

Transformer les opérations

Un serveur, des révisions et un losange qui tient

losange

TP1 et TP2

Jupiter

révision

composition

Une position n'a de sens que dans une version du texte. Depuis 1989, l'OT déplace les éditions plutôt que de renommer les lettres ; voici ce que cette idée a coûté et ce qu'elle a rapporté.

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Éditeur OT » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

Dans le projet « éditeur OT », vous complétez cinq clauses d'une fonction de transformation sur des opérations qui parcourent tout le document, puis vous observez le protocole fourni : un serveur qui numérote les révisions, trois clients qui gardent une opération en vol et composent les suivantes dans un buffer, et des expériences où l'ordre d'acceptation change le texte final sans empêcher la convergence. Le compte rendu vous demande un losange complet, des scénarios personnels et une explication du rôle des révisions.

Ce fascicule ne donne aucune de ces clauses. Il raconte l'histoire de l'idée, de l'éditeur GROVE de 1989 à Google Docs ; il déroule à la main le losange sur une représentation plus simple que celle du projet, des opérations à une seule position ; il explique les deux propriétés TP1 et TP2, pourquoi la seconde a résisté si longtemps et comment un serveur central permet de s'en passer ; il rapporte enfin le débat, toujours ouvert, entre les partisans de l'OT et ceux des CRDT, et ce que font les systèmes réels. Rien n'est à rendre : lisez-le comme un récit, un soir, avec le projet à portée de main.

2 Une position appartient à une version

Tout le sujet tient dans une observation que vous avez faite dès l'expérience A : « insérer à l'indice 1 » ne désigne pas la même chose avant et après qu'un autre a supprimé l'indice 0. Commençons par mesurer le souci, puis par le résoudre sur le cas le plus petit possible.

2.1 Les positions ne commutent pas

Deux insertions au même indice, appliquées dans deux ordres, donnent deux textes. Voici la mesure, en quelques lignes.

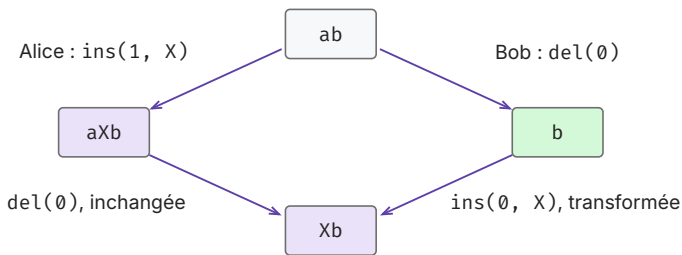
```
1 insérer = fn texte, p, c ->
2   String.slice(texte, 0, p) <> c <> String.slice(texte, p..-1//1)
3 end
4 alice_puis_bob = "ab" |> insérer.(1, "X") |> insérer.(1, "Y")
5 bob_puis_alice = "ab" |> insérer.(1, "Y") |> insérer.(1, "X")
6 IO.inspect({alice_puis_bob, bob_puis_alice})
```

CONSOLE {"aYXb", "aXYb"}

Si chaque réplique applique les éditions dans l'ordre où elles lui parviennent, Alice et Bob finissent avec deux textes différents. Un CRDT répond en donnant à chaque lettre un nom qui ne bouge plus ; c'est l'autre projet. La *transformation opérationnelle* (*operational transformation*, OT) répond en gardant les positions mais en *adaptant* chaque édition à ce qui a été appliqué avant elle sur la copie réceptrice.

2.2 Le losange, à la main

Prenons la représentation la plus simple qui soit, celle des premiers articles : une opération est soit `ins(p, c)`, insérer le caractère `c` à la position `p`, soit `del(p)`, supprimer le caractère de position `p`. Alice et Bob partent de `ab`. Alice produit `ins(1, x)` et voit `aXb` ; Bob produit `del(0)` et voit `b`. Chacun doit maintenant intégrer l'édition de l'autre.



Le losange. Bob reçoit `ins(1, x)` : après sa suppression, la position 1 est la fin du texte, l'insertion doit être déplacée en 0. Alice reçoit `del(0)` : son insertion était après la position 0, la suppression reste en 0. Les deux chemins donnent `x b`.

