A Nonlinear Representation of Page History in P2P Wiki System

Sawsan Alshattnawi, Gérôme Canals, Pascal Molli

Nancy-Université, LORIA/INRIA Lorraine , Campus Scientifique, BP
239, F-54506 Vandœuvre-lès-Nancy, France,
e-mail: {alshattn, canals, molli}@loria.fr

Abstract Awareness about the document evolution is an important part in collaborative editing systems. It is represented always by the versions history. The representation of the document history in centralized collaborative editing systems is linear. However, in distributed collaborative editors, there is no central server and the users can work asynchronously or in isolation and some versions may be produced concurrently, in this case, the history is no long linear. The existing history representations are limited because they don't provide any information about the concurrence on the document history. We introduce here a non linear representation for the page history in P2P wiki systems. The concurrency information about the page versions is provided; the user can explore the page versions that resulted under the user's control or produced by the server in case of merging concurrent modifications.

1 Introduction

Providing history of versions in collaborative editing systems with each document is very important. The history contains all the revisions for one document since its creation, a new revision is created in the history when a user commits his modifications. The history helps the user to understand the evolution of the document, to compare between two versions, to see what are the operations that convert one version to another, who made these operations and he can revert to precedent version.

Please use the following format when citing this chapter:

Alshattnawi, S., Canals, G., Molli, P., 2008, in IFIP International Federation for Information Processing, Volume 286; Towards Sustainable Society on Ubiquitous Networks, eds. Oya, M., Uda, R., Yasunobu, C., (Boston: Springer), pp. 151–160.

Some collaborative editors are based on state-based approach [10]. State-based approach takes into account just the final and initial states of the document. When the user wants to see the evolution from one revision to another, the two versions are *diffed*, *diff* algorithm compares between two files and compute the changes made to a current file by comparing it to a former version of the same file, and the difference between these two revisions is presented. The problem with this approach that there is no information about the operations that transform one state to the other.

Existing decentralized collaborative editing systems are based in operation-based approach [7]. In this approach the operations that transform one revision to another is stored in a history. The operations are kept in a patch that is sent to other sites to be integrated. The patch is a non mutable object and therefore the set of operations that transform one revision to another is the same over all sites. State-based approach is not suitable for decentralized systems because the *diff* algorithm may give different set of operations over each site.

In Distributed Version Control Systems (DVCS) [3], the operations that resulted from the *diff* algorithm between the committed version and the precedent version is reserved in a patch. The patch is diffused to other wiki servers where it will be integrated and recorded by the editor on a log file. When the user gets the history, it is easy to present the difference between versions.

The documents in traditional systems are always produced by the users, in the Copy-Modify-Merge paradigm [8] the central server control all the operations of distributed participants. When it detects concurrent modifications, he asks the user to merge them manually. So, the existing history representation in this case may be enough. However, in DVCS, there is no central server and the users can work asynchronously or in isolation, in this case the merge may be done by the server. The classical way to notify users about merge results is to modify the file itself with conflict blocks. The user who gets the page must search for conflict blocks to know if the page is resulted by the server or by the user. In the history representation there is no indication about the page state.

In a P2P wiki like in DVCS, concurrent changes may occur because of asynchronous work and may be saved on different servers. Merges are not executed when pages are saved but when remote changes are received by each site. To ensure eventual consistency [12], merges are fully automatic and performed by wiki servers. In this case the page is *server-produced*. While when the merge is done under the control of the user the page is *user-produced* [4]. Our objective in this paper is to represent the history of P2P wiki pages by adding information about the pages state if it either user-produced or server-produced. The evolution of the pages concerning the concurrence may help the user to understand the intension of each user and resolving conflicts. This paper is organized as follow: at the next section we mention the history representation in different collaborative editing systems, in section 3 we present our mechanism for detecting the pages states, section 4 presents our history representation in P2P wiki systems and we conclude our work in section 5.

