Distributed Semantic Wiki: Kolflow Project -Task 5- State of the art (D5.1)

Hala Skaf-Molli and Pascal Molli,
GDD Team, Nantes University
Hala.Skaf@univ-nantes.fr, Pasca.Molli@univ-nantes.fr

June 12, 2012

Abstract

This paper presents the state of the art on Distributed Semantic Wikis. This work is part of the Kolflow project, more precisely it is part of the task 5, **D(5.1)**.

1 Semantic Wikis

Wikis are the most popular tools of Web 2.0, they provide an easy to share and contribute to global knowledge. The encyclopedia Wikipedia is a famous example of a wiki system. In spite of their fast success, wiki systems have some drawbacks. They suffer from search and navigation [6], it is not easy to find information in wikis [2]. They have also scalability, availability and performance problems [19, 8] and they do not support offline works and atomic changes [3]. To overcome these limitations, wiki systems have evolved in two different ways: semantic wikis and peer-to-peer wikis.

Semantic wikis such as Sematic MediaWiki [6], IkeWiki [14], SweetWiki [2] and SWooki [17] are a new generation of collaborative editing tools, they allow users to add semantic annotations in the wiki pages. These semantic annotations can then be used to find pertinent answers to complex queries. Semantic wikis can be viewed as an efficient way to better structure wikis by providing a means to navigate and answer questions or reason, based on annotations. Semantic wikis are an extension of wiki systems that preserve the same principles of wikis such as simplicity in creating and editing wikis pages. Semantic wikis embed semantic annotations in the wiki content by using Semantic Web technologies such as RDF and SPARQL. In semantic wikis, users collaborate not only for writing the wiki pages but also for writing semantic annotations. Traditionally, authoring semantics and creation ontologies has mainly been in the hand of "ontologists" and knowledge management experts. Semantic wikis allow mass collaboration for creating and emerging ontologies. Some existing semantic wikis allows the emergence of the ontology, MediaWiki(SMW) [6] and SweetWiki [2].

Others use the wikis as a collaborative ontologies editors. IkeWiki [14] aims to create an instance of existing ontologies, while OntoWiki [1] aims the creation of ontology schema. In a semantic wiki, users add semantic annotations to wiki page text to represent relations and properties on this page. In SMW users choose their own vocabularies to type links. For instance, a link between the wiki pages "France" and "Paris" may be annotated by a user as "capital".

Content of wiki page of "France"	Content of semantic wiki page of "France"
France is located in [Europe]	France is located in [locatedIn::Europe]
The capital of France is [Paris]	The capital of France is [hasCapital:: Paris]

These annotations express semantic relationships between wikis pages. Semantic annotations are usually written in a formal syntax so they are processed automatically by machines and they are exploited by semantic queries. There are two approaches of semantic wikis [2]:

• *The use of ontologies for wikis*: requires the load of an existing ontology. The advantage is to build controlled vocabularies [14] but it can be too rigid for emergent domains where ontologies are not clearly defined.

• *The use of wikis for ontologies*: semantic wikis let users choose their own vocabularies [6]. Semantic annotations are integrated directly in the wiki text. Semantic data appear in their context. The main advantage is to allow the emergence of an ontology.

In spite of their success, semantic wikis do not support a multi-synchronous work mode. Their current model provides only one copy of a semantic wiki page. The state of a semantic wiki page is always the current visible one on the server, intermediate inconsistent states are always visible. Consequently, transactional changes are not be supported neither the isolated work mode nor off-line editing mode.

2 P2P wikis

The basic idea of peer-to-peer wikis is to replicate wiki pages on the peers of a P2P network.

The main problem is to ensure the consistency of copies. Strong consistency of copies such as sequential consistency or 1-copy serializability cannot be currently ensured on P2P networks. P2P wikis rely on algorithms with high communication complexity or on algorithms that do not support dynamicity of P2P networks [13]. All existing P2P wikis are using an optimistic replication algorithm that ensures only a weaker consistency such as causal consistency, eventual consistency or CCI consistency (causality, convergence, intentions). A wiki deployed on a P2P network takes natural advantages of a P2P network i.e. faults-tolerance, better scalability, infrastructure cost sharing and better performances. P2P wikis can be divided into two categories:

Partial replication. The replication is "partial" if a single page has a fixed number of copies [8], [3] and [10]. Partial replication is generally implemented

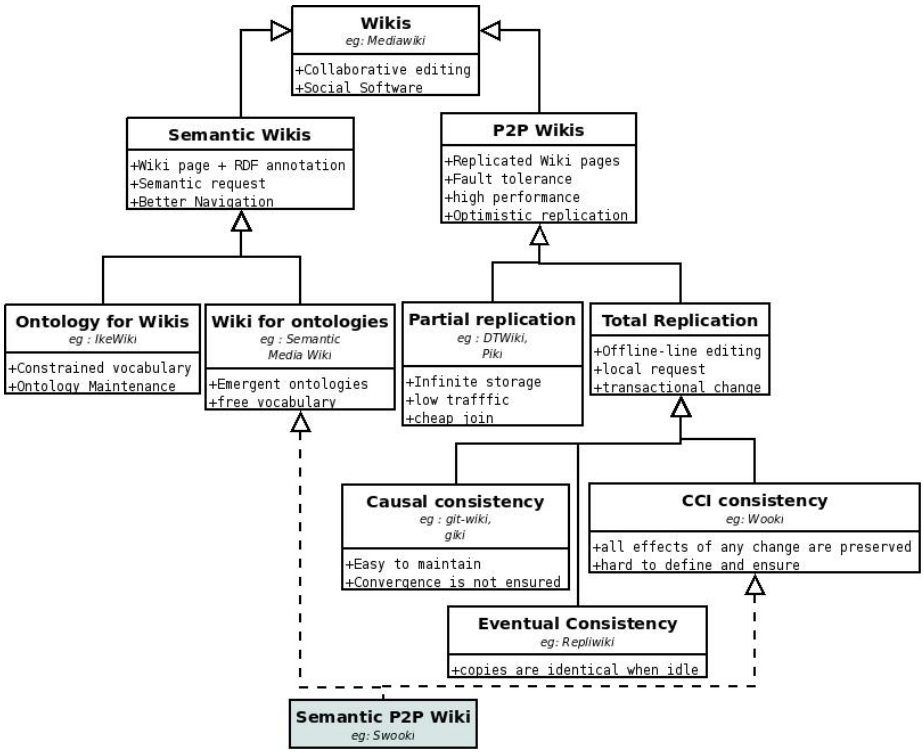

Figure 1: Wiki, Semantic Wiki and P2P Wiki

on Distributed Hash tables (DHT). The main advantages of partial replication are: (i) a virtual infinite storage i.e. adding a peer in the network increases storage capacity, (ii) a less traffic generation than the total replication approach (iii) a cheap join of a new site to the P2P network.

