OntoVote: a scalable distributed votes collecting mechanism for ontology drift on P2P platform

Yanfeng Ge, Yong Yu, Xing Zhu, Shen Huang and Min Xu
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, 200030, P.R.China
{gyf, yyu, zhuxing, s_huang, xm}@sjtu.edu.cn

Abstract

Ontologies provide potential support for knowledge and content management on P2P platform. Although we can design ontology beforehand for an application, it is argued that in P2P environments, static or predefined ontology cannot satisfy the ever-changing requirements of all users. So we propose every user should make proposals for what kind of ontology is the most up to his need. Collecting all these proposals (or votes) helps to the drift of ontology. This paper presents OntoVote, a scalable distributed votes collecting mechanism based on application-level broadcast trees and describes how OntoVote can be applied to ontology drift on P2P platform by discussing several problems involved in the voting process.

1 Introduction

With the rapid development of ontology technology and P2P computing in the past few years, it has been suggested that knowledge and content management on P2P platform make use of ontologies to provide enhanced services to users [Fensel *et al.*, 2002]. To attain this object, some limited but beneficial attempts have been made and even more research plans are on the agenda. For example, the open source project Edutella [Nejdl *et al.*, 2002] aims to provide an RDF-based metadata infrastructure for P2P applications building on the JXTA framework [JXTA]. [Sato *et al.*, 2002; Nodine *et al.*, 2000; Arumugam *et al.*, 2002; *etc*] try to gather and share information and knowledge with the help of ontologies in P2P or other distributed environments.

These attempts presume that ontologies have been constructed beforehand and what they are concerned about is how to use ontologies to exchange knowledge and to enable efficient and accurate semantic search in distributed environments. In many application scenarios, such predefined ontologies cannot catch up with the ever-changing requirements of users. Instead, ontology should drift with the appearance of new application requirements. But just as [Fensel *et al.*, 2002] has stated, one cannot expect any maintenance to happen on the ontologies in P2P environments (in fact, users will not often know what is in the ontologies on the machine, let alone that they perform maintenance on them) and as a result, we must design mechanisms that allow the ontologies to update themselves, in order to cope with ontological drift. [Fensel *et al.*, 2002] has proposed several informal mechanisms that use metaphors from social science (opinion-forming, rumor-spreading, etc).

In this paper, we propose a more formal mechanism of ontology drift that is based on every user's participating in proposing the modification of ontology according to his demands of the application. To relieve the burden of users, proposals can be obtained by mining user activities (so called emergent semantics [Maedche and Staab, 2001], e.g., by mapping the modification of a directory name to the modification of a concept in ontology) or by providing users with a basic ontology together with visualization tools with which the users can make modifications easily. The modifications cause every user to hold a local ontology. These ontologies are characterized by:

- They are partly overlapped, but the same concepts may be expressed in different words.
- There are a lot of noisy semantics, owing to the wrong activities of users (e.g., a rookie of a domain may modify the domain ontology wrongly).
- Most of them cannot represent all aspects of the requirements of users.

In order to align concepts, to filter out noisy semantics, and to indicate the principal direction of the development of user requirements, *we propose these local ontologies be combined together to construct a common ontology*. With a common ontology, we can also improve the efficiency of semantic search by avoiding too many mappings between ontologies.

One possible way to combine the ontologies from all users is votes collecting: we collect the proposals of all users to make some analyses; only the semantics hold by a majority of the users (or we can set a threshold for the proportion of users) is adopted by the common ontology. The various minor semantics collected can also be treated in different ways according to its value in use, which we will describe in details afterwards.

Practically, a voting organizer (such as a chairman or a tally clerk) is needed to accomplish the voting task. This organizer can be considered as a server and serves for the common interests of a community by publishing messages to and receiving messages from all other voters. But in P2P environments, it may be hard to find any volunteer to serve the community for no evident good. Moreover, using a server to collect votes will bring about scalability and single node failure problems as discussed in many P2P researches. To get rid of such problems, we use OntoVote, a scalable distributed votes collecting mechanism based on application-level broadcast trees, to collect votes on P2P platform.

The rest of the paper is organized as follows. Section 2 describes the design of OntoVote. Section 3 applies OntoVote to the process of ontology drift. Section 4 makes a conclusion of our work and proposes our future work.

