INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Concurrency Awareness in a P2P Wiki System

Sawsan Alshattnawi — Gérôme Canals — Pascal Molli

N° 6425

January 2008

Thème COG

ISSN 0249-6399 ISRN INRIA/RR--6425--FR+ENG

INRIA
LORRAINE

Concurrency Awareness in a P2P Wiki System

Sawsan Alshattnawi*, Gérôme Canals †, Pascal Molli ‡

Thème COG — Systèmes cognitifs
Projet ECOO

Rapport de recherche n° 6425 — January 2008 — 18 pages

Abstract: Currently, Wikis are the most popular form of collaborative editors. Recently, some researches have been done to shift from traditional centralized architecture to fully decentralized wikis relying on peer-to-peer networks. This architecture improves scalability and fault-tolerance, but subtly changes the behavior of wiki in case of concurrent changes. While traditional wikis ensure that all wiki pages have been reviewed by, at least, a human, some pages in P2P wiki systems can be the result of an automatic merge done by the system. This forces P2P wiki systems to integrate a concurrency awareness system to notify users about the status of wiki pages. The particular context of a P2P wiki system makes traditional awareness mechanisms inadequate. In this paper, we present a new concurrency awareness mechanism designed for P2P wiki systems.

Key-words: Awareness in Collaborative Systems, Architectures and Design of Collaborative Systems, Platforms for Collaboration, Web Infrastructure for Collaborative Applications, Web- and Internet-Enabled Collaboration

* alshattn@loria.fr, ECOO Project, Nancy-University, LORIA, INRIA Centre - Nancy Grand Est
† gerome.canals@loria.fr, ECOO Project, Nancy-University, LORIA, INRIA Centre - Nancy Grand Est
‡ molli@loria.fr, ECOO Project, Nancy-University, LORIA, INRIA Centre - Nancy Grand Est

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

la conscience de la concurrence dans un système de Wiki sur un réseau Pair à Pair

Résumé : Actuellement, les wikis ont rendu les éditeurs collaboratifs très populaires. Récemment, plusieurs travaux ont proposé de remplacer l'architecture centralisée traditionnelle par une architecture complètement décentralisé sur un réseau pair-à-pair. Une telle architecture améliore le passage à l'échelle et la tolérance aux pannes. Ce pendant, elle conduit à traiter différemment les modifications concurrentes. Alors que dans un wiki classique les changements concurrents sont fusionnés sous la conduite d'un utilisateur, ils sont réalisés automatiquement par le serveur dans un wiki P2P. Certains pages peuvent ainsi être accessible aux utilisateurs alors que leur contenu n'a jamais été contrôlé par ses auteurs. Pour faire face à ce problème, nous proposons d'intégrer au wiki P2P un mécanisme de conscience de changements concurrents. Ce mécanisme détecte les pages produites automatiquement et indique aux utilisateurs les zones dans ces pages qui proviennent de la fusion de modifications concurrents. Ce mécanisme est construit pour respecter les contraintes d'échelle et d'autonomie liées au contexte P2P.

Mots-clés : Travail collaboratif, wiki P2P , modifications concurrentes , Conscience de groupe

Contents

1 Introduction

Currently, Wikis are the most popular form of collaborative editors. They allow users connected to the web to concurrently edit and modify a shared set of wiki pages. Current wikis are built over a centralized architecture: the whole set of pages reside on a single server that controls all operations of distributed participants.

Recently, some researches have been done to shift from this traditional centralized architecture to fully decentralized wiki relying on peer-to-peer networks [21, 19, 31]. Expected benefits of this new approach are scalability, better performance, fault-tolerance, infrastructure cost sharing, self-organization, better support to nomad users and resistance to censorship [4].

A P2P Wiki is composed of a P2P network of autonomous wiki servers. Each wiki page is replicated over the whole set of server. A change performed on a wiki server is immediately applied to the local copy and then propagated to the other sites. A remote change, when received by a server, is merged with local changes and then applied to the local copy. The P2P wiki system is correct if it ensures eventual consistency i.e. the system is correct if all copies are identical when the system is idle [15, 23] and intentions are preserved [24].

Our objective it to host a site like Wikipedia on a P2P wiki. Wikipedia [2] has currently collected more than 9,000,000 articles in more than 250 languages. Wikipedia has more than 13 millions of page requests per day. 200,000 changes are made every day, with however a total mean number of edit per article actually less than 40 [1]. Actually, Wikipedia needs a costly infrastructure to handle the load. Hundreds of thousands of dollars are spent every year to fund this infrastructure. A P2P wiki system would allow to distribute the service, tolerate failures, improve performances, resist to censorship and share the cost of the underlying infrastructure. We envisage large scale overlay networks, i.e. up to hundred of thousands of wiki servers. In our opinion, this will not change the number of edit and/or contributors per article and for each article, the number of sites that will produce changes is only a small subset of the overlay. Actually, a very large majority of wikipedia articles has less than 1000 contributors.

The behavior of a P2P wiki differs from traditional wikis when concurrent changes are occurring. In a traditional wiki such as MediaWiki, two users (say u_1 and u_2) can edit concurrently the same page. The first user that finishes (e.g. u_1) saves his change as usually. This creates a new version of the page. When user u_2 saves his change, the server detects a concurrent change and asks user u_2 to manually merge his change with the concurrent change from user u_1. When the merge is done, user u_2 saves and a new version of the page is produced. The merge process of concurrent changes is always under the control of a user. Thus, *all* versions of a wiki page are produced by a user and have been reviewed by this user before they were created.

In a P2P wiki, concurrent changes may occur and be saved on different servers.

Merges are not executed when pages are saved but when remote changes are received by each sites. To ensure eventual consistency, merges are fully automatic and performed by wiki servers. Therefore, it can happen that the current visible wiki page has been produced by the wiki server and its content can be meaningless. This is a serious problem for the credibility of the wiki system.

This problem is illustrated in figure 1 where two sites connected through P2P network concurrently update a wiki page. The initial state of the page is the simple string *"A snake is a mammal"*. Two users access this page on the two servers and update it to correct it. At site 1, a user replace "snake" by "cat". This change is applied to the local copy of the page and produces the version *"A cat is a mammal"* which is considered correct by this user. At the same time, a user connected to site 2 replaces "mammal" by "reptile" producing the version *"A snake is a reptile"* which is considered as correct by this user.

