

Concurrency and Automatic Conflict Resolution



Frank Rosner · Sep 4 '18 · 1 min read

#distributedsystems #concurrency #versioncontrol #databases

Introduction

Modern software applications are often required to be reliable and scalable. By combining multiple unreliable components into one bigger, distributed system, we can achieve higher reliability and scalability than what would have been possible with a single component.

However, distributed systems are much harder to reason about, implement correctly, and maintain compared to a single instance deployment. There are many issues you have to deal with in distributed systems:

- unreliable clocks
- unbounded network delays
- limited network bandwidth
- non-static member set
- heterogeneous components
- arbitrary process pauses
- ...

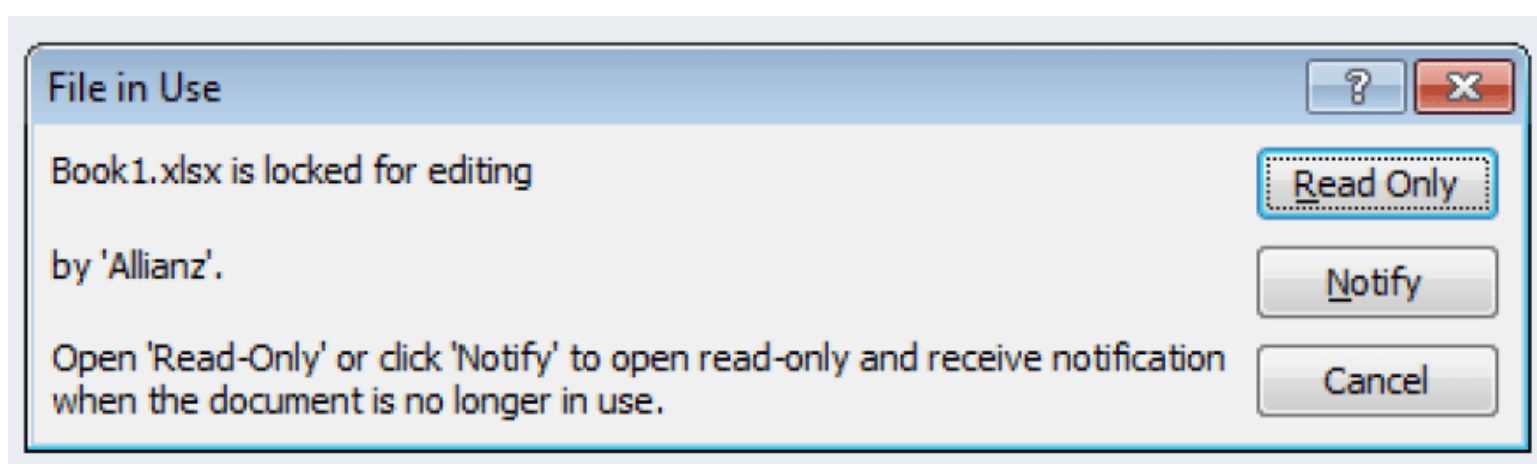
In this post we want to focus on one particular issue: *Concurrency and conflicts*. Two operations are considered to be concurrent if they are not aware of each other. Concurrent operations can be executed interleaved or sequentially - there is no meaningful ordering between them. Concurrency is about how a system is *designed* and should not be confused with *parallelism*,

which defines how a system is *operating*, i.e. running operations at the same time.

Whenever you are dealing with concurrency you most likely have to deal with conflicts. A conflict happens whenever a resource is modified concurrently, i.e. by more than one concurrent operation. We can look at concurrency and conflicts from very different levels of abstraction: We might be talking about two people editing the same document, two processes increasing the same counter in a database, or two CPU cores modifying the same memory address.

How can we deal with conflicts? Well, a common strategy is to avoid them using synchronization techniques such as locking. Consider a desktop text processing application: Whenever someone opens a document it gets locked and cannot be modified by anyone else. Databases often support explicit locking by the client but also use locking internally to isolate transactions.