Écrivons `transform(a, b)` pour « l'opération `a`, adaptée au texte obtenu après `b` », les deux opérations ayant été produites sur le même texte de départ. Les règles tiennent en quelques cas. Une insertion placée avant la position de l'autre opération ne bouge pas ; placée après, elle se décale d'une case vers la droite si l'autre a inséré, vers la gauche si l'autre a supprimé. Une suppression suit la même logique. Deux insertions à la même position demandent une convention, par ex. l'ordre des noms de sites, qui doit être la même chez tout le monde. Enfin, deux suppressions de la même lettre ne doivent pas supprimer deux fois : la seconde devient l'opération vide. Voici ces règles en Elixir, puis le losange vérifié ; chaque insertion porte son site, qui sert de départageur.

```

1 defmodule Position do
2   def apply(texte, {ins, p, c, _}) do
3     {avant, apres} = String.split_at(texte, p)
4     avant <> c <> apres
5   end

```

```

6
7  def apply(texte, {:del, p}) do
8      {avant, apres} = String.split_at(texte, p)
9      avant <> String.slice(apres, 1..-1//1)
10  end
11
12  def apply(texte, :rien), do: texte
13
14  # transform(a, b) adapte a au texte obtenu après b.
15  def transform({:ins, pa, c, sa}, {:ins, pb, _, sb})
16      when pa < pb or (pa == pb and sa < sb),
17      do: {:ins, pa, c, sa}
18  def transform({:ins, pa, c, sa}, {:ins, _, _, _}),
19      do: {:ins, pa + 1, c, sa}
20  def transform({:ins, pa, c, sa}, {:del, pb}) when pa <= pb,
21      do: {:ins, pa, c, sa}
22  def transform({:ins, pa, c, sa}, {:del, _}),
23      do: {:ins, pa - 1, c, sa}
24  def transform({:del, pa}, {:ins, pb, _, _}) when pa < pb,
25      do: {:del, pa}
26  def transform({:del, pa}, {:ins, _, _, _}), do: {:del, pa + 1}
27  def transform({:del, pa}, {:del, pb}) when pa < pb, do: {:del, pa}
28  def transform({:del, pa}, {:del, pb}) when pa > pb,
29      do: {:del, pa - 1}
30  def transform({:del, _}, {:del, _}), do: :rien
31  end
32
33  alice = {:ins, 1, "X", :alice}
34  bob = {:del, 0}
35  bob2 = Position.transform(bob, alice)
36  alice2 = Position.transform(alice, bob)
37  par_alice = "ab" |> Position.apply(alice) |> Position.apply(bob2)
38  par_bob = "ab" |> Position.apply(bob) |> Position.apply(alice2)
39  IO.inspect({par_alice, par_bob, alice2})

```

```
CONSOLE  {"Xb", "Xb", {:ins, 0, "X", :alice}}
```

Les deux chemins se rejoignent. Cette égalité, pour toute paire d'opérations produites sur le même texte, est la première propriété de transformation.

TP1, LA PROPRIÉTÉ DU LOSANGE

Pour deux opérations a et b définies sur le même texte t , appliquer a puis $\text{transform}(b, a)$ donne le même texte qu'appliquer b puis $\text{transform}(a, b)$. Pourquoi c'est vrai ici : chaque règle ne fait que recalculer où se trouve, après l'autre opération, la lettre que visait la première ; la seule liberté est le départage de deux insertions au même endroit, et il est identique des deux côtés.

Le projet n'utilise pas cette représentation. Ses opérations décrivent tout le document par une suite de `retain`, `insert` et `delete`, ce qui permet de transporter une édition entière en un seul message et de la composer avec la suivante. Le raisonnement est le même, la mécanique est différente, et elle vous appartient.

3 De GROVE à Google Docs

L'OT a trente-sept ans, et son histoire est celle d'une idée simple dont les détails ont résisté longtemps. Les dates ci-dessous sont celles des publications ; chacune a sa place dans la suite du fascicule.

année	qui	quoi
1989	Clarence Ellis et Simon Gibbs	l'éditeur GROVE et l'algorithme dOPT : la première transformation d'opérations
1995	David Nichols, Pavel Curtis, Michael Dixon, John Lamping	Jupiter, à Xerox PARC : un serveur central, chaque client ne transforme que face à lui
1996	Matthias Ressel, Doris Nitsche-Ruhland, Rul Gunzenhäuser	adOPTed, et les deux propriétés TP1 et TP2
1998	Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, David Chen	GOT et le modèle CCI : convergence, causalité, intention
2003	Abdessamad Imine, Pascal Molli, Gérard Oster, Michaël Rusinowitch	des contre-exemples à TP2 trouvés par un prouveur automatique

année	qui	quoi
2006	Gérald Oster, Pascal Molli, Pascal Urso, Abdessamad Imine	TTF : des pierres tombales pour rendre TP2 vraie
2009	Google	Wave, un éditeur temps réel fondé sur l'OT à serveur central
2010	Google	Google Docs explique publiquement son usage de l'OT