Then both changes are propagated to the network and reach site 1 and site 2. At both sites the servers merge these concurrent changes and update the local copies of the page, producing a new version in which both changes are integrated. Both copies are then consistent according to eventual consistency criterion and have the same value: *"A cat is a reptile"*. Note that the two users that edited the page are not aware of this new value. A third user accessing this page later on any of the two servers will then get a page version that would probably be considered as meaningless by its authors themselves.

To overcome this problem, we introduce *concurrency awareness*. Concurrency awareness is a mechanism that makes users aware of the status of the pages they access regarding concurrency: is the page *server-produced*, i.e. resulting from an automatic merge, or *user-produced* ? In addition, concurrency awareness indicates which region of the page has been merged automatically.

In our scenario, both sites should converge to a state where "cat" and "reptile" must be marked as concurrent changes. This makes the third user aware about the fact that these two changes have been made concurrently on this page and that what he is currently reading have not been reviewed by a human and may be meaningless.

Many awareness frameworks have been developed in the CSCW community [9]. Most of them have been designed for synchronous groupware [10] are not adequate in our purely asynchronous context. Some asyn-

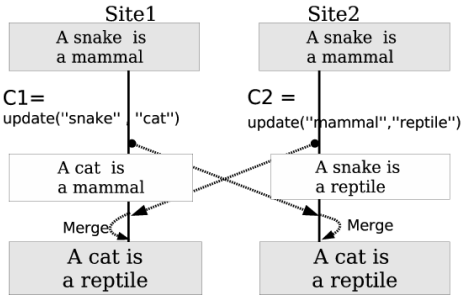

Figure 1: Automatic merge problem

chronous groupware systems [11, 20] or Version Control Systems [29] handle concurrent updates and are able to display concurrent and conflicting changes. However, we have some constraints in our context that make most existing approaches inadequate. We need an awareness system that fit the requirements of a P2P network. We cannot use a central server for delivering awareness. We must also be sure that the awareness system has no impact on the replication system and generates no violation of eventual consistency or intentions. Finally, we have to be sure that the complexity in time and space of our awareness algorithm is independant of the number of sites. Currently, no systems fit all these constraints.

In a traditional asynchronous collaborative system a workspace is owned by one user and any update of this workspace is issued, monitored and controlled by this user. In a P2P wiki system, the workspace is managed by a wiki server and an update is performed by the server when remote changes are available. In a classical system, when a user updates his workspace, he is aware that some concurrent changes can be integrated. The system delivers awareness during the update process and users can perform additional requests to the system to get more informations about what has been done during the merge. In a P2P wiki system, any user can request any page at any time and concurrency awareness has to be delivered at this time. The fact that the workspace is managed by the wiki server is a big context change for awareness systems.

For these reasons, we have developed a new awareness mechanism designed for P2P wiki constraints. It is fully asynchronous, it requires no central server, it delivers the same awareness on each site if the system is idle, it requires no workspace for wiki users. This paper introduces and describes this concurrency awareness mechanism for a P2P wiki. This paper is organized as follows: section 2 gives an outline of our P2P wiki. Section 3 presents our concurrency awareness mechanism and its implementation. Section 4 is about editing server produced pages that contain awareness information. Finally, sections 5 and 6 discuss the related work and conclude.

2 Work context

This section gives more details about our approach to build a fully decentralized, P2P wiki system. We first briefly present the overall architecture of our system, called Wooki [31], and then discuss consistency maintenance and concurrent histories.

2.1 Wooki: a P2P Wiki

A Wooki system is a set of interconnected wiki servers that forms a P2P overlay network. In this overlay, each server plays the same role. Like in any P2P network, membership is dynamic. Wooki servers can join or leave the network at any time. Wiki pages are replicated over all the members of the overlay. Each server then hosts a copy of the pages and can autonomously offer the wiki service. Page copies at each site are maintained by an optimistic replication mechanism that disseminates changes and ensures consistency.

Expected benefits of this architecture are the classical ones of P2P systems. Wooki is more scalable than a centralized one thanks to massive data replication and multiple servers. Wooki is also more fault tolerant. In case of a server crash, any request can be adressed to any other server of the netwok. Thanks to its dynamic membership aspect, Wooki offers a support to nomadic users. A user that needs to disconnect from the network can create a replica of the wiki pages and start a wiki server on his workstation. He can then access and edit the wiki while being disconnected. When he reconnects, the replication mechanism will propagate his changes to the network.

Wooki is built over a fully decentralized architecture with no master server and no master or reference copy. Viewing, editing a page and saving a modification can be done at any server. A modification at one server is immediately applied to the local replica and then broadcasted to the other servers. A remote modification, when received by a server, is integrated to the local replica. If needed, the integration process merge this modification with concurrent modifications, either local or received from a remote server.

As many classical wikis, wooki represents a change introduced by a user during an edit sesion as *patches*. A patch is a delta between two successive versions of a wiki page and contains the sequence of elementary operations required to transform one version into another. Patches are computed when a user saves his modifications by applying a *diff* algorithm between the new version of the page and its predecessor version. A patch represents a change to one single page. Patches are sequentially numbered by the server and receive a unique identifier formed by the pair $< Site_{id} , Patch_{no} >$. Patches are the Wooki replication unit: a Wooki server disseminates locally produced patches to the network and integrates remote patches to its local replica.

2.2 Page replication in the Wooki system

A Wooki P2P wiki server is a wiki server enhanced with an optimistic replication system composed of two main components: an integration component that applies patches to local wiki pages and merge concurrent patches if needed, and a dissemination component that broadcasts patches to the P2P network and receives patches from remote servers.

The Wooki integration mechanism is based on the Woot algorithm [21]. WOOT ensures eventual consistency [15] and intentions preservation [25] 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 indepently of the delivery order of patches. This allows to achieve eventual convergence.

A wooki page is a sequence of lines and the integration algorithm works at the line level. When a line is inserted in a page, it receives a unique and non mutable identifier. Woot never destroies lines in its stored data. The Woot *Delete* operation consists just in marking as *not visible* the deleted line. Although this can be considered as a drawback in the general case, it is acceptable in the context of a wiki system which keeps track of all the successive versions of all the wiki pages it hosts. Finally, a Woot integration returns always a linear history, i.e. a free of conflict history, in which *all* operations are kept. Of course, the algorithm works at the textual level. It is far from guaranteeing the meaningfulness of a merge result. Even if the merge result is free of conflict, it need to be checked by a human reader.