2 OntoVote

In practice, a voting process can be divided into three successive phases. The first is a *preparing phase*, notifying all participants to get ready their votes. The second is a *collecting phase*, collecting votes from all participants. And the third phase is devoted to *publishing the voting results* to all participants.

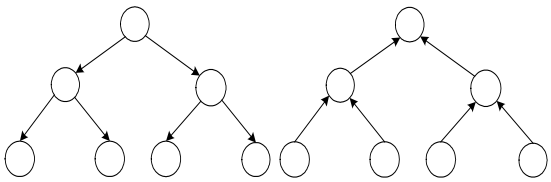

Figure 1. The phases of votes collecting. (a) Phase One: preparing phase and Phase Three: results publishing phase. (b) Phase Two: collecting phase.

OntoVote realizes the voting process in a fully decentralized manner. It uses broadcasts and a reverse operation on an application-level broadcast tree to fulfill the three phases of a voting. As in Figure 1, at the first and the third phase in (a), notifying messages and voting results are published from root to leaves. At the second phase in (b), a node on the tree first collects votes from all of its children, then it sums up the votes from the children together with its own votes, and submits the total votes to its parent.

To make use of broadcast trees, OntoVote supposes there are a large number of groups on P2P platform. Every peer can choose several groups to join in. Every group forms a reliable application-level broadcast tree with mechanisms introduced in some researches [Castro *et al*., 2002; Zhuang *et al*., 2001; *etc*], which is out of the scope of this paper (one can simply view a broadcast tree as a tree). Votes collecting happens inside a group. OntoVote provides best-quality votes collecting service (i.e., collecting exactly one copy of votes from every participant) by extending the existing application-level broadcast mechanisms.

2.1 Basic Implementation of OntoVote

In this section we introduce the basic implementation of OntoVote, mainly discussing two important procedures: *getCredential()* and *deliver(msg)*. *getCredential* validates a voting. *deliver* handles the messages on broadcast trees.

Voting Validity

Before a new round of voting in a group is initiated, the root node of the broadcast tree of the group calls *getCredential* to get a voting credential for the voting. A voting credential is granted only when a majority of group members are online so that they are capable to take participant in the voting, otherwise, the voting results plotted by a minority of group members will be invalid and misleading. Current version of OntoVote simply assumes that all votes are available, so a voting credential is always granted. This assumption is correct if all peers publish their votes to rendezvoux, which are online most of the time, just as in JXTA [JXTA].

Voting Process

The three phases of a voting process is realized in the procedure *deliver(msg)*, which is called whenever a node receives a message whose destination is the node itself. The parameter *msg* contains the received message. The pseudo code for this procedure, simplified for clarity, in shown in Figure 2.

The following variables are used in the pseudo code: *msg.type* is the message type, which may be PREPARE, SUBMIT or PUBLISH, corresponding to the three phases of a voting. *msg.groupID* indicates the group the message belongs to. *groups* is a set of groups that the node has joined in, *groups[].children* and *groups[].parent* are bi-directed links of the broadcast tree of the group. To avoid conflicts among different voting tasks, we treat each voting process as a transaction and use *transID* to distinguish them from each other globally and uniquely. *trans* is a set of transactions that the node is involved in, *trans[].votePool* is the vote pool which keeps an entry for the votes from every child (i.e., keep the votes from every child separately).

After the root node has got a voting credential, it sends PREPARE messages to its children, thus starting the new round of voting. On receiving a PREPARE message, a node sets up a new transaction environment by clearing the vote pool for the transaction (line 2). If the node is a leaf node, it sends a SUBMIT message to itself to start the collecting phase (lines 3, 4). Otherwise, it recursively passes on the message to its children (lines 5, 6).

