X, Y) : boolean b
9      let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i])
10 merge (X, Y ) : payload Z
11     let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

3.2. forráskód. State-based increment-only számláló

Látható, hogy ezesetben nem elegendő egy integer a reprezentációhoz, hanem egy integer tömbre van szükség, melynek mérete megegyezik a lehetséges replikák számával. Könnyen belátható, hogy előbbi esetén már két replika esetén sem adható megfelelő `merge` metódus. Tegyük fel, hogy mindkét replika értéke 2, melyen az egyik 3-at, míg a másik csak 2-t növelt. Ekkor az egyik replikán értéke 5, míg a másikon 4, az állapot, ahová konvergálni szeretnénk pedig (mivel összesen 5 darab növelés

történt) 7. Ez a két payloadból nem határozható meg, szükség lenne valamilyen szinkronizáció, konszenzus bevezetésére, ami sértené a strong eventual consistencyt.

Ehelyett az adatszerkezetet a korábban ismertetett módon reprezentálva, az `increment` függvény megvalósítható, ha minden replika a saját azonosítójának megfelelő indexen lévő értéket növeli. A számláló értéke ebben az esetben a reprezentáló tömb összegzésével adható vissza. Így már adható megfelelő `merge` függvény is, mely a kapott két tömbből úgy állítja elő a harmadikat, hogy minden indexen veszi a kapott két érték maximumát. Belátható, hogy ez a függvény idempotens, kommutatív és asszociatív. Az objektumok között megadható a rendezés is. Úgy tűnhet, a reprezentáció feltételezi, hogy a replikák száma pontosan ismert, míg műveletalapú esetben erre nem volt szükség. Valójában azonban a másik módszer esetében szükséges ismerni a replikákat az üzenetek küldéséhez, így nem tettünk plusz megszorítást a rendszer egészére nézve.

Ennek az adatszerkezetnek számlálóvá való bővítése nem magától értetődő. Ha műveletalapú esethez hasonlóan a `decrement` művelet a saját azonosítójának megfelelő indexen lévő értéket csökkenti, két helyen is problémába ütközünk: egyrészt a frissítések ekkor nem monoton növelnek, másrész a `merge` maximumfüggvénye sem működne. A megoldáshoz még egy integer tömb bevezetésére van szükség az egyes replikákon történt `decrement` műveletek számolásához. A `merge` ekkor mindkét tömbön maximumot vesz, a `compare` pedig kibővül a másik tömb vizsgálatával is.

4. fejezet

Szövegszerkesztés CRDT-k segítségével

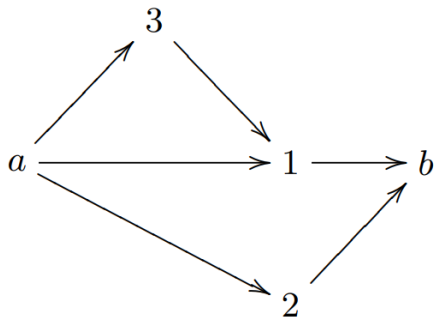
A CRDT koncepció tehát alkalmazható sok jól ismert adatszerkezet megvalósítására, viszont nem magától értetődő, hogyan lehet használni valós idejű szövegszerkesztésre. Az alábbiakban az erre kidolgozott algoritmusok alapötletét mutatom be és hasonlítom össze, valamint egy konkrét implementációt, az Yjs keretrendszer alapjait ismertetem.

4.1. Megközelítések

Az először kifejlesztett algoritmusok közé tartozik a `WOOT` [20] és a `Treedoc` [21] [22]. Bár hatékonyságban mindkettő elmarad a később kidolgozott módszerekhez képest [23], számos, máshol is alkalmazott alapgondolat már ezeknél megjelent. Később ezekre épülve már kidolgozásra kerülhettek hatékonyabb algoritmusok, mint például a `Logoot/LSEQ` [24] [25] vagy az `RGA` [26]. Az algoritmusok időkomplexitása látható a 4.5 ábrán.