The dissemination component is in charge of broadcasting patches over the overlay network. Given that patches are non-mutable objects, a classical P2P approach for disseminating data can be used. The component must however offer guarantees about the delivery of all patches to all servers, including temporarily disconnected servers. Wooki uses a gossip protocol [28] to broadcast to connected servers. Disconnected servers that reconnect need then to run an anti-entropy protocol [7] to synchronize their copy with another server by obtaining the patches they missed while disconnected. The dissemination protocol offers a unique guarantee on the delivery order of patches: all patches originating from the same site are delivered at all sites in the order they were produced (FIFO site to site).

3 Delivering Concurrency awareness

As introduced in section 1, a wooki page can be either a *user-produced page* or a *server-produced page*. While user produced pages are produced under the control of a user, server produced pages, that results from an automatic merge, are produced out of the control of a user and may content mismatches due to concurrency.

To overcome this problem, we introduce *concurrency awareness*. Concurrency awareness is a mechanism in charge of recognizing server produced pages and of highlighting the effects of concurrent updates inside these pages. This makes explicit the regions of the page that are subject to concurrency mismatches.

Concurrency awareness is activated each time a user requests a page to view it, either he contributed to the page or not.

Our awareness mechanism is made of two components :

- a log analyzer, in charge of detecting server produced states and of computing the page regions that need to be highlighted. The log analyser works on the patch history of a page,

- an awareness visualization tool, in charge of adding concurrency awareness information to a server produced wiki page. This tool adds decorations to a wiki page at the time it is extracted from the storage system.

Before to go in more details about these components, we introduce some definitions related to concurrent histories in order to clearly establish th role of our awareness mechanism.

3.1 Concurrent histories in Wooki

Figure 2 illustrates a concurrent history that can happen in the Wooki system. In this example, three Wooki servers are connected. Each one hosts a copy of a wiki page. At the beginning of the scenario, the state of the page is a user produced state Sn (white circle labeled u). At site 1, patch p_{n1} is produced. This is of course a non-concurrent patch. The resulting state n_1 is labeled u. The patch is then broadcasted to the network. At the same time, site 2 produces patch p_{n2}, reaches the user-produced state n_2 and broadcasts p_{n2}. Then site 2 produces patch p_{n3} and broadcasts it. The resulting state at site 2 is n_{23}, labeled u.

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. Any user requesting the page at that stage should be informed of this status. In addition, higlighting the page region impacted by patches p_{n1} and p_{n2} will help him understanding potential concurrency mismatches.

A simular situation occurs upon reception of p_{n3} at site 1, resulting in the server produced state n_{123}. At that stage, the page region that should be highligthted is the one impacted by patches p_{n1}, p_{n2} and p_{n3}. Indeed, p_{n1} is concurrent to p_{n2} and to p_{n3}.

We now introduce some definitions about page states and patch concurrency. Part of these definitions are taken from [26].

Recall that Woot has no requirement on the delivery ordering of patches. Therefore, the state of a page results only from the *set* of patches applied to that page, regardless of the order they were received and applied.

Definition 1 *Page State*

A page state PS is defined by the set of patches applied to the page and is defined by:

1. The initial page state is $PS = \{\}$.
2. A patch P applied to a page transforms its state PS to $PS' = PS \bigcup \{P\}$.

Based on this definition of a page state, we can define the generation context of a patch, which captures the state on which the patch was produced, and the resulting state.

Definition 2 *Patch Generation Context*

For a patch P, its generation context $GC(P)$ is $GC(P) = PS \bigcup \{P\}$, where PS is the page state from which P was produced (i.e. the state of the wiki page at the time a user requested its edition).

The notion of patch generation context allows the introduction of a precedence relation between patches that captures causality.

Definition 3 *Patch precedence* $\rightarrow$

Given a patch P_i and a patch P_j, P_i precedes P_j, $P_i \rightarrow P_j \; iff \; P_i \neq P_j$ and $GC(P_i) \subset GC(P_j)$.

Applied to our example, the above definitions give us:

- $GC(p_{n1}) = \{p_0, \ldots, p_n, p_{n1}\}$,
- $GC(p_{n2}) = \{p_0, \ldots, p_n, p_{n2}\}$,
- $GC(p_{n3}) = \{p_0, \ldots, p_n, p_{n2}, p_{n3}\}$,

from which we deduce that $p_{n2} \rightarrow p_{n3}$.

We can now define the notion of concurrency between patches. As usually, two patches are said to be concurrent (or conflicting) if they are not causally related.

Definition 4 *Patch concurrency* $\|$

Given two patches P_i and P_j, P_i is concurrent to P_j, $P_i \parallel P_j, iff$ neither $P_i \rightarrow P_j$ nor $P_j \rightarrow P_i$.
In our example, we have $p_{n1} \parallel p_{n2}$ and $p_{n1} \parallel p_{n3}$.

For practical reasons, we extend this definition of patch concurrency to introduce the idea of a patch being concurrent to a page state. A patch is concurrent to a page state if this page state is not included in the generation context of the page.

Definition 5 *State/patch concurrency* $\parallel$

Given a patch P_i and a page state PS, P_i is said to be concurrent to PS, iff $PS \not\subset GC(P_i)$

On the contrary, a patch is not concurrent to a state if this state is included in the generation context of the patch. This captures the idea that this state has been viewed when the patch is generated. From this definition of state/patch concurrency, we can now give a precise specification of a *server-produced page*,and of a *user-produced page*.

Definition 6 *Server Produced page*

A wooki page is said to be server produced iff its actual state PS results from the application of a patch P to its previous state PS' and $PS' \parallel P$.

Definition 7 *User Produced page*

A wooki page is said to be user produced iff its actual state PS results from the application of a patch P to its previous state PS' and $PS' \nparallel P$.

Returning to our example, we can examine the situation at site 2 and site 3. At site 2, p_{n2} and p_{n3} are local and sequentially produced. They are obviously non concurrent, and state $n23$ is user produced because $n_2 \nparallel p_{n3}$. The situation is a bit different at site 3 since it does not produce any patch, but 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. Indeed, $Sn = \{p_0, \ldots, p_n\}$, $GC(p_{n2}) = \{p_0, \ldots, p_n, p_{n2}\}$, and $Sn \subset GC(p_{n2})$. When p_{n1} is received, it is considered as concurrent, and the resulting state $n12$ is server-produced. This comes from $n2 = \{p_0, \ldots, p_n, p_{n2}\}$ while $GC(p_{n1}) = \{p_0, \ldots, p_n, p_{n2}\}$. 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.