2 Revision Histories in Collaborative Editors

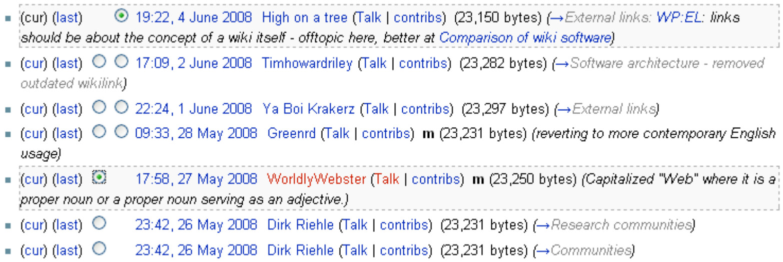
In traditional version control systems such as CVS [5] and subversion [1], no information about the evolution of one state into another is used. They adopted the state-based merge where the information about the state is used. Every time the user commits a file to the CVS repository, a new revision is created. Each revision is identified by a number (for example, 1.1 or 1.11.3.5). Along with each revision is stored the author, the date, and a commit comment. In the revision history view, there is a link near each revision to display *diffs* between that revision and the previous one, and the users is allowed to display *diff* between arbitrary revisions. For conserving space, RCS [14] stores deltas i.e difference between successive revisions. RCS uses the program *diff* which first computes the longest common substring of two revisions, and then produces the delta from that substring. The delta is simply script consisting of deletion and insertion commands that generate one revision from the other.

In Wikipedia [2], every article may be edited anonymously or with a user account. The ”History” page attached to each article contains every single past revision of the article. Revision history of a wiki is traditionally maintained as a linear chronological sequence. The informations that we can get from this history representation are the date and time of every edit, the user name, the modification size and from the history we can compare between any two versions or any version with the actual one. Figure 1 represents the revision history for a page *wiki*[1] taken from the encyclopedia *Wikipedia*. From the page history the user can see the difference between that edit and the current version from the link *cur*, the link *last* shows changes between that edit and the previous version, radio buttons can be used to select any two versions of the page. Near each version the date and the time, the user or the contributor name or IP address, edit summary, the nature of the modification and the size, *m: minor edit* to indicate that the modifications were not over the content , and finally the button *undo* to delete the ancient modifications.

Some approaches are proposed to represent the history for giving additional information about the evolution of the wiki page. In [11] the author proposed a tree representation where each edge has a weight indicates the similarity between the two corresponding page revisions. The tree structure reflects actual evolution of page content, revealing reverts, vandalism, and edit wars.

Another work presented in [15], provides an overview of a document’s evolution by analyzing differences between multiple revisions of one document. They provide information about how a group has contributed to a document or how a modification has influenced the current version of a document. The existing approaches [11, 15] try to show the violence over the pages. Unfortunately, they don’t take into account the concurrency.

[1] http://en.wikipedia.org/wiki/Wiki



- (cur) (last) ◉ 19:22, 4 June 2008 High on a tree (Talk | contribs) (23,150 bytes) (→External links: WP:EL: links should be about the concept of a wiki itself - offtopic here, better at Comparison of wiki software)
- (cur) (last) ○ ○ 17:09, 2 June 2008 Timhowardriley (Talk | contribs) (23,282 bytes) (→Software architecture - removed outdated wikilink)
- (cur) (last) ○ ○ 22:24, 1 June 2008 Ya Boi Krakerz (Talk | contribs) (23,297 bytes) (→External links)
- (cur) (last) ○ ○ 09:33, 28 May 2008 Greenrd (Talk | contribs) m (23,231 bytes) (reverting to more contemporary English usage)
- (cur) (last) ◉ 17:58, 27 May 2008 WorldlyWebster (Talk | contribs) m (23,250 bytes) (Capitalized "Web" where it is a proper noun or a proper noun serving as an adjective.)
- (cur) (last) ○ 23:42, 26 May 2008 Dirk Riehle (Talk | contribs) (23,231 bytes) (→Research communities)
- (cur) (last) ○ 23:42, 26 May 2008 Dirk Riehle (Talk | contribs) (23,231 bytes) (→Communities)

Fig. 1 the pages histories in wikipedia

3 Work Context

The work presented in this paper is based on the concurrency awareness mechanism and the results obtained from our work presented in [4]. The objective of the concurrency awareness mechanism is to recognize the server produced pages and the user produced pages and if the page is server produced to highlight the effects of concurrent updates inside these pages. This makes explicit the regions of the page that are subject to concurrency mismatches. Our major contribution, that we introduce here, to P2P wiki is a nonlinear representation to the page history that reflect the page state concerning the concurrency.