The main drawback of locking is that depending on the amount of contention in your system processes might have to pause their execution waiting for locks to be released. You need to consider and deal with deadlocks in case two processes are waiting for a lock held by the other one. If contention is high in databases, transaction throughput might suffer. If contention is high for a Microsoft Excel document, people might start being at each other's throats:



In some cases however we can live without any synchronization. For example: Collaborative editors like Google Docs or Etherpad, distributed version control systems like Git, or distributed databases like DynamoDB or Cassandra deal with (some of the) conflicts without any synchronization.

The remainder of this post is structured as follows. First we want to look at conflict resolution concepts in general, discussing the advantages and disadvantages of two commonly used techniques to deal with conflicts. Afterwards we will introduce two alternatives that provide automatic conflict resolution. We are closing the blog post by summarizing the main findings.

Dealing With Conflicts

The Right Tool For The Job

There are many different algorithms and techniques available for conflict resolution. Finding the right algorithm requires you to look at your problem very closely. Here are a few questions that you might want to ask yourself:

- How many conflicts do you anticipate to happen? Is a conflict an exception or happening on a regular basis?
- How many parties are able to make concurrent modifications?
- What is the connection between the involved parties? Dealing with conflicts that happen on the same machine can be easier than having to resolve conflicts through an unreliable network connection.
- Is it OK to lose data? If conflict resolution by simply selecting one of the conflicting updates and discarding all the others ones is applicable, you can reach a consistent state fairly easily.

If you are aware of your requirements, you can select one of the many possible techniques or algorithms to deal with conflicts. One important point remains: In order to deal with conflicts we might have to detect them first. Luckily there are different methods available that version the resource changes by keeping track of some form of modification history. This can be achieved, e.g. by explicitly tracking the complete history of all changes like it is done in Git, or using vector clocks [1, 2] as it is done in DynamoDB.

First of all let's look at two approaches to "resolve" conflicts that are easy to implement but come with some obvious disadvantages: *Discarding concurrency* and *resolution delegation*.

Discarding Concurrency

Also sometimes known as *last write wins* (LWW), this technique deals with conflicts by discarding all but one operation involved in the conflict. It is used in Cassandra, e.g., to resolve conflicting updates happening on multiple replicas. The obvious disadvantage is that resolving a conflict effectively means losing data.

When using this technique you should make sure to avoid conflicts if possible and be ready to deal with lost updates. In Git, this technique is also [available](#) by using the command line arguments `--theirs` or `--ours` but you should know what you are doing in order not to upset your colleagues.

Resolution Delegation

Resolution delegation makes any detected conflict explicit by returning both versions of the value on the next read request and ask the client to resolve the conflict. Git, e.g., uses this technique if the merge algorithm cannot

automatically combine two versions of the same file by stopping the merge and asking the user to manually resolve the conflict.

While being more graceful than the previous technique, resolution delegation comes at a price. Firstly, conflict detection is an expensive operation. Maintaining a complete history of changes might be feasible for a source code repository but not for a database. Vector clocks are more efficient but still impose an operational overhead for every write. Secondly, by delegating the conflict you are not really resolving it. The problem just gets moved to someone else.

Luckily there are other alternatives that do not lose any updates but still resolve conflicts without delegation. Let's look at two notable examples of automatic conflict resolution techniques: *Operational transformation*, and *conflict-free replicated data types*.