Clarence Ellis et Simon Gibbs décrivent en 1989, dans l'article « Concurrency control in groupware systems », un éditeur de plans à plusieurs mains, GROVE, et l'algorithme dOPT qui le fait fonctionner sans verrou : chaque site applique ses propres éditions immédiatement, puis transforme celles des autres à leur arrivée, en s'appuyant sur des vecteurs d'état pour savoir quelles éditions sont concurrentes et sur des priorités pour départager les égalités. C'est le geste fondateur. Une faille de dOPT, connue sous le nom de « dOPT puzzle », est repérée au milieu des années 1990 : dans certaines suites de trois éditions concurrentes, les sites divergent. Matthias Ressel, Doris Nitsche-Ruhland et Rul Gunzenhäuser proposent en 1996 l'algorithme adOPTed et, surtout, isolent les deux propriétés qu'une fonction de transformation doit vérifier pour qu'un système sans serveur converge : TP1, que vous venez de voir, et TP2, qui occupe la section suivante.

Entre-temps, à Xerox PARC, David Nichols, Pavel Curtis, Michael Dixon et John Lamping ont construit Jupiter (1995), un monde virtuel partagé dont les fenêtres se modifient à plusieurs. Leur choix d'architecture, un serveur central par lequel passe toute édition, simplifie radicalement la transformation. En 1998, Chengzheng Sun et ses collègues formulent le modèle CCI, qui ajoute à la convergence et à la causalité une troisième exigence, la « préservation de l'intention », et proposent l'algorithme GOT. Les années 2000 sont celles de la vérification : les contre-exemples de 2003 et la réponse de 2006 sont le sujet de la section 4.

Le grand public rencontre l'OT en 2009 avec Google Wave, présenté à la conférence Google I/O. Son livre blanc, signé David Wang, Alex Mah et Soren Lassen, décrit un protocole à serveur central où chaque client ne garde qu'une opération en vol et compose les suivantes en attendant l'accusé ; vous reconnaîtrez le protocole du projet. Wave est abandonné dès 2010 et confié à la fondation Apache, mais son moteur survit : en septembre 2010, l'équipe de Google Docs publie une série de trois billets, dont le dernier, signé John Day-Richter, explique que le nouvel éditeur repose sur la transformation opérationnelle. Joseph Gentle, qui a travaillé sur Wave, réimplémente ensuite son modèle dans ShareJS, devenu ShareDB, l'une des bibliothèques OT les plus utilisées sur le web.

4 TP2, la propriété qui résiste

TP1 porte sur deux opérations. Dès qu'un système sans serveur doit intégrer une édition arrivée après *plusieurs* éditions concurrentes, une question nouvelle se pose : dans quel ordre les lui faire traverser ? Si la réponse change le résultat, les sites divergent.

4.1 L'énoncé

Soit trois opérations o_1 , o_2 et o_3 définies sur le même texte, deux à deux concurrentes. Un site qui a déjà appliqué o_1 puis o_2 adaptée à o_1 doit transformer o_3 contre o_1 , puis contre o_2 adaptée. Un autre site, qui a appliqué o_2 puis o_1 adaptée à o_2 , transforme o_3 contre o_2 , puis contre o_1 adaptée. Par TP1, les deux sites affichent le même texte avant d'intégrer o_3 . La *deuxième propriété de transformation*, TP2, demande que les deux versions de o_3 soient la même opération. En clair : adapter une opération à une séquence ne doit pas dépendre de l'ordre dans lequel cette séquence a été transformée.

