

Causal trees: towards real-time read-write hypertext

Victor Grishchenko
Ural State University
51 Lenina ave
Ekaterinburg, Russia
victor.grishchenko@gmail.com

ABSTRACT

Impressive advances in knowledge compilation and organization were demonstrated by the Wikipedia project which currently approached its social scalability threshold. It is conjectured that the next major impediment for further scaling of wikis is centralization. The paper explores the vision of federated wikis, where changes are filtered and disseminated by a web of trust. The objective is to automate information dissemination the way hyperlink automated associations and search engines automated search. The immediate technical problem was to find a version control and change propagation technology for those specific conditions. Classical approaches were found inapplicable, so a new "causal trees" model was developed. The paper discusses both theoretical aspects and implementation experience with the new technology.

1. INTRODUCTION

While the Web contains a tremendous wealth of information, that information is scattered into myriads of pieces. Search engines focus on spotting the top ten pieces out of a myriad. Wikipedia had demonstrated the tremendous potential of crowdsourcing, fusing together small pieces of information contributed by volunteers. Secret for the Wikipedia's success is the ability to provide an easily digestible brief compilation of available knowledge on the topic as well as leads for deeper investigation.

Still, the recent slowdown in Wikipedia growth [24] demonstrates that the technology has hit some glass ceiling of scalability. The hypothesis is that due to its centralized architecture, Wikipedia cannot accommodate more specific knowledge, thus locking itself in the borders of the eBritannica vision. More specific and/or local knowledge cannot be evaluated by volunteer moderators easily; each particular topic will gather much less eyeballs – so quality may suffer. Finally, the advance into the "long tail" will require more technical resources.

Naturally, more specific content might be accommodated by smaller specialized wikis. This, in turn, raises issues of fragmentation, duplication of effort, contributor inclusion/exclusion etc.

The original motivation behind the research is to create a suitable change propagation mechanism for the Bouillon social wiki [19]. The ultimate goal of the Bouillon project is to achieve both technical and social decentralization of wikis, still avoiding content/community fragmentation. The general vision is to perceive different wikis as different versions/projections of the same language/knowledge space. The key process is to let wiki sites easily exchange content to propagate "beneficial mutations" while filtering out "mistakes". Thus, moderation and editor conflicts are replaced by a pure market process: each participant is free to "buy" some new changes, as well as to reject.

An important additional requirement is to let changes propagate in real-time. Speculatively, as people become more and more densely connected, they are more and more aware of each other's details. As a consequence, communications naturally gravitate to compact and speedy update-only forms. So, a secondary objective is to re-energize wikis by introducing real-time collaboration features and associated usecases.

1.1 Relevant projects

Retrospectively, the read-write web vision was popular since long before the Web itself. The historical prototypes involve NLS by Engelbart [10] in 60s, Xanadu by Nelson [16] in 70s, Intermedia [15] in 80s. A relevant contemporary technology is wiki [2], a freely-editable hypertext. It is interesting, that the inventor of wikis Ward Cunningham proposed the Folk Memory [18] concept featuring change propagation by a network of peers – a technology described as peer-to-peer wikis in the current terms. WebDAV [13] was a high-profile effort to introduce writability and versioning into HTTP; it did not reach widespread use.

On the other side, there is some number of currently existing real-time or near-real-time, web-based or standalone, commercial or open-source editors [12, 11, 1]. The most notable ones are SubEthaEdit (standalone, Mac OS X) and Google Docs (web based). Such editors do not focus on the hypertext or publishing aspect, their primary objective is real-time collaboration of closed groups. Usually, such applications employ either diff/patch or operational transformation (OT) technology for version management and change

propagation. Google Docs functioning, for example, reminds of the classic CVS [6] workflow with commit/update cycles happening every minute (employs JavaScript implementation of diff/patch algorithms [8]). Classical diff/patch-based and OT-based solutions imply existence of a central server to serialize and store changes. Albeit, some novel approaches assume decentralized architecture, such as decentralized version control systems (git [7], Codeville [5], Bazaar [3], etc) or federated wiki projects XWiki-Concerto [17] and Wooki [25] employing post-OT framework WOOT [20].

1.2 Relevant version control technologies

1.2.1 Source code version control and diff/patch

diff and *patch* are classic UNIX tools for file comparison, change serialization and application. The underlying algorithm of the *diff* program is the longest common subsequence search. When serializing changes, *diff* identifies the changed piece of the text both by its position and its context (the preceding and following pieces of the text). Later, the *patch* utility may reapply changes to a slightly modified version of the text – even if the context of the change itself has changed a little. More or less, contemporary version control systems revolve around diff/patch. Their underlying abstraction is a directed acyclic graph where nodes correspond to versions of the text and arcs correspond to changes (diffs).