When a node receives a SUBMIT message from a child, it adds the votes from the child to its vote pool in line 7 (If the child has submitted votes before, the old submitted votes in the vote pool will be replaced with new submitted votes). After all children of the node have submitted votes, the node adds the votes in the pool and its own votes together, then submits the total votes to its parent (lines 10 to

```
deliver(msg)
1 switch msg.type is
2 PREPARE:  empty trans[msg.tranID].votePool
3             if(isLeafNode(msg.groupID))
4               send SUBMIT message to self
5             else ∀ node in groups[msg.groupID].Children
6               send(msg, node)
7 SUBMIT:  addToPool(msg.votes, msg.source)
8           if(isRootNode(msg.groupID) AND allChildrenSubmittedVotes())
9             analyze voting results and send PUBLISH message to self
10          else if (allChildrenSubmittedVotes())
11            msg.votes=countVotes(trans[msg.tranID].votePool)+local votes
12            send(msg, groups[msg.groupID].parent)
13PUBLISH:  updateKB(msg)
14            ∀ node in groups[msg.groupID].Children
15              send(msg, node)
```

Figure 2. The implementation of deliver

12). These lines may be called for several times allowing for node failures, which will be discussed in details in the next section.

After getting votes from all of its children, the root node extracts some useful knowledge from the votes and republishes the knowledge to its children (lines 8, 9), thus any node in the group can update its local knowledge base (lines 13 to 15).

As is seen, the basic implementation of OntoVote is very simple. But if we want to collect votes in a distributed fashion reliably and efficiently on network, there are still more things to be considered.

2.2 Reliability

On account of the unreliable nature of Internet, a broadcast tree may break at any time, including during voting process. If we collect votes from rendezvoux, the rate of failure will decrease sharply, but we still cannot avoid node failures completely. Therefore, OntoVote proposes repairing the broadcast tree for the best-quality collecting purpose.

Periodically, each node in the tree sends a heart beat message to its children, if any, and the children respond with answering messages. A child suspects its parent is faulty when it fails to receive heartbeat messages and so does the parent, i.e. when two nodes lose in touch with each other, every one of them will suspect the other has failed. But in fact, any one of them may be really faulty, or none of them are faulty, but the link between them is broken.

Upon detection of the failure of its parent, a child tries to connect to a new parent. To maintain the performance of fully distributed votes collecting, the tree's balance should be approximately retained, so the new parent is chosen from the tree nodes that are nearly at the same level with the old parent in a uniformly-random way.

If node failures occur during publishing phases (i.e., the first or the third phase in Figure 1), it is a trivial task to ensure that every node on the tree would receive the published message: the new parent sends all published messages it has received or will receive to the new child, and the child either relays the messages to it children or discards the messages, according to whether or not it has received the identical messages before.

However, if node failures occur during collecting phase, things become a bit more complicated: if a node fails at the very time that some children have submitted votes to it and some not, how about these submitted votes? If the failing node has disconnected from the network, the submitted votes will be lost. To avoid losing votes, the children of the failing node can resubmit votes to a new parent. But if the failing node has not disconnected from the network or if the failing node has submitted votes to parent before its failure, straightforward resubmitting will result in redundant votes on the broadcast tree. In the rest of this section, we first put forward the repairing protocol of OntoVote for collecting phase, then we show that this protocol satisfy our best-quality collecting purpose by leaving out as few votes as possible and by avoiding counting in the same copy of votes repeatedly.

Repairing Protocol for Collecting Phase

As in Figure 3, let *FN* be a failing node from the aspect of its child *CN*. *CN* reconnects to a new parent *NPN*. We use *P[X]* to denote the parent of a node *X*. ΔT is a configurable time limit. After *CN* reconnects to *NPN*, the repairing protocol goes as follows:

(1) If *CN* has not submitted votes to *FN* yet, then *CN* will submit votes to *NPN* after it collects votes from all of its children.

Else if *CN* has submitted votes to *FN*, but the interval from the submission of *CN* to the failure of *FN* is still within the time limit ΔT, then *CN* will submit votes to *NPN* immediately.

Else *CN* submits a null vote as a placeholder to *NPN* immediately.

(2) After *NPN* adds the votes from *CN* to vote pool:

If *NPN* has never submitted votes before (that is, *NPN* is still waiting for some other children), or if *NPN* finds that the votes from *CN* are null, then stop.

Else *NPN* recounts the total votes (including that of *CN*) and resubmits the total votes to *P[NPN]*. The recounting and resubmitting process will be iterated up the tree until some ancestor of *NPN* that has not submitted votes yet.

(3) If *CN* has submitted votes to *FN* and the interval from the submission of *CN* to the failure of *FN* is still within the time limit ΔT, then *FN* will delete the entry of *CN* in its vote pool. If *FN* has also submitted votes to *P[FN]*, it will recount the total votes it has collected (excluding that of *CN*) and resubmit the total votes to *P[FN]*. The recounting and resubmitting process will be iterated up the tree until some ancestor of *FN* that has not submitted votes yet.