Total replication. The replication is "total" if all pages are replicated on all servers [19], [5], [4]. The main advantages of total replication compared to partial replication are: (i) off-line editing i.e. even when disconnected, a peer can continue editing, this is mandatory for mobile work, (ii) transactional changes i.e. by allowing off-line work, it is possible to generate a change that concerns multiple wiki pages (iii) local requests execution i.e. as all data are local, all requests can be processed locally.

Compared to partial replication, total replication has some drawbacks: (i) total replication generates more traffic for write operations i.e. when a change is made on a copy, it is propagated to all copies and not to a fixed number of copies, (ii) when a peer is joining the network, it requires transferring the complete state of the wiki not just the range of wiki pages that the peer is responsible to.

In this paper, we focus on the state of the art of distributed semantic wiki. Such a system combines advantages of P2P wikis and Semantic wikis. We will

detail two systems: SWOOKI and DSMW. Both of these systems are based on the **the use of wikis for ontologies** approach such as Semantic MediaWiki. However, Swooki is **peer-to-peer wiki** based on **total replication** and **CCI consistency** and DSMW is a **peer-to-peer wiki** based on **partial replication** and **CCI consistency**.

3 SWooki : A peer-to-peer Semantic Wiki

A P2P semantic wiki is a P2P network of autonomous semantic wiki servers (called also peers or nodes) that can dynamically join and leave the network. Every peer hosts a copy of all semantic wiki pages and an RDF store for the semantic data. Every peer can autonomously offer all the services of a semantic wiki server. When a peer updates its local copy of data, it generates a corresponding operation. This operation is processed in four steps:

1. It is executed immediately against the local replica of the peer,
2. it is broadcasted through the P2P network to all other peers,
3. it is received by the other peers,
4. it is integrated to their local replica. If needed, the integration process merges this modification with concurrent ones, generated either locally or received from a remote server.

The system is correct if it ensures the CCI consistency model (see section **??**).

3.1 Data Model

The data model is an extension of Wooki [19] data model to take in consideration semantic data. Every semantic wiki peer is assigned a global unique identifier named $NodeID$. These identifiers are totally ordered. As in any wiki system, the basic element is a semantic wiki page and every semantic wiki page is assigned a unique identifier $PageID$, which is the name of the page. The name is set at the creation of the page. If several servers create concurrently pages with the same name, their content will be directly merged by the synchronization algorithm. Notice that a URI can be used to unambiguously identify the concept described in the page. The URI must be global and location independent in order to ensure load balancing. For simplicity, in this paper, we use a string as page identifier.

Definition 1 *A semantic wiki page Page is an ordered sequence of lines* $L_B L_1, L_2, \ldots L_n L_E$ *where* L_B *and* L_E *are special lines.* L_B *indicates the beginning of the page and* L_E *indicates the ending of the page.*

Definition 2 *A semantic wiki line L is a four-tuple* $<$ *LineID, content, degree, visibility* $>$ *where*

• $LineID$ is the line identifier, it is a pair of ($NodeID$, $logicalclock$) where $NodeID$ is the identifier of the semantic wiki server and $logicalclock$ is a logical clock of that server. Every semantic wiki server maintains a logical clock, this

clock is incremented when an operation is generated. Lines identifiers are totally ordered so if $LineID_1$ and $LineID_2$ are two different lines with the values ($NodeID_1$, $LineID_1$) and ($NodeID_2$, $LineID_2$) then $LineID_1 < LineID_2$ if and only if (1) $NodeID_1 < NodeID_2$ or (2) $NodeID_1 = NodeID_2$ and $LineID_1 < LineID_2$.

• *content* is a string representing text and the semantic data embedded in the line.

• *degree* is an integer used by the synchronization algorithm, the degree of a line is fixed when the line is generated, it represents a kind of loose hierarchical relation between lines. Lines with a lower degree are more likely generated earlier than lines with a higher degree. By definition the degree of L_E and L_B is zero.

• *visibility* is a boolean representing if the line is visible or not. Lines are never really deleted they are just marked as invisible. For instance, suppose there are two lines in a semantic wiki page about "France" , "France" is the identifier of the page.

France is located **in** [locatedIn :: Europe]
The capital of France is [hasCapital :: Paris]

Suppose these two lines are generated on the server with $NodeID = 1$ in the above order and there are no invisible lines, so the semantic wiki page will be internally stored as.

L_B
((1,1), France is located **in** [locatedIn :: Europe], 1, true)
((1,2), The capital of France is [hasCapital :: Paris], 2, true)
L_E

Text and semantic data are stored in separate persistent storages. Text can be stored in files and semantic data can be stored in RDF repositories, as described in the next section.

Semantic data storage model RDF is the standard data model for encoding semantic data. In P2P semantic wikis, every peer has a local RDF repository that contains a set of RDF statements extracted from its wikis pages. A statement is defined as a triple (Subject, Predicate, Object) where the subject is the name of the page and the predicates (or properties) and the objects are related to that concept. For instance, the local RDF repository of the above server contains: R = {("France", "locatedIn", "Europe"), ("France", "hasCapital", "Paris") }. As for the page identifier, a global URI can be assigned to predicates and objects of a concept, for simplicity, we use a string. We define two operations on the RDF repositories:

• insertRDF(R,t): adds a statement t to the local RDF repository R. • deleteRDF(R,t): deletes a statement t from the local RDF repository R.

These operations are not manipulated directly by the end user, they are called implicitly by the editing operations as shown later.

3.2 Editing operations

A user of a P2P semantic wiki does not edit directly the data model. Instead, she uses traditional wiki editing operations, when she opens a semantic wiki page, she sees a view of the model. In this view, only visible lines are displayed. As in a traditional semantic wiki, she makes modifications i.e. adds new lines or deletes existing ones and she saves the page(s). To detect user operations, a diff algorithm is used to compute the difference between the initial requested page and the saved one. Then these operations are transformed into model editing operations. A *delete* of the line number n is transformed into a *delete* of the n^{th} visible line and an *insert* at the position n is transformed into *insert* between the $(n-1)^{th}$ and the n^{th} visible lines. These operations are integrated locally and then broadcasted to the other servers to be integrated. There are two editing operations for editing the wiki text: *insert* and *delete*. An update is considered as a delete of old value followed by an insert of a new value. There are no special operations for editing semantic data. Since semantic data are embedded in the text, the RDF repositories are updated as a side effect of text replication and synchronization. (1) *Insert(PageID, line,* l_P *,* l_N*)* where $PageID$ is the identifier of the page of the inserted line. *line* is the line to be inserted. It is a tuple containing $<$ LineID, content, degree, visibility $>$. l_P is the identifier of the line that precedes the inserted line. l_N is the identifier of the line that follows the inserted line. During the insert operation, the semantic data embedded in the line are extracted, RDF statements are built with the page name as a subject and then they are added to the local RDF repository thanks to the function $insertRDF(R, t)$. (2) The *delete(PageID, LineID)* operation sets the visibility of the line identified by $LineID$ of the page $PageID$ to false. The line is not deleted physically, it is just marked as deleted. The identifiers of deleted lines must be kept as a tombstones. During the delete operation, the set of RDF statements contained in the deleted line is deleted from the local RDF repository thanks to the $deleteRDF(R, t)$.