4.1.1. WOOT

A `WOOT` megközelítés esetében a műveletek pozíciók helyett karaktereket reprezentáló elemekre vonatkoznak, ebből adódóan az elemeknek egyedien azonosíthatónak kell lenniük. Például az `insert` $(a \prec e \prec c)$ művelet az e elemet beszújra az a és b elemek közé, a `del` (a) pedig kitörli az a azonosítójú karaktert. Ekkor előfor-



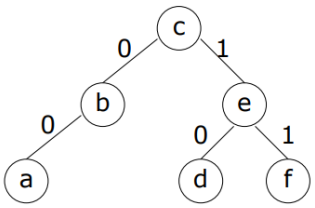
4.1. ábra. WOOT alapötlete [20]

dulhat, hogy valamely replika olyan `insert` műveletet kap, melyhez tartozó a vagy c elemet már lokálisan törölte. Hogy ez ne okozzon problémát, a törölt elemeket az algoritmus nem törli ténylegesen, csupán megjelöli, ezeket a megjelölt elemeket az irodalom "tombstone"-oknak nevezi.

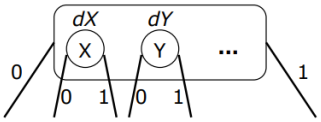
A `WOOT` alapötlete, hogy minden elemhez nyilvántart referenciákat, hogy mely két elem közé kell beszúrni azonban nem elég az elemek sorbarendezéséhez, ennek feloldásához további egyedi karakterazonosítókra, és algoritmusokra van szükség. Ez látható a 4.1 ábrán is, ahol a lehetséges karaktersorozatok: $a312b$, $a321b$, és $a231b$.

4.1.2. Treedoc

A `Treedoc` megközelítés szintén egyedi azonosítókat használ az elemek megkülönböztetésére, ez az azonosító azonban az elem helyét is meghatározza: ebből adódóan az azonosítók értékkészletének tetszőlegesen sűrűnek kell lennie (egyébként létezhetne olyan eset, mikor a beszúrás nem lehetséges két elem közé). A valós számok halmaza ilyen, azonban sok karakter beszúrása esetén a szükséges pontosság kezelhetetlen lenne. Ehelyett az azonosítók egy lehetséges utat reprezentálnak egy n-áris fában (4.2. ábra). A fa megfelelő bejárásával megkapható az ábrázolt karaktersorozat. Az `insert` műveletkor a replika meghatároz egy új azonosítót, mely a megfelelő csúcshoz vezet a fában, és beszúrja azt. Mivel előfordulhat, hogy két replika ugyanazt az azonosítót generálja, a fa egyes csúcsaiba egyszerre több karakter is kerülhet (4.3. ábra), ezek egyértelmű sorrendjét rendezett replika azonosítók segítségével lehet biztosítani (ezen karakterek közé aztán hasonló módon szúrhatóak be elemek). `Delete` művelet hasonlóan az út meghatározásával lehetséges. Ekkor a csúcs a `WOOT`-hoz ha-



4.2. ábra. Treedoc dokumentum. [21]
b azonosítója 0, *d*-é 10



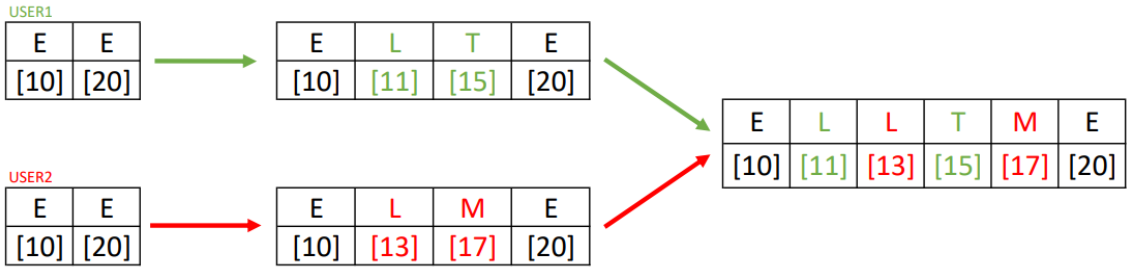
4.3. ábra. Konkurens beszúrások esetény több karakter egy csúcsban[21]

sonlóan tombstone-ként kerül megjelölésre, hiszen más karaktereket azonosító úton szerepelhet. A sok beszúrás és törlés miatt a fa mérete (és ezzel arányosan a generált azonosítók mérete, melyek az üzenetekben elküldésre kerülnek) megnövekedhet, valamint szerkezete kiegyensúlyozatlanná válhat (beszúrás és törlés költségét növeli), ezért kiegyensúlyozó és szemétgyűjtő algoritmusokra is szükség lehet.

