The Knowledge Engineering Review, Vol. 18:3, 257–263.
DOI: 10.1017/S0269888904000062 Printed in the United Kingdom

OntoVote: a scalable distributed vote-collecting mechanism for ontology drift on a 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, PR China; e-mail: gyf@sjtu.edu.cn, yyu@sjtu.edu.cn, zhuxing@sjtu.edu.cn, s_huang@sjtu.edu.cn, xm@sjtu.edu.cn

Abstract

Ontologies provide potential support for knowledge and content management on a P2P platform. Although we can design ontologies beforehand for an application, it is argued that in P2P environments static or predefined ontologies 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 apt to his need. Collecting all these proposals (or votes) helps the drift of ontologies. This paper introduces OntoVote, a scalable distributed vote-collecting mechanism based on application-level broadcast trees, and describes how OntoVote can be applied to ontology drift on a 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 a P2P platform make use of ontologies to provide enhanced services to users (Fensel *et al*., 2002). To attain this objective, some attempts have been made and 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, undated). Sato *et al.* (2002), Nodine *et al.* (2000), Arumugam *et al.* (2002) and others 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, ontologies should drift with the appearance of new application requirements. However, just as Fensel *et al.* (2002) have stated, one cannot expect any maintenance to happen on the ontologies in P2P environments (in fact, users will not often know what the ontologies on the machine are, let alone 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) have proposed several informal mechanisms that use metaphors from social science (opinion-forming, rumour-spreading, etc).

In this paper, we propose a more formal mechanism of ontology drift that is based on every user's participation in proposing the modification of the ontology according to his demands of the application. To relieve the burden on users, proposals can be obtained by mining user activities (so-called emergent semantics (Maedche & Staab, 2001), e.g. by mapping the modification of a

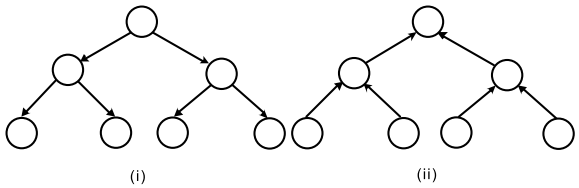

Figure 1 The phases of vote collecting: (i) preparing phase and result-republishing phase; (ii) collecting phase

directory name to the modification of a concept in the ontology) or by providing users with a basic ontology together with visualisation tools with which the users can easily make modifications. The modifications cause every user to hold a local ontology. These local ontologies are characterised thus:

- They are partly overlapped, but the same concepts may be expressed in different words.
- There are a lot of noisy semantics, owing to wrong activities by users (e.g. a rookie in 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 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 vote-collecting: we collect the proposals of all users to make some analyses; only the semantics held by a majority of users is adopted by the common ontology. The various minor semantics collected can also be treated in different ways according to value in use, which we will describe in detail afterwards.

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

2 OntoVote

Practically, an organiser (such as a chairman or a tally clerk) is needed to accomplish a voting task. The organiser can be regarded as a server and serves the community by publishing messages to and receiving messages from all 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 vote-collecting mechanism based on application-level broadcast trees, to collect votes on a P2P platform.

OntoVote divides a voting process into three successive phases. The first is the *preparing phase*, notifying all participants to ready their votes. The second is the *collecting phase*, collecting votes from all participants. The third is devoted to *republishing the voting results* to all participants.

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

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

votes from every participant under an unreliable network condition. The details about how OntoVote realises this can be found in our technical report.

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 briefly showed 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 ongoing PICQ project. PICQ is dedicated to paper sharing on a P2P platform with semantic web technologies. In PICQ, users are grouped according to their research interests. Every group has a common ontology. New fields or new application requirements will introduce new concepts in to the ontology. The common ontology is used to support more powerful semantic search services.

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 visualisation tools to create a local ontology. The visualisation 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, the 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 *CO* (Common Ontology) 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. Go to (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 should the system do if there is a conflict of opinions? In the following sections, we will discuss these problems one by one.

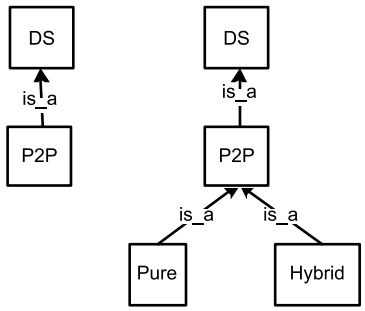

Figure 2 Two different local ontologies

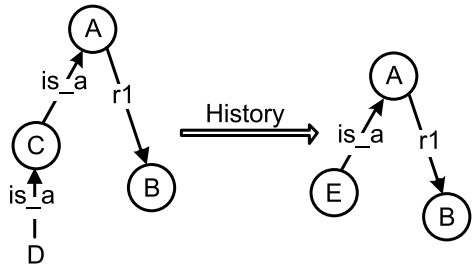

Figure 3 Tracking modifications from initial common ontology to current local ontology

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). They propose using RDF statements (i.e. triples) instead of resources or literals as the smallest trackable pieces of knowledge. The two basic types of update 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.

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 has been made. We have treated modification proposals as votes and OntoVote requires two identical votes to be merged during the collecting phase. However, two batch updates that represent the same proposals may not match exactly (e.g. in Figure 2 two users both propose the sub-tree rooted at the concept ''P2P'' be deleted, but the two sets of removed triples cannot match exactly. One set contains two more triples than the other). To get rid of this problem, we define some parameterised patterns, which actually reflect the intentions of users, for batch updates. 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 2 can now be matched according to our definition of the sub-tree deletion pattern, which is parameterised by the link from the root node of the deleted sub-tree to its parent node.

