"The best thing for being sad," replied Merlin, beginning to puff and blow, "is to learn something. That's the only thing that never fails. You may grow old and trembling in your anatomies, you may lie awake at night listening to the disorder of your veins, you may miss your only love, you may see the world about you devastated by evil lunatics, or know your honour trampled in the sewers of baser minds. There is only one thing for it then — to learn. Learn why the world wags and what wags it. That is the only thing which the mind can never exhaust, never alienate, never be tortured by, never fear or distrust, and never dream of regretting. Learning is the only thing for you. Look what a lot of things there are to learn - pure science, the only purity there is. You can learn astronomy in a lifetime, natural history in three, literature in six. And then, after you have exhausted a milliard lifetimes in biology and medicine and theo-criticism and geography and history and economics - why, you can start to make a cartwheel out of the appropriate wood, or spend fifty years learning to begin to learn to beat your adversary at fencing. After than you can start again on mathematics, until it is time to learn to plough." T.H. White - The Once and Future King

Concur: The First Correct Concurrency Control Algorithm

© James Smith

Imperial College London

2nd June 2016

Motivation

Problem

Actually there seem to be three problems that need solving:

- Is this algorithm correct?
- Is this algorithm novel?
- Assuming the first two can be solved, then what?

I can solve the first problem today hopefully and give some insights into the second. I'm open to suggestions on the third...

What are concurrency control algorithms?

There may be other definitions, but here I'm talking about the kinds of algorithms that are used by collaborative text editors.

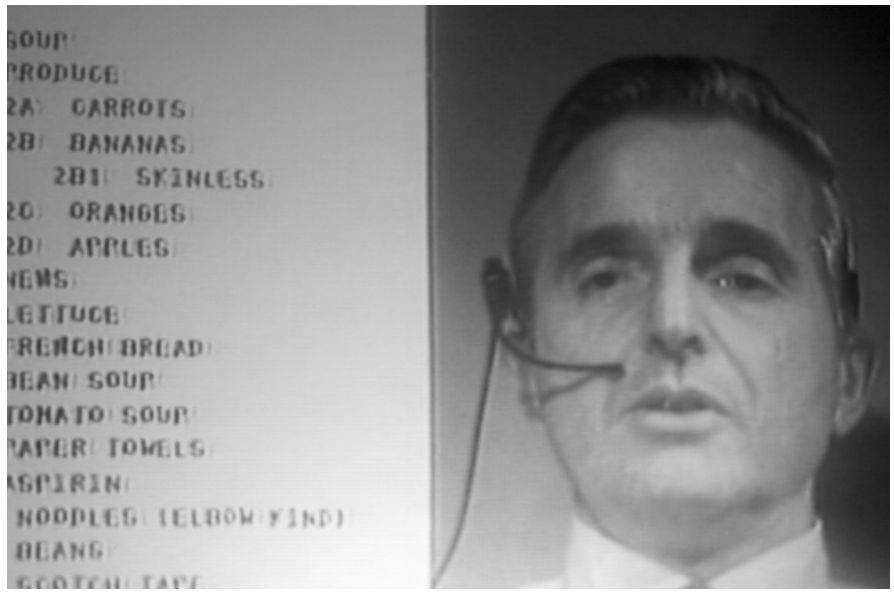
Example

Example

What are concurrency control algorithms?

They have a long history starting around 1968...

"The Mother of All Demos"
Douglas Engelbart 1925 - 2013



[youtube] Part 8 - Real time collaboration

What are concurrency control algorithms?

Contributions from academia started around 1989...

- Ellis and Gibbs.[1]

Algorithm	Transformation properties	System
dOPT	TP1	GROVE
selective-undo	TP1, TP2, RP, IP1, IP2, IP3	DistEdit
adOPTed	TP1, TP2, IP1	JOINT EMACS
Jupiter	TP1	
GOT	None	REDUDE
GOTO	TP1, TP2	REDUCE, CoWord, CoPPT, CoMaya
AnyUndo	IP1, TP1, TP2	REDUCE, CoWord, CoPPT, CoMaya
SCOP	TP1	NICE
COT	TP1 (No IP1 necessary)	REDUCE, CoWord, CoPPT, CoMaya
TIBOT	TP1	
SOCT4	TP1	
SOCT2	TP1, TP2, RP	
MOT2	TP1, TP2	

[wikipedia] Wikipedia - Operational transformation

[1] Ellis and Gibbs. Concurrency Control in Groupware Systems. SIGMOD, 18(2):399–407, 1989.

What are concurrency control algorithms?

There have been a myriad implementations in industry...

Atmail, Axigen, Blogtronix, BSCW, Calliflower, Collaba, Collaber, Collanos Workplace, EditGrid, FirstClass, GForge, Google Docs, Google Wave, Group-Office, i-sense, Intraboom, Kolab, Mikogo, Mixedink, PabloDraw, PBworks, Powermeeting, QikPad, Samepage, Saros, SubEthaEdit, ShareJS, ShareDB, tmsEKP, Wiggio, Writeboard, Xaitporter, etc.

[wikipedia] Wikipedia - List of collaborative software

So what's the real problem?

They just don't work very well...

"Floobits works pretty well most of the time."

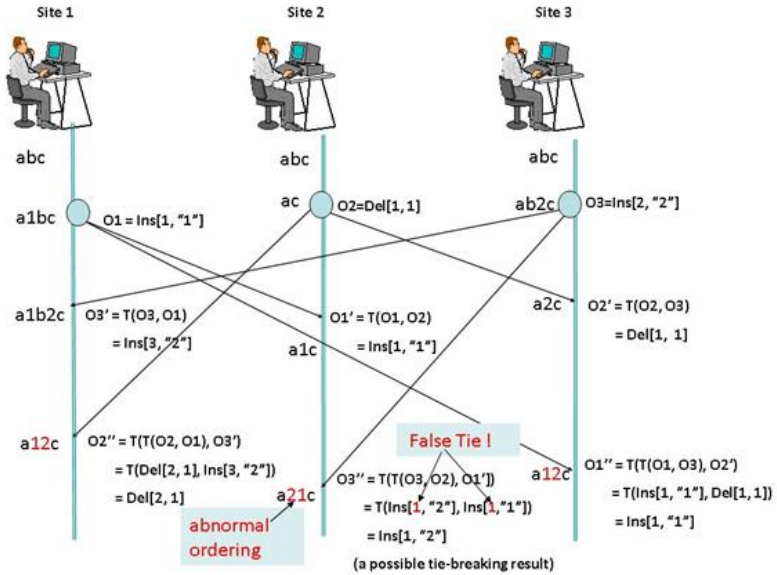
Matt Kaniaris - Co-founder, Floobits

"Unfortunately, implementing OT sucks. There's a million (sic) algorithms with different tradeoffs, mostly trapped in academic papers. The algorithms are really hard and time consuming to implement correctly...Wave took 2 years to write and if we rewrote it today, it would take almost as long to write a second time."