3.3 Correction Model

This section defines causal relationships and intentions of the editing operations for our P2P semantic wiki data model.

3.4 Causality preservation

The causality property ensures that operations ordered by a precedence relation will be executed in the same order on every server. In WOOT, the precedence relation relies on the *semantic causal dependency*. This dependency is explicitly declared as preconditions of the operations. Therefore, operations are executed on a state where they are legal *i.e.* preconditions are verified. We define causality for editing operations that manipulate text and RDF data model as:

Definition 3 ***insert Preconditions*** *Let Page be the page identified by* $PageID$*, let the operation op=*Insert($PageID$, *newline*, p , n)*, newline =< LineID, c,*

d, v> generated at a server $NodeID$, R is its local RDF repository. The line newline can be inserted in the page $Page$ if its previous and next lines are already present in the data model of the page $Page$.

$$\exists i \; \exists j \; LineID(Page[i]) = p \wedge \; LineID(Page[j]) = n$$

Definition 4 ***Preconditions of delete operation*** *Let $Page$ be the page identified by $PageID$, let op =* Delete($PageID$, dl) *generated at a server $NodeID$ with local RDF repository R, the line identified by dl can be deleted (marked as invisible), if its dl exists in the page.*

$$\exists i \; LineID(Page[i]) = dl$$

When a server receives an operation, the operation is integrated immediately if its pre-conditions are evaluated to true else the operation is added to a waiting queue, it is integrated later when its pre-conditions become true.

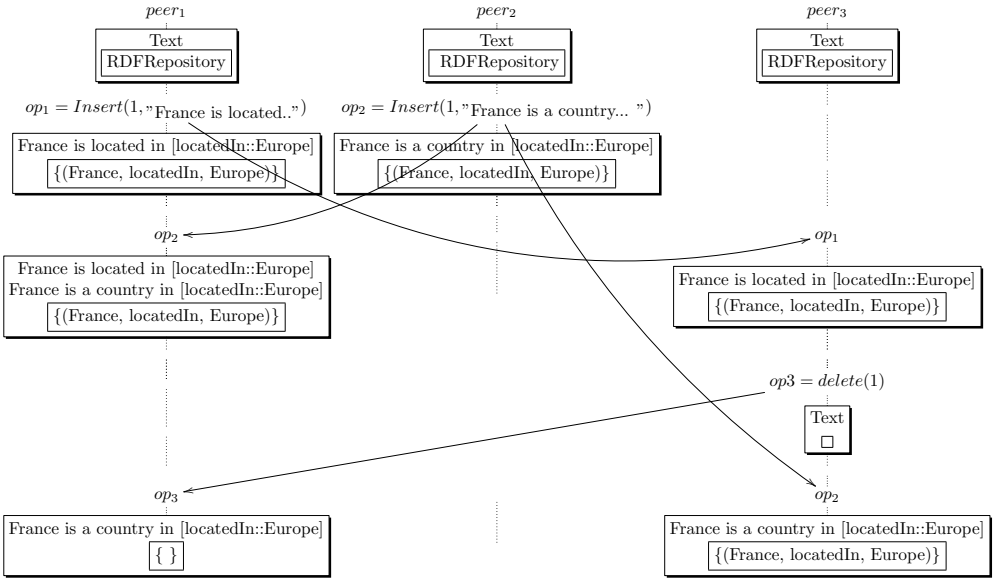

Figure 2: Semantic inconsistency after integrating concurrent modifications

3.5 Intentions and Intentions preservation

The intention of an operation is the visible effect observed when a change is generated at one peer, the intention preservation means that the intention of the operation will be observable on all peers, in spite of any sequence of concurrent operations. We can have a naive definition of intention for *insert* and *delete*:

• The intention of an insert operation *op= Insert(PageID, newline, p , n)* when generated at site $NodeID$, where *newline* =< nid, c, d, v> is defined as: (1) The content is inserted between the previous and the next lines and (2) the semantic data in the line content are added to the RDF repository of the server.

• The intention of a delete operation *op= delete(pid, l)* when generated at site S is defined as : (1) the line content of the operation is set to invisible and (2) the semantic data in the line content are deleted from the RDF repository of the server.

Unfortunately, it is not possible to preserve the previous intention definitions. We illustrate a scenario of violation of these intentions in figure 2. Assume that three P2P semantic wiki servers, $peer_1$, $peer_2$ and $peer_3$ share a semantic wiki page about "France". Every server has its copy of shared data and has its own persistence storage repository. At the beginning, the local text and the RDF repositories are empty. At $peer_1$, $user_1$ inserts the line "France is located [located In::Europe]" at the position 1 in her copy of the "France" page. Concurrently, at $peer_2$ $user_2$ inserts a new line "France is a country in [located In::Europe]" in her local copy of "France" page at the same position and finally at $peer_3$ $user_3$ deletes the line added by $user_1$. When op_2 is integrated at $peer_1$, the semantic annotation is present two times in the text and just one time in the RDF repository. In fact, the RDF repository cannot store twice the same triple. When op_3 is finally integrated on $peer_1$, it deletes the corresponding line and the semantic entry in the RDF repository. In this state, the text and the RDF repository are inconsistent. Concurrently, $peer_3$ has integrated the sequence [op_1;op_3;op_2]. This sequence leads to a state different than the state on $peer_1$. Copies are not identical, convergence is violated.

The above intentions cannot be preserved because the effect of executing op_3 changes the effect of op_2 which is independent, of op_3 i.e. op_3 deletes the statement inserted by op_2, but op_3 has not seen op_2 at generation time.

3.6 Model for Intention preservation

It is not possible to preserve intentions if the RDF store is defined as a set of statements. However, if we transform the RDF store into multi-set of statements, it becomes possible to define intentions that can be preserved.