4.2 Un contre-exemple à la main

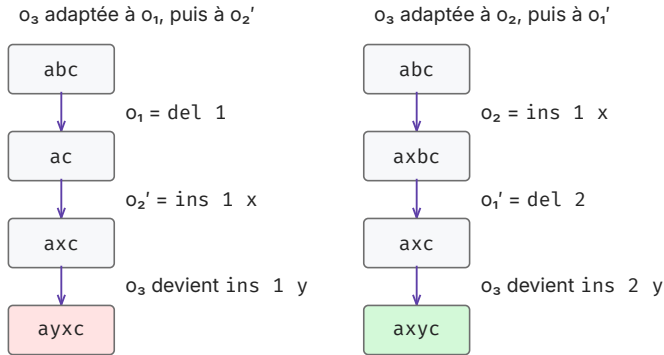
Nos règles de la section 2 vérifient TP1. Vérifient-elles TP2 ? Reprenons la session où le module `Position` est défini et posons trois opérations sur `abc` : Alice supprime `b`, Carol insère `x` devant `b`, Bob insère `y` derrière `b`.

```

1 o1 = {:del, 1}
2 o2 = {:ins, 1, "x", :carol}
3 o3 = {:ins, 2, "y", :bob}
4 o2_apres_o1 = Position.transform(o2, o1)
5 o1_apres_o2 = Position.transform(o1, o2)
6 par_o1 = Position.transform(Position.transform(o3, o1), o2_apres_o1)
7 par_o2 = Position.transform(Position.transform(o3, o2), o1_apres_o2)
8 texte = "abc" |> Position.apply(o1) |> Position.apply(o2_apres_o1)
9 IO.inspect({texte, par_o1, par_o2})
10 IO.inspect(Position.apply(texte, par_o1))
11 IO.inspect(Position.apply(texte, par_o2))

```

CONSOLE {"axc", {:ins, 1, "y", :bob}, {:ins, 2, "y", :bob}} puis "ayxc" puis "axyc"



Les deux ordres de transformation de o_3 . À gauche, la suppression de b est traitée d'abord : x et y se retrouvent tous deux en position 1, et le départage par site place y devant. À droite, y est d'abord décalé derrière x , puis la suppression le ramène en 2.

Voyez-vous d'où vient la divergence ? Sur le texte de départ, x est devant b et y derrière : leur ordre relatif est fixé par le texte. Quand la suppression de b est traitée en premier, les deux positions se confondent, et notre règle de départage, qui ne regarde que les noms de sites, oublie cet ordre. Le site :bob passe avant :carol, et y saute devant x . Dans l'autre ordre, y a déjà été décalé derrière x avant que la suppression n'intervienne. Deux sites, deux textes, aucune règle violée localement.

4.3 Les contre-exemples, puis la parade

Ce scénario n'est pas une maladresse de notre module. En 2003, Abdessamad Imine, Pascal Molli, Gérard Oster et Michaël Rusinowitch codent les fonctions de transformation publiées par Ellis et Gibbs, par Ressel et ses coauteurs, par Sun et ses coauteurs, et les soumettent à un prouveur automatique, SPIKE. Pour chacune, l'outil trouve des contre-exemples à TP2, souvent de la forme que vous venez de voir : une suppression qui fait se rencontrer deux insertions. La conclusion de l'article est que l'intuition ne suffit pas, et que la vérification mécanique est nécessaire pour des fonctions que des dizaines de lecteurs avaient relues.

La réponse vient des mêmes auteurs en 2006. Dans « Tombstone Transformation Functions », Gérard Oster, Pascal Molli, Pascal Urso et Abdessamad Imine changent le modèle plutôt que les règles : une lettre supprimée reste dans le document interne comme *pierre tombale*, invisible mais comptée dans les positions. Une suppression ne fait plus se rencontrer deux insertions, puisqu'aucune position ne disparaît, et TP2 devient vraie. Vous reconnaissez l'idée centrale du projet CRDT : les deux familles, nées en opposition, se retrouvent sur la même parade.

UN TEST DU LOSANGE NE TESTE PAS TP2

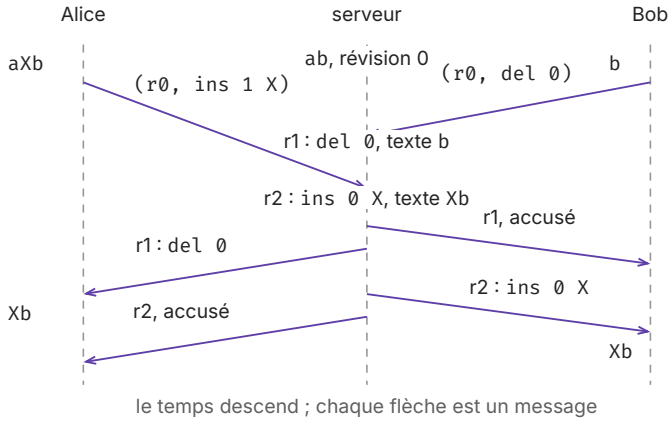
Le test fourni par le projet énumère des paires d'opérations sur une même source et vérifie leur losange : c'est TP1. TP2 porte sur des triplets, et aucune paire ne peut la détecter. Le protocole du projet n'en a pas besoin, pour la raison expliquée dans la section suivante ; mais si vous écriviez un éditeur entre paires, sans serveur, cette propriété redeviendrait obligatoire, et elle est difficile.