Joseph Gentle - ex Google Wave engineer

So what's the real problem?

Academics don't think they work very well either...



"Due to the need to consider complicated case coverage, formal proofs are very complicated and error-prone, even for OT algorithms that only treat two character-wise primitives (insert and delete)"[2]

[2] 2010

Back to a more precise definition

A concurrency control algorithm has three distinct parts:

- A set of operational transformations for operations on the chosen data type
- A function that composes these transformations so that one sequence of operations can be transformed relative to another
- A system that utilises this function to enable two or more copies of the same resource to be kept in line over a network

I think that any correct solution *must* solve the problems inherent in each of these parts separately and sequentially. Unfortunately, an almost universal pitfall seems to be muddling them up.

The main contributions

These roughly correspond to each of the aforementioned parts:

1. A set of comprehensive (you could almost say canonical), string-wise operational transformations
2. A recursive function that composes these transformations and is guaranteed to terminate
3. A distributed algorithm including a protocol that makes use of this recursive function
4. The correct treatment of latency

The last of these I initially overlooked but it is crucial if the system is to work well in practice.

Some bonus contributions

I think I have created a system that works well in practice:

- Beyond the operational transformations, which are tied to the data type obviously, the algorithm is agnostic to the data type
- The recursive function that composes operational transformations is guaranteed to terminate for (discrete) data type and any set of operational transformations that obey a single criterion
- At least in the case when the data type is plain text, the recursive function appears to terminate quickly and is therefore extremely fast and could doubtless be made faster
- *The system is completely and demonstrably fault tolerant*

Some bonus contributions

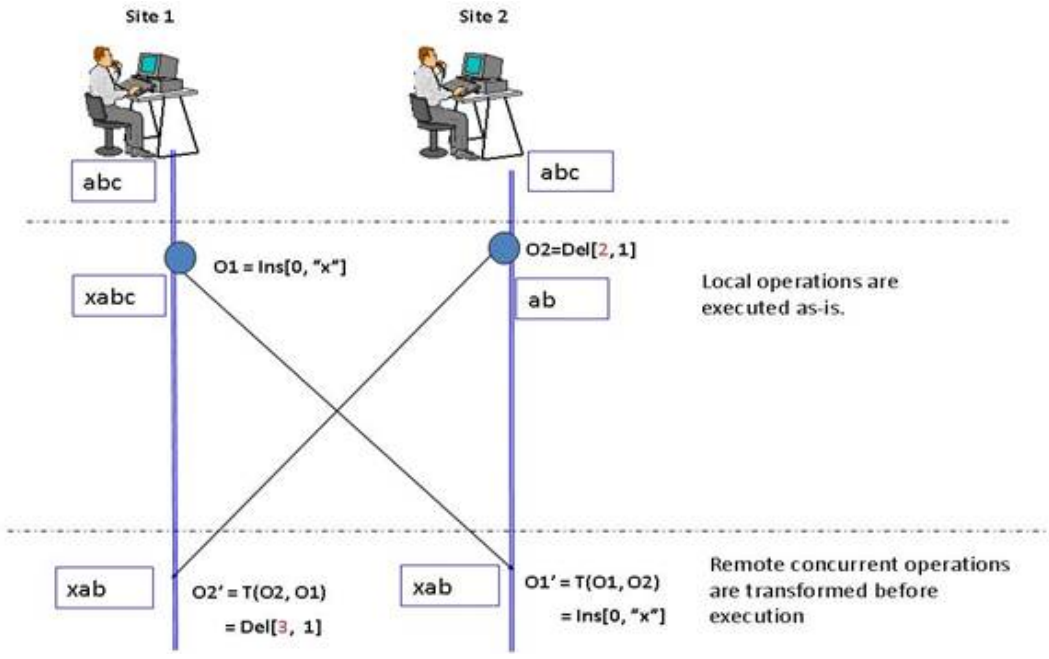
I have a proof assistant called Florence as a test-bed for these:

- The plain text data type can be augmented to include user selections or to track the active document, etc
- The whole system can be extended to a rich, multi-document one with session handling, etc

This turned out to be a lot of hard work!
And finally:

- Because of the absence of any need for locking, databases and such like, it's possible to envisage scenarios where these resource intensive approaches can be done away with
- Because the recursive function is fast and in fact parallelisable, and because the system is simple and fault tolerant, it could turn out to be massively scalable