The mechanism is built over P2P wiki system called Wooki [16]. A Wooki network is dynamic p2p network where any site can join or leave at any time. Each site has a unique identifier named *siteid*. Site identifiers are totally ordered. Each site replicates wiki pages of other sites. A Wooki site generates a sequence of operations, when a wooki user saves his page. These operations are integrated locally on the wooki page by executing the Woot algorithm [9], and broadcasted to be integrated on all other sites.

WOOT ensures eventual consistency [6] and intentions preservation [13] for linear structures. Integrating a remote operation consists in line arising a dependency graph between the operations. The algorithm guarantees that the linearisation order is the same on all sites independently of the delivery order of patches. This allows to achieve eventual consistency.

In the concurrency awareness mechanism, when a patch is received and integrated in Wooki system the concurrency is checked and if the concurrency is detected then the mechanism analyzes the log file for determining the common version from which the concurrence operations must be presented to the users. We explain the mechanism by the following example, suppose that we have three site connecting to Wooki server and editing the same Wooki page. The sequence of the operation generation is shown in the figure 2. The three users got the page version Sn, at site 1 the user modifies the page by generating the patch P_{n1} and at site 2 the user modifies the page by the generation of P_{n2} and P_{n3}. These patches is propagated and integrated over all the connected sites of the network. At the reception of each patch the

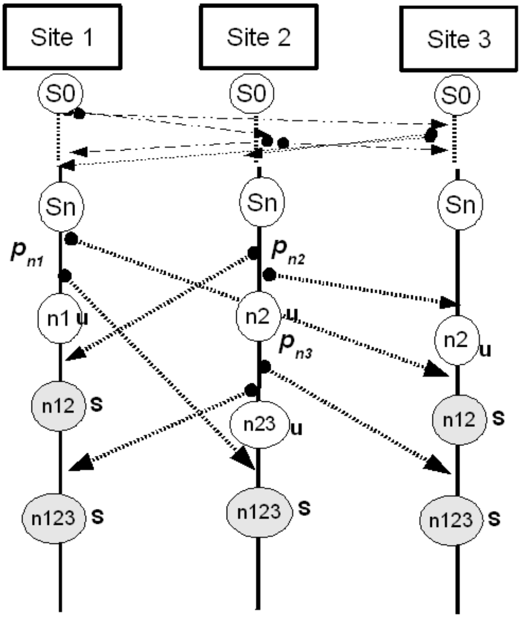


Fig. 2 the patch generation and integration over all sites

concurrency is detected. For example, at site 1, when p_{n2} is received, its integration will obviously require a merge with p_{n1}, resulting in the server-produced state n_{12}, labeled S in the figure, because p_{n1} is concurrent with p_{n2} . Any user requesting the page at that stage should be informed of this status. In addition, highlighting the page region impacted by patches p_{n1} and p_{n2} will help him understanding potential concurrency mismatches.

When site 2 produces patch p_{n2}, reaches the user-produced state n_2. Then site 2 produces patch p_{n3} and broadcasts it. The resulting state at site 2 is n_{23}, labeled u. When site 2, now, receives p_{n1}, we have $p_{n3} \parallel p_{n1}$[2], then the resulting state is server-produced page n_{123}. Here the log must be analyzed. The algorithm extract from the log the last applied patch. Then, it checks its concurrency with all patches in the log. Each concurrent patch is added to a set of concurrent patches. Then, the algorithm computes the common ancestor state of this set. Finally, it adds to the set all patches posterior to this state. So, at site 2, the last integrated patch is p_{n1} and we have $p_{n3} \parallel p_{n1}$, and $p_{n2} \parallel p_{n1}$ so this would return the set contained $\{p_{n3}, p_{n2}, p_{n1}\}$.