Definition 5 ***RDF repository*** *is the storage container for RDF statements, each container is a* multi-set of RDF statements*. Each RDF repository is defined as a pair (T, m) where T is a set of RDF statements and m is the multiplicity function $m : T \rightarrow \mathcal{N}$ where $\mathcal{N} = 1, 2.....$*

For instance, the multi-set $R =$ { ("France", "LocatedIn", "Europe"),("France", "LocatedIn", "Europe"),("France", "hasCapital", "Paris") } can presented by $R =$ { ("France", "LocatedIn", "Europe")^2, ("France", "hasCapital", "Paris")^1 } where 2 is the number of occurrence of the first statement and 1 is this of the second one.

Definition 6 ***Intention of insert operation*** *Let S be a P2P semantic wiki server, R is its local RDF repository and $Page$ is a semantic wiki page. The intention of an insert operation op=* Insert($PageID$, *newline*, p , n) *when generated at site S, where newline =< nid, c, d, v> and T is the set (or multi-set) of RDF statements in the inserted line, is defined as: (1) The content is*

inserted between the previous and the next lines and (2) the semantic data in the line content are added to R.

$$\exists i \quad \wedge \exists i_P < i \; LineID(Page[i_P]) = p \tag{1}$$
$$\wedge \exists i \leq i_N \; LineID(Page[i_N]) = n \tag{2}$$
$$\wedge Page'[i] = newline \tag{3}$$
$$\wedge \forall j < i \; Page'[j] = Page[j] \tag{4}$$
$$\wedge \forall j \geq i \; Page'[j] = Page[j-1] \tag{5}$$
$$\wedge R' \leftarrow R \uplus T \tag{6}$$

Where $Page'$ and R' are the new values of the page and the RDF repository respectively after the application of the insert operation at the server S and $\uplus$ is the union operator of multi-sets. If a statement in T already exists in R so its multiplicity is incremented else it is added to R with multiplicity one.

Definition 7 ***Intention of delete operation*** *Let S be a P2P semantic wiki server, R is the local RDF repository and Page is a semantic wiki page. The intention of a delete operation op=* delete($PageID$, ld) *where T is the set (or multi-set) of RDF statements in the deleted line, is defined as: (1) the line ld is set to invisible and (2) the number of occurrence of the semantic data embedded in ld is decreased by one, if this occurrence is equal to zero which means these semantic data are no more referenced in the page then they are physically deleted from the R.*

$$\exists i \quad \wedge PageID(Page'[i]) = ld \tag{7}$$
$$\wedge visibility(Page'[i]) \leftarrow false \tag{8}$$
$$\wedge R' \leftarrow R - T \tag{9}$$

Where $Page'$ and R' are the new values of the page and the RDF repository respectively after the application of the delete operation at the server S and $-$ is the difference of multi-sets. If statement(s) in T exists already in R so its multiplicity is decremented and deleted from the repository if it is equal to zero.

Let us consider again the scenario of the figure 2. When op_2 is integrated on $peer_1$, the multiplicity of the statement ("$France$","$locatedIn$","$Europe$") is incremented to 2. When op_3 is integrated on $peer_1$, the multiplicity of the corresponding statement is decreased and the consistency between text and RDF repository is ensured. We can observe that $Peer_1$ and $Peer_3$ now converge and that intentions are preserved.

3.7 Algorithms

As any wiki server, a P2P semantic server defines a *Save* operation which describes what happens when a semantic wiki page is saved. In addition, it defines *Receive* and *Integrate* operations. The first describes what happens upon receiving a remote operation and the second integrates the operation locally.

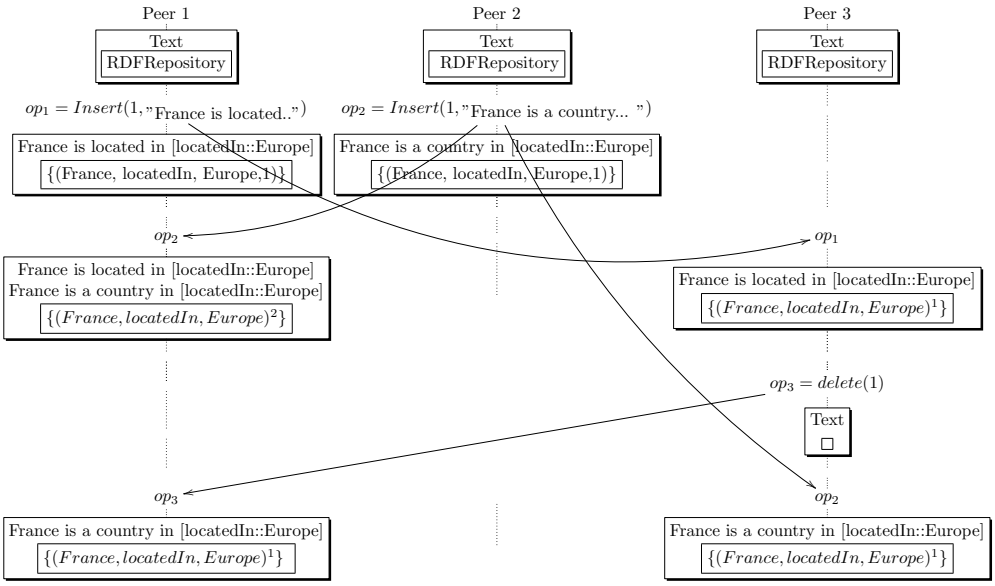

Figure 3: Convergence after integrating concurrent modifications

Save operation During saving a wiki page, a *Diff* algorithm computes the difference between the saved and the previous version of the page and generates a patch. A *patch* is the set of delete and insert operations on the page (Op = *Insert(PageID, line,* l_P *,* l_N*)* or Op = *Delete(PageID, LineID)*). These operations are integrated locally and then broadcasted to other sites in order to be executed as shown below.

```
Upon Save(page, oldPage) :
   let P ← Diff(page, oldPage)
 for each op ∈ P do
   Receive(op)
 endfor
Broadcast(P)
```

At this level of description, we just make the hypothesis that $Broadcast(P)$ will eventually deliver the patch P to all sites. More details are given in section **??**.

Delivery Operation When an operation is received (cf figure 4) its preconditions are checked (cf figure 5). If they are not satisfied, the operation is added to the waiting log of the server, else according to the type of the operations some steps are executed.