The case of distributed version control systems is more complicated than pairwise file comparison. The file modification history is non-linear; versions are just partially ordered. Thus, out of two merged versions, no version is “old”; some additional information is needed to detect which pieces were added or deleted in each version. Typically, VCS analyzes file modification histories, the simplest case being the 3-way merge [22]. Examples of more sophisticated algorithms include git’s octopus or Codeville’s “who wins” algorithms.

Version-control approach has extensive drawbacks in the case of distributed real-time collaborative editing. In a real-time environment, changes are introduced at a higher rate, so the number of versions is higher. In an *asynchronous* distributed environment with non-complete connectivity, the number of possible versions of the text depends on the number of possible change recombinations. Thus, the number of versions combinatorially depends on the number of users and the number of edits they make. Some estimations show 2^N possible versions, where N is the number of changes to a document during some period of time. Be it N^2 , it does not scale well anyway. So, in a large distributed group of collaborators, even listing different versions of a text might be a challenge.

Other problems with the approach include complex patch interdependencies, possible ambiguity in patch context identification and the dependence of the resulting text on the order of patch application. Partially, that problems might be resolved by introducing a central coordinating entity, but the objective was the opposite. Another possibility typically employed by VCSs is to escalate complex merging problems to the end user; that is undesirable in our case as the intended user audience (“average” users) lacks necessary skills.

1.2.2 Operational transformation

Operational transformation is a theoretical framework for handling concurrent changes of some shared document; it is typically employed by real-time collaborative editors. Each user’s edits are decomposed into *operations* (often, an operation is either addition or removal of a particular character). *Transformations* of operations are needed to incorporate concurrent remote changes into a local copy of a document. Operational transformation research typically lists three requirements [23]:

1. convergence (all copies converge)
2. causality preservation (the original order of execution of the operations does not change at different copies)
3. intention preservation (independent operations do not interfere)

The operational transformation theory is generally regarded as too complex. Some recent works on OT include a thesis by J. Preston [21].

A notable post-OT work is WOOT [20] (“WithOut Operational Transformation”), which makes some progress in developing change control model for unbounded (peer-to-peer) environments with the projected goal of implementing peer-to-peer Wikipedia. In plain words, WOOT attaches some metadata to every letter of a document: an unique id, id of the preceding letter, id of the following letter. That differs from the positional approach of earlier OT models. WOOT algorithms focus on document assembly according to the aforementioned requirements. In a peer-to-peer environment no locking is possible, so WOOT is made lock-free. Currently, WOOT is employed by XWiki-Concerto [17] and Wooki [25] projects. Still, even simpler practical framework was required to meet the objectives of this research.

2. CAUSAL TREES

As mentioned, the main purpose of the Bouillon project is to build a real-time read-write medium where changes are propagated and filtered by a social network of participants which approach mimics (and aims to automate) natural social processes. That vision converts to a list of technical requirements for the change control technology.

1. it must process fine-grained changes to the text
2. it must be able to work in real-time
3. it must be absolutely decentralized, use no special “central” entities
4. it must satisfy the aforementioned OT requirements
5. it must not list/poll every collaborator as the circle of collaborators is potentially unbounded
6. it must not even try to list every version of the text, as the number of possible “versions” is subject to combinatorial explosion
7. merge of diverged versions must (MUST) be effortless independently of their modification history intricacies, as manual intervention can not be requested

As we need a readable text, not an executable program, some minor mess might be tolerated. It is a common situation when some bit of correctness is sacrificed for some

more scalability. The WOOT is the most close match to the requirements, except the author considered it too complicated.

2.1 The CT model

The causal trees model builds on rather simple rules. Every text consists of atoms; each atom either exists or not (i.e. a letter, an HTML tag). So, any change may be represented as a set of atom removals and insertions. Atom removals are represented as insertions of special “backspace” atoms. If an atom b was initially inserted just after an atom a , then b was *caused* by a . *Inactive* atom is an atom that causes an active backspace atom. Every atom has an unique numeric id (must be unique in the document scope, better be unique globally). So, an atom is a triple $(id, causing_id, letter)$.

An atom’s id is always greater than the id of the causing atom. Also, all atoms contributed by a given participant have monotonically increasing ids. Technically, a 64-bit id consists of a 32-bit timestamp and 32 bits of anti-collision pseudo-random padding. Considering the Birthday paradox [4], $\sim 2^{16}$ users will have to push a button at the same page at the same second to reliably reproduce id collision.