[2] we use the symbol || to denote the concurrency between patches

The common ancestor is Sn, this means that the integrated operations since Sn must be highlighted. At this stage, receiving any patch that is not concurrent with the last one will change the state from server-produced page state to user-produced page state. The situation is a bit different at site 3 since it does not produce any patch, but it just receives and integrates patch produced by site 1 and 2. When p_{n2} arrives, it is considered as non concurrent and the resulting state $n2$ is user-produced. When p_{n1} is received, it is considered as concurrent, and the resulting state $n12$ is server-produced. Here, $p_{n1} \parallel p_{n2}$ and the page region that potentially contains concurrency mismatch is the one impacted by $\{p_{n1}, p_{n2}\}$. Finally p_{n3} is received. It is also a concurrent patch and the resulting state $n123$ at site 3 is server-produced. In this mechanism, the user-produced page state means that the page is reviewed by someone and we consider it as valid, while the server-produced page state means that the page may contain some mismatch and it needs to be reviewed.

4 Representing the Histories in P2P wiki

The user might want to see the evolution of a wiki page: he wants to see when the page was user-produced and when it was server-produced, and what the patches that made the page server-produced and what are the patches that convert the page from server-produced to user-produced page. The user can also see who made these patches, where the patches were made, when they were made, why and from which version the concurrency is happened i.e the common ancestor. In this section we present a way to visualize the history with the concurrency information.

In P2P wiki, the history is not linear and some versions may be produced concurrently. We represent the history by a directed acyclic graph. The graph E consists of vertices and edges $E(V,E)$. Where the vertices (V) represent the page versions and the edges (E) represent the patches that convert one version into another. For our example in section 3, the history is shown in figure 3.

We will introduce some definitions related to the history before establishing our history representation.

- the history H at site S of page P, $H_s(P_i)$ is the set of revisions $v1 \ldots vn$ of the page P when integrated the patch i and the set of patches $1 \ldots i$ that convert one revision into another.
- when another patch $i+1$ is integrated the version history is $H_s(P_{i+1})$= $H_s(P_i)$+ $v(n+1)$.
- every page revision has a state PS this state may be user-produced or server-produced
- every applied patch i has a generation context GC_i. This context is to determine if this patch is locally generated or received from other sites, and the precedence and the concurrency relations with the other patches in the log file. These informations can be easily computed thanks to the concurrency detection mechanism. The generation context of pn_3, GC_{pn_3} is: pn_2 happened before pn_3 and $p_{n3} \parallel p_{n1}$

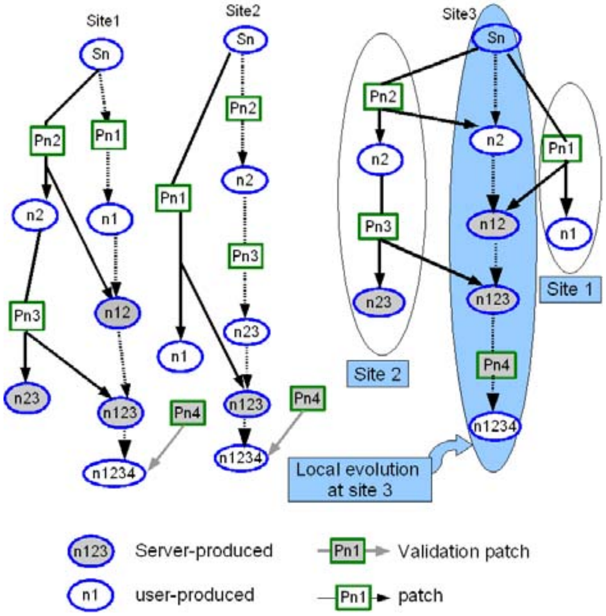
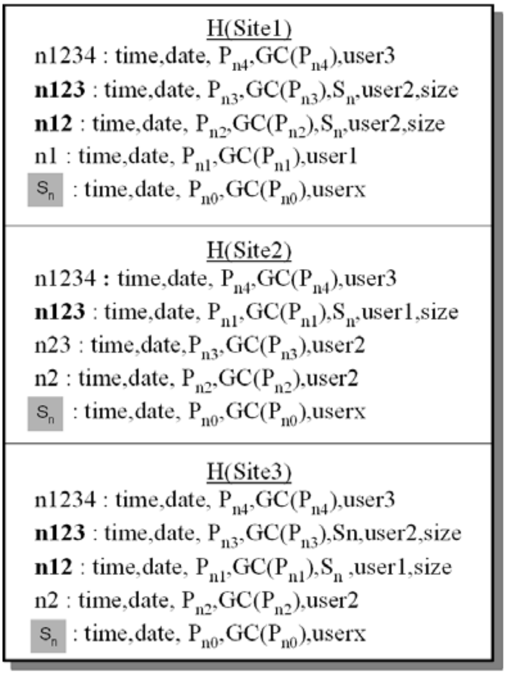