At this stage, we have $p_{n3} \parallel p_{n1}$, but $p_{n3} \nparallel p_{n2}$. However, we think that the page region that should be highlighted is the one impacted these three patches. Before to describe how we compute this set of patches, it is worth noting that after the delivery of these three ptches, we have an identical state at each site - this is guaranteed by Woot - labeled as server produced at each site. However, the state sequence is different at each site because the patch history is different. Consequently, the sequence of status - user or server produced- is also different at each site.

Let's now discuss how to compute the set of patches whose effects will be highlighted by the concurrency awareness mechanism. From the above discussion, it is clear that just computing the set of patches that are concurrent to the last integrated one is not enough. In addition, this would not return the same result at eache site. Indeed:

- at site 1, the last integrated patch is p_{n3} and we only have $p_{n3} \parallel p_{n1}$, so this would return $\{p_{n3}, p_{n1}\}$ but not p_{n2},
- at site 3, it is exactly the same,
- 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 $\{p_{n3}, p_{n2}, p_{n1}\}$.

Our approach is the following. When a patch P is applied to a state PS, resulting into state PS', we extract from the history of PS' all patches posterior (in the sense of patch precedence) to the state that has been viewed by both P and all its concurrent patches in PS. This state is the common ancestor, i.e. the state from which P and all its concurrent patches derive. This state can be computed by the intersection of the generation context of P with the generation context of its concurrent patches.

Definition 8 *Concurrent History of a patch P in a state PS*

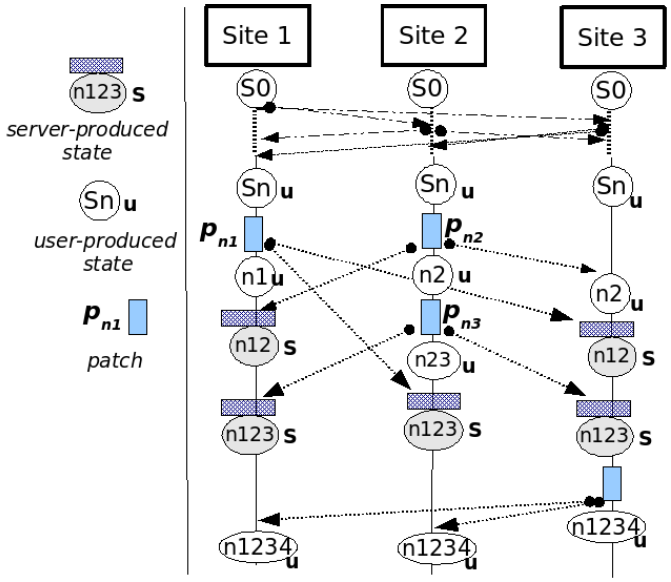

Figure 2: A Concurrent history in a P2P wiki

The Concurrent History of a patch P applied to a state PS, noted $CH(PS, P)$, is defined as $CH(PS, P) = PS \bigcup \{P\} - \bigcap GC(P), GC(p_i)$, where $p_i \in PS$ and $p_i \parallel P$.

Note that if $PS \not\parallel P$, then $CH(PS, P) = \{\}$. The concurrent history is not empty only if the applied patch is concurrent to the actual state of the page.

In our example, we now have:

- at site 1, when p_{n3} is integrated, we have $CH(n12, p_{n3}) = \{p_{n1}, p_{n2}, p_{n3}\}$. Indeed, $p_{n3} \parallel p_{n1}$, and $GC(p_{n3}) = \{p_0, \ldots p_n, p_{n2}, p_{n3}\}, GC(p_{n1}) = \{p_0, \ldots p_n, p_{n1}\}$, thus $GC(p_{n3}) \bigcap GC(p_{n1}) = \{p_0, \ldots p_n\}$.
- at site 3, the same occurs,
- at site 2, when p_{n1} is integrated, we now have $CH(n12, p_{n3}) = \{p_{n1}, p_{n2}, p_{n3}\}$.

Finally, a user at site 3 requests the page, finds it as server-produced and wants to solve some mismatches due to concurrency. He does it by editing the page and producing a new patch p_{n4}. This patch is obviously not concurrent to current state $n123$, since $GC(p_{n4}) = n123 \bigcup \{p_{n4}\}$. The page status at site 3 is changed to user-produced.

At the other sites, if no other modification of the page occurs concurrently, patch p_{n4} is also recognized as non concurrent to state $n123$ when integrated. The resulting state, $n1234$ is thus labeled as user produced at all sites.

This is the way a user reviewing the page can correct a concurrency mismatch. In some cases, no modification is required: although being server produced, the page content is considered correct by the user. In this case, we simply generate and broadcast an empty patch that will change the page status but not its content.

To conclude on this example, it is important to note that the concurrent history is only dependent of the patch and the state on which it is applied, but is independant of the site. In the final situation of our example, the three sites store the same state, and the concurrent history computed at this stage will be the same on the three sites.

3.2 The Log analyzer

The log analyzer analyzes patch logs to determine the status of a page and to extract the concurrent part of a page history. This component is present at each wooki server. It locally computes page status and concurrent histories for the local replica.

The component provides two operations:

- *computeConcStatus*$(State, Patch)$ is called each time a patch, either local or remote, is integrated to a page. This operation compute the concurrency status of the page and stores it.

- *getConcurrentHistory*(*PatchLog*) is called when a server produced page is requested. This operation returns the concurrent part of the patch log.

Since the Wooki engine does not use any ordering mechanism, either for merging or for broadcasting patches, the log analyzer needs to implement a mechanism for concurrency detection. We introduce this concurrency detection mechanism before presenting algorithms for the two operations.

3.2.1 Implementing concurrency detection

Detecting concurrency in a large scale system is a difficult problem. Charron-Bost [6] showed that causality can only be captured completely with a mechanism that would have a size $O(N)$ where N is the number of sites. This clearly do not scale, and means that a scalable mechanism need to be based on a trade-of between acuracy and size.

A straightforward implementation of concurrency detection from definitions given in 3.1 can be the following:

1. each site logs the patches it successively integrates to the local pages. A site maintains a patch log per page.
2. each patch is labelled and disseminated with its generation context. Of course, this context can contain only patch identifiers,
3. state/patch concurrency and concurrent histories are computed from basic set operations.

This implementation is similar to hash histories [16], but is not scalable because of the unbounded number of patches that can grow infinitely. This renders the approach inappropriate for our P2P context.