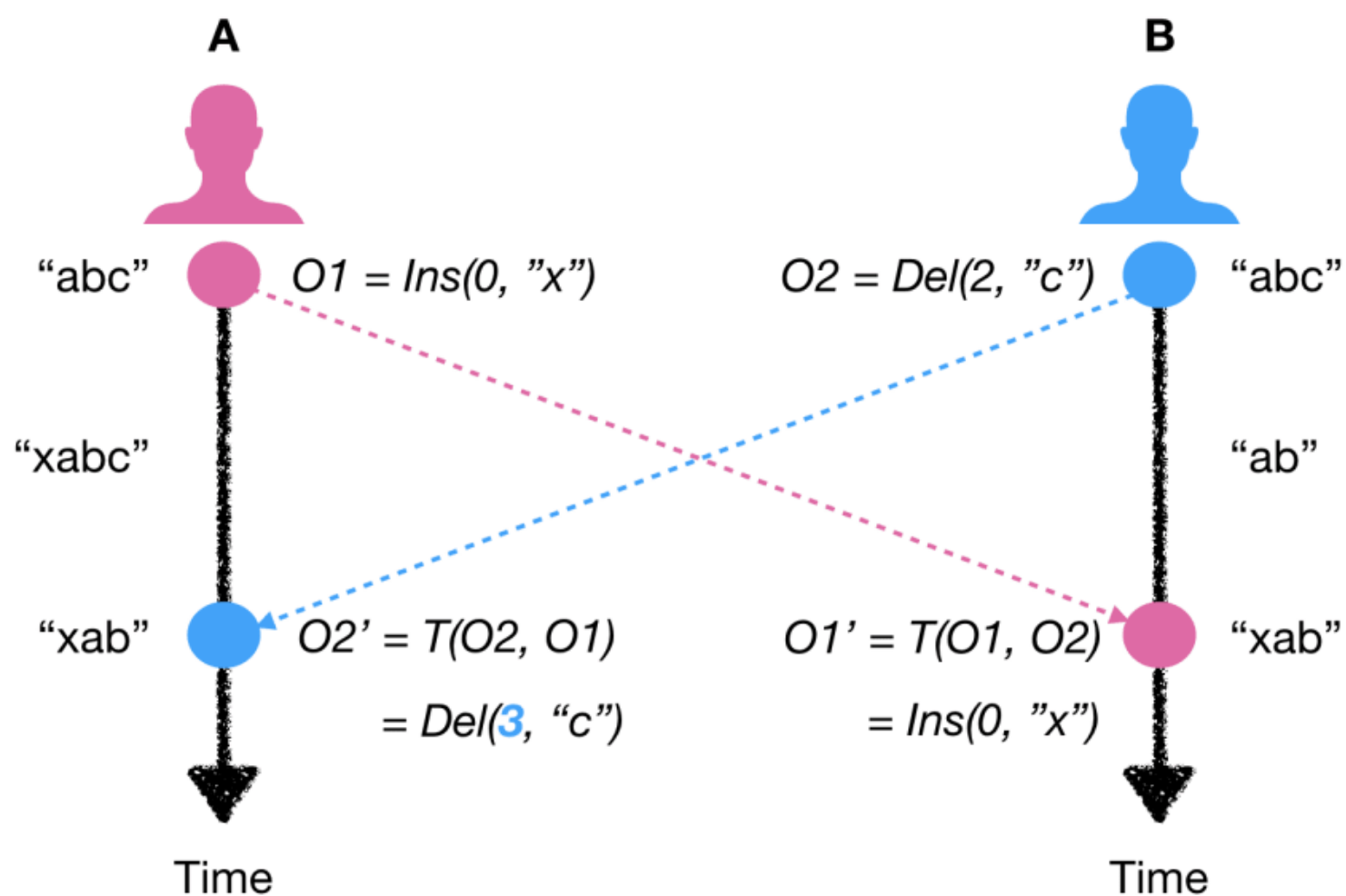
Automatic Conflict Resolution

Operational Transformation

Operational transformation (OT) [3] was designed to achieve consistency within collaborative text editors. Researchers developed different extensions and variations over the years and applications such as Google Wave were the outcome.

The main idea behind OT is to keep clients performing concurrent modifications in sync by sharing the operations they perform and making each client apply them asynchronously. This way, all clients should eventually reach the same consistent state. A problem arises if by the time an operation arrives at a client it already modified the document itself, which might cause a conflict with the incoming operation. This is where the *transformation function* comes into play.