1. A set of comprehensive, string-wise operational transformations

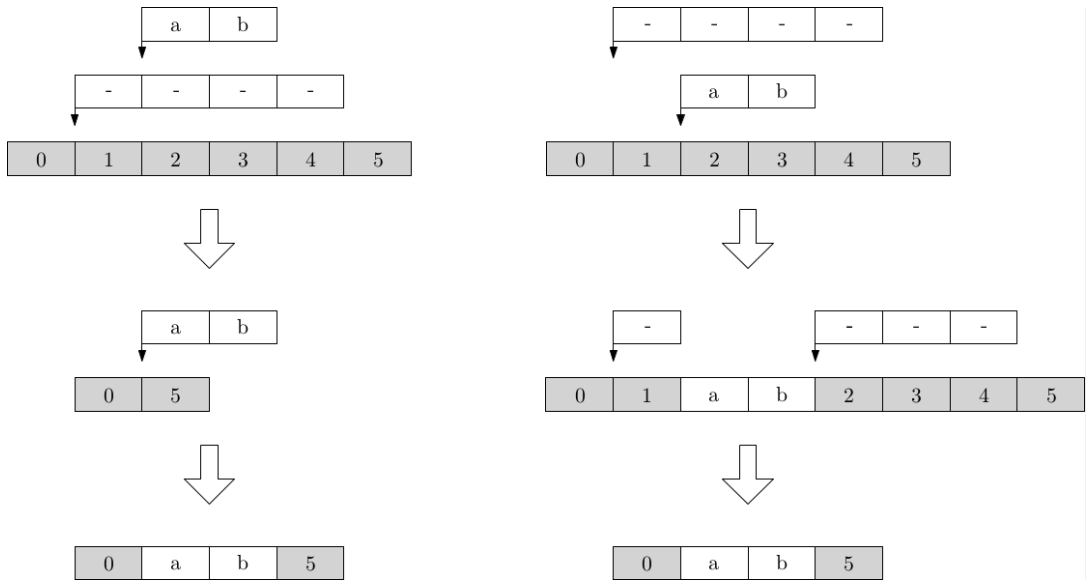


$$i(0,x) \backslash d(2,1) = i(0,x) \quad d(2,1) \backslash i(0,x) = d(3,1)$$

Roughly speaking if the operation to be transformed is to the *right* of the transforming operation, it is shifted accordingly. If it is to the *left*, it is left unchanged.

1. A set of comprehensive, string-wise operational transformations

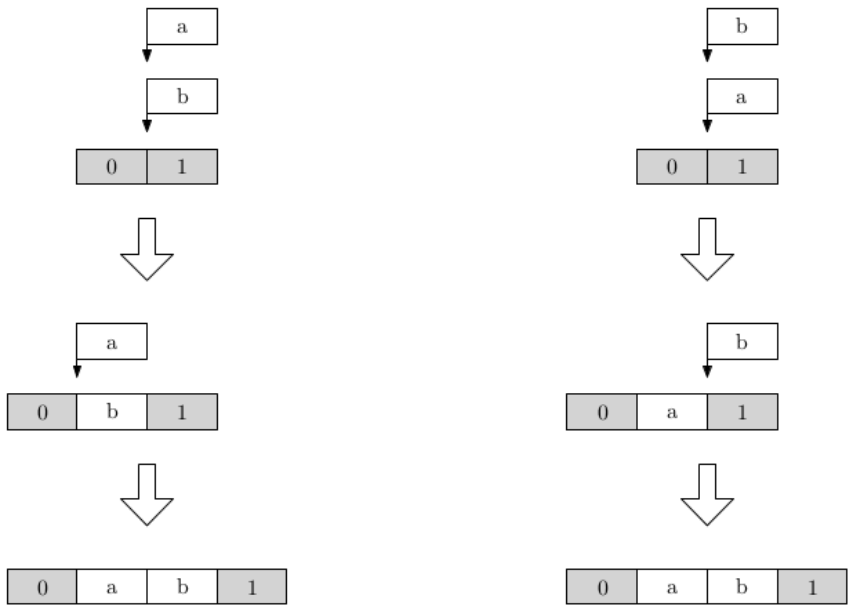
Things get a bit harder when two concurrent operations overlap...



The insert gets shifted to the left edge of the delete. The delete gets split in two and its rightmost part gets shifted to the right.

1. A set of comprehensive, string-wise operational transformations

The only subtle case is when both of the operations are insert operations that happen to insert their strings at the same place.



This is known as a tie. In this case I leave the lexicographically lesser of the two insert operations in place.

1. A set of comprehensive, string-wise operational transformations

The aim with these transformations is that the combined effect of the operations is the same regardless of the order in which they are executed, provided the second is suitably transformed relative to the first. Formally:

$$\tau; \rho \backslash \tau \equiv \rho; \tau \backslash \rho$$

This is known as transformation property 1, or TP1.

Theorem

Our string-wise operational transformations satisfy TP1. □

To get this we go through around a dozen cases and be careful in the implementation.

A digression: Preservation of intention

Suppose we define delete operations to specify the contents of the string that they are deleting rather than just their length. Then we have enough information to define inverses for both inserts and deletes. And now suppose we do the following:

$$\tau \backslash \rho = \rho^{-1} \quad \rho \backslash \tau = \tau^{-1}$$

Then:

$$\tau ; \rho \backslash \tau = \tau ; \tau^{-1} = \epsilon = \rho ; \rho^{-1} = \rho ; \tau \backslash \rho$$

Here ϵ is the empty operation and TP1 is somewhat pointlessly satisfied.

A digression: Preservation of intention

Why bother?!

To make the point that operational transformations can satisfy TP1 withouth preserving intention, although we would like it if they did.

Let's start by defining the concept of intention:

$$\begin{aligned} [\![i(n,s)]\!] &= (n,s) \\ [\![d(n,l)]\!] &= \{n, \ldots n+l-1\} \end{aligned}$$

The intention of an insert is its position together with the string of characters it inserts. The intention of a delete is just the set of indices of the characters it deletes.

A digression: Preservation of intention

We now employ a little sleight of hand...

Looking back at the figures we see that characters keep their initial indices after operations have been applied. With the initial indices being kept in this manner, the preservation of intention is easy to formalise.

If τ is either an insert or a delete and ρ is an insert we have:

$$[\![\tau \backslash \rho]\!] = [\![\tau]\!]$$

In all cases when both τ and ρ are deletes we have:

$$[\![\tau \backslash \rho]\!] = [\![\tau]\!] \backslash [\![\rho]\!]$$

A digression: Preservation of intention

When the transforming operation is a delete and the transformed operation an insert some of the cases can be a little more involved. If the left hand corner of the insert overlaps the delete, the insert is effectively moved immediately to the delete operation's right:

$$[\![\tau \backslash \rho]\!] = \begin{cases} [\![\tau \uparrow \rho^+]\!] & \tau \simeq \rho \vee \tau > \rho \\ [\![\tau]\!] & \textit{otherwise} \end{cases}$$

Here $\tau \simeq \rho \vee \tau > \rho$ is formalism lifted from the paper and just means τ and ρ overlap in the required way.

Anyway, we can go away and check all of these cases.