Naturally, the causality relation defines a partial order. Technically, atoms are stored in append-only *causality feeds* and every feed complies with that order: the causing atom always precedes any of its caused atoms. Still, the particular order of atoms varies from feed to feed depending on the order of arrival. The causality relation forms *causality trees* of atoms, where each causing atom acts as a parent node to its caused atoms. Any modification of a text is represented as attachment of new subtrees. A *causality string* is a sequence of atoms where each atom causes the next.

Linear (textual) form of the causal tree is defined as its depth-first preorder traversal. Order of children is defined by their age: younger goes first. “Younger” is a synonym for “larger id”. The resulting sequence is actually a *weave*, a version control data structure containing every piece of the file that ever existed, put in their natural order. Given a weave, it is easy to construct any historical version of a file or to compute a difference between any two versions. The actual text is constructed by removing inactive atoms from the weave. A simple scenario of a word “Test” being corrected to “Text” is shown Fig. 2.1, including all the mentioned data structures.

The CT format provides two possibilities to embed metadata. First, pieces of rich metadata might be attached using special symbols (like the backspace symbol). Second, given a causality string, its concatenated 32-bit paddings may contain some metadata according to some convention, e.g. name of the author, a public PGP key or a signature.

2.2 Propagation

From the standpoint of a user, the social dissemination process is masked by an ordinary open/save cycle. A user is supposed to be subscribed to a certain number of *sources*. Those sources may correspond to institutions, communities, groups, etc. Each source hosts multiplicity of feeds identified by free-form strings. A source might be accessed in one of three modes: read-only, read-write or real-time.

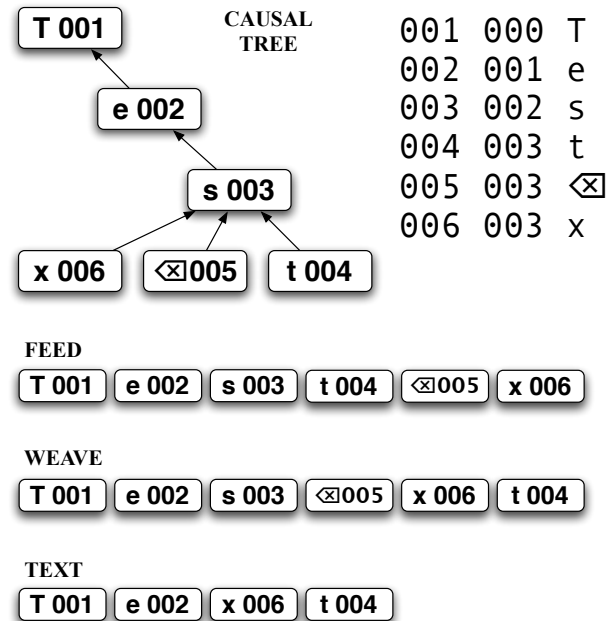


Figure 1: Causal trees and related formats. Top-right: feed in the tabular form, each row corresponds to an atom (id, causing id, the actual letter). The bar-and-cross sign stands for backspace.

When a user requests some page, feeds for that page are downloaded from all available sources and merged. The resulting composite feed is shown to the user. Normally, there is a locally saved “default” feed which keeps the last saved state; so, any changes since the last save are highlighted. The user is free to introduce own changes as well as to accept/reject any imported change.

Real-time feeds immediately get result of every user’s keystroke. Read-write feeds, including the default feed, are synchronized to the state of the composite feed when the user saves the page. The annoyance of highlighted diffs is supposed to stimulate the user to do that (or to use some form of auto-save). As pressing Ctrl+S is much simpler than introduction of own changes, feeds of different sources are supposed to converge in the long run (contrary to the possibility of total divergence making incremental merging impossible).

2.3 Algorithms

Main algorithms of the CT model are text assembly and feed synchronization. The text assembly algorithm makes one N -step loop to walk the tree and compose the weave. Another N -step loop iterates over the weave in reverse order, filters out inactive atoms and produces the resulting text. Depending on the data structures used, the overall complexity is $O(N)$ or $O(N \log N)$. Version diffs are trivially built by comparing states of an atom in each respective version.

The feed synchronization algorithm has to bring a feed to an equal state with the composite feed, i.e. to make their sets of active non-backspace atoms identical. The synchronization algorithm must minimize length of the feed; prac-

tically, the “historical garbage”, i.e. removed atoms, must not be transferred if possible. The actual algorithm makes several passes over the composite feed to (a) transfer any active non-backspace atom and any its causing atom (transitively) and (b) transfer any active backspace atom if its causing atom is already present in the subject feed.