4.1.3. Logoot/LSEQ

A `Logoot` módszere az előzőhöz hasonlóan tetszőlegesen sűrű értékkészletből választott egyedi azonosítókra épül, melyek meghatározzák a hozzájuk tartozó elem pozícióját a szövegben. Az eredeti módszerben ezek az elemek sorokat reprezentálnak, de természetesen karakterekre is alkalmazható. A módszer alapötlete, hogy az azonosítók megfelelően reprezentálhatóak listákkal. Tegyük fel, hogy adott két elem, rendre $[4], [5]$ azonosítóval. A két elem közé beszúrni a két lista közötti azonosító meghatározásával lehet: mivel nincs egész szám 4 és 5 között, egyszerűen bővítjük az első listát egy generált számmal, legyen ez 7. Ezután az azonosítók: $[4], [4, 7], [5]$. Ha két replika azonos számot generál, szintén rendezett replika azonosítók döntenek a sorrendben. Az eljárás tetszőlegesen sokszor ismételhető, így bármelyik pozícióra újabb karakterek szúrhatóak be. A törlés egyszerűen a megfelelő azonosító megadásával történik. Fontos különbség az eddigi algoritmusokkal szemben, hogy törlésnél nincs szükség tombstone-okra, mivel az elemek nem hivatkozzák egymást. Az azonosítók hossza azonban szintén nagy lehet, ezért beszúrásnál fontos, hogyan generálunk új számokat. Az `LSEQ` erre hatékonyabb módszert használ a `Logoot`nál.

Probléma a két algoritmussal, hogy azonos pozícióra történő konkurens beszúrások átfedhetnek, ahogy az a 4.4. ábrán látható. Tegyük fel,



4.4. ábra. Logoot probléma

hogy adott a `<([10],E),([20],E)>` `Logoot` objektum. Az egyik replika két karaktert szúr be, mely után `<([10],E),([11],L),([15],T),([20],E)>` állapot alakul ki. Ezzel egyidőben egy másik replika szintén két karaktert szúr be `<([10],E),([13],L),([17],M),([20],E)>` állapotot eredményezve. Az összefésülés eredménye ekkor `<([10],E),([11],L),([13],L),([15],T),([17],M),([20],E)>`. Látható, hogy a felhasználók által egymás mellé szánt karakterek nincsenek egymás mellett.

4.1.4. RGA

A `Replicated Growing Array (RGA)` megközelítés egyirányú láncolt listát használ a szöveg reprezentálásához. Két műveletet definiál: az `addRight(v,a)` segítségével az `a` elemet egyből a `v` elem után szúrja be. Hasonlóan az eddigi megközelítésekhez, azonos pozícióra történő beszúrások feloldására rendezett replika azonosítók használhatóak. Törlés hatására tombstone-ok keletkeznek, hasonlóan a `WOOT`-hoz, hogy a konkurens beszúrások végrehajthatóak maradjanak. A módszer újítása, hogy hash táblát használ a hatékonyság növeléséhez.

4.2. Yjs

Az Yjs [28] [29] egy ingyenesen elérhető, CRDT implementációkat tartalmazó keretrendszer. Megosztott adatszerkezeteket ("shared types") biztosít, mint például `Map`, `Array` vagy `Text`. Utóbbihoz a `YATA` CRDT [27] egy módosított, optimalizált változatát használja. Ez a fent ismertetett algoritmusokhoz nagyon hasonló, egy kétirányú láncolt listát használ a reprezentációhoz, a beszúráshoz pedig saját algoritmusát. A láncolt listából adódóan törlésnél tombstone keletkezik, ennek kontro-

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

4.5. ábra. Időkomplexitás [23] [27]

lálása szemétgyűjtő algoritmussal történik. A műveletek gyorsítása érdekében pedig az elemeket fa-struktúrába szervezi.