Theorem

Our string-wise operational transformations preserve intention. □

1. A set of comprehensive, string-wise operational transformations

Unfortunately, even a set of correct character-wise operational transformations seemed impossible to come by for some time.[3]

- Ellis and Gibbs 1989
- Ressel et al. 1996
- Suleiman, Cart & Ferrie 1997
- Sun et al. 1998

The SPIKE theorem prover found mistakes in each set of operational transformations. These authors also proposed the first formally correct set of character-wise operational transformations.

If you abridge my own string-wise operational transformations they are in agreement, including the treatment of tied inserts.

---

[3] Abdessamad Imine, Pascal Moland, Gerald Oster, and Michael Rusinowitch. Proving Correctness of Transformation Functions in Real-time Groupware. In European Conference on Computer Supported Cooperative Work, pages 277–293. Springer, 2003.

So why were a set of correct operational transformations so difficult to come by?

Remember I stated that any correct solution must solve the problems inherent in each of the parts separately and sequentially.

This hasn't been done. Furthermore, the parts themselves are often more complicated. Nearly all algorithms in the literature are based on a peer-to-peer network, for example. This means the use of vector clocks and the like, trying to enforce precedence of operations or making do without it, managing buffers, etc.

Without any clarity, unreasonable correctness criteria were put on operational transformations when all you really need is TP1 and the preservation of intention.

The main contributions

1. A set of comprehensive, string-wise operational transformations ✓
2. A recursive function that composes transformations and is guaranteed to terminate
3. A distributed algorithm including a protocol that makes use of the recursive function
4. The correct treatment of latency

2. A recursive function that composes transformations and is guaranteed to terminate

Operations come in sequences not just singly. So sequences of operations have to be transformed relative to one another.

Consider the case of a single operation being transformed relative to a sequence of two operations. Intuitively:

$$\rho\backslash(\tau_1;\tau_2) = (\rho\backslash\tau_1)\backslash\tau_2 = \rho\backslash\tau_1\backslash\tau_2$$

However what if $(\rho\backslash\tau_1) = \rho_1;\rho_2$? Regardless we need a formula for a sequence of two operations being transformed relative to a single operation. Somewhat less intuitively:

$$(\tau_1;\tau_2)\backslash\rho = \tau_1\backslash\rho;\tau_2\backslash(\rho\backslash\tau_1)$$

2. A recursive function that composes transformations and is guaranteed to terminate

These identities appeared a few years after Ellis and Gibbs.[4]

Since transformed operations are occasionally split in two we get what I call fragmentation:

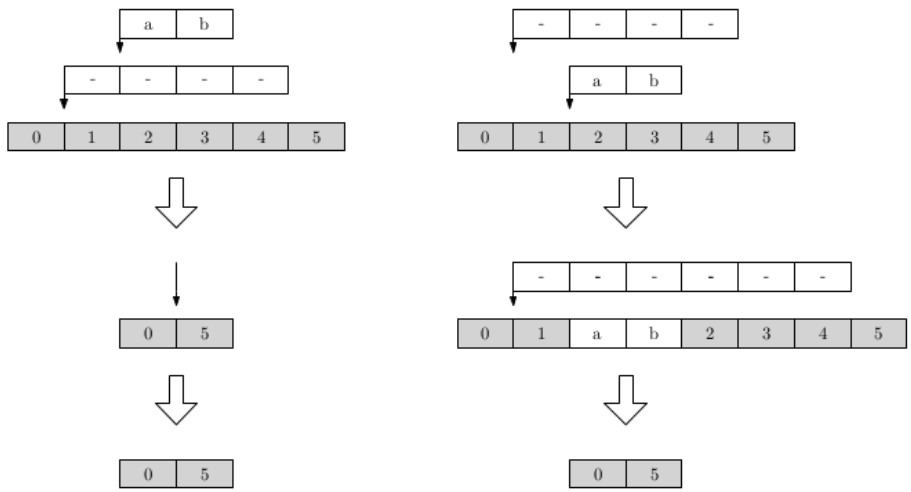
$$\begin{aligned}
\rho\backslash(\tau_1;\tau_2) &= \rho\backslash\tau_1\backslash\tau_2 \\
&= (\rho\backslash\tau_1)\backslash\tau_2 \\
&= (\rho_1;\rho_2)\backslash\tau_2 \\
&= \rho_1\backslash\tau_2;\rho_2\backslash(\tau_2\backslash\rho_1) \\
&= \rho_1\backslash\tau_2;\rho_2\backslash(\tau_3;\tau_4)
\end{aligned}$$

It's by no means certain that this process is going to terminate.

[4] Gordon Cormack. A Calculus for Concurrent Update. In Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pages 269–279. ACM, 1995.

2. A recursive function that composes transformations and is guaranteed to terminate

This has been known since Cormack. He got around it by skimping on the operational transformation that would otherwise cause an insert to split a delete in two:



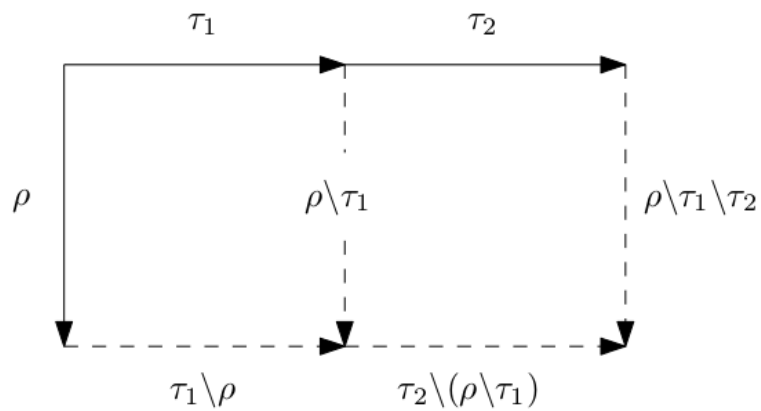
Note that TP1 is still satisfied, however these operational transformations no longer preserve the intention of the insert, in fact it's effectively just thrown away!