5 Un serveur qui compte les révisions

Jupiter, en 1995, a montré comment échapper à TP2 : en interdisant la situation qui la rend nécessaire. Si toutes les éditions passent par un serveur qui les range dans un ordre unique, personne n'a jamais à transformer une opération contre une séquence « dans l'autre ordre ».

5.1 L'idée de Jupiter

Chaque client ne parle qu'au serveur, et le serveur ne parle qu'à des clients un par un. Toute paire client-serveur partage un petit espace à deux dimensions : combien d'opérations du client le serveur a acceptées, combien d'opérations du serveur le client a intégrées. Dans cet espace, intégrer une opération distante revient toujours à transformer *une* opération locale contre *une* opération distante, puis à recommencer : seule TP1 intervient. Google Wave, puis le projet, reprennent ce schéma en numérotant les opérations acceptées : la *révision* r est l'opération numéro r dans l'ordre du serveur, et un client envoie son édition accompagnée de la révision sur laquelle il l'a faite.



Alice et Bob éditent `ab`, révision 0, et envoient chacun leur opération faite sur cette révision. Le serveur accepte Bob en premier (r1), puis transforme l'insertion d'Alice contre r1 avant de l'accepter (r2). Chaque client intègre les révisions dans l'ordre ; sa propre révision lui sert d'accusé.

Suivons la frise. Le serveur reçoit `del(0)` faite sur la révision 0 ; rien n'a été accepté depuis, il l'applique telle quelle et la numérote r1. Il reçoit ensuite `ins(1, X)`, faite elle aussi sur la révision 0 ; une opération a été acceptée depuis, r1, contre laquelle il transforme l'insertion, qui devient `ins(0, X)`, numérotée r2. Côté clients, Bob reçoit r1 : c'est sa propre opération, l'accusé de réception ; puis r2, qu'il applique. Alice reçoit r1 alors que son insertion est encore en vol : elle transforme `del(0)` contre son opération en attente, l'applique, et voit `xb` ; puis r2, son accusé. Le serveur ne stocke pas « la vérité » sur le texte, il fabrique un ordre, et cet ordre suffit.

Voici le cœur du serveur, écrit avec notre module `Position` : un historique des opérations acceptées, et pour chaque message reçu, une transformation contre tout ce qui a été accepté depuis la révision annoncée.

```

1 defmodule Serveur do
2   # L'état : le texte et l'historique des opérations acceptées,
3   # dans l'ordre des révisions. Un client envoie son opération
4   # avec la révision sur laquelle il l'a faite.
5   def accepte({texte, historique}, {revision, op}) do
6     depuis = Enum.drop(historique, revision)
7     op =
8       Enum.reduce(depuis, op, fn acceptee, courante ->
9         Position.transform(courante, acceptee)
10       end)
11     {Position.apply(texte, op), historique ++ [op]}
12   end
13 end
14
15 etat = {"ab", []}
16 etat = Serveur.accepte(etat, {0, {:del, 0}})
17 etat = Serveur.accepte(etat, {0, {:ins, 1, "X", :alice}})
18 IO.inspect(etat)

```

```
CONSOLE {"Xb", [{:del, 0}, {:ins, 0, "X", :alice}]}
```

L'historique conserve les opérations *déjà transformées*, dans l'ordre des révisions : c'est ce que les clients reçoivent, et c'est contre ces versions qu'une édition tardive est adaptée. Notez le sens des arguments de `transform` : l'opération acceptée est toujours celle qui fixe le départage, exactement comme dans le projet où l'opération déjà ordonnée est placée à gauche.