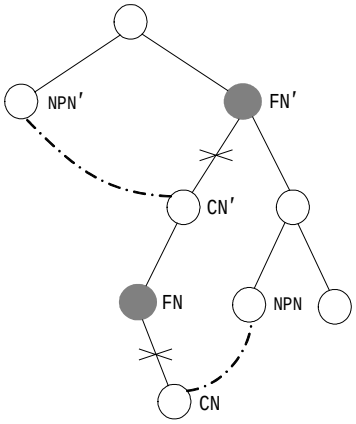

Figure 3. Repairing broadcast trees for best-quality collecting

Explanation of the Repairing Protocol

To explain the repairing protocol, we first neglect the time limit ΔT let ΔT =0). The main idea of the protocol is that when a node finds its parent is faulty, it doesn't know whether the submitted votes are lost or not, so it resubmits votes to a new parent, and at the same time, the original ancestors of the node try to eliminate the votes from the node by recounting and resubmitting the total votes in a bottom up manner. The repairing protocol is passive in that although the old submitted votes may have been relayed to many ancestors, they are rolled back to start afresh.

How well does this repairing protocol perform? In particular, can it really guarantee we collect exact one piece of votes from every node online even when several nodes of the tree fail concurrently or subsequently?

To illustrate the robustness of this protocol, assume in Figure 3, *CN* loses in touch with *FN* and reconnects to *NPN*. If *CN* has not submitted votes to *FN* yet, it is all right for *CN* to collect votes from all children and to submit the votes to *NPN*. Otherwise, *CN* resubmits votes to *NPN* and the previously submitted votes to *FN* should be eliminated thoroughly. If *FN* is not disconnected from the network, *P[FN]* may find *FN* is still alive, so *FN* and its ancestors can delete the copy of votes of *CN* from their vote pools. Assume before this deleting process is performed till some ancestor *CN'*, *CN'* finds its parent *FN'* is also faulty and reconnects to *NPN'*, then the deleting process will be continued on the new path. Meanwhile, *FN'* will start a new deleting process for the votes of *CN'*, which contains the votes of *CN*. If *FN* is disconnected from the network, then *P[FN]* will delete votes of *FN*, which also contains the votes of *CN*. Such recursive call of the deleting process ensures the old copy of the votes of *CN* is deleted completely.

Obviously, the resubmitting process and the deleting process for the votes of *CN* are executed along two different paths to root, so if the processes can reach the root, they are not likely to always arrive at the same time. To avoid losing votes or introducing redundant votes, after all children have submitted votes, the root node should wait for a period of time that is long enough for the two processes to be finished. But the problem is that while the root is waiting for the repairing processes for one node, another node may happen to fail. The passive repairing processes will be called again, despite that the votes submitted by the node may have reached the root. As a result, the root will be trapped in an ever-waiting deadlock.

We adjust the passive repairing protocol by introducing some active ingredient to avoid ever-waiting: If a node has submitted votes to its parent long before it finds the parent is dead (i.e., beyond a time limit ΔT which is much longer than the collecting time of the node), then it simply assumes that the parent has also submitted votes and there are several ancestors that have received the votes. Because the parent (if it is not disconnected from the network) and every one of the ancestors that have kept the votes longer than ΔT will try their best (just as the node itself does) to relay the votes to the root, it is not necessary for the node to resubmit votes.

With this compromised protocol, the loss of votes will still occur if several nearest ancestors of a node disconnect from the network concurrently, but the probability of such loss is greatly less than before (the above mentioned scenario).

2.3 Efficiency

Recall that before a node submits votes to its parent, it will sum up the votes in the vote pool together with its own votes. OntoVote does not export the summation method but leaves it to application level. The implementation of the method is highly related to the efficiency of OntoVote: when more and more votes are collected, message packets that encapsulate votes will become larger and larger.

Without additional disposal, the packets will overwhelm the network and the scalability of OntoVote will be no better than that of a client-server model. So we should follow several principles in the design of this method.