2. A recursive function that composes transformations and is guaranteed to terminate

To continue, are these identities even correct? And can we even couch them as identities?

$$\begin{aligned} \rho\backslash(\tau_1;\tau_2) &= \rho\backslash\tau_1\backslash\tau_2 \\ (\tau_1;\tau_2)\backslash\rho &= \tau_1\backslash\rho;\tau_2\backslash(\rho\backslash\tau_1) \end{aligned}$$

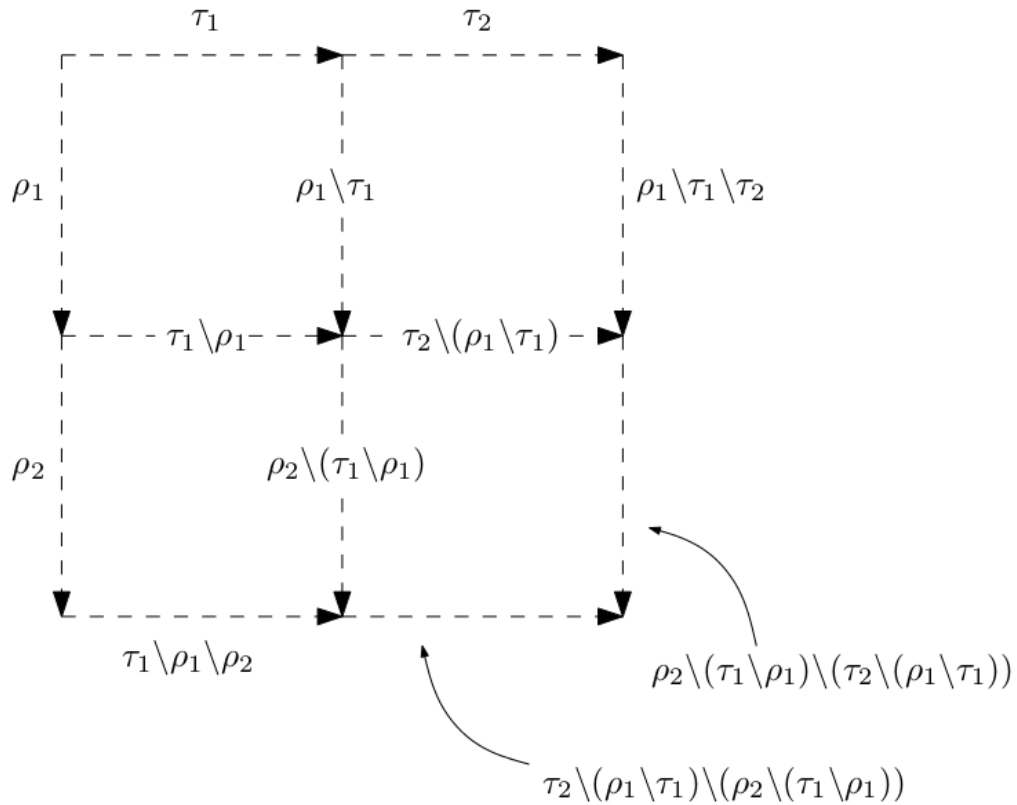
A little abstract rewriting will help!



The dotted lines signify possibly more than one operation.

2. A recursive function that composes transformations and is guaranteed to terminate

What about an identity for $(\tau_1;\tau_2)\backslash(\rho_1;\rho_2)$ in the general case?



$$(\tau_1;\tau_2)\backslash(\rho_1;\rho_2) = \tau_1\backslash\rho_1\backslash\rho_2;\tau_2\backslash(\rho_1\backslash\tau_1)\backslash(\rho_2\backslash(\tau_1\backslash\rho_1))$$

2. A recursive function that composes transformations and is guaranteed to terminate

At this point I claim two things:

- That these identities can be used as reductions in a recursive function that computes the transformation of any sequence of operations relative to any other

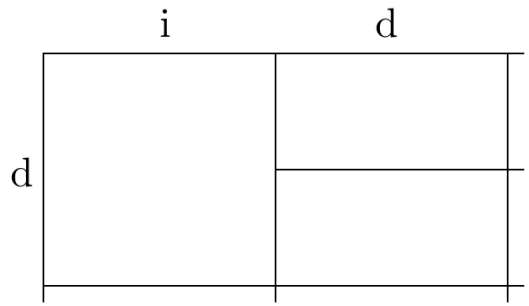
$$\rho \backslash (\tau_1 ; \tau_2) \rightsquigarrow \rho \backslash \tau_1 \backslash \tau_2$$
$$(\tau_1 ; \tau_2) \backslash \rho \rightsquigarrow \tau_1 \backslash \rho ; \tau_2 \backslash (\rho \backslash \tau_1)$$
$$(\tau_1 ; \tau_2) \backslash (\rho_1 ; \rho_2) \rightsquigarrow \tau_1 \backslash \rho_1 \backslash \rho_2 ; \tau_2 \backslash (\rho_1 \backslash \tau_1) \backslash (\rho_2 \backslash (\tau_1 \backslash \rho_1))$$

- That this function always terminates

In order to justify these claims the diagrams must be proved to be commutative, or in other words Church-Rosser.

2. A recursive function that composes transformations and is guaranteed to terminate

In order to do this I streamline the diagrams, replacing each dotted line with a single line if the result of the transformation is a single operation, or a double line if the result of the transformation might be two operations:



The three sub-diagrams that have single operations on the top and left sides are called elementary diagrams. Now you can see that the whole diagram commutes provided that each sub-diagram does.

2. A recursive function that composes transformations and is guaranteed to terminate

This approach was inspired by decreasing diagrams...[5]

...and makes use of Newman's lemma:

Lemma

No infinite sequences and locally Church-Rosser $\Rightarrow$ *Church-Rosser.*

---

[5] Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. A Geometric Proof of Confluence by Decreasing Diagrams. Journal of Logic In Computing, 10(3):437–460, 2000.