Our implementation is based on *R-entries patch vectors* [30]. A patch vector is a concise representation of a set of patches, where patches are grouped by their origine site. Thanks to the FIFO ordering of patches for site-to-site communication, it is sufficient to keep the identifier of the last received patch for each site that generated at least one patch. A patch vector is thus a set of pairs $< S_{id}, P_{id} >$, where S_{id} is a site identifier, and P_{id} identifies the last patch received from S_{id}. A patch vector can be used to represent the generation context of a patch and the state of a page. By labelling states and patches with their corresponding patch vectors and using the classical rules to update and compare vector clocks, it is possible to detect state/patch and patch/patch concurrency. A R-entry patch vector is a patch vector with s a bounded number R of entries. If the system contains more that R sites that produce patches, then multiple sites may share the same entry in the vector. Assignment of sites to entries is just done by a *modulo R* function: site i is assigned to entry e where $e = i \ modulo \ R$.

For a system containing N sites producing patches, if $N \leq R$, a R-entry vector behaves exactly like a vector clock and exactly captures all causal dependencies in the system. If $N > R$, it is shown in [30] that a $R - entry \ Vector$ is a plausible clock. This means that:

- all causal dependencies are captured by the clock,
- the clock does not detect false conflicts,
- it may happens that a conflict in the system is not detected by the clock, but reported as a causal relationship.

Authors report an evaluation on the clock that shows an average error rate of 0.20 for 100 sites and a 3-entries vector. Obviously, choosing the size of the vector is important part of the trade-off. The size may of course depend on the wiki usage. If we consider Wikipedia, a 1000-entry vector will behave like a vector clock for the very large majority of articles and will support exceptional cases, e.g. articles with up to 10.000 contributing sites, with an error rate remaining quite low.

In our context, using a $R-entry \ vector$ based concurrency detector means that some server-produced states will not be detected, and that some concurrent histories may be incomplete. More concretely, the system may in some cases not deliver awareness information that would be useful, but will never introduce unuseful awareness information. Provided the number of non detected conflicts remains low, we consider that this is a acceptable trade-off.

The concurrency detection mechanism offers two primitives:

- isConcurrent(V1, V2):Boolean ; check the concurrency by comparing the two vectors using classical vector clock rules. Two identical vectors are considered to be concurrent.

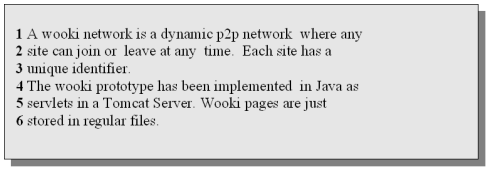

Figure 3: the initial page state

- getCommonAncestor($\{V_i\}$):Vector ; compute the common ancestor of a set of vectors as defined in 3.1. This consists in retaining the min value in the set for each entry.

3.2.2 The algorithms

The log analyzer algorithms are based on a R-entry vector concurrency detector. Page and patches are labelled with a R-entry vector.

The *computeConcStatus*() operation is very simple :

```
computeConcStatus(State, P)
PageId = P.getPageId()
Page = State.getPageById(PageId)
if isConcurrent(Page.getVector(), P.getVector()) then
  Page.setConcStatus (ServerProduced)
else
  Page.setConcStatus (UserProduced)
end if
```

The *getConcurrentHistory()* is a bit more complex. 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 the result set. Then, the algorithm computes the common ancestor state of this set. Finally, it adds to the result set all patches posterior to this state.

```
getConcurrentHistory( Log )
Patch lp = Log.getLastPatch()
ResultSet = {lp}
for all patch P_i ∈ Log do
  if isConcurrent(lp.getVector(),P_i.getVector()) then
    ResultSet = ResultSet ⋃{P_i}
  end if
end for
Ac = getCommonAncestor(ResultSet)
for all patches P_j ∈ Log, Ac.getVector() ← P_j.getVector() do
  resultSet = ⋃{P_j}
end for
return( ResultSet )
```

3.3 Visualization of Concurrent Modifications

To illustrate our awareness visualization tool, we use an example based on the concurrent history from the previous section. The initial state of the wiki page is given in figure 3.

From this initial state, a user connected at site 1 inserts two lines between line 3 and line 4 and saves. This modification corresponds to patch p_{n1} in our example history. At the same time, a user connected at site 2 updates line 3 and saves, producing patch p_{n2}. Later, the same user at the same site deletes lines 4 to 6 and saves, producing patch p_{n3}. These modifications are presented in figure 4. Assume that patches exchange occurs in the same order that in figure 2.

Our awareness visualization mechanism delivers awareness information to the user by highlighting the effects of the concurrent part of the history in the page it returns when the requested wooki page is server produced.

When the server receives a $GET(pageId)$ request from a user, the text content of the corresponding page is extracted from the page storage and passed to the html renderer. This renderer transforms the wiki syntax into HTML. If the page is a server produced page, this process is affected by the awareness mechanism: it inserts

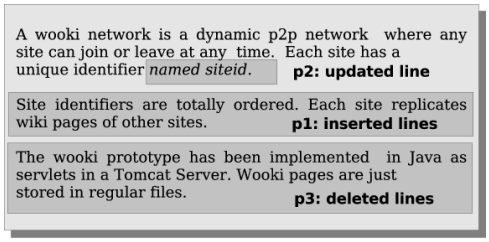

Figure 4: the operations executed over the page wiki

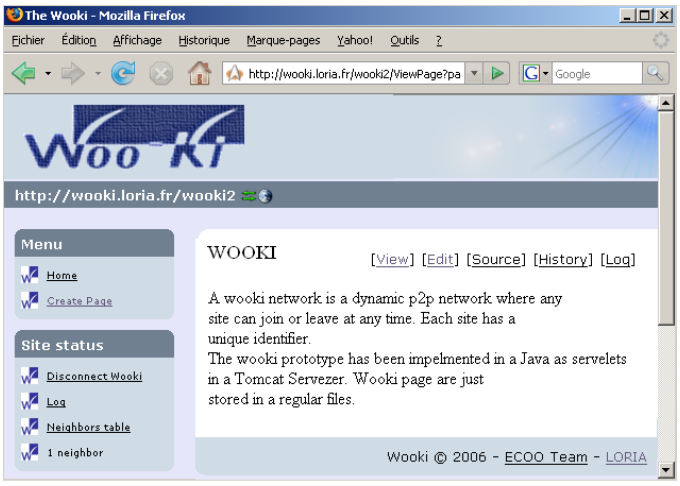

Figure 5: the initial page content

awareness directives into the text content of the page during the extraction. These directives are computed from the concurrent part of the page history. A decorated wiki page is thus passed to the html renderer that interprets these directives to produce HTML code accordingly.