2.4 Discussion

The explained causal trees architecture shares some concepts with OT, except for no transformation. It is similar to WOOT, but simpler. Compared to the diff/patch approach, the architecture does not rely on contexts: all atoms are uniquely identified, thus the position-identification problem is bruteforced. Interestingly, causal trees bear some resemblance to the historical Xanadu [16] versioned hypertext model.

Being compared to source code version control systems, the distinctive feature of the CT model is its preparedness for small incremental real-time updates, instant branching and merging.

The general idea of identifying every letter in a document (or even globally) might seem eccentric by the standards of 80s. Currently, the associated computational costs are tolerable to negligible, as the text itself no longer occupies a significant share of bandwidth or storage. For example, all texts of the Russian Web fit into a single 4U server in an indexed form.

Although the proposed model is expressed in slightly different terms than OT, the three requirements of OT are met. Convergence is obvious: given the same set of atoms, the resulting text is the same. So the causality requirement: if a is (transitively) causing b then b will surely be applied after a . If a and b are causally incomparable, the order of application is mostly irrelevant; still, it is normally preserved by implementations. Also the intention preservation: if a user contributes a chain of symbols in the natural way, that original sequence will not be interrupted by any external symbol until somebody will explicitly insert one.

Although any CT merge is technically unambiguous, the resulting text might be slightly messy. Nothing smart could be done about it, as we can't merge texts semantically. As a program will surely fail if trying to be smart then better it be simple and consistent.

3. IMPLEMENTATION EXPERIENCE

Three prototypes were developed to evaluate different scenarios for real-time social wiki backed by a CT engine. The actual algorithms used for CT manipulations greatly varied depending on available data structures and other specific conditions. Attempts to implement the CT logic in JavaScript and C greatly contributed to the simplification of the model.

Feature requirements for the prototypes were: to make a CT-backed WYSIWYG editor able to act as a real-time collaborative editor and to perform all the necessary inline change highlighting functionality. (Note: “real-time” stands for 1-2 sec user-to-user change propagation time.)

3.1 DOM-based prototype

DOM-based prototype was written in JavaScript and plugged into an existing WYSIWYG browser-based editor (FCKEditor) to add real-time collaborative editing ability. The prototype, in fact, imitated a non-JavaScript implementation, as it relied on W3C DOM manipulations. The approach could be practical if embedded into a browser engine as C code; JavaScript's DOM traversal performance is too poor. JavaScript DOM also lacks any natural possibility to attach metadata to text nodes, so an ugly hack was used. Change serialization/application was implemented by the “polymerase” algorithm. Namely, a bi-iterator named “polymerase” traverses in parallel the causality tree and the document's object model (depth-first order) to detect differences and to perform synchronization. All the content behind the iterator is in perfect match: user-made local DOM changes are converted to atoms, imported remote changes are reflected in the DOM tree.

The server part functioned as a regular HTTP server responding with preassembled HTML pages or raw feeds. Pages contained the JavaScript code. Technically, it was implemented in C++ as a 500-line libevent-based [9] HTTP server. The real-time update effect was achieved by Comet [14] HTTP polling for the feed tail. The server employed a number of performance optimizations. For example, the data format lets dynamic content to be partially cached, so the client retrieves only the delta. Standard HTTP compression partially compensated bloatedness of the feed format, so the actual transfer consumed just about 5 times more bandwidth than the original text. As the client gets a preassembled page first and feed downloading proceeds in the background, feed size was not an issue.

3.2 HTML-based prototype

The second prototype was JavaScript-friendly; it was based on manipulations with the innerHTML property. As the CT code composed the HTML, it was passed to the browser for rendering and editing. The Google's JavaScript diff implementation was employed to isolate changes made by the user to the HTML. The prototype was practically usable. The single most important performance trick was HTML chunk caching. It eliminated the need for instant recalculation of the whole HTML, so the prototype had acceptable performance.

First two prototypes took 500-800 lines of JavaScript code each, not counting the libraries. Besides usual JavaScript troubles, the most significant problem with the these prototypes was the fact they processed HTML. An HTML document, differently from a plain text, has internal structure that might be violated. E.g. when merging CTs of two perfectly correct versions of an HTML page, the result might be incorrect. To minimize the danger of HTML tag invalidation all tags were substituted for single Unicode symbols using a dictionary. Other HTML-related problems remained unsolved. For example, merging of concurrent changes may introduce two tags which brace overlapping ranges. An opportunistic approach is to let an existing HTML engine to parse the “tag soup”. Other possibilities involve mechanisms of automatic conflict resolution (such as rejecting the conflicting change).