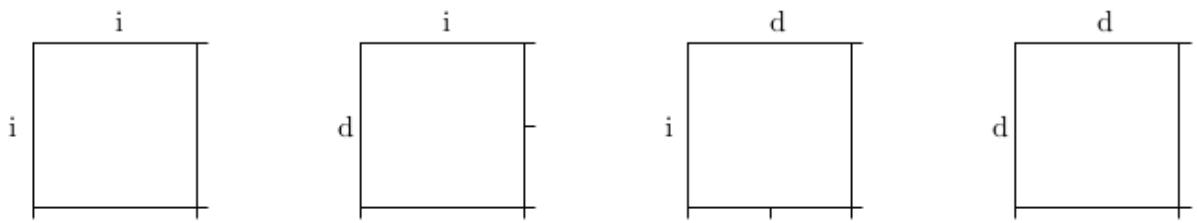
2. A recursive function that composes transformations and is guaranteed to terminate

The theorem that the elementary diagrams are Church-Rosser, in other words that the diagrams are locally Church-Rosser, is simply a restatement of the theorem that the operational transformations obey TP1:

Theorem

The elementary diagrams are Church-Rosser.

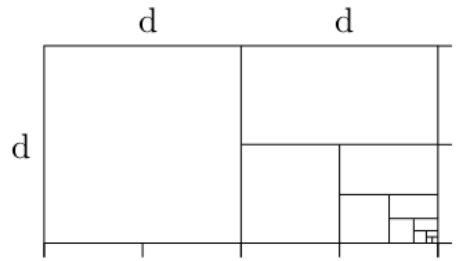
Proof.



□

2. A recursive function that composes transformations and is guaranteed to terminate

Note that deletes do not split deletes. If they did then infinite sequences would occur:



A little experimentation suggests that infinite series can never occur. This intuition can be made precise:

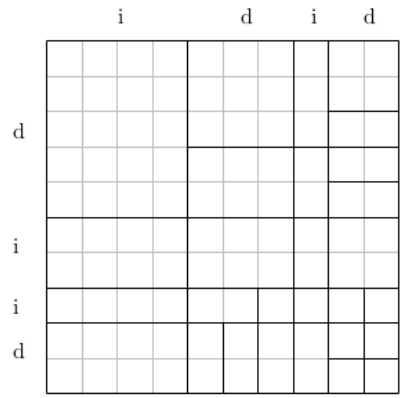
Theorem

All possible diagrams are Church-Rosser. □

So TP1 is satisfied for sequences of operations not just single ones.

2. A recursive function that composes transformations and is guaranteed to terminate

In fact a simple argument suffices in the discrete case.



I call this my "squared paper" argument. It rests on the fact that the size of transformed operations never increases:

$$|\tau| \leqslant |\tau \backslash \rho|$$

In fact this argument holds for any data type provided a metric can be found and the above inequality satisfied.

The main contributions

1. A set of comprehensive, string-wise operational transformations ✓
2. A recursive function that composes transformations and is guaranteed to terminate ✓
3. A distributed algorithm including a protocol that makes use of the recursive function
4. The correct treatment of latency

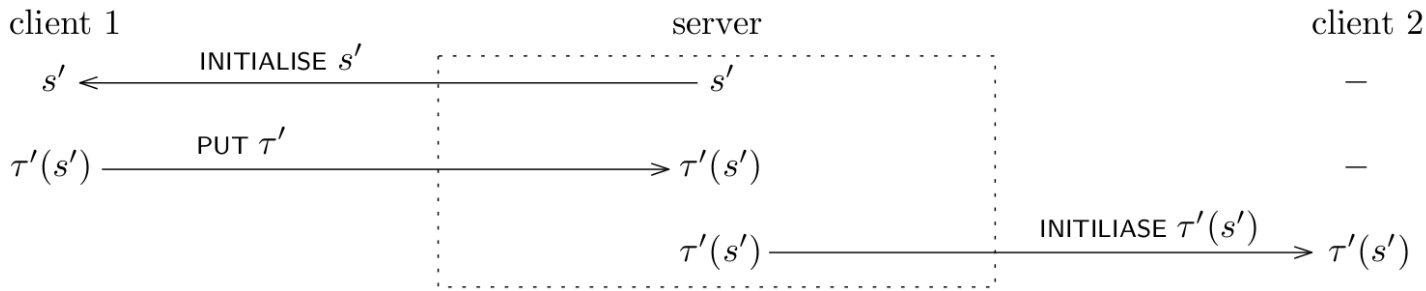
3. A distributed algorithm including a protocol that makes use of the recursive function

This sits on top of the HTTP or HTTPS protocols and is therefore based on the client-server model. The protocol is simple, there are three types of transaction:

1. INITIALISE : the server responds with a copy of its document
2. PUT : the client puts a sequence of operations on the server, the server confirms
3. GET : the client requests its pending operations, the server duly responds

3. A distributed algorithm including a protocol that makes use of the recursive function

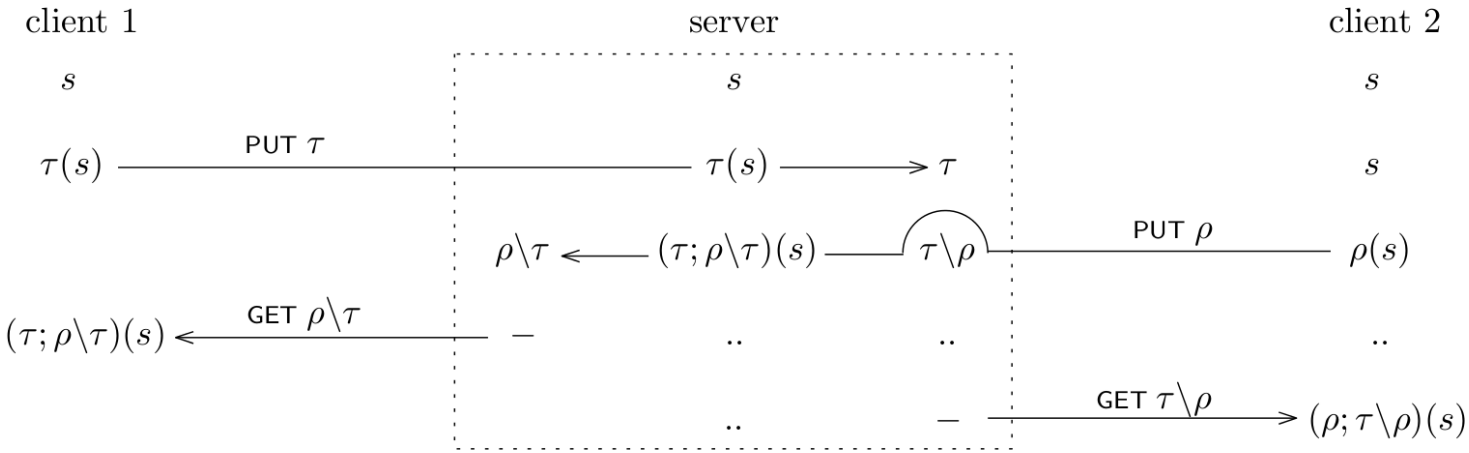
At least initially I'll make the incorrect assumption of "wholly global time" and stick with just two clients in order to get the salient points across...