A könyvtár nagy előnye a flexibilitása, a biztosított CRDT-k mind szerver-kliens mind P2P környezetben működnek. Ezenfelül előnye, hogy számos online text-editorral kompatibilis különböző kötéseken, bindingokon keresztül. A 4.1 példán egy Monaco editorral való összekapcsolása látható, WebSocket alapú kommunikációval.

```
let ydocument = new Y.Doc()
let provider = new WebsocketProvider(serverAddress,
                                     ydocument)
let type = ydocument.getText('monaco')

const editor = monaco.editor.create(
                         document.getElementById('monaco-editor'),
                         {
                           value: '',
                           language: "javascript",
                           theme: 'vs-dark'
                         })

let monacoBinding = new MonacoBinding(type,
                                      editor.getModel(),
                                      new Set([editor]),
                                      provider.awareness)
```

4.1. forráskód. Yjs példa

A fentieken túl egyszerűen létrehozhatóak szobák, mely tartalmát csak bizonyos

felhasználók szerkeszthetik. További előnye, hogy az offline állapotba került kliensek újrakapcsolódáskor automatikusan szinkronizálnak. CRDT-k esetén az undo redo műveletek kivitelezése problémás lehet, ezt az Yjs testreszabható formában valósítja meg.

A könyvtárra számos alkalmazás épül, nem csak szövegszerkesztők, de például rajzprogramok is megvalósíthatók segítségével.

5. fejezet

Konklúzió

A szövegszerkesztés bár egy felhasználónál egyszerűen megvalósítható, több felhasználós, konkurens megvalósítása, megfelelés a CCI feltételeknek nem triviális feladat.

Az első megközelítés, melyet vizsgáltunk, az Operational transformation. Ez a konkurens műveleteken végzett transzformációkkal oldja fel a konfliktusokat. Sajnos ezek az algoritmusok gyakran igényelik központi szerver jelenlétét, és bizonyos esetekben hibásan működhetnek. Ennek ellenére sok területen alkalmazzák az iparban.

A másik vizsgált megközelítés a Conflict-free replicated data type-ok használata. Ezek az elosztott adatszerkezetek biztosítják a Strong eventual consistency-t. Ebből kifolyólag nincs szükség konszenzusra, roll-backekre, mivel minden konfliktus garantáltan helyben feloldható lesz, valamint a rendszer egy állapothoz konvergál. A CRDT-k két nagy csoportra oszthatóak, aszerint, hogy a változtatások milyen módon propagálódnak a hálózaton. A State-based megközelítés esetében először helyben hajtódik végre a változtatás, majd az adatszerkezet teljes reprezentációja elküldésre kerül, és a fogadó oldalon összefésüléssel állítható elő az új állapot. Az összefésülő függvény szükségképpen idempotens, kommutatív és asszociatív tulajdonságú. Az Op-based irány ezzel szemben műveleteket elküldésével továbbítja a változtatásokat, ezek a műveletek kommutatívak. Az állapotalapú megközelítés kevésbé megbízható hálózatokon is jól működik, azonban a műveletalapú megközelítéssel szemben a hálózati adatforgalom nagyobb. Az utóbbi használata terjedt el szövegszerkesztésre, melyre már számos CRDT ismert. Fent bemutatásra került a `WOOT`, `Treedoc`, `Logoot/LSEQ`, `RGA` ezek közül csak néhány. Újabb kutatások a

fenti osztályok szintézisére, az OT és CRDT megközelítés egyesítésére koncentrálnak: ezeknek az eredménye például a Causal Tree [30] és a Pure Operation-Based Replicated Data Types [31].

Az OT és CRDT mindegyike kielégítően használható kollaboratív szerkesztésre. Általánosságban elmondható, hogy az időkomplexitása a CRDT alapú rendszereknek jobb, tárkomplexitása ezeknek viszont rosszabb az OT-hoz képest [32]. Egy harmadik, kevésbé kutatott módszer a Differentional synchronization [33], mely szintén egy központi szerver segítségét igényli.

A fentiekből látható, hogy számos jól használható, érdekes ötlet született a témában, és az ezekre épülő, valós idejű kollaboratív alkalmazások számos ember munkáját könnyítik meg nap mint nap.