3.3 Wikitext-based prototype

The third prototype processed wikitext; it was a dramatic simplification compared to HTML as wiki markup is much more relaxed and inter-symbol dependencies are more local and sparse. Changes in visible markup are also easily detected. To satisfy the WYSIWYG requirement, a special “wikiwyg” syntax highlighting mode was introduced. Namely, markup was rendered in watered colors, while the text was formatted according to the markup (larger size for headers, bold, italic, etc). As the author had absolutely tired of JavaScript, the third prototype is made in C++ using the QT toolkit. Generally, the third prototype is dramatically simpler and is able to communicate via FTP, SSH or filesystem. Key algorithms were implemented in a 100-line C++ file.

So finally, prototypes proved the approach to be cheaply implementable in several different ways. The causal trees model and the algorithms are practical and working.

4. CONCLUSION

As a version control technology, causal trees enable real-time co-authoring in a distributed hypertext environment. Causal trees are suitable for such extreme conditions as unbounded groups of collaborators, the associated instant branching and merging, divergent replicas, etc.

Causal-tree-backed hypertext might be deployed in a broad range of scenarios: in a peer-to-peer network, as a public web service or as a private intranet wiki that mostly pulls from outside but rarely pushes. By reconfiguring the topology of change propagation, users are free to build elaborated editorial processes. The technology is quite flexible in that regard.

Finally, the approach is proven to be workable and very simple to implement.

5. REFERENCES

- [1] ACE collaborative text editor. <http://ace.sf.net>.
- [2] Article on wikis at the original WikiWikiWeb site. <http://c2.com/cgi/wiki?WikiWikiWeb>.
- [3] Bazaar distributed vcs. <http://bazaar-vcs.org/>.
- [4] Birthday problem. http://en.wikipedia.org/wiki/Birthday_paradox.
- [5] Codeville ditributed vcs. <http://codeville.org/>.
- [6] FSF page for Concurrent Versions System. <http://www.nongnu.org/cvs>.
- [7] The git version control system homepage. <http://git.or.cz>.
- [8] Google diff-match-patch library. <http://code.google.com/p/google-diff-match-patch/>.
- [9] libevent - an event notification library. <http://monkey.org/~provos/libevent/>.
- [10] NLS demo (1968) video. <http://sloan.stanford.edu/mousesite/1968Demo.html>.
- [11] SubEthaEdit product page. <http://www.codingmonkeys.de/subethaedit/>.
- [12] SynchroEdit collaborative editor product page. <http://www.synchroedit.com/>.
- [13] Webdav resources. <http://www.webdav.org/>.
- [14] Wikipedia entry on the Comet AJAX technique. [http://en.wikipedia.org/wiki/Comet_\(programming\)](http://en.wikipedia.org/wiki/Comet_(programming)).
- [15] Wikipedia entry on the intermedia hypertext system. [http://en.wikipedia.org/wiki/Intermedia_\(hypertext\)](http://en.wikipedia.org/wiki/Intermedia_(hypertext)).
- [16] Xanadu project homepage. <http://xanadu.com>.
- [17] Xwiki concerto p2p wiki project. <http://concerto.xwiki.com/>.
- [18] CUNNINGHAM, W. Folk memory: A minimalist architecture for adaptive federation of object servers. <http://c2.com/doc/FolkMemory.pdf>, 1997.
- [19] GRISHCHENKO, V. S. Bouillon: A wiki-wiki social web. In *CSR (2007)*, V. Diekert, M. V. Volkov, and A. Voronkov, Eds., vol. 4649 of *Lecture Notes in Computer Science*, Springer, pp. 139–145.
- [20] OSTER, G., URSO, P., MOLLI, P., AND IMINE, A. Data consistency for p2p collaborative editing. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work* (New York, NY, USA, 2006), ACM, pp. 259–268.
- [21] PRESTON, J. A. *Rethinking consistency management in real-time collaborative editing systems*. PhD thesis, Georgia State University, 2007.
- [22] RITCHER, B. A trustworthy 3-way merge. <http://www.cmcrossroads.com/content/view/full/6725/120/>.
- [23] SUN, C., AND ELLIS, C. A. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work* (1998), pp. 59–68.
- [24] USER:DRAGONS_FLIGHT. Wikipedia log analysis. http://en.wikipedia.org/wiki/User:Dragons_flight/Log_analysis.
- [25] WEISS, S., URSO, P., AND MOLLI, P. Wooki: a p2p wiki-based collaborative writing tool. <http://hal.inria.fr/inria-00156190/en/>.