The waiting log is visited after the integration and the operations that satisfy their preconditions are removed from the log and integrated. The function $ContainsL(PageID, id)$ tests the existence of the line in the page, it returns true if this is the case. The function $isVisible(LineID)$ tests the visibility of

```
Upon Receive(op) :
 if isExecutable(op) then
   if type(op) = insert then
     IntegrateIns(op)
   if type(op) = delete then
     IntegrateDel(op)
 else
      waitingLog ← waitingLog ⋃ {op}
 endif
```

Figure 4: Receive operation

```
isExecutable(op) :
 if type(op) = del then
   return
        containsL(PageID,LineID)
        and isVisible(LineID)
 else
   return ContainsL(PageID,l_P)
        and ContainsL(PageID, l_N)
 endif
```

Figure 5: isExecutable Operation

```
IntegrateDel(LineID) :
   IntegrateDelT(LineID)
   IntegrateDelRDF(LineID)
```

Figure 6: IntegrateDel operation

```
IntegrateDelT(LineID) :
   Page[LineID]. visibility ←false
```

Figure 7: IntegrateDelT Operation

the line.

Integrate operation The integration of an operation is processed in two steps (cf figure 6): (1) text integration and (2) RDF statements integration. To integrate a text *delete* operation (cf. figure 7), the visibility flag of the line is set to false whatever is its content. To integrate RDF statements (cf figure 8), a counter is used to implement a multi-set RDF repository. A counter is attached to every RDF triple, the value of the counter corresponds to the number of occurrence of the triple in the repository. During the delete operation, the counter of the deleted statements is decreased, if the counter is zero the statements are physically deleted from the repository.

To integrate an insert operation (cf figure 9) the *line* has to be placed among all the lines between l_P and l_N, some of these lines can be previously deleted or inserted concurrently and the inserted semantic data are integrated. To

```
IntegrateDelRDF(LineID) :
let   S ← ExtractRDF(LineID)
if S ≠ ∅ then
 for each triple ∈ S do
      triple .counter−−
     if   triple .counter = 0 then
       deleteRDF(R,triple)
     endif
endif
```

Figure 8: IntegrateDelRDF operation

```
IntegrateIns(PageID, line, l_P , l_N) :
   IntegratedInsT(PageID, line, l_P , l_N)
   IntegrateInsRDF(line)
```

Figure 9: IntegrateIns Operation

```
IntegrateInsT(PageID, line, l_P , l_N) :
let S' ←
subseq(Page[PageID]), l_P, l_N)
if S = ∅ then
  insert(PageID, line, l_N)
else
   let i ← 0
   let d_min ← min(degree(S'))
   let F ← filter(S', degree = d_min)
   while (i < |F| − 1)and(F[i] <_id line)
   do  i ← i +1
      IntegrateInsT(PageID,line,F[i−1],F[i])
endif
```

Figure 10: Integrate insert text operation

```
IntegrateInsRDF(line) :
let  S ← ExtractRDF(line)
if S ≠ ∅ then
  for each triple ∈ S do
    if  Contains(triple) then
        triple .counter++
    else
       insertRDF(R,triple)
    endif
endif
```

Figure 11: IntegrateInsRDFOperation

integrate a line in a wiki page, we use the integration algorithm defined in [19]. This algorithm (cf. figure 10) selects the sub-sequence S' of lines between the previous and the next lines, in case of an empty result, the line is inserted before the next line. Else, the sub-sequence S' is filtered by keeping only lines with the minimum degree of S'. The remaining lines are sorted according to the line identifiers order relation $<_{id}$ [9], therefore, *line* will be integrated in its place according $<_{id}$ among remaining lines, the procedure is called recursively to place *line* among lines with higher degree in S'. To integrate the semantic data (cf figure 11), the RDF statements of the inserted line are extracted and added to the local RDF repository. If the statements exist already in the repository, their counter is incremented, otherwise, they are inserted into the RDF repository with a counter value equals to one as shown below.

To summarize, causality as defined in section 3.4 is ensured by the *Receive* algorithm. Convergence for text is already ensured by the WOOT algorithm [9]. Convergence for semantic data is trivially ensured by the multi-set extension of the RDF repository. The intention preservation for a text is demonstrated in [9]. Here, we are concerned with the intention of semantic data as defined in 3.5. The intention of an *insert* operation is trivially preserved by the algorithm *IntegrateInsRDF*. Since a possible way to implement a *multi-set* is to associate a counter to every element. In the same way, the algorithm *IntegrateDelRDF* preserves the intention of the *delete* operation. The basic idea behind all these algorithms is to reach convergence and preserve intentions whatever is the order of reception of operations. This implies that these algorithms "force" commutativity of operations. If operations are commuting then all concurrent executions are equivalent to a serial one. In our system, users can start a transaction just by switching to the offline mode and end a transaction by switching to online mode. We chose this way to interact with users in order to keep the system simple. If a user produces a consistent change, as all operations of any transaction are commuting and ensure the same effects, then all concurrent execution

of transactions generate a correct state.

The SWOOKI prototype has been implemented in Java as servlets in a Tomcat Server and demonstrated in [11]. This prototype is available with a GPL license on sourceforge at `http://sourceforge.net/projects/wooki` and it is also available online at: `http://wooki.loria.fr/wooki1`.

3.8 Synthesis

SWooki is a peer-to-peer semantic wikis combines both advantages of semantic wikis and P2P wikis. The fundamental problem is to develop an optimistic replication algorithm that ensures an adequate level of consistency, supports P2P constraints and manages semantic wiki page data type.

SWooki is based on total replication this allows to query, access, reason and retrieve data locally from any peer without the need for search mechanisms nor transfer of the semantic data between peers to resolve queries Therefore, "querying the network" [18] becomes querying any peer. (ii) Total replication enables transactional changes *i.e.* atomic changes across multiple pages. Supporting transactional changes is a very important feature in the context of semantic wiki. In semantic wiki, a wiki page presents a concept of an ontology, so a modification of one concept may requires atomic changes to other concepts. If the change is not atomic, it means that intermediate changes will be visible to others users and to concurrent requests. This can lead to confusion for other users and to false results for requests.

SWooki [17] combines the advantages of P2P wikis and semantic wikis. It is based on an optimistic replication algorithm that ensures the CCI consistency model. It supports the off-line work mode and transactional changes. Its main limitations are the total replication and the collaboration model. Every peer of the network hosts a replica of wiki pages and a replica of the semantic store. Users cannot choose the pages that they want to replicate neither the period of synchronization. Modifications are propagated on page saving. The collaborative community is implicit. Users cannot choose to whom propagate modifications. Changes propagation is under the control of system and not the users. All connected peers receive and integrate changes. A disconnected peer will receive the modifications of the others at reconnection thanks to the anti-entropy algorithm.

4 Multi-synchronous Semantic Wiki Approach (DSMW)

Multi-synchronous semantic wikis allow users to build their own cooperation networks. The construction of the collaborative community is declarative, in the sense, every user declares explicitly with whom he would like to cooperate. Every user can have a multi-synchronous semantic wiki server installed on her machine. She can create and edit her own semantic wiki pages as in a normal

semantic wiki system. Later, she can decide to share or not these semantic wiki pages and decide with whom to share.

