Detecting, Managing and Querying XML Replicas and Versions in Peer-to-Peer Environments

Deise de Brum Saccol[1], Nina Edelweiss, Renata de Matos Galante[2]

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

Advisor: Nina Edelweiss; Co-advisor: Renata de Matos Galante

{deise, nina, galante}@inf.ufrgs.br

Level: PhD Candidate

Program: Pós-Graduação em Ciência da Computação - Universidade Federal do Rio Grande do Sul (PPGC/UFRGS)

Admission date: August/2004

Expected conclusion date: July/2008

Concluded tasks: qualifying exam (jan/2006) and thesis proposal (aug/2006)

Abstract: *Peer-to-Peer* (P2P) systems seek to provide sharing of computational resources, which may be duplicated or versioned over several peers. Duplicate resources (i.e. replicas) are the key to better query performance and availability. On the other hand, multiple versions can be used to support queries on the lineage of resources and the evolution of history. However, traditional P2P systems are not aware of replicas and versions, which cause complexity at the logical level and inefficiency at the physical level. To solve these problems, we propose an environment for detecting, managing and querying replicas and versions of XML documents in a P2P context. The proposed version detection mechanism is based on *Naïve Bayesian* classifiers. Thus, our approach turns the detection problem into a classification problem. In this paper, we present the results of various experiments that show that our approach produces very good results, both in terms of recall and precision measures. The proposed environment can also be used for plagiarism detection, web page ranking, and software clone identification.

Keywords: Semistructured and XML databases, temporal databases, information integration and interoperability, information retrieval, Web data services.

[1] This work has been partially supported by CNPq under grant No. 142396/2004-4, PERXML under grant No. 475.743/2004-0 and DIGITEX - CTInfo under grant No. 550.845/2005-4.

[2] This work has been partially supported by CNPq under grant No. 481516/2004-2 (Edital Universal) and Fapergs under grant No. 0412264 (Auxílio PROAPP).

1. Introduction

Peer-to-peer (P2P) systems refer to a class of applications that use distributed resources to perform tasks in a decentralized context [1]. The usability of P2P systems is mainly dependent on techniques used to find and retrieve results. However, the searching optimization faces two problems. The first problem is the existence of multiple resource representations. The existence of duplicated resources may be interesting for increasing the performance, since the user poses a query and the results are returned from a specific peer, obeying fast response time criteria. Though, to take advantage of resource replication, it is necessary to identify these multiple representations.

The second problem arises from the evolutionary behavior of some resources. The evolving issue is a fundamental aspect in any persistent information system and it is more evident in XML domain, with frequent structure and content changes. The evolution aspect must be managed to allow historical retrieving, for example by using versions. Although the version management is well known for managing co-authoring on software engineering [2], the detection issue is still a big challenge in other scenarios. For plagiarism detection, comparing file checksums is enough for detecting exact replicas, but insufficient for partial copies [3]. By considering partial copies as versions, such plagiarism can be detected. The web page ranking process can also take advantage of the detection mechanism by ranking new versions of existent top-ranked pages [4]. At last, the software clone problem that arises during the system development may have a negative impact on their maintenance [5]. By considering the clone as a resource version, such effect can be reduced.

To solve these issues, this paper proposes *DetVX*, an environment for detection, management and querying of XML replicas and versions in a P2P context. We define a suitable architecture for a P2P system to deal with replicated and versioned documents. The paper presents the activities and metadata necessary to detect, represent, manage and query those replicas and versions in this context. The main purpose is a replica and version detection mechanism, based on document similarity. The paper is organized as follows: Section 2 presents the related works. A brief description of proposed environment is discussed in Section 3. Section 4 presents the focus of our work, the replica and version detection mechanism. Some query capabilities are discussed in Section 5. Finally, concluding remarks are presented in Section 6.