Fig. 3 `pages histories representation`

- the size of the concurrence: when the set of concurrent patches is extracted, in the case of server-produced page, the size of concurrence may be computed by counting the number of operations that are included in these patches. Because WOOT works at the line level then the size means how many lines are impacted by the concurrent operations. This size may give the user the impression that the version that has more concurrent operations has more priority to be reviewed.

In figure 3, we can see the evolution from *Sn*, suppose that we got this version by applying the patch P_{n0} made over sitex . *Sn* is a user-produced page and it is represented by a transparent vertex. At site 1 when P_{n1} is generated the state is stayed user-produces because this patch is not concurrent with the page state *Sn*. While when P_{n2} is received, the concurrency is checked and detected with the last patch, for this reason the page state is converted to server-produced with a shadowed vertex. The straight dotted line represents the page evolution over the local site while the other lines represent the patches and the states in the remote sites. For example at site 3, we show at the figure the local evolution and the remote integrated patches from site1 and site2. We distinguish the operation that transforms the server-produced



H(Site1)

n1234 : time,date, P_{n4},GC(P_{n4}),user3

n123 : time,date, P_{n3},GC(P_{n3}),S_n,user2,size

n12 : time,date, P_{n2},GC(P_{n2}),S_n,user2,size

n1 : time,date, P_{n1},GC(P_{n1}),user1

S_n : time,date, P_{n0},GC(P_{n0}),userx

H(Site2)

n1234 **:** time,date, P_{n4},GC(P_{n4}),user3

n123 : time,date, P_{n1},GC(P_{n1}),S_n,user1,size

n23 : time,date,P_{n3},GC(P_{n3}),user2

n2 : time,date, P_{n2},GC(P_{n2}),user2

S_n : time,date, P_{n0},GC(P_{n0}),userx

H(Site3)

n1234 : time,date, P_{n4},GC(P_{n4}),user3

n123 : time,date, P_{n3},GC(P_{n3}),Sn,user2,size

n12 : time,date, P_{n1},GC(P_{n1}),S_n ,user1,size

n2 : time,date, P_{n2},GC(P_{n2}),user2

S_n : time,date, P_{n0},GC(P_{n0}),userx

Fig. 4 the page history over the three sites

page to user-produced page by a shadow edge. At site 2, when *Pn*4 is integrated, it is not concurrent with any patch in the log and then it will change the state from server-produced to user-produced, it is represented as a shadow edge. The samething will happen sur site 1. Note that the state that have at least two outgoing edges is a common state while the state that is resulted from at least two incoming edges is a server-produced page.

Suppose that the user at site 3 validates the concurrency by generating the patch *Pn*4. This patch is considered as a validation patch even at site 1 and site 2 because it is not concurrent with last version state. When the user see this representation, he will understand that user 3 reviewed the page and he validated it by some operations in *Pn*4. The integration of this patch is shown at site 2 in the figure.

The question that must be asked now, is this representation is scalable i.e suitable for any number of sites in the network? This representation is suitable when the network size is very small but when we have a very huge network the graph will be complex and can't fit easily at the user's screen. For this reason, we adapted this representation to be seemed the current wiki history representation with some additional decorations for adding the concurrency informations. We represent it as shown in figure 4 and the function of each field is as follow:

1. the common ancestor has a shadowed background;
2. the server-produced versions in bold;
3. the user-produced versions appear normally;
4. the common ancestor version appears near the server-produced versions; when the user point to a server-produced version the versions until the common ancestor will be highlighted;
5. a link to the patch generation context is provided with every patch identifier;
6. we can indicate the size of concurrence in the page near the server-produced page. Being aware of the number of changes others users have made, helps the user to better understand the evolution of the page and easily collaborate with others to resolve the conflicts.
7. the date,time and the user name of the generation of each version. In addition, the case is like in the traditional wiki systems, the user can compare between any two versions but in this case the effects of concurrent modifications will be presented in a distinguish way.