To understand our principles, one can view a message packet that contains votes as a sheet. Every vote has its entry on the sheet. The number of entries a sheet can hold is limited. By combining the entries on several sheets together to form a new sheet and by relaying the new sheet to parent node, a voting participant fulfils his collecting task. Here are the design principles described with the above metaphor:

- Merge identical entries. To save on the space of a sheet, votes devoted to the same candidate should be merged together, that is, each entry should maintain a counter of the votes that fall in this entry. Actually, this is the original meaning of votes counting.
- Filter minor entries. Because the number of the entries a sheet can hold is limited, one should filter out the least counting entries when the sheet overflows. The contents of the least counting entries are likely to be noisy or unimportant to most users, so it is acceptable to filter them out.
- Choose a proper-sized sheet. Each application should choose a proper-sized sheet by simulation or by probability analysis to avoid a phenomenon that we call "Entry Jolts": although some vote may be very large altogether, it happens to be filtered out every time before it can accumulate. "Entry Jolts" is determined by such factors as data makeup, data distribution on the network, filtering algorithms and sheet sizes, etc. To reduce the probability of "Entry Jolts", a large-sized sheet is preferred, but we'd better get an upper bound of the size of the sheet, size larger than which brings about no evidently-good performance but more consumption of the network bandwidth.

In addition to the implementation of the summation method, other factors such as the balance of a broadcast tree, which has been addressed above, also affect the efficiency of OntoVote. But they are beyond our considerations.

3 Application of OntoVote to Ontology Drift

In section 1, we give our thought that colleting votes from all users can drive the drift of a common ontology and in section 2, we show that it is possible to collect votes in P2P environments with OntoVote. In this section, we will try to apply OntoVote to the process of ontology drift. Our implementation of ontology drift is part of the knowledge acquisition module developed under our undergoing PICQ project. PICQ is dedicated to paper sharing on P2P platform with semantic web technologies. In PICQ, users are grouped according to research interest. Every group has a common ontology. New fields or new application requirements will introduce new concepts or attributes in the ontology. The common ontology is used to provide more powerful semantic search service.

3.1 Process of Ontology Drift

We use the following process to cope with the drift of a common ontology:

(1) When a new group is created, the creator provides a basic common ontology.

(2) When a new user joins the group, he is provided with the currently available common ontology. If the user is dissatisfied with the common ontology, he can modify it with ontology visualization tools to create a local ontology. The visualization tools can map ontology elements to application elements (e.g., directories, bookmarks, etc) to hide ontologies from elementary users. They can also provide powerful support for expert users to modify ontologies directly and easily. In any case, the modifications of the local ontology are translated into the user's votes on the common ontology. By tracking user modifications, system can also partly do the mapping between the local ontology and the common ontology.

(3) At a proper time, all votes are collected and the common ontology is modified with the voting results.

(4) After the new common ontology is published to all peers, the mapping between the local ontology and the new common ontology is adjusted again.

(5) Iterate the above process.

To put it simply, the whole is an iterated process. Let *LO* (Local Ontology) denote the local ontology of a user and let *CO* (Common Ontology) denote the common ontology of a community. *LC* is the mapping between *LO* and *CO*. The iterated process can be described as follows:

(1) *LO*=*CO*, *LC*=Identity Mapping

(2) User modifies *LO*. System records user modifications and automatically adjusts *LC*.

(3) At a proper time, system translates modifications into votes, collects all votes and modifies *CO* background.

(4) Publish *CO* to every peer and adjust *LC* again.

(5) Goto (2).

There are some issues that need to be further addressed in the management of the ontologies involved in this process, e.g., how to track user modifications? Why do we keep the mapping between the common ontology and the local ontology and how to do this mapping (Obviously, it is not enough to do the mapping between two ontologies just by tracking the changes of either ontology)? How to derive a user's modification proposals (or votes) on the common ontology from his modifications of the local ontology? What the system should do if there is a conflict of opinions? In the following sections, we will discuss these problems one by one.

3.2 Tracking User Modifications

The problem for tracking changes within ontologies or within a knowledge base has been addressed in [Kiryakov and Ognyanov, 2002]. [Kiryakov and Ognyanov, 2002] proposes using RDF statements (i.e. triples) instead of resources or literals as the smallest trackable pieces of knowledge. The two basic types of updates in a repository are addition and removal of a statement. To track series of updates that are bundled together according to the logic of the application, it uses *batch update* that works with the repository in a transactional fashion.