2. Related Work

P2P systems introduce many problems on data management and querying, mainly because of the distribution aspect. The result quality of user queries may be measured by some metrics, such as the size of the result set, query satisfaction in the results and processing time [6]. Thus, there is usually a relation between cost and result quality that must be balanced. However, the searching optimization must deal with the existence of replicated and multiversioned resources, which is not addressed in current P2P systems.

Existent works on version management focuses on maintaining and efficiently retrieving the resource versions [7]. Some techniques were proposed for transaction time management in temporal databases [8], artifact versions in object-oriented databases [9] and change management in semistructured data [10]. Version control systems model files as text line sequences, storing the last version and using reverse editing scripts to retrieve previous versions [11]. But these systems do not preserve the original logic structure and do not support complex queries, and thus are inadequate for XML versions. These gaps are addressed in [12] and [13], respectively.

Moreover, these works focus on version management rather than detection. However, version detection is essential in our motivating application, since the anonymity/distributed nature of P2P environments prevents users from identifying resources from which the new version or replica is being created. Thus, we propose an automatic detection mechanism based on file similarity. Existent researches on change detection can be used as a basis for measuring similarity. Some approaches use *diff* algorithms to detect differences between files [14]. Nevertheless, *diff* results are a delta script with no semantic information regarding the similarity between files. Another possibility is to analyze their ordered tree representations by calculating the edit distance, i.e., the minimum cost to transform one tree into another using basic operations [15]. Also, tree edit distance results do not contain meaningful information related to the similarity level that could be used to detect resource versions.

There are several similarity functions, either for atomic values (e.g. *Levenshtein* or *Edit Distance* (Edit) [16], *Guth* [17] and *N-grams* [16]) or for documents [18][19][20]. However, these functions are often suitable for some specific requirements. For example, some XML similarity functions focus more on structure, while others focus more on content. For the version detection problem, many different features must be considered together. In this paper, we define a similarity function that considers several characteristics with different weights to achieve a more flexible approach.

Besides choosing a similarity function, the threshold value has to be defined. This is not a trivial task, even if it is automatically defined or manually chosen by a user. Some works address threshold definition [21], but there is not a widely accepted approach. Usually the similarity value is semantically poor, and different functions produce different distributions (different interpretations for the similarity values [22]). In other words, the result quality can vary among functions when a specific threshold is chosen (a threshold chosen for on set of files may not be adequate for another set of files). In order to obtain more robust methods we need to eliminate the dependence of threshold. Thus, our approach turns the version detection issue into a classification problem, for which no threshold definition is necessary. This is done by using *Naïve Bayesian* classifiers, a simple probabilistic classification with strong independence assumptions [23].

Finally, there are some works on document classification based on *Naïve Bayesian* classifiers [24]. In these works, the goal is to assign a single document to the category that is most relevant. In our proposal, the goal is to categorize pairs of files in two classes (i.e., versions and non-versions). There are a set of features that must be considered when defining the requirements for being version. We define these requirements in a similarity function and apply the classification technique for version detection.

3. *DetVX* Environment

In the proposed environment (**Det**ection of Replicas and **V**ersions of **X**ML Documents), each peer acts both as client and server. Files are stored according to the super peer architecture, that works as follows: the user poses a query on a specific peer; if this peer is not able to locally answer it, then the query is routed to the respective super peer; the super peer verifies the available peers that are able to answer it and resend it to those peers; the peers process the query and return the results to the super peer; finally, the super peer resends the results to the user.

The peer grouping criterion is based on the knowledge domain of the documents (described by an ontology). Each super peer has at least one associated ontology and it can aggregate peers with documents related to different domains. One peer that stores files related to different domains can also be linked to different super peers. There is one central entity responsible for managing the environment (quite common in P2P systems that adopt the partially centralized architecture). Each super peer contains metadata that describe information about aggregated peers and files. Metadata describe the super peers, application domains and some management information.