The approach we adopt to highlight the effects of concurrent patches is the following:

- non affected lines appear normally, without any visual modification,
- deleted lines are kept visible in the document, but marked as deleted with an overriding thin line,
- inserted lines are colored,
- updated lines appear with the old value as deleted and the new value as inserted,

Figure 5 depicts the wooki system appearence when a user requested a user produced page corresponding to the initial state of our example. This page appears as a regular wiki page.

Figure 6 illustrates what the user get when he requests the example page at site 1, after the integration of patch p_{n2}. At this stage, the page is server-produced and the concurrent history is $\{p_{n1},p_{n2}\}$. The status of the page is marked with a flag on the page title. Effects of all the edit operations appearing in these two patches are highlighted.

More concretely in this case, line 3 has been updated by site 2 (patch p_{n2}). So line 3 appears two times: the first occurrence corresponds to the old value and appears overrided with a thin line, while the second occurrence corresponding to the new value appears with a colored background. The two lines that have been inserted at site 1 (patch $pn1$). These lines appear also with a colored background. The other lines, i.e. lines 1 and 2 and the last paragraph appear normally since they are not impacted by the concurrent history at this stage.

Figure 7 illustrates what a user will get if he requests the page at site 2, after the integration of patches $pn2$ and p_{n3}. Note that this page is user produced. Thus, no concurrency awareness is provided. Line 3 is updated, and the last paragraph is deleted.

Finally, figure 8 illustrates the page the user gets at the same site 2 after the integration of the concurrent patch p_{n1} issued from site 1. At this stage, the concurrent history is $\{p_{n1},p_{n2},p_{n3}\}$. All the effects of these 3

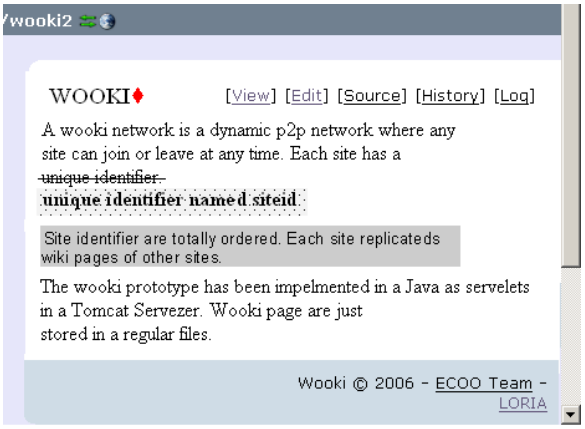

Figure 6: Visualizing a server produced page: case 1

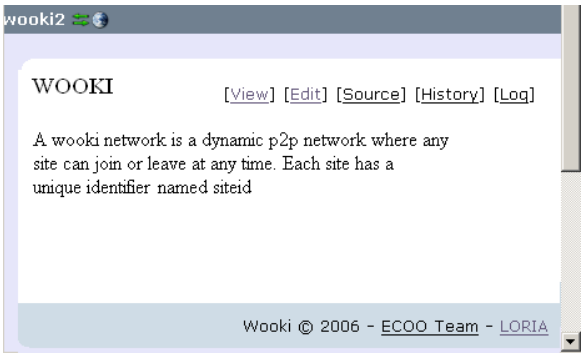

Figure 7: Visualizing a server produced page: case 2

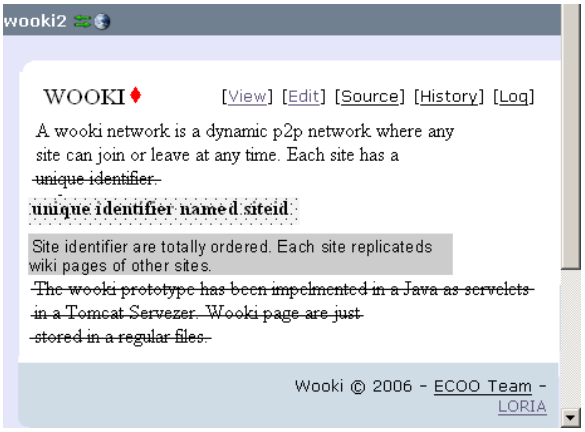

Figure 8: Visualizing a server produced page: case 3

patches are highlighted. The effects of patches p_{n1} and p_{n2} are highlighted as in figure 6. In addition, the effects of patch p_{n3} are now visible: the deleted lines appears again in the page, but overrided with a thin line.

4 Editing server produced pages

Server produced pages can of course be edited as any regular page. In many cases, a server produced page is edited in order to correct a concurrency mismatch. To help users in this task, awareness directives are kept in the edit area. This text area is a standard wiki edit area: the user creates and modify content using a regular wiki syntax.

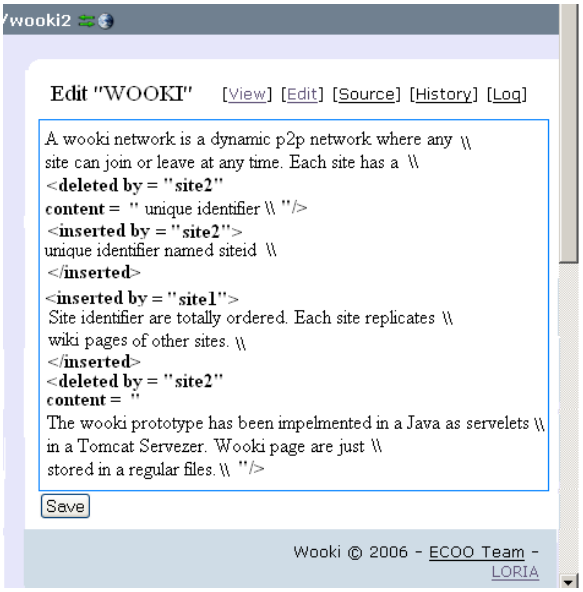

Figure 9: the edit text area for the example

When the server receives an $EDIT(pageId)$ request, it computes the annotated wiki page as it does for a GET request. This wiki page is stored, and returned to the user in its text area.

At the save time, i.e. when the server receives a $SAVE(pageId)$ request, the new value and the stored value of the page are both filtered to remove any awareness directive. Then, the difference between the new and the stored value is computed to generate the corresponding patch.

Awareness directives are in fact special tags inserted in the text content of the page. Deleted lines appear as the value of the CONTENT attribute of the <deleted> tag. Inserted lines are encapsulated by a <inserted> </inserted> pair. Filtering the page content consists in removing all these tags.