Irodalomjegyzék

[1] Chengzheng Sun és tsai. “Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems”. *ACM Trans. Comput.-Hum. Interact.* 5.1 (1998. márc.), 63–108. ISSN: 1073-0516. DOI: 10.1145/274444.274447. URL: https://doi.org/10.1145/274444.274447.

[2] Ning Gu, Jiangming Yang és Qiwei Zhang. “Consistency Maintenance Based on the Mark & Retrace Technique in Groupware Systems”. *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*. GROUP ’05. New York, NY, USA: Association for Computing Machinery, 2005, 264–273. ISBN: 1595932232. DOI: 10.1145/1099203.1099250. URL: https://doi.org/10.1145/1099203.1099250.

[3] Kumawat Santosh és Ajay Khunteta. “A Survey on Operational Transformation Algorithms: Challenges, Issues and Achievements”. *International Journal of Computer Applications* 3 (2010. júl.). DOI: 10.5120/787-1115.

[4] Nicolas Vidot és tsai. “Copies Convergence in a Distributed Real-Time Collaborative Environment”. *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*. CSCW ’00. New York, NY, USA: Association for Computing Machinery, 2000, 171–180. ISBN: 1581132220. DOI: 10.1145/358916.358988. URL: https://doi.org/10.1145/358916.358988.

[5] Haifeng Shen és Chengzheng Sun. “Flexible Merging for Asynchronous Collaborative Systems”. *CoopIS/DOA/ODBASE*. 2002.

[6] Abdessamad Imine és tsai. “Proving Correctness of Transformation Functions in Real-Time Groupware”. *Proceedings of the Eighth Conference on European*

Conference on Computer Supported Cooperative Work. ECSCW'03. USA: Kluwer Academic Publishers, 2003, 277–293.

[7] Google Drive. *What's different about the new Google Docs: Making collaboration fast*. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html. [Hozzáférés: 2020.01.30.] 2010.

[8] Inc. AppJet. *Etherpad and EasySync technical manual. [Online]*. `https://github.com/ether/etherpad-lite/blob/e2ce9dc/doc/easysync/easysync-full-description.pdf`. 2011.

[9] Maher Suleiman, Michèle Cart és Jean Ferrié. "Serialization of Concurrent Operations in a Distributed Collaborative Environment". *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*. GROUP '97. New York, NY, USA: Association for Computing Machinery, 1997, 435–445. ISBN: 0897918975. DOI: `10.1145/266838.267369`. URL: `https://doi.org/10.1145/266838.267369`.

[10] Chengzheng Sun és Clarence Ellis. "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements". *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW '98. New York, NY, USA: Association for Computing Machinery, 1998, 59–68. ISBN: 1581130090. DOI: `10.1145/289444.289469`. URL: `https://doi.org/10.1145/289444.289469`.

[11] Rui Li és Du Li. "A Landmark-Based Transformation Approach to Concurrency Control in Group Editors". *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*. GROUP '05. New York, NY, USA: Association for Computing Machinery, 2005, 284–293. ISBN: 1595932232. DOI: `10.1145/1099203.1099252`. URL: `https://doi.org/10.1145/1099203.1099252`.

[12] Marc Shapiro és tsai. "Conflict-free Replicated Data Types". *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*. Szerk. Xavier Défago, Franck Petit és Vincent Villain. 6976. köt. Lecture Notes in Computer Science. Grenoble, France: Springer, 2011. okt.,

386–400. old. DOI: 10.1007/978-3-642-24550-3_29. URL: https://hal.inria.fr/hal-00932836.

[13] Marc Shapiro és tsai. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, 2011. jan., 50. old. URL: https://hal.inria.fr/inria-00555588.

[14] Martin Kleppmann és Alastair Beresford. "A Conflict-Free Replicated JSON Datatype". *IEEE Transactions on Parallel and Distributed Systems* PP (2016. aug.). DOI: 10.1109/TPDS.2017.2697382.

[15] Russell Brown és tsai. "Riak DT Map: A Composable, Convergent Replicated Dictionary". *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*. PaPEC '14. New York, NY, USA: Association for Computing Machinery, 2014. ISBN: 9781450327169. DOI: 10.1145/2596631.2596633. URL: https://doi.org/10.1145/2596631.2596633.