I also assume sequentiality on both the clients and the server. This is a valid assumption given a careful implementation.

3. A distributed algorithm including a protocol that makes use of the recursive function

Both clients and server end up with copies that are in line by TP1:



This is crucial:

> The second client's operations do not become the first client's pending operations nor are they applied to the server's copy of the document without first being transformed relative to second client's own pending operations

The main contributions

1. A set of comprehensive, string-wise operational transformations ✓
2. A recursive function that composes transformations and is guaranteed to terminate ✓
3. A distributed algorithm including a protocol that makes use of the recursive function ✓
4. The correct treatment of latency

4. The correct treatment of latency

I mentioned earlier that I have created a system that works well in practice. This is because latency is treated correctly. I order to do this I abandon the previous wholly notion of global time and adopt Lamport's "happens before" relation.

I also simplify the protocol, bundling PUT and GET transactions into one UPDATE transaction:

1. INITIALISE : the server responds with a copy of its document
2. UPDATE : the client puts a sequence of operations on the server, the server responds with the client's pending operations, suitably transformed.

4. The correct treatment of latency

Next I introduce the fact that clients keep not one copy of the document but two:

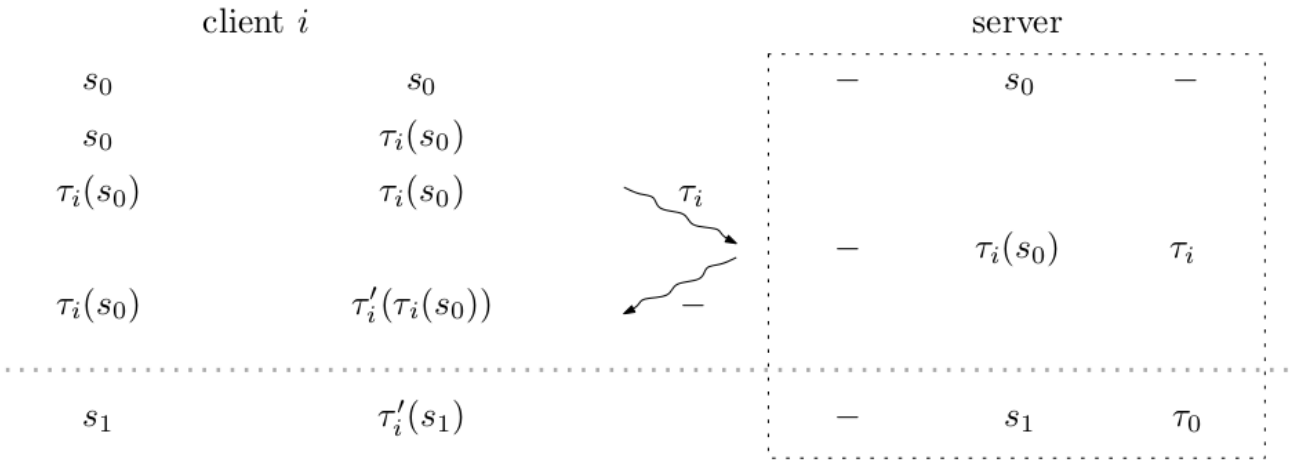
1. A working copy, the one formalised thus far
2. An editable copy considered to be the value of the input field made available to the user

You can never make any claims about editable copies being in line. You simply cannot, given that the user can interact with the system at any time.

From now on I work with an arbitrary, albeit fixed, number of clients, rather than just two. The client involved in a particular transaction at any time is the i'th client, whilst any other client is the j'th client.

4. The correct treatment of latency

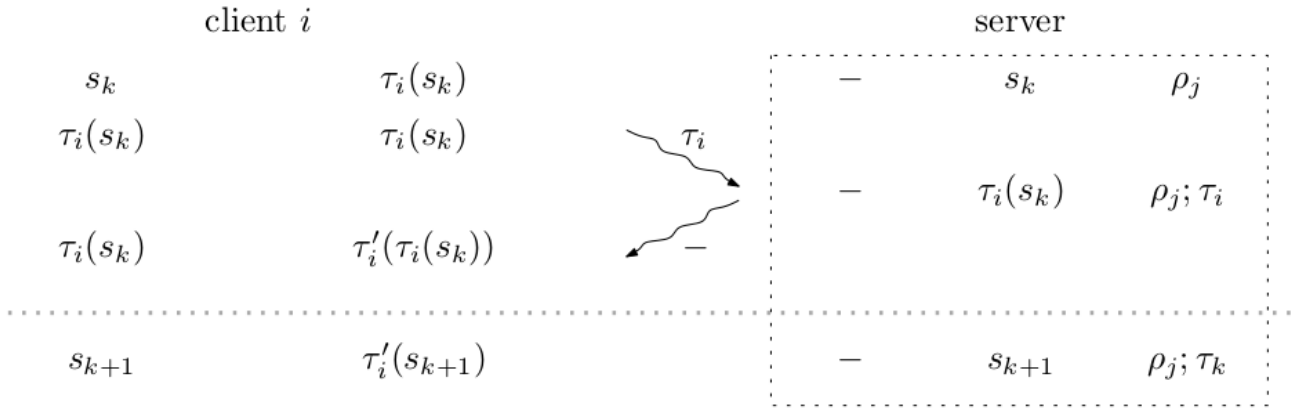
The first UPDATE transaction:



The i'th client's editable copy changes from $\tau_i(s_0)$ to $\tau_i'(\tau_i(s_0))$ whilst the transaction is in progress. Its working copy and the server's copy end up being in line. I rename τ_i to τ_0 and set $s_1 = \tau_0(s_0)$.