Consider the following example as illustrated in the figure below: Client *A* and *B* are both concurrently modifying a document "abc". *A* wants to insert the character "x" at position 0 ($O1 = Ins(0, "x")$). At the same time, *B* is deleting character "c" at position 2 ($O2 = Del(2, "c")$).



Now both exchange their operations to enable the other client to catch up with the modifications. However when A receives the delete operation, the character is no longer at position 2. This is why all remote operations are passed through a transformation function T . In our particular example, $O2'$ is computed by transforming $O2$ given $O1$, which in this case corrects the position of the character to delete to 3, because the preceding operation $O1$ inserted a character at an earlier position.

Although in our given example applying the transformation function looked rather trivial, it is very difficult to prove the correctness of a given transformation function. [4] Proofs from published papers were later found to be incorrect and OT algorithms turned out to be very difficult to implement. However, there is an alternative to OT which is actively being researched: Conflict-free replicated data types.

Conflict-Free Replicated Data Types

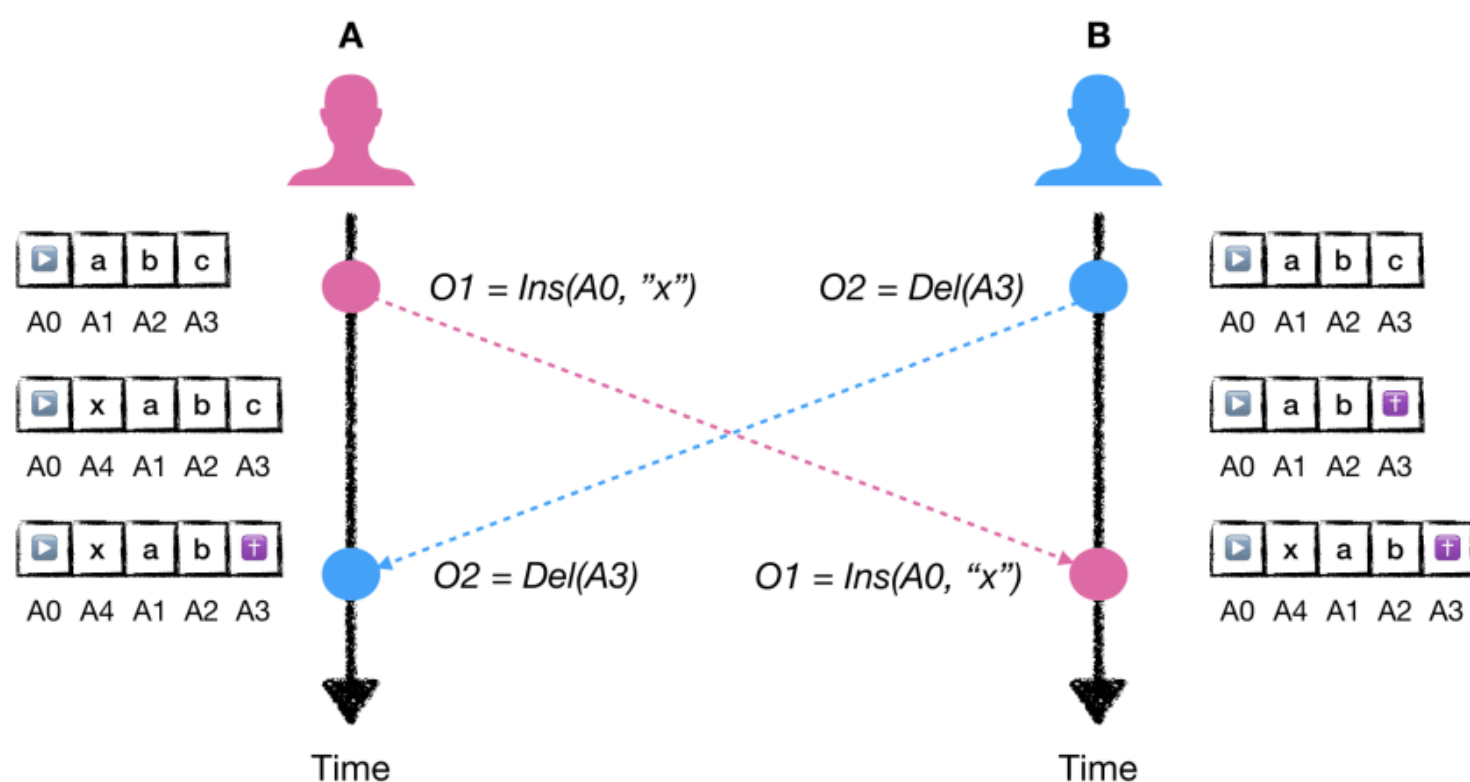
A conflict-free replicated data type (CRDT) [5] is a data structure which can be replicated in a concurrent application without the need for manual conflict resolution. Depending on the implementation, updates can be sent as deltas (corresponding to operations in OT), or by sharing the complete state. If your clients are communicating frequently it is more efficient to only share the individual operations. If the communication happens through a very unreliable network however, e.g. mobile or satellite, sending the complete CRDT every once in a while might be the better option.