In order to provide the mechanism functionalities for the replica and version detection, this work proposes the architecture shown in Figure 1. The user interacts with the system through the user interface, which allows registering peers and files (using the peer manager). To connect a peer, it is necessary to choose a proper super peer, based on the file application domain (through the ontology manager). After connecting, the system verifies if the new shared files are replicas or versions of existent files already available in the network (through the replica and version manager). Finally, the user can pose queries (using the query processor). These modules are detailed in the next sections.

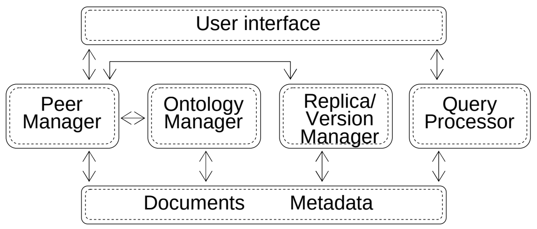

Figure 1. *DetVX* architecture

Metadata play a fundamental role in the proposed architecture and they are presented next.

3.1. Files, Documents and Metadata

File refers to a physical representation in a peer; *document* refers to a representation of an object in the real world. Thus, one document can be stored as many files, either because it is replicated or versioned. Each file has registering and modification date, local identifier (to identify a file in a peer; based on a hash function result) and global identifier (to identify a file in the entire network).

To manage this information, this approach relies on metadata, represented in XML and classified in two levels. Administrative super peer metadata describe the super peers, aggregated peers, local identifiers, hash results, registering and modification date for each file in each peer. They are accessed to verify peer changes and for maintenance at peer (re)connections. Super peer metadata specify the available replicas and versions in each super peer and the timestamps for each element in a file located in a peer. These metadata are updated whenever a new file is registered and they are extensively used for query processing. Timestamps are inferred form the file modification time in which the element is represented. Whenever a new version is detected, the timestamps need to be updated.

3.2. Peer Manager and Ontology Manager

The peer manager is responsible for peer (re)connection management and periodic verification of file changes. To check the occurrence of changes since the last connection, the system runs a hash function for each file; then the results are compared with metadata. After connecting a peer, all the registered files are available for sharing. However, P2P present a dynamic behavior of connection, i.e., the peer can be connected in a time t and disconnected in a time $t+1$. For metadata maintenance, the system periodically runs a checking service that verifies the on-line peers in a time t. Peers that have not been connected for a period of time p are removed from the administrative naming service.

The ontology manager is responsible for maintaining the ontology repository (set of OWL files) and associating ontologies to super peers. Information about the associated ontologies is stored in a XML file. Similarity techniques based on tree edit distance can be used for matching the ontologies to XML files. Load balancing must also be considered. In this work, the load balancing is done based on the number of peers. The goal is to keep the same number of peers in each super peer. The load balancing must agree with the grouping criterion. Thus, the regrouping may imply on associating ontologies to other super peers. More details about the peer manager and the ontology manager can be found in [25].

Implementation. We are currently developing a graphic tool for peer management based on JXTA platform [26]. The system will allow managing the super peers, peers and corresponding shared files. We are also working on the ontology manager, by defining and implementing a tool for matching ontologies to XML documents. By matching the ontology to a XML document, the system can connect the peer to a proper super peer that is described by a specific ontology. The matching phase considers the concept name, the structure similarity and some stemmer algorithms. The ontologies are generated from an integration process among the conceptual schemas that describe the XML documents. The ontology generation is not a contribution of our work and it is under development by other students.