5.2 Une opération en vol, un buffer qui compose

Un client qui attend un accusé peut continuer à taper. Le livre blanc de Wave et le protocole du projet font le même choix : une seule opération en vol, et toutes les éditions suivantes *composées* dans un buffer, c'est-à-dire fondues en une seule opération équivalente, envoyée quand l'accusé arrive. Composer et transformer sont deux outils différents. Composer concerne deux éditions *successives* du même auteur, et produit l'édition qui fait les deux ; transformer concerne deux éditions *concurrentes*, et produit l'adaptation de l'une à l'autre. Une opération distante qui arrive avant l'accusé est transformée contre l'opération en vol, puis contre le buffer, avant d'atteindre le texte visible ; symétriquement, l'opération en vol et le buffer sont adaptés pour que le serveur puisse les accepter plus tard.

À RETENIR

Avec un serveur central, chaque transformation oppose une opération à une opération, dans un ordre fixé par les révisions : TP1 suffit. Le serveur ne juge pas le contenu, il numérote. Le prix est une dépendance : sans serveur joignable, un client peut taper, mais rien ne circule.

EN PRATIQUE

Dans vos scénarios du palier 4, notez pour chaque message sa révision de départ et le nombre d'opérations acceptées depuis. Ce nombre est exactement le nombre de transformations que le serveur effectuera ; le prédire avant d'envoyer vous dit où chercher si un texte final vous surprend.

6 OT contre CRDT : le débat

Les deux projets d'éditeur de ce cours correspondent à deux écoles qui discutent depuis vingt ans. Nous rapportons leurs arguments tels qu'ils sont écrits, sans arbitrer.

Du côté de l'OT, Chengzheng Sun, David Sun, Agustina Ng et Weiwei Cai publient en 2018 et 2019 deux longs articles intitulés « Real Differences between OT and CRDT ». Leur constat de départ est factuel : trente ans après son invention, l'OT équipe « la vaste majorité » des éditeurs collaboratifs en service, Google Docs en tête, tandis que les CRDT, nés vers 2006 avec WOOT et présentés comme une technique « post-OT », y sont « rarement trouvés ». Leur analyse soutient que les CRDT de texte paient leurs promesses par des coûts cachés, identifiants par caractère et pierres tombales conservées, et qu'ils effectuent en réalité une forme de transformation, déplacée dans la structure de données.

Du côté des CRDT, Martin Kleppmann avance depuis 2016 un argument d'architecture : les algorithmes OT en production reposent sur un serveur central, et les variantes entre pairs ont une histoire de fonctions publiées puis réfutées, celle que raconte la section 4 ; un CRDT fonctionne sans serveur, hors ligne, et se prête au chiffrement de bout en bout. Son exposé « CRDTs: The Hard Parts » (2020) admet en retour que les CRDT ont leurs propres difficultés, l'entrelacement et la taille des métadonnées en tête. Témoignage éloquent, Joseph Gentle, auteur de ShareJS et

ancien de Wave, publie en septembre 2020 un billet intitulé « I was wrong. CRDTs are the future », après avoir vu cet exposé.

Les deux camps convergent pourtant sur un point. Le résumé de l'article sur Fugue (2023) note que l'anomalie d'entrelacement « touche à la fois les CRDT et la transformation opérationnelle » ; les pierres tombales de TTF (2006) avaient déjà importé l'idée CRDT dans l'OT ; et Sun et ses coauteurs reconnaissent que les CRDT transforment, à leur manière. La frontière passe moins entre deux mathématiques qu'entre deux hypothèses sur le réseau : un serveur qui ordonne, ou pas.

	OT, à serveur central	CRDT à états
ce qui circule	des opérations et des révisions	des états complets
ce qu'il faut prouver	TP1 pour chaque paire d'opérations	les trois lois d'un demi-treillis
sans serveur	TP2 devient nécessaire	rien ne change
doublons et désordre	refusés par le protocole	absorbés par la fusion
mémoire	le texte et un historique à compacter	le texte et ses pierres tombales

7 Au-delà du projet

Google Docs reste le système OT le plus utilisé au monde, et le billet de 2010 en décrit les grandes lignes : un serveur qui ordonne, des clients qui transforment, et des opérations qui portent aussi la mise en forme. ShareDB, l'héritière de ShareJS, généralise l'approche en séparant le protocole des *types* d'opérations : un type texte, un type JSON, un type texte riche, chacun avec ses fonctions `apply`, `transform` et `compose`, exactement la séparation que le projet vous fait pratiquer entre `Operation`, `TextOT` et `Session`. Etherpad, dont l'équipe a rejoint celle de Wave en 2009, décrit ses éditions par des *changesets* qui parcourent tout le document ; la représentation du projet, une suite de `retain`, `insert` et `delete` qui consomme la source entière, descend de cette famille, reprise par Wave puis par `ot.js`, dont la démonstration de Tim Baumann fait voir chaque étape.