4.1 Multi-synchronous Collaboration Model

The replication of data and the communication between servers is made through *channels* (feeds). The channel usage is restricted to few servers with simple security mechanisms that requires no login and complex access control. Capabilities fit perfectly these requirements [7]. The key point is that channels are read-only for consumers and can be hosted on hardware of users. When a semantic wiki page is updated on a multi-synchronous semantic wiki server, it generates a corresponding operation. This operation is processed in four steps: (1) It is executed immediately against page, (2) it is published to the corresponding *channels*, (3) it is pulled by the authorized servers, and (4) it is integrated to their local replica of the page. If needed, the integration process merges this modification with concurrent ones, generated either locally or received from a remote server.

The system is correct if it ensures the CCI (Causality, Convergence and Intention Preservation) consistency model. Multi-synchronous semantic wikis use the Logoot synchronization algorithm [20] to integrate modifications. Logoot ensures the CCI consistency model. More precisely, Logoot ensures convergence and preserves the intentions of operations if the causality of the operations is preserved.

4.2 Collaboration Scenarios

This section presents two scenarios of collaboration in multi-synchronouns semantic wikis. We suppose several professors collaborate together through a semantic wiki to prepare lectures, exercises and exams. Later, they want to make lectures available for students and they want to integrate relevant feedbacks from students.

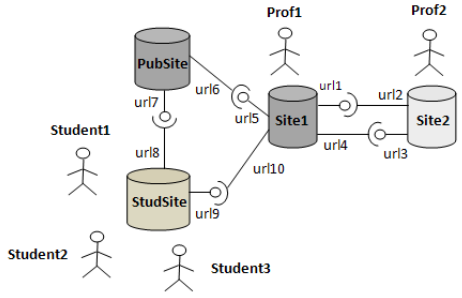

Figure 12: multi-synchronous semantic wikis scenario

Scenario 1: Collaboration among professors For simplicity, we suppose there are two professors $prof_1$ and $prof_2$ (see figure 12). Every professor has her own multi-synchronous semantic wiki, $site_1$ for $prof_1$ and $site_2$ for $prof_2$, respectively.

1. $prof_1$ edits three semantic wiki pages $lesson_1$, $exercises_1$ and $exam_1$ on her site $site_1$. For instance, the page $lesson_1$ edited by $prof_1$ has three lines:

```
  Introduction:
In "multi-synchronous" work, parties work independently in parallel.
[Category::Lesson] [forYear:2009]
```

2. $prof_1$ wants to publish her modifications on $lesson_1$ to $prof_2$. Therefore, she creates a *feed* url_1 that contains change set. Finally, she communicates the address of the feed to $prof_2$.

3. $prof_2$ subscribes to this channel and creates a feed url_2 to pull modifications. The result is a local semantic wiki page $lesson_1$ that has the same content as $lesson_1$ of $prof_1$.

4. $prof_2$ edits his local copy of the $lesson_1$ page. $lesson_1$ has now five lines:

```
 Introduction:
In "multi-synchronous" work mode, parties work independently in parallel.
[Category::Lesson] [forYear:2009]. This mode is based on divergence and
synchronization phases.
```

5. In her turn, $prof_2$ shares his modifications on $lesson_1$ with $prof_1$. He creates a feed and publishes his modifications. $prof_1$ subscribes to the feed and pulls the modifications published by $prof_2$ and finally integrates these modifications on her local copy of $lesson_1$. The integration process merges the remote modifications with concurrent ones, if any, generated locally by $prof_1$. The integration process has to ensure the convergence of all copies on $lesson_1$ if all generated operations on $site_1$ and $site_2$ are integrated on both sites.

The advantages of this collaboration model are, on the one hand, there is no need for centralized server to cooperate. On the other hand, every professor works on her or his own copy in isolation, publishes and pulls changes whenever she wants, *i.e.* the changes propagation is under the control of the user. This collaboration process is not supported in classical semantic wikis. The table 1 represents a part of the scenario.

Scenario 2: Collaboration among professors and students In this scenario, $prof_1$ wants to make $lesson_1$ available for her students while she continues to make corrections and minor modifications on $lesson_1$. In order to provide her students with the courses, $prof_1$ publishes them on a public site *pubSite* that can be either her proper public wiki site or the site of the university. *pubSite* is accessible by the students for read only to

$Site_1$	$Site_2$
Edit($lesson_1$)	
Edit($exercises_1$)	
Edit($exam_1$)	
CreatePushFeed(f_1,q_1)	
	CreatePullFeed(f_1, f_2)
Edit($lesson_1$)	Edit($lesson_1$)
Push(f_1)	
	CreatePushFeed(f_3,q_2)

Table 1: Multi-synchronous collaboration scenario

maintain the courses consistency. However, $prof_1$ manages to integrate relevant feedbacks from students provided by the students sites.

Professor can make continuous improvement of the lessons and can make continuous integration of the modifications and make only consistent modifications visible. Without multi-synchronous support all incremental changes will be visible to both end users and to semantic request engines. Every participant professor and student want to control the visibility of their modifications and want to control the integration of others' modifications.

5 Multi-synchronous Semantic Wiki System

This section presents the data model and the algorithms for multi-synchronous Semantic Wiki systems (M2SW). The data model is defined an ontology. Therefore, it is possible to querying and reasoning on the model itself and make future extension. For instance, it is possible to make queries like: (1)"list all unpublished changes", (2) "list all published changes on a given channel", (3) "list unpublished change set of a given semantic wiki page", (4) "list all pulled change sets", etc.

5.1 DSMW Ontology

The M2SW ontology is defined as an extension of existing ontologies of semantic wikis [2]. In this section, we present the M2SW ontology and detail only its new vocabulary and their properties.

- **WikiSite** : this concept corresponds to a semantic wiki server. A site has the following properties:
  - $siteID$: this attribute contains the URL of the site.
  - *logicalClock* : this attribute has a numeric value. Every semantic wiki server maintains a logical clock, this clock is used to identify patches and operations in an unique way in the whole network.
  - $hasPush$, $hasPull$ and $hasPage$: the range of these properties are respectively a push feed, a pull feed and a semantic wiki page. A wiki site has several push feeds, pull feeds and several pages .
- **SemanticWikiPage**: this concept corresponds to a normal semantic wiki page. It has the following properties:
  - $pageID$: this attribute contains the URL of the page.
  - *hasContent* the range of this property is a String, it contains text and the semantic data embedded in the semantic wiki page.