4. Replica and Version Manager

The replica (duplicate) detection mechanism aims to verify if a file is an exact copy of any other file stored in any peer belonging to the same super peer network. The replica detection is done by comparing the file hash result with the hash results already stored in its super peer metadata. Two files *f1* and *f2* are replicas if: *HashFunction(f1)=HashFunction(f2).*

The version detection mechanism seeks to verify if two files are versions of the same document (based on file similarity). For measuring similarity, we define a set of attributes (i.e., features) that must be considered for being a version. The detection mechanism performs three activities:

similarity analysis (responsible for applying a similarity function to each pair of files), *classification* (responsible for detecting the versions by using *Naïve Bayesian* classifiers) and *metadata management* (not presented in this paper, but details may be found in [27]). The detection mechanism can be used for both linear and branched versioning. It only differs in the structure of the tree that is traversed during the *similarity analysis* phase [27]. Similarity analysis between files is described in the next section.

4.1. Similarity Analysis

This task is responsible for generating a similarity value for each pair of files. Three types of evolution are considered: content, structure and structure/content evolution. In our work, *structure* and *structure and content* evolution are grouped together.

Content Evolution. The following features are observed:

- *Diff* results: by using a *diff* algorithm and analyzing the input files and the delta result, we can determine the percentage of elements that have not changed in *f2*. The assumption is that the bigger percentage of unchanged elements, the larger chance the files are versions of the same document.
- Matched and unmatched elements: this feature analyses the string similarity between the elements that have changed (i.e., unmatched elements). The more similar the respective unmatched elements, the larger is the chance that the files are versions of the same document.
- Element change relevance: this feature analyses the relevance of individual changes (e.g., an unmatched element *birthdate* has more relevance than an unmatched element *address*. The smaller change relevance they present, the larger chance the files are versions of the same document.

The similarity function *simC* between two files *f1* and *f2* is defined as: $simC(f1,f2) = (w_1*F_1 + w_2*F_2 + w_3*F_3 + ... + w_n*F_n)$. Where w_n is a factor that weights the importance of a specific feature F_n. A factor may be positive or negative (if it influences the similarity growth or reduction, respectively). Considering $w_x, w_{x+1},\dots w_y$ as positive factors and $w_z, w_{z+1},\dots w_q$ as negative factors, we assume that $w_x + w_{x+1} +...+w_y = 1$ and $-1<=w_z + w_{z+1} +...+ w_q <= 0$.

The features discussed are combined to produce the following content similarity function: $simC(f1,f2) = w_1*P + w_2*S + w_3*R$. Where: *P* is the percentage of matched elements, *S* is the mean similarity of the unmatched elements and *R* is the average of domain relevances of the unmatched elements ($\{P|P \in [0,1]\}$, $\{S|S \in [0,1]\}$, $\{R|R \in [0,1]\}$). To calculate *P*, we use a function that returns the percentage of matched elements based on the *diff* result. *S* is calculated by using a (combination of) string similarity function(s) and it is defined as the average of the similarity values for the unmatched elements.

Structure and Content Evolution. In addition to *diff results* and *element change relevance* discussed in the last section, another feature is considered:

- Added and removed elements: by using a *diff* algorithm and analyzing the delta result, we can observe the number of new elements (i.e., added elements) and the number of removed elements (i.e., deleted elements) between the first and the second file. We redefine the term *matched* to refer to an element that has the same structure and content in both files Thus, the function used to compute the similarity including structure evolution is as follows: $simE(f3,f4) = simC(f3,f4) + w4*A + w5*D$. Where: *simC* is the content similarity value, *A* is the percentage of added elements and *D* is the percentage of deleted elements ($\{A|A \in [0,1]\}$, $\{D|D \in [0,1]\}$).

After measuring the file similarity, the version detection task is performed, as next described.

4.2. Version Detection (Classification)

This task is responsible for deciding if two files *f1* and *f2* are versions of the same document. The classification phase receives a set of files (training set), each labeled with a class label, and outputs a model (classifier). The classifier assigns a class label (version or non-version) to each pair of files (testing set). The results quality is measured by recall and precision, classical measures commonly used in information retrieval [19].