The main idea of CRDTs is similar to the one of OT. The difference lies in the fact that clients do not have to transform incoming messages as they already contain all the information required to resolve conflicts with the local state.

While in OT clients have to remember previous operations explicitly to then transform new operations given this history, CRDTs persist enough information within the data structure to create operations that can be merged without any transformation.

Looking at our collaborative text editor example from the previous section, we can use so called sequence CRDTs. Replicated Growable Arrays (RGA) [6] is one example that we will take a closer look at now. I stumbled upon it while watching a conference talk about JSON CRDTs [7].

The figure below illustrates how our two clients are concurrently modifying the document "abc". This time however we are representing the document not as a simple string of characters, but adding a unique identifier to each position. This identifier, e.g. *A1*, is composed of the a globally unique client identifier (*A*) and a client-unique numerical identifier (*1*).



Whenever a client applies an operation it can reference the character by identifier rather than by position. As the identifier is unique the operation will be still valid when received by another client, even if concurrent modifications happened in the meantime. If client *A* tells *B* that it inserted "x" after *A0*, *B* knows at which position to insert. Deletions happen in a similar fashion but we are not actually deleting the character from the data structure but simply marking it deleted with a so called tombstone.

Two open questions remain: How do we assign new identifiers, and how do we deal with two concurrent inserts at the same position?

New identifiers are generated as follows: Every client keeps track of the highest numerical identifier associated with its client ID. Whenever it inserts a new character, it increments the highest numerical identifier and assigns a new identifier composed of its own client ID and the increased number.

Concurrent inserts at the same position need to be resolved in a way that leads to a consistent state. Let's say both *A* and *B* wanted to insert "*x*" and "*y*" respectively after *A0*. If we applied the operations without any special rule, *A* would end up with "*xyabc*" and *B* would get "*yxabc*". As there is no correct answer it is really just a matter of deciding for one version consistently across all clients. To achieve that we are only inserting at the requested position if there is no higher identifier at the next position. Otherwise we keep walking to the right until we find a position with a smaller identifier and insert there. Because $B4 > A4$, the final document would be "*yxabc*".

Besides RGA there are many more CRDTs at your disposal. The [Akka distributed](#) module, e.g., provides counters, sets, and maps based on CRDTs together with different consistency guarantees when used within Akka cluster mode. [Redis enterprise](#) offers a CRDT based replication architecture. Since version 2.0, [Riak](#) has types based on CRDTs in their portfolio.

Nevertheless, CRDTs are not a silver bullet. They only fit for a certain set of problems and induce additional storage overhead because they need to keep track of enough history information within the data structure to enable merging incoming operations or state.

Conclusion

We have seen that distributed systems come with a lot of different challenges. Concurrency in distributed systems becomes harder to deal with due to unreliable communication between nodes. If two processes concurrently modify the same resource, conflicts occur.