Figure 9 illustrates the text area for a user editing the wiki page of our current example, in a state where all three patches where integrated. This could happen at either site 1, site 2 or site 3. Imagine this happen at site 3, when a third user reviewing the page wants to correct it.

To undo the deletion of the last paragraph, he just needs to remove the corresponding $< delete >$ tag, but keep the content value as regular lines. He can of course update these lines. To keep the inserted lines by site1, he just needs to keep the text as it is. An example of the text area at the save time and its effect is given in figure 10

When the user saves, the text saved and the stored version are filtered and any awareness tags removed. The result of this filtering is shown in figure 11. These two versions are then *diffed* to produce a patch that inserts an updated version of the last paragraph.

5 Related Work

Many approaches were proposed to provide awareness in collaborative editing systems [9, 14].

Workspace awareness is built for real time groupware with a small group of users. Workspace awareness delivers knowledge about "who, what and where". Who is currently present in the workspace ? who is doing that ? what are they currently doing ? on what object ? Where are they currently working or looking ? Awareness is delivered through awareness widgets such as radar views, telepointers, multi-user scrollbars [13]. In a real-time context, concurrency awareness in implicitely delivered by workspace awareness. Unfortunately, workspace awareness widgets are not suitable in a fully asynchronous context like our.

Worspace awareness for past interactions answers the "who, what, where, when and how" for workspace events in the past. Who was there and when ? what had a person been doing ? where has a person been ? when did that event happen ? How did that operation happend, how did this object come to be in this state ? It is clear that concurrency awareness is related to the last question. It gives users knowledge about the state of the object - which is a wiki page in our context- and it explains how this wiki page come to be in this state. The problem is how to compute this information in P2P context and how to really deliver this information to

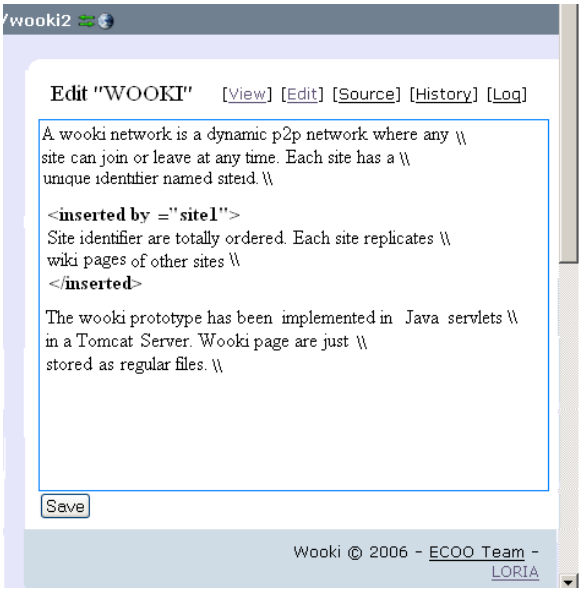

Figure 10: the edit area at the save time

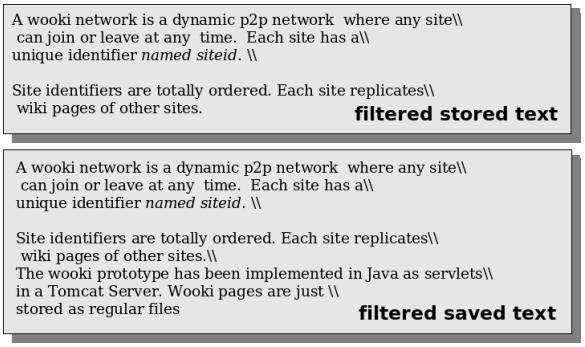

Figure 11: Filtered versions at save time

the user. Workspace awareness for past interaction is mainly available in wiki systems through the history of wiki pages. It is interesting to notice that in P2P wiki, the history of wiki page is no more linear. The total order of versions in a trditional wiki is replaced by a partial order of operations in a P2P wiki. Concurrency awareness in an attempt to make users aware about the effects of the linearization of this partial order.

Change awareness [27] answers the question "is anything different since I last looked at the work ?". If change awareness is an important issue for collaborative system, it is clear that concurrency awareness and change awareness are orthogonal problems. Concurrency awareness answers the question "Has this page been merged automatically ? in this case show me where concurrent changes occur since last reviewed state ?".

The State Treemap [17] try to answer a different question: "what is the status of my workspace according to committed or uncommitted concurrent changes". The state treemap has been designed to make users aware about divergence that exist between different workspaces. State treemap delivers a kind of awareness that can be classified in workspace awareness for multi-synchronous editing [8] while concurrency awareness belongs to workspace awareness for past interaction.

Others work tried to quantify the amount of changes introduced by the last "update" operation and localize them within documents. It can be classified as an extension of change awareness. In [22], authors focused on collaborative editing of structured documents. When remote operations are applied on local document, it triggers the computation of a metric that quantify and localize changes according to the hierarchical structure of document.In [18], author focused on collaborative editing of files on a file system. In both cases, these metrics try answer the questions "where remote changes are located and what size they have ?".

Distributed Version Control Systems (DVCS)[3] allow users to work asynchronously with no central server. Changes are stored as patches in a direct acyclic graph represented the partial ordering of patches. It is possible

to extract from this graph all information required for computing concurrency awareness. However, in Version Control System and DVCS, the classical way to notify users about merge results is to modify the file itself with conflict blocks. In the following example, the conflict block notifies the user of this workspace that these two lines have been modified concurrently and concurrent changes are overlapping.

```
<<<<<<< driver.c
    exit(nerr == 0 ? SUCCESS : FAILURE);
=======
    exit(!!nerr);
>>>>>>> 1.6
```

Tools have different strategies in the way they manage these conflicts. Some are preventing to commit or to update the file before conflicts are solved (as in CVS [5]). This is not a problem in the context of the copy-modify-merge paradigm. But, in a P2P wiki context, if we apply the same strategy, each server will stop to integrate remote patches until conflicts are resolved by a user. Different wiki servers can stop on different conflicts and the whole system can be blocked waiting manual conflict resolutions.

Other tools like Git [12] allow users to publish blocks of conflicts. This causes no particular problem when all participants are humans. If a wiki server can publish blocks of conflicts, this means that when integrating a remote change, the system generates new changes just to deliver conflict awareness. If each server adopt this strategy, there are cases where the P2P system can start an infinite loop. It starts with a simple conflict, next generates conflicts of conflicts and the conflicts blocks grow infinitely.