4. The correct treatment of latency

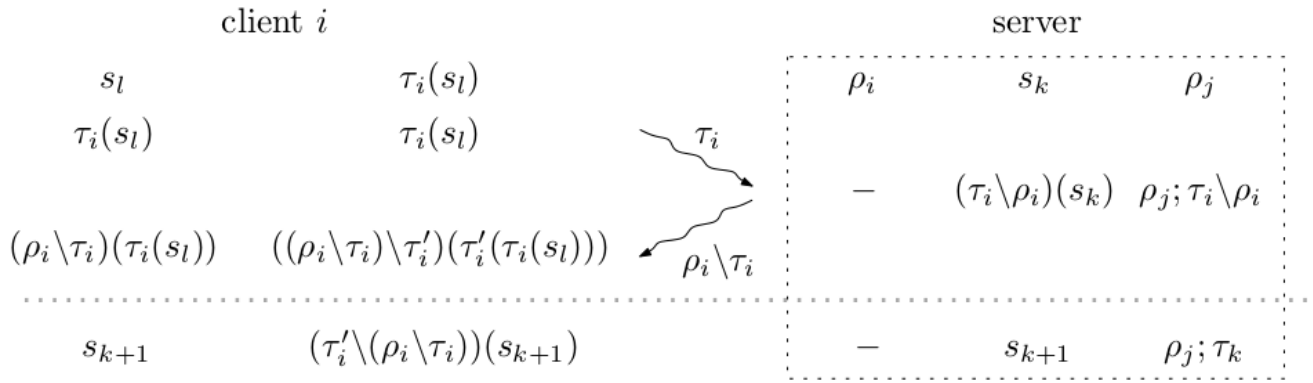
The k'th UPDATE transaction with no pending operations:



The induction hypothesis is that the i'th client's working copy and server's copy are in line. I reuse τ_i, I'll keep on doing so, renaming it to τ_k and appending it to the pending operations of all the other clients. Again I set $s_{k+1} = \tau_k(s_k)$.

4. The correct treatment of latency

The k'th UPDATE transaction with pending operations:



I'll prove in a moment that $\rho_i(s_l) = s_k$. I rename $\tau_i \backslash \rho_i$ to τ_k and append it to the pending operations of all the other clients as before. Then it just remains to prove that $(\rho_i \backslash \tau_i)(\tau_i(s_l)) = s_{k+1}$. Note that the pending operations are transformed before being applied to the editable copy, let's not worry about that now!

4. The correct treatment of latency

Lemma

$(\rho_i \backslash \tau_i)(\tau_i(s_l)) = s_{k+1}$

Proof.

I make use of $\rho_i(s_l) = s_k$, proved in a moment, together with TP1:

$$\begin{aligned}
(\rho_i \backslash \tau_i)(\tau_i(s_l)) &= (\tau_i ; \rho_i \backslash \tau_i)(s_l) \\
&= (\rho_i ; \tau_i \backslash \rho_i)(s_l) \\
&= (\tau_i \backslash \rho_i)(\rho_i(s_l)) \\
&= \tau_k(s_k) \\
&= s_{k+1}
\end{aligned}$$

□

4. The correct treatment of latency

Lemma

$\rho_i(s_l) = s_k$

Proof.

Note s_l is the value of the i'th client's and the server's copy at the end of the $l-1$'th transaction. Then observe that ρ_i is $\tau_l; ...\tau_{k-1}$ and since $\tau_l(s_l) = s_{l+1}$ all the way up to $\tau_{k-1}(s_{k-1}) = s_k$. □

And with a simple inductive argument we're done:

Theorem

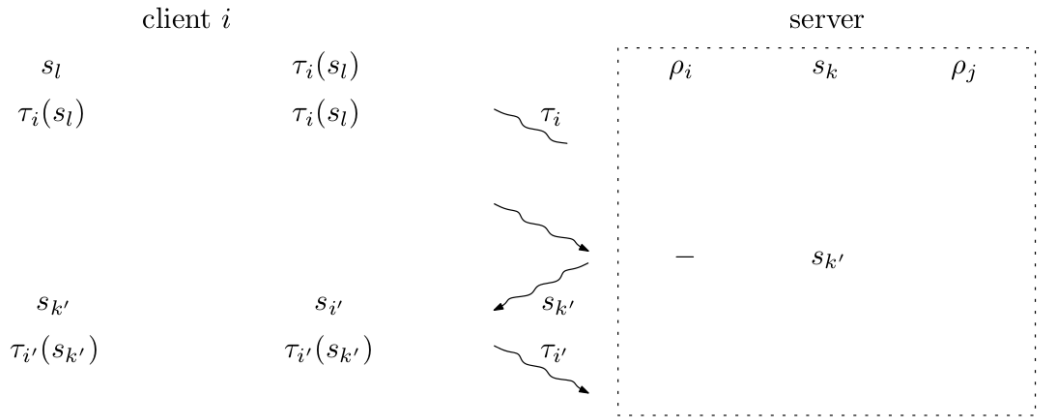
At the end of the k'th transaction the i'th client's and the server's copy are in line.

The main contributions

1. A set of comprehensive, string-wise operational transformations ✓
2. A recursive function that composes transformations and is guaranteed to terminate ✓
3. A distributed algorithm including a protocol that makes use of the recursive function ✓
4. The correct treatment of latency ✓

Bonus contribution: Fault tolerance

Suppose the k'th transaction times out. The client i'th client re-initialises to become the i''th client, keeping its editable copy:



It can optionally keep its editable copy in which case a new sequence of operations $\tau_{i'}$ is generated.

Note that it doesn't matter whether the transaction fails before or after the request reaches the server.

Future directions

- I can't stress enough that it really doesn't matter whether latency is measured in milliseconds or minutes. The algorithm would work in space!
- Consider a data type that is a set of bank accounts or some such, with operations being transfers. Lack of preservation of intention would just mean there's not enough money in one particular account.
- What I'd call a star topology would mean a set of super-clients gathering operations from say, a hundred to a thousand clients each, before aggregating them and passing them on to the server.

Thank you

Thanks for your attention!

James Smith
http://djalbat.com
jecs@imperial.ac.uk