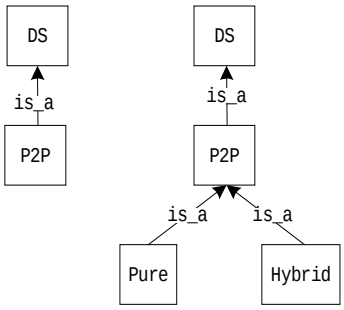

Figure 4. Two Different Local Ontologies

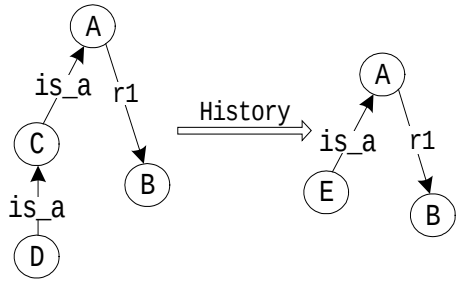

Figure 5. Tracking Modifications from Initial Common Ontology to Current Local Ontology

In regard to reflecting the modification proposals of users, we think *atomic update* (additions or removals of statements) plus *batch update* is almost appropriate and only a small change is made: because we treat modification proposals as votes and OntoVote requires identical votes to be merged, but two batch updates that represent the same proposals may not match exactly (e.g., in Figure 4, both of two users propose the sub-tree rooted at the concept "P2P" be deleted, but the two sets of removed triples can't match exactly. One set contains two more triples than the other.), so we propose the application should induce some patterns from batch updates, which reflect the intention of a user. Two batch updates are matched if and only if both their patterns and the parameters of the patterns are matched. For instance, the deleting of the two sub-trees rooted at "P2P" in Figure 4 can be matched if a pattern for the deleting of a sub-tree is defined and the root node of the sub-tree is used as a parameter of the pattern.

Our approach for tracking modifications is illustrated in Figure 5. The left structure of the figure denotes the initial common ontology that the user imported from the community; the right structure is the local ontology. A history of modifications is recorded in a way something like that of [Kiryakov and Ognyanov, 2002]:

History:
1. remove sub-tree(C): remove <C is_a A>, remove<D is_a C>
2. add <E is_a A>

3.3 Maintenance of Ontology Mapping

In P2P applications, every user should be allowed to hold a local ontology to keep his personality. He uses this local ontology to raise queries. The queries are translated into the common ontology before being sent to other peers to improve search efficiency and the recall of search results. So it is of great importance to maintain the mapping between the local ontology and the common ontology.

By tracking the modifications of the local ontology and the common ontology in step (2) and step (3) in the process of ontology drift, we can partly do the mapping between them. For instance, if a resource in either ontology is renamed, the mapping can be adjusted. But how about adding a new resource? How can we know whether or not the new resource in one ontology can be mapped to some old resource in another ontology? This problem may be solved with emergent semantics [Maedche and Staab, 2001; Fensel *et al.*, 2002]: after the user adds a new resource, he queries with this new resource for a period of time. During this period, emergent semantics helps to find out the mappings between the new resource and some other resources in the common ontology or other local ontologies (e.g. same file categorized to different concepts indicates alignment). The derived mappings are expressed with probabilities, based on the number of the instances that indicating alignment. Different peers may find different mappings, or same mappings with different probabilities. At a proper time, all new resources and the mappings among them are collected and coordinated with a new round of voting. Among the resources that can be mapped to each other, the one that wins the vote is adopted by the common ontology, the noisy semantics (votes that are below a threshold) is discarded and the rest are mapped to the one accepted. This process is something like the revision of a dictionary in social life: after a new word emerges, people use this word in their communication and every one gets a scrap of the meanings of the word (e.g., find synonyms of the word). When a dictionary is revised, all meanings of the word is collected and validated, and if necessary, the word is lexicalized.

In practice, the mapping results that are automatically obtained and maintained may not always be sound, so a semi-automatic ontology mapping mechanism is preferred, i.e., advanced users are allowed to manually correct the

mapping results before or after any round of votes collecting.

3.4 Generation of Votes

Before votes collecting, the modifications of the local ontology should be translated into the modification proposals (votes) on the common ontology. Or else, just as section 3.2 has stated, although the proposals of two users are identical, they cannot be merged.