If the system never stabilize, it violates eventual consistency. In the concurrency awareness presented in this paper, awareness visualization does not modify the state of wiki pages. It can be considered as a view on data. Consequently, it cannot break eventual consistency.

6 Conclusion and future work

A P2P Wiki subtly changes the behavior of traditional wikis in case of concurrent editing. In a P2P wiki, some pages may result from an automatic merge: they are produced by the wiki server outside the control of any user.

We have introduced *concurrency awareness*, a mechanism that makes users aware of the status of a wiki page regarding concurrency and capable of highlighting page regions subject to concurrency mismatches. We have also proposed an implementation of our mechanism that fits P2P requirements, and in particular that is scalable. We used R-entry vectors to implement a scalable concurrency vector. This approach offers a trade-off between scalability and accuracy of the concurrency detection that is acceptable in the context of an awareness mechanism.

This awareness mechanism is activated when a user accesses a wiki page [1]. It extracts the concurrent history from the log, and highlights all lines impacted by these operations.

A first version of our prototype, Wooki, has already been released (http://wooki.sourceforge.net/). We are actually working on the next release that will include a first version of concurrency awareness.

Our work points many open issues that need further investigation.

First, we need to conduct a more complete evaluation of the approach from both the user point of view and the accuracy of the concurrency detection. In particular, the impact of the size of the R-entry vector on the error rate, and the impact of the error rate on the acceptability of the approach need to be examined. We plan a usage study in real settings as soon as the awareness mechanism is available in our prototype.

Awareness can also be improved. Many useful views on the document can be computed. For example, the system can classify and highlight concurrent modifications depending on their type (inserted/deleted lines), on the user who introduced the modification or on the originator site. An aggregated view about concurrency over a whole wiki (a set of pages) can also be computed and presented to users.

We plan although to investigate the way in which version histories can be visualized. As in any wiki system, Wooki maintains the complete version history of each wiki page. However, while in traditional wikis version histories are allways linear, in a wooki server the history integrates patches issued from remote sites and is by nature a concurrent history. How to present such an history in a useful and understandable way is still an open question in Wooki.

Finally, we also think about adding two simple mechanism. The first one would consists in just notifying user when a patch he produced has been merged. The second one consists in reusing concurrency awareness

[1] Of course, we can use caching techniques to improve performance

to provide change awareness. When a site reconnect after a disconnected period, we can compute awareness from the set of patches integrated to the local wiki at the reconnection time. This would provide an interesting awareness about what happened in the wiki during the disconnected period for free.

References

[1] Wikipedia Statistics. *Online http://stats.wikimedia.org/*, (2006).

[2] Wikipedia. The Free Encyclopædia that Anyone Can Edit. *Online http://www.wikipedia.org/*, (2006).

[3] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting Geographically-Distributed Software Development. *Software Configuration Management: Icse Scm-4 and Scm-5 Workshops: Selected Papers*, 1995.

[4] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[5] B. Berliner. CVS II: Parallelizing software development. *Proceedings of the USENIX Winter 1990 Technical Conference*, 341:352, 1990.

[6] B. Charron-Bost. Combinatorics and geometry of consistent cuts: Application to concurrency theroy. In *Proceedings of the Internationnal Workshop on Parallel and Distributed Algorithms*, pages 45–56, 1989.

[7] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[8] P. Dourish. The parting of the ways: divergence, data management and collaborative work. In *ECSCW'95: Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 215–230, Norwell, MA, USA, 1995. Kluwer Academic Publishers.

[9] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pages 107–114, Toronto, Ontario, 1992. ACM Press.

[10] C. A. Ellis, S. J. Gibbs, and G. Rein. Groupware: some issues and experiences. *Commun. ACM*, 34(1):39–58, 1991.

[11] R. S. Fish, R. E. Kraut, and M. D. P. Leland. Quilt: a collaborative tool for cooperative writing. In *Proceedings of the ACM SIGOIS and IEEECS TC-OA 1988 conference on Office information systems*, pages 30–37, New York, NY, USA, 1988. ACM.

[12] Git - fast version control system. http://git.or.cz.

[13] S. Greenberg, C. Gutwin, and M. Roseman. Semantic telepointers for groupware, 1996.

[14] C. Gutwin and S. Greenberg. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Computer Supported Cooperative Work (CSCW)*, 11(3):411–446, 2002.

[15] P. Johnson and R. Thomas. RFC677: The maintenance of duplicate databases, 1976.

[16] B. Kang, R. Wilensky, and J. Kubiatowicz. The hash history approach for reconciling mutual inconsistency. *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 670–677, 2003.

[17] P. Molli, H. Skaf-Molli, and C. Bouthier. State treemap: an awareness widget for multi-synchronous groupware. In *7th International Workshop on Groupware - CRIWG'2001*, Darmstadt, Germany, September 2001.

[18] P. Molli, H. Skaf-Molli, and G. Oster. Divergence awareness for virtual team through the web. In *Integrated Design and Process Technology, IDPT 2002*, Pasadena, CA, USA, June 2002. Society for Desing and Process Science.

[19] J. Morris. DistriWiki:: a distributed peer-to-peer wiki network. *Proceedings of the 2007 international symposium on Wikis*, pages 69–74, 2007.

[20] C. Neuwirth, D. Kaufer, R. Chandhok, and J. Morris. Issues in the design of computer support for co-authoring and commenting. *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 183–195, 1990. PREP.

[21] G. Oster, P. Urso, P. Molli, and A. Imine. 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.

[22] S. Papadopoulou and M. C. Norrie. How a structured document model can support awareness in collaborative authoring. In *3rd International IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing*, New York, USA, November 2007.

[23] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.

[24] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. 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.

[25] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. 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.

[26] D. Sun and C. Sun. Operation context and context-based operational transformation. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 279–288, New York, NY, USA, 2006. ACM.

[27] J. Tam and S. Greenberg. A framework for asynchronous change awareness in collaborative documents and workspaces. *International Journal of Human-Computer Studies*, 64(7):583–598, 2006.

[28] Th., R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A. M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, November 2003.

[29] W. Tichy. RCS - A System for Version Control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[30] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distrib. Comput.*, 12(4):179–195, 1999.

[31] S. Weiss, P. Urso, and P. Molli. Wooki: a p2p wiki-based collaborative writing tool. In *Web Information Systems Engineering*, Nancy, France, December 2007. Springer.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399