[16] Ahmed Bouajjani, Constantin Enea és Jad Hamza. "Verifying Eventual Consistency of Optimistic Replication Systems". *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014, 285–296. ISBN: 9781450325448. DOI: 10.1145/2535838.2535877. URL: https://doi.org/10.1145/2535838.2535877.

[17] Yasushi Saito és Marc Shapiro. *Optimistic Replication*. Technical Report. Refer to rep:syn:1500 instead. Microsoft Research, 2003. URL: https://hal.inria.fr/inria-00444768.

[18] Seth Gilbert és Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". *SIGACT News* 33.2 (2002. jún.), 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[19] Marc Shapiro és tsai. *Conflict-free Replicated Data Types*. Research Report RR-7687. 2011. júl., 18. old. URL: https://hal.inria.fr/inria-00609399.

[20] Gérald Oster és tsai. “Data Consistency for P2P Collaborative Editing”. *ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*. Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work. http://portal.acm.org/. Banff, Alberta, Canada: ACM Press, 2006. nov., 259 –268. old. URL: https://hal.inria.fr/inria-00108523.

[21] Mihai Leția, Nuno Preguiça és Marc Shapiro. *CRDTs: Consistency without concurrency control*. Research Report RR-6956. INRIA, 2009, 16. old. URL: https://hal.inria.fr/inria-00397981.

[22] Nuno Preguiça és tsai. “A commutative replicated data type for cooperative editing”. *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*. Montreal, Québec, Canada: IEEE Computer Society, 2009. jún., 395–403. old. DOI: 10.1109/ICDCS.2009.20. URL: https://hal.inria.fr/inria-00445975.

[23] Mehdi Ahmed-Nacer és tsai. “Evaluating CRDTs for Real-time Document Editing”. *11th ACM Symposium on Document Engineering*. Szerk. ACM. Mountain View, California, United States, 2011. szept., 103–112. old. DOI: 10.1145/2034691.2034717. URL: https://hal.inria.fr/inria-00629503.

[24] Stéphane Weiss, Pascal Urso és Pascal Molli. *Logoot: a P2P collaborative editing system*. Research Report RR-6713. INRIA, 2008, 13. old. URL: https://hal.inria.fr/inria-00336191.

[25] Brice Nédelec és tsai. “LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing”. *13th ACM Symposium on Document Engineering (DocEng)*. 10 pages. Florence, Italy, 2013. szept., 37–46. old. DOI: 10.1145/2494266.2494278. URL: https://hal.archives-ouvertes.fr/hal-00921633.

[26] Hyun-Gul Roh és tsai. “Replicated abstract data types: Building blocks for collaborative applications”. *J. Parallel Distrib. Comput.* 71 (2011), 354–368. old.

[27] Petru Nicolaescu és tsai. “Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types”. *GROUP ’16*. 2016.

[28] Petru Nicolaescu és tsai. “Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types”. *Engineering the Web in the Big Data Era*. Szerk. Philipp Cimiano és tsai. Cham: Springer International Publishing, 2015, 675–678. old. ISBN: 978-3-319-19890-3.

[29] Kevin Jahns. *Yjs*. `https://github.com/yjs/yjs`. 2019.

[30] Victor Grishchenko. “Deep Hypertext with Embedded Revision Control Implemented in Regular Expressions”. *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*. WikiSym ’10. Gdansk, Poland: Association for Computing Machinery, 2010. ISBN: 9781450300568. DOI: 10.1145/1832772.1832777. URL: https://doi.org/10.1145/1832772.1832777.

[31] Carlos Baquero, Paulo Almeida és Ali Shoker. “Pure Operation-Based Replicated Data Types”. (2017. okt.).

[32] David Sun és tsai. *Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors*. 2019. máj.

[33] Neil Fraser. “Differential Synchronization”. *DocEng’09, Proceedings of the 2009 ACM Symposium on Document Engineering*. 2 Penn Plaza, Suite 701, New York, New York 10121-0701, 2009, 13–20. old. URL: http://neil.fraser.name/writing/sync/eng047-fraser.pdf.