Currently, our system generates the votes in a rather simple way, which is mainly based on the mapping between the common ontology and the local ontology. Below we discuss how our approach works in various conditions.

Firstly, if the mapping between the common ontology and the local ontology is well maintained, and the pattern of the modification has been defined by the application, then the translation is straightforward. For instance, in Figure 6, assume concept “DC” is mapped to concept “Distributed Computing” and concept “P2P” is mapped to concept “Peer-to-Peer”. If a user adds two new concepts “Pure” and “Hybrid” under concept “P2P” in his local ontology, then we think the user proposes adding these concepts under concept “Peer-to-Peer” in the common ontology; If the user deletes the tree rooted at “DC” completely, then we think the user proposes deleting the tree rooted at “Distributed Computing” completely.

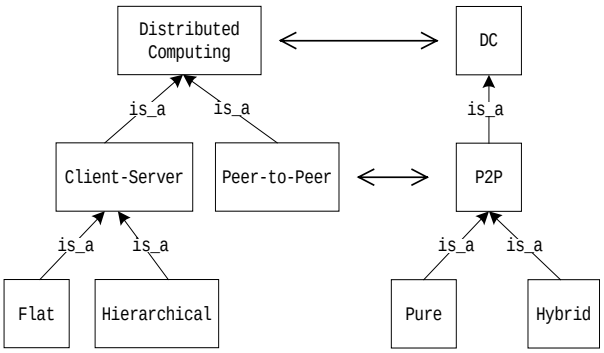

Figure 6. Common Ontology and Local Ontology

Secondly, if no appropriate mapping information is obtained, then we either ignore the modification or transform the modification into a list of sub-modifications, according to the specification of the modification pattern. For example, in Figure 6, if no corresponding concept of “DC” is found in the common ontology, then we either discard the vote or split the vote to generate a new one to delete the sub-tree rooted at the concept “peer-to-peer”.

Thirdly, for some modifications (e.g., additions of statements), the mapping information is not necessary at all, i.e., additions of statements that have no relations with other resources in the common ontology are allowed in our system.

The last but not the least, advanced users are allowed to modify the local copy of the common ontology directly, if they are pleased to do that. We also allow a user to replace his local ontology with the common ontology, if he is dissatisfied with his local ontology or he does not want his opinions diverge too much from those of the masses, thus the old modification records are cleared and the new modifications of the local ontology can be more easily translated into those of the common ontology.\

It should be noted that our system tries to translate every record in the modification history into a vote. If the modifications do not interact, this approach may work well. However, sometimes an appropriate combination of additions and removals may trigger complex "non monotonic" updates of the ontology, e.g., a user first adds “B” as a sub-concept of “A” and adds “C” as a sub-concept of “B”, and later, he finds that “C” is unnecessary, so he deletes it and adds “B” as a sub-concept of “A” directly. This sequence of modifications will be translated into a list of votes. If most members of the community believe that “B” is a sub-concept of “A”, it is likely that the last proper modification of the user will be accepted. However, if many users make the similar wrong modifications before, the wrong modifications may also be accepted. To filter out the wrong modifications, we’d better find out the last determination of the user (or the real intention of the user) before we construct a vote from a sequence of related modifications. Unfortunately, our system has not realized this object yet, and we will leave it to the future.

3.5 Resolving Conflicts

It is obvious that there are conflicts among all collected proposals, e.g., some users suggest deleting concept “Peer-to-Peer”, while others suggest adding “Pure” as a sub-concept of “Peer-to-Peer”.

One straightforward way to resolve a conflict is to choose among the conflicting proposals the one with the largest proportion. However, such simple conflict-resolving mechanism will bring about the instability of the common ontology, especially when the proportion of the adopted proposal is not overwhelming.