Ce qui change à l'échelle tient en quatre mots. *Historique* : un serveur ne peut pas garder toutes les révisions depuis la création d'un document de dix ans, il prend des instantanés et n'accepte plus les éditions faites sur une révision trop ancienne.

Curseurs : la position du curseur de chaque personne est une position comme une autre, qu'il faut transformer contre chaque opération reçue pour qu'elle reste au bon endroit. *Annulation* : annuler sa propre édition quand d'autres ont édité entre-temps demande de transformer l'opération inverse, et c'est un problème à part entière. *Texte riche* : gras, titres et listes deviennent des attributs attachés aux caractères, et chaque attribut a ses propres cas de transformation. Le projet vous laisse volontairement ces quatre chantiers.

8 Pour aller plus loin

Concurrency control in groupware systems, Clarence A. Ellis et Simon J. Gibbs (1989).

<https://doi.org/10.1145/67544.66963>

L'article d'origine, pour voir dOPT tel qu'il a été écrit, avec ses vecteurs d'état et ses priorités. Accès par la bibliothèque numérique de l'ACM.

High-latency, low-bandwidth windowing in the Jupiter collaboration system, David A. Nichols, Pavel Curtis, Michael Dixon et John Lamping (1995).

<https://doi.org/10.1145/215585.215706>

Le serveur central et l'espace à deux dimensions. La section sur l'algorithme est courte et se lit avec la frise de ce fascicule sous les yeux.

Google Wave Operational Transformation, David Wang, Alex Mah et Soren Lassen (2010).

<https://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html>

Le livre blanc, archivé chez Apache : une opération en vol, un buffer composé, un serveur qui numérote. C'est le protocole du projet, écrit par ceux qui l'ont déployé.

Proving Correctness of Transformation Functions in Real-Time Groupware, Abdessamad Imine, Pascal Molli, Gérald Oster et Michaël Rusinowitch (2003).

<https://inria.hal.science/inria-00107652>

Les contre-exemples à TP2 trouvés par un prouveur. À lire pour mesurer ce qu'une relecture humaine laisse passer.

Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems, Gérald Oster, Pascal Molli, Pascal Urso et Abdessamad Imine (2006).

<https://inria.hal.science/inria-00109039>

La réponse : garder les lettres supprimées. Le pont entre les deux projets d'éditeur du cours.

Real Differences between OT and CRDT for Co-Editors, Chengzheng Sun, David Sun, Agustina Ng et Weiwei Cai (2018).

<https://arxiv.org/abs/1810.02137>

Le plaidoyer du camp OT, long et argumenté. Lisez-le avec le fascicule du projet CRDT, pour entendre les deux voix.

I was wrong. CRDTs are the future, Joseph Gentle (2020).

<https://josephg.com/blog/crdts-are-the-future/>

Le billet d'un praticien de l'OT qui change d'avis, avec ses raisons et ses réserves. Court et personnel.

Visualization of OT with a central server, Tim Baumann.

<https://operational-transformation.github.io/>

La démonstration interactive citée par le README : vous y rejouez le losange et le protocole à révisions, opération par opération, après avoir écrit les vôtres.

9 Une question pour la route

Alice tape un mot, Bob supprime une lettre au milieu de ce mot, puis Alice appuie sur « annuler ». Que doit faire l'éditeur ? Annuler, c'est appliquer l'opération inverse de la sienne ; mais le texte a changé depuis, et cette inverse doit être transformée contre l'édition de Bob. Que devient-elle si la lettre que Bob a supprimée faisait partie du mot à retirer ? Et si Bob appuie lui aussi sur « annuler » juste après ? Essayez d'écrire la règle, sur la représentation à positions de ce fascicule, pour une insertion et une suppression ; puis cherchez le cas où votre règle hésite. L'annulation collaborative n'a pas de solution unanimement acceptée, et c'est une bonne nouvelle pour qui aime les problèmes ouverts.

Pour toute question sur ce fascicule ou sur le projet, vous pouvez me contacter par email, comme pour les autres séances.