In order to resolve conflicts we might have to detect them first. This can be achieved by keeping track of some form of history. As soon as we detect a conflict we can either resolve it automatically or delegate the conflict resolution to the next level, e.g. the application code or the user. While discarding all but one conflicting write is a commonly used technique it comes with data loss.

Automatic conflict resolution techniques that promise no data loss exist such as OT or CRDTs. We are not getting the advantages for free, however. OT turns out to be hard to reason about and implement correctly. CRDTs have significant memory overhead as they store some history information explicitly within the data structure. Both OT and CRDTs are only applicable to a specific problem set and there is no one-size-fits-all solution, unfortunately.

Did you use OT or CRDTs before? Did you ever lose data without being aware that your application uses LWW to resolve conflicts? Let me know

your thoughts in the comments.

References

- [1] Colin J. Fidge (February 1988). "Timestamps in Message-Passing Systems That Preserve the Partial Ordering" (PDF). In K. Raymond (Ed.). Proc. of the 11th Australian Computer Science Conference (ACSC'88). pp. 56–66. Retrieved 2009-02-13.
- [2] Mattern, F. (October 1988), "Virtual Time and Global States of Distributed Systems", in Cosnard, M., Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France: Elsevier, pp. 215–226
- [3] Ellis, C.A.; Gibbs, S.J. (1989). "Concurrency control in groupware systems". ACM SIGMOD Record. 18 (2): 399–407.
- [4] Du Li & Rui Li (2010). "An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems". 19 (1): 1–43.
- [5] Shapiro, Marc; Preguiça, Nuno; Baquero, Carlos; Zawirski, Marek (2011), Conflict-Free Replicated Data Types, Lecture Notes in Computer Science, 6976 (Proc 13th International Symposium, SSS 2011), Grenoble, France: Springer Berlin Heidelberg, pp. 386–400
- [6] Roh, H.G., Jeon, M., Kim, J.S. and Lee, J., 2011. Replicated abstract data types: Building blocks for collaborative applications. Journal of Parallel and Distributed Computing, 71(3), pp.354-368.
- [7] Kleppmann, M. and Beresford, A.R., 2017. A conflict-free replicated JSON datatype. IEEE Transactions on Parallel and Distributed Systems, 28(10), pp.2733-2746.

dev.to is a social network for software developers.

Lean more

(open source and free forever)



Frank Rosner + FOLLOW

My professional interests are cloud and big data technologies, machine learning, and software development. I like to read source code and research papers to understand how stuff works.

@frosnerd FROSnerd FROSner


Add to the discussion



PREVIEW

SUBMIT



zeddotes 

Sep 5 '18 

Great read.



REPLY



Frank Rosner  

Sep 5 '18 

Glad you liked it!



REPLY



Frank Rosner  

Sep 4 '18 

Here you go [@maphengg](#) :) Fulfilling your request and writing about CRDTs :D



REPLY

[code of conduct](#) - [report abuse](#)

Classic DEV Post from Sep 13

Follow Friday: Which DEVs would you recommend following?



Helen Anderson

Follow Friday: Which DEVs would you recommend following?



Another Post You Might Like

Fantastic Faults and What to Call Them



Vaidehi Joshi

A fault is really just anything in our system that is different from what we expect it to be. Whenever some piece of our system deviates from its expected behavior, or whenever something unexpectedly occurs in our system, that behavior itself is a fault!



Another Post You Might Like

Ticking Clocks in a Distributed System



Vaidehi Joshi

Ticking clocks in a distributed system We often spend a lot of our lives blissfu...





Path to become a junior+ data engineer?

FAN 2 TUNNING - Oct 12



Unveiling Ballerina Non-Blocking Architecture

Vinod Kavinda - Oct 9



Ordering Distributed Events

Vaidehi Joshi - Oct 30



ESCAPE/19 - An Unbiased Multi-Cloud Conf in NYC

ESCAPE/19: The Multi-Cloud Conference - Oct 2

[Home](#) [About](#) [Privacy Policy](#) [Terms of Use](#) [Contact](#) [Code of Conduct](#)

DEV Community copyright 2016 - 2019 