To understand this problem, assume there are totally 100 users in the community. At the beginning, 60 percent of them suggest adding the concept “Peer-to-Peer” (or concepts that are equivalences of “Peer-to-Peer”) to the common ontology, thus “Peer-to-Peer” is accepted in the common ontology. In the second round of voting, assume 30 users suggest deleting the concept “Peer-to-Peer” (these 30 users may come from the previously 60 percent of users, or advanced users who modify common ontology directly, or users who reimport and modify their local ontologies, etc.), while 10 users add sub-concepts “Pure” under “Peer-to-Peer” and the rest make no modifications. After the voting, if the system chooses to delete “Peer-to-Peer”, then the local ontologies that still record the addition of “Peer-to-Peer” will propose adding “Peer-to-Peer” to the common ontology again in the next round of voting. Because the proportion of this proposal is still large enough, it may be adopted again by the common ontology, which causes the instability of the common ontology.

Intuitively, We can get rid of the instability problem by tracking the history of voting. But till now, this idea has not been tested yet. Instead, we take a rather simpler measure to ease the instability problem, i.e., we set different thresholds for modifications with different patterns to be accepted in the common ontology. Because the common ontology is mainly used to improve search efficiency and the recall of search results, it is better to contain more resources than not. So we set low thresholds for additions of statements and high thresholds for deleting operations. In this way, when a conflict occurs, it is more likely that a proposal with an overwhelming vote proportion exists, and that the opposite proposals may be too trivial to bring about instability.

4 Conclusions and Future Work

Ontology drift is important in many requirement-sensitive P2P applications. We proposed collecting the modification proposals (votes) from all users to drive ontology drift. To collect votes on P2P platform, we presented OntoVote, a scalable distributed votes collecting mechanism based on application-level broadcast trees. OntoVote is reliable in that it leaves out as few votes as possible and avoids counting in the same copy of votes repeatedly. We also summarized several design principles of vote counting for OntoVote to work efficiently. And finally, we tried to apply OntoVote to the process of ontology drift with the discussion of several problems encountered in the application.

In future, we will research ontology drift further by obtaining more general modification patterns of ontology. we will also try to make the drifting process more stable. Besides, we will research some further issues of voting, such as voting security and the using of the opinions of authoritative members. There is also a problem that common ontology based on voting will neglect the views of an individual (or a small group of people) that brings real innovation and original perspectives on community's point of view. We will find out whether intercommunication between peers helps to solve this problem.

References

[Fensel *et al.*, 2002] Dieter Fensel, Steffen Staab, Rudi Studer and Frank van Harmelen. Peer-2-Peer Enabled Semantic Web for Knowledge Management. *Ontology-based Knowledge Management: Exploiting the Semantic Web*, Wiley, London, UK, 2002.

[Sato *et al.*, 2002] Hiroyuki Sato, Yutaka Abe and Atsushi Kanai. Hyperclip: a Tool for Gathering and Sharing Metadata on Users' Activities by Using Peer-to-Peer Technology. *WWW2002 Workshop on Real world RDF and Semantic web applications (2002).*

[Nodine *et al.*, 2000] Marian Nodine, Jerry Fowler, Tomasz Ksiezyk, Brad Perry, Malcolm Taylor, Amy Unruh. Active Information Gathering in Infosleuth. In *International Journal of Cooperative Information Systems* 9:1/2, 2000, pp. 3-28.

[Arumugam *et al.*, 2002] Madhan Arumugam, Amit Sheth, I. Budak Arpinar. Towards Peer-to-Peer Semantic Web: A Distributed Environment for Sharing Semantic Knowledge on the Web. *WWW2002 Workshop on Real world RDF and Semantic web applications (2002).*

[Nejdl *et al.*, 2002] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek *et al.*. EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *proc. of WWW11,* May 2002,Hawaii.

[JXTA] Project JXTA: An open, innovative collaboration. White paper, available at www.jxta.org.

[Castro *et al.*, 2002] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec and Antony Rowstron. Scribe: A Large-scale and Decentralized Application-level Broadcast Infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No, 8,* October 2002.

[Zhuang *et al.*, 2001] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz and John Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video(N OSSDAV 2001),* Port Jefferson, NY, June 2001.

[Maedche and Staab, 2001] Alexander Maedche, Steffen Staab. Ontology Learning for the Semantic Web. *IEEE Intelligent Systems*, 16(2):72-79, March/April 2001.

[Kiryakov and Ognyanov, 2002] Atanas Kiryakov and Damyan Ognyanov. Tracking Changes in RDF(S) Repositories. In *proc. of 13th International Conference on Knowledge Engineering and Knowledge Management EKAW02*, Siguenza, Spain, 1-4 Oct. 2002.