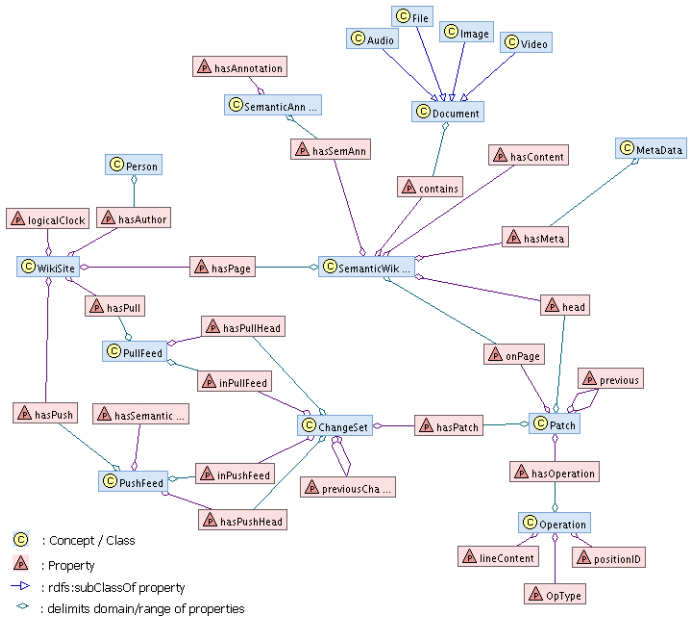

Figure 13: Multi-synchronous ontology

  - *head* : this property points to the *last patch* applied to the page.

- **Operation** : this concept represents a change in a line of a wiki page. In our model, there are two editing operations : *insert* and *delete*. An update is considered as a delete of old value followed by an insert of a new value. An operation has the following properties:

  - $operationID$: this attribute contains the unique identifier of the operation. $operationID$ is calculated by: $operationID = \text{concat}(Site.siteID, Site.logicalClock++)$, the *concat* function concatenates two strings.
  - $opType$: this attribute contains the type of the operation, it can be either an *insert* or a *delete*.
  - $positionID$: denotes the position of the line in the wiki page. This identifier is calculated by the *Logoot* algorithm[20].
  - $lineContent$: is a string representing text and the semantic data embedded in the line.

- **Patch** : a patch is a set of operations. A patch is calculated during the save of the modified semantic wiki page using the Logoot algorithm. A patch has the following properties:

  - $patchID$: is a unique identifier of the patch. Its value is calculated by : $patchID = \text{concat}(Site.siteID, Site.logicalClock++)$
  - $onPage$: the range of this property is the page where the patch was applied.
  - $hasOperation$: this property points to the operations generated during the save of the page.

- *previous*: points to the precedent patch.

- **ChangeSet** : a change set contains a set of patches. This concept is important in order to support transactional changes. It allows to regroup patches generated on multiple semantic wiki pages. Therefore, it is possible to push modifications on multiple pages. *ChangeSet* has the following properties:
  - $changSetID$: is a unique identifier of a change set. Its value is calculated as : $changeSetID$ = concat($Site.siteID$,$Site.logicalClock++$)
  - $hasPatch$ property points to the patches generated since the last push.
  - $previousChangeSet$: points to the precedent change set.
  - $inPushFeed$: the range of this property is $PushFeed$. This property indicates the $PushFeed$ that publishes a $ChangeSet$.
  - $inPullFeed$: the range of this property is $PullFeed$. This property indicates the $PullFeed$ that pulls a $ChangeSet$.

- **PushFeed** : this concept is used to publish changes of a $WikiSite$. It is a special semantic wiki page. It inherits the properties of the $SemanticWikiPage$ concept and defines its own properties:
  - $hasPushHead$: this property points to the *last* published *changeSet*.
  - $hasSemanticQuery$: this property contains a semantic query. This query determines the content of the push feed. For instance, the query can be "find all Lessons", this will return all the pages in the class (category) Lessons. To answer $hasSemanticQuery$, reasoning and querying capabilities of semantic wikis are used.

- **Pull Feed** : this concept is used to pull changes from a remote $WikiSite$. A pull feed is related to one push feed. In the sense that it is impossible to pull unpublished data. A pull feed is also a special semantic wiki page. It inherits the properties of the *Semantic Wiki Page* concepts and defines it own properties:
  - $hasPullHead$: this property points to the last pulled change set pulled.
  - $relatedPushFeed$: this property relates a pull feed to the URL of its associated push feed.

We can extend and build on $MS2W$ ontology. The $MS2W$ ontology is maintained by the $MS2W$ developers. It is defined in OWL DL allowing to querying and reasoning on the patches, ChangeSet, PushFeed, etc. SPARQL can be used to query the MS2W data. For instance, it is possible to list all published patches on a push feed:

$$Published \equiv \exists(hasPatch^{-1}).\exists(inPushFeed^{-1}).PushFeed$$

5.2 Algorithms

As any semantic wiki server, a multi-synchronous semantic wiki server defines a *Save* operation which describes what happens when a semantic wiki page is saved. In addition, we define special operations : *CreatePushFeed*, *Push*, *CreatePullFeed*, *Pull* and *Integrate* for the multi-synchronous semantic wiki. We use the Logoot algorithm [20] for the generation and the integration of the *insert* and *delete* operations. In the following, detail these operations for a semantic wiki server called *site*.

Save Operation During the saving a wiki page, the *Logoot* algorithm computes the difference between the saved and the previous version of the page and generates a patch. A *patch* is a set of delete and insert operations on the page ($Op = (opType, operationID, positionID, lineContent)$). *Logoot* calculates the $positionID$, $lineContent$ and the $opType$ of the operation. These operations are integrated locally and then eventually published on a push feed.

```
On Save(page : String, page:String) :
  Patch(pid=concat(site.siteID, site . logicalClock + +))
  foreach op ∈ Logoot(page, page) do
    Operation(opid=concat(site.siteID,site . logicalClock+ +))
    hasOperation(pid,opid)
  endfor;
  previous(pid,page.head)
  head(page,pid)
  onPage(pid,page)
```

CreatePushFeed Operation The communication between multi-synchronouns semantic wiki servers is made through feeds. The *CreatePushFeed* operation creates of a push feed. A push feed is a special semantic wiki page that contains a query that specifies the pushed data. It is used to publish changes of a wiki server. Authorized sites can access the published data. *CreatePushFeed* operation calls the *Push* operation.

```
On CreatePushFeed(name:String,request:String):
  PushFeed(name)
  hasSemanticQuery(name,request)
  hasPush(site,name)
  call Push(name)
```