To sum up, our approach for tracking modifications can be illustrated with Figure 3. 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 as follows:

History:

1. remove sub-tree (<C is_a A>): 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 (see Section 3.1), 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 & 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 the mappings between the new resource and the other resources (e.g. same file categorised to different concepts indicates alignment). The derived mappings are expressed with probabilities, based on the number of the instances that indicate alignment. At a proper time, all new resources and mappings 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) are 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 everyone gets a scrap of the meanings of the word (e.g. finds synonyms of the word). When a dictionary is revised, all meanings of the word are collected and validated and, if necessary, the word is lexicalised.

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 vote-collecting.

3.4 Generation of votes

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

Our system generates the votes in two steps. In the first step, the records that reflect the current modification intentions of a user are selected from the modification history. In the second step, the selected records are translated into the modification proposals on the common ontology according to the mapping between the local ontology and the common ontology.

To understand the necessity of the first step, suppose a user first adds ''B'' as a sub-concept of ''A'' and adds ''C'' as a sub-concept of ''B''. Later, he finds that ''B'' is unnecessary, so he deletes it and adds ''C'' as a sub-concept of ''A'' directly. With this list of modifications, if ''B'' has not been accepted as a sub-concept of ''A'' in the common ontology yet, then the user will not want it to be added; if ''B'' has been accepted in the common ontology, then the user will want it to be deleted. Obviously, different records should be selected according to the current state of the local ontology and the common ontology. Our system does this job by finding the differences between the local ontology and the common ontology and decides which modifications bring about the differences.

In the second step, various conditions may occur. First, if the mapping between the common ontology and the local ontology is well maintained, and the pattern of the modification has been defined, then the translation is straightforward. For instance, in Figure 4, 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

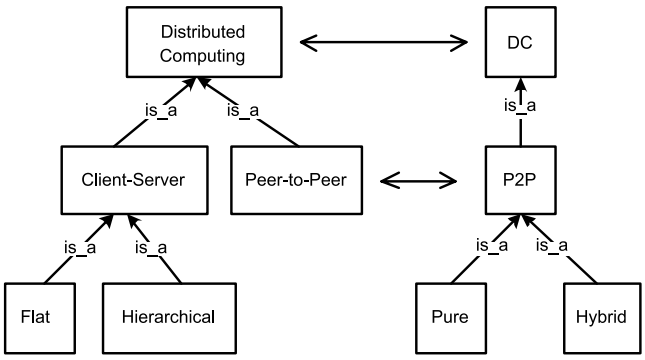

Figure 4 Common ontology and local ontology

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.

Second, 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 4, 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 that deletes the sub-tree rooted at the concept ''Peer-to-Peer''.

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

Last but not least, advanced users are allowed to modify the local copy of the common ontology directly, if they want to. We also allow a user to replace his local ontology with the common ontology, if he is dissatisfied with his local ontology or does not want his opinions to 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.

3.5 Resolving conflicts

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

Our way to resolve conflicts is as follows: first, choose among all conflicting proposals the one with the largest proportion, then exclude those that directly conflict with that selected, then select the second one from the rest of the proposals, and iterate this process until the conflicting set is empty.

Although this conflict-resolving mechanism is simple, it may bring about the instability of the common ontology. To understand this problem, assume there are in total 100 users in the community. At the beginning, 60% of them suggest adding the concept ''Peer-to-Peer'' (or concepts that are equivalent to ''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 25 users may come from the previous 60% of users, or advanced users who modify common ontology directly, or users who re-import and modify their local ontologies etc.), while 10 users add sub-concepts ''Pure'' under ''Peer-to-Peer''. After the voting, the system chooses to delete ''Peer-to-Peer'', while 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 instability in the common ontology.

Intuitively, we can avoid the instability problem by tracking the history of the common ontology. But until now this idea has not been tested. 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 deleting proposal is accepted, its opposite proposal is less likely to be accepted again in the next round of voting.

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 a P2P platform, we introduced OntoVote, a scalable distributed vote-collecting mechanism based on application-level broadcast trees. Finally, we tried to apply OntoVote to the process of ontology drift with a discussion of several problems encountered in the process.

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

References

Arumugam, M, Sheth, A and Arpinar, IB, 2002, ''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)*.

Castro, M, Druschel, P, Kermarrec, A-M and Rowstron, A, 2002, ''Scribe: a large-scale and decentralized application-level broadcast infrastructure'' *IEEE Journal on Selected Areas in Communication (JSAC)* **20**(8).

Fensel, D, Staab, S, Studer, R and van Harmelen, F, 2002, ''Peer-2–Peer enabled Semantic Web for knowledge management'' in, *Ontology-Based Knowledge Management: Exploiting the Semantic Web* Wiley.

JXTA, undated, ''Project JXTA: an open, innovative collaboration'', white paper, available at www.jxta.org.

Kiryakov, A and Ognyanov, D, 2002, ''Tracking changes in RDF(S) repositories'' *Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management EKAW02*.

Maedche, A and Staab, S, 2001, ''Ontology learning for the Semantic Web'' *IEEE Intelligent Systems* **16**(2) 72–79.

Nejdl, W, Wolf, B, Qu, C, Decker, S, Sintek, M *et al*., 2002, ''EDUTELLA: a P2P networking infrastructure based on RDF'' *Proceedings of WWW11*.

Nodine, M, Fowler, J, Ksiezyk, T, Perry, B, Taylor, M and Unruh, A, 2000, ''Active information gathering in Infosleuth'' *International Journal of Cooperative Information Systems* **9**(1/2) 3–28.

Sato, H, Abe, Y and Kanai, A, 2002, ''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)*.

Zhuang, SQ, Zhao, BY, Joseph, AD, Katz, RH and Kubiatowicz, J, June 2001, ''Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination'' *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video(N OSSDAV 2001)*.