We note that the versions order is not the same over sites. However, the patches that have a causal relation are integrated according to this relation. For example, over site 2, $Pn2$ is generated before $Pn3$, and therefore $Pn2$ must be integrated before $Pn3$ over all sites. When the system is stable i.e all the generated operations are propagated and integrated aver all sites, the final state and the common ancestor state are the same and set of integrated patches between these two states are also, the same over all sites. For example, at the page version $n123$ the system is stable, the page state $n123$ and the common state Sn are the same over all sites and the set of patches between Sn and $n123$ equal to $\{Pn1, Pn2, Pn3\}$.

5 Conclusion

we have presented here a non linear representation to the pages histories in P2P wiki systems. This representation is made ,firstly, by a directed acyclic graph and then it is made as the same form of the existing history representation in wiki for the scalability reasons. The objective of this representation is to show the page versions that are concurrently produced, and at any moment the user can compare between any two versions and see the operations that are integrated concurrently. The second step of this work is to evaluate this representation from the user point of view.

References

1. Open Source Software Engineering Tools. Online http://subversion.tigris.org/ (2006)
2. Wikipedia. The Free Encyclopædia that Anyone Can Edit. Online http://www.wikipedia.org/ (2006)
3. Allen, L., Fernandez, G., Kane, K., Leblang, D.B., Minard, D., Posner, J.: Clearcase multisite: Supporting geographically-distributed software development. In: Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management, pp. 194–214. Springer-Verlag, London, UK (1995)
4. Alshattnawi, S., Canals, G., Molli, P.: concurrency awareness in P2P wiki . In: The 2008 International Symposium on Collaborative Technologies and Systems (CTS 2008), pp. 285–294. IEEE (2008)
5. Berliner, B.: CVS II: Parallelizing software development. Proceedings of the USENIX Winter 1990 Technical Conference **341**, 352 (1990)
6. Johnson, P.R., Thomas, R.H.: RFC677: The maintenance of duplicate databases (1976)
7. Lippe, E., van Oosterom, N.: Operation-based merging. SIGSOFT Softw. Eng. Notes **17**(5), 78–87 (1992). DOI http://doi.acm.org/10.1145/142882.143753
8. Molli, P., Skaf-Molli, H., Bouthier, C.: State treemap: an awareness widget for multi-synchronous groupware. In: 7th International Workshop on Groupware - CRIWG'2001. Darmstadt, Germany (2001)
9. Oster, G., Urso, P., Molli, P., Imine, A.: Data consistency for p2p collaborative editing. In: Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006, Banff, Alberta, Canada, November 4-8, 2006. ACM (2006)
10. Robbes, R., Lanza, M.: Versioning Systems for Evolution Research. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution, (IWPSE '05), pp. 155–164. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/IWPSE.2005.32
11. Sabel, M.: Structuring wiki revision history. In: Proceedings of the 2007 international symposium on Wikis, (WikiSym '07), pp. 125–130. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1296951.1296965
12. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys (CSUR) **37**(1), 42–81 (2005)
13. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Transactions on Computer-Human Interaction (TOCHI) **5**(1), 63–108 (1998)
14. Tichy, W.: RCS - A System for Version Control. Software - Practice and Experience **15**(7), 637–654 (1985)
15. Viégas, F.B., Wattenberg, M., Dave, K.: Studying cooperation and conflict between authors with history flow visualizations. In: Proceedings of the 2004 conference on Human factors in computing systems, (CHI '04), pp. 575–582. ACM Press (2004). DOI 10.1145/985692.985765. URL http://portal.acm.org/citation.cfm?id=985765
16. Weiss, S., Urso, P., Molli, P.: Wooki: a p2p wiki-based collaborative writing tool. In: Web Information Systems Engineering. Springer, Nancy, France (2007)