Push Operation This operation creates a change set corresponding to the pages returned by the semantic query and adds it to the push feed. Firstly, the semantic query is executed, then the patches of the pages returned by the query are extracted. These patches are added to the change set if they have not been published on this push feed yet.

```
On Push(name:String):
  ChangeSet(csid=concat(site.siteID,site . logicalClock++))
  inPushFeed(csid, name)
  let published ← { ∃x ∃y ∧ inPushFeed(y,name) ∧ hasPatch(y,x) }
  let patches ← { ∃x ∀p ∈ execQuery(name.hasSemanticQuery) ∧ onPage(x,p)}
  foreach patch ∈ {patches − published} do
    hasPatch(csid, patch)
  endfor
  previousChangeSet(csid, name.hasPushHead)
  hasPushHead(name,csid)
```

CreatePullFeed Operation As the replication of data and the communication between multi-synchronouns semantic wiki servers are made through feeds, pull feeds are created to pull changes from push feeds on remote peers to the local peer (cf figure 14). A pull feed is related to a push feed. In the sense that it is impossible to pull unpublished data.

```
On CreatePullFeed(name:String, url:URL)
   PullFeed(name);
   relatedPushFeed(name,url)
   call  Pull(name);
```

Figure 14: CreatePullFeed operation

```
On ChangeSet get(cs : ChangeSetId ,url)
    if ∃x previousChangeSet(cs,x)
       return x
    else   return null;
```

Figure 15: get a ChangeSet operation

```
On Pull(name:String):
   while ((cs ← get(name.headPullFeed,
          name.relatedPushFeed)≠ null)
   let p  ← {∃x ∧ inPushFeed(x,name)}
   if cs ∉ {p} then
       inPullFeed(cs,name)
       call  Integrate(cs)
   endif
   hasPullHead(name,cs)
  endwhile
```

Figure 16: Pull Operation

```
Integrate(cs:ChangeSet):
  foreach patch ∈ cs do
    previous(patch,patch.onPage.head)
    head(patch.onPage,patch)
    foreach op ∈ hasOperation.patch
        do call  logootIntegrate(op)
    endfor
  endfor
```

Figure 17: IntegrateOperation

Pull Operation This operation fetches for published change sets that have not pulled yet (cf figure16). It adds these change sets to the pull feed and integrate them to the concerned pages on the pulled site.

get Function This function allows to retreive a ChangeSet (cf figure 15).

Integration operation The integration of a change set is processed as follows (cf figure 17). First all the patches of the change set are extracted. Every operation in the patch is integrated in the corresponding semantic wiki page thanks to the Logoot algorithm.

DSMW algorithms ensure the causality and the CCI model (Causality, Convergence, Intention) as demonstrated in [12]

DSMW is implemented as an extension of Semantic MediaWiki [6]. Feeds, ChangeSets and Patches are represented as special semantic wiki pages. They stored in a special namespace to prevent user modification. The Logoot algorithm has been implemented in PHP and integrated in Mediawiki relying on the hook mechanism. The push, pull, createPushFeed and createPullFeed operations are available in special administration pages of MediaWiki. This extension is designed to respect the simplicity of the wikis while supporting the MS2W model and the result is an easy way to construct a P2P network of semantic wikis based on Semantic MediaWiki. DSMW was demonstrated in different conference [15, 16].

6 Conclusion

DSMW extends a semantic wiki with multi-synchronous capabilities. Multi-synchronous semantic wikis allow users to build their own cooperation networks. The construction of the collaborative community is declarative. Every user declares explicitly with whom he would like to cooperate. The replication of the wiki pages on semantic wikis servers and the synchronization periods are variant and under the control of the users. The MS2W model enhances the existing semantic wikis by supporting transactional changes and the off-line work mode. Hence, multiple dataflow oriented workflows can be supported. In addition, the model takes natural advantages of a P2P network, i.e. faults-tolerance, better scalability, infrastructure cost sharing and better performance.

References

[1] S. Auer, S. Dietzold, and T. Riechert. Ontowiki - A tool for social, semantic collaboration. In *International Semantic Web Conference*, 2006.

[2] M. Buffa, F. L. Gandon, G. Ereteo, P. Sander, and C. Faron. Sweetwiki: A semantic wiki. *Journal of Web Semantics*, 6(1):84–97, 2008.

[3] B. Du and E. A. Brewer. Dtwiki: a disconnection and intermittency tolerant wiki. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 945–952, New York, NY, USA, 2008. ACM.

[4] Git. git based wiki: http://atonie.org/2008/02/git-wiki, 2008.

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

[6] M. Krötzsch, D. Vrandecic, M. Völkel, H. Haller, and R. Studer. Semantic wikipedia. *Journal of Web Semantic*, 5(4):251–261, 2007.

[7] H. Levy. *Capability-Based Computer Systems.* Butterworth-Heinemann Newton, MA, USA, 1984.

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

[9] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, Banff, Alberta, Canada, November 2006. ACM Press.

[10] C. L. Patrick Mukherjee and A. Schurr. Piki - a peer-to-peer based wiki engine. In *Eighth International Conference on Peer-to-Peer Computing*, pages 185–186. IEEE, 2008.

[11] C. Rahhal, H. Skaf-Molli, and P. Molli. Swooki: A peer-to-peer semantic wiki. In *The 3rd Semantic Wikis workshop, co-located with the 5th Annual European Semantic Web Conference (ESWC), Tenerife, Spain*, June 2008.

[12] C. Rahhal, H. Skaf-Molli, P. Molli, and S. Weiss. Multi-synchronous collaborative semantic wikis. In *10th International Conference on Web Information Systems Engineering - WISE '09*, volume 5802 of *LNCS*, pages 115–129. Springer, October 2009.

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

[14] S. Schaffert. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. *1st Workshop on Semantic Technologies in Collaborative Applications*, 2006.

[15] H. Skaf-Molli, G. Canals, and P. Molli. Dsmw: a distributed infrastructure for the cooperative edition of semantic wiki documents. In A. Antonacopoulos, M. J. Gormish, and R. Ingold, editors, *ACM Symposium on Document Engineering*, pages 185–186. ACM, 2010.

[16] H. Skaf-Molli, G. Canals, and P. Molli. Dsmw: Distributed semantic mediawiki. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, editors, *ESWC (2)*, volume 6089 of *Lecture Notes in Computer Science*, pages 426–430. Springer, 2010.

[17] H. Skaf-Molli, C. Rahhal, and P. Molli. Peer-to-peer semantic wikis. In *DEXA'09: 20th International Conference on Database and Expert Systems Applications*, 2009.

[18] S. Staab and H. Stuckenschmidt, editors. *Semantic Web and Peer-to-peer*. Springer, 2005.

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

[20] S. Weiss, P. Urso, and P. Molli. Logoot : a scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS*. IEEE, 2009.