Implementation. Several experiments were carried out to analyze the results. The results obtained with the *Naïve Bayesian* classifier have high quality. The experiments were split in two groups: detection in content evolution and in content/structure evolution. For each group, four activities were done: data acquisition, training and testing, and results analysis. For these experiments, the training and testing data were randomly simulated (i.e., the feature values for the similarity functions were synthetically generated), in order to assess both the results quality and the classifier scalability. We have generated the feature values, as presented in Section 4.1.

We firstly considered the content evolution type. In this case, the weights were set in 0.5, 0.5 and -0.5, respectively. We ran 9 testing data iterations, with testing set size varying from 100 to 1000 pairs of files. The experiments were executed considering two scenarios for the training set (9000 pairs of files):

- Equally distributed between the classes (versions and non-versions): the obtained results show that the mean recall and the mean precision were, 92.13% and 92.4%, respectively. Even the worst case for recall (88.89% in experiment 4) and precision (87.27% in experiment 4) were still good. The classifier correctly detected over 92% of the existent versions, and over 92% of the detected versions were correctly classified;
- Unequally distributed between the classes (80% of versions and 20% of non-version): the obtained results show that the mean recall and the mean precision were, 99.83% and 91.35%, respectively. Even the worst case for recall (99.28% in experiment 7) and precision (89.13% in experiment 5) were still good. The classifier correctly detected over 99% of the existent versions, and over 91% of the detected versions were correctly classified.

Some experiments were also executed for content and structure evolution type. The results were also good (high rates for recall and precision). The experiment descriptions can be found in [27].

4.3. Version History Representation

From *n* files representing *n* versions (*n>1*), we generate a new representation that contains the document history. This representation is stored in the corresponding super peer as a new file (*H-Doc*). *H-Doc* files can be accessed for faster query processing for some queries that ask for the document history. Thus, it is not necessary to access all the versions spread over the super peer network. Timestamps (*TS* and *TE*) are responsible for validating the elements in specific versions. *TS* is initialized with the file modification date in which the element was added/updated. *TE* is initialized with *now* and is modified whenever the element is updated/removed in another version.

Implementation. We have implemented a tool named *XVersion* [28] that receives a set of document versions and outputs the *H-Doc* file, by processing the differences between the versions and using timestamps. This implementation has used the *XyDiff* algorithm [29].

5. Query Processor

After detecting replicas and versions, temporal queries can be posed on the original files (located in the peers) or on the consolidated representation (located in the super peers). To evaluate which files must be accessed to answer a query, our approach relies on metadata described in Section 3.1. Let us consider the following query example [30]: retrieve the history of an element e_j – for instance, get the history of the element *address*. To answer this query, the system searches the metadata, looking for all the versions of the element *address*. The last version of this element is represented by *TE=now*. Another possibility for this query is to check if there is a generated *H-Doc* representation for this file (*attribute HDoc="YES"*). In this case, the system can access this file in the super peer, as follows. Considering an element *e* in a document *D*, some temporal restrictions can be used based on a specific date *x* or an interval *x* and *y*. Some clauses are: *select_Before (e, x)*, *select_After (e, x)*, *select_Between (e, x, y)*, *select_Now (e)*, *select_Before (D, x)*, *select_After (D, x)*, *select_Between (D, x, y)*, and *select_Now (D)*.

Implementation. Query capabilities were implemented in our tool named *XVersion* [28], using the *Qizx/Open* library [31]. By using temporal restrictions, historical information is returned by accessing only one file. Some query examples are: 1) get the current content of the element *salary*. To answer this query, the tool searches for the elements that *TE=now*; 2) get the salaries before *10/11/2006*. To answer this query, the tool searches for the element *salary* that TS<*10/11/2006*.

6. Concluding Remarks

The problem of version and replica detection is critical in many scenarios, including software clone identification, Web page ranking, plagiarism detection, and P2P searching. In order to increase efficiency and effectiveness in such systems, this paper describes the architecture and the functionalities of *DetVX* environment. We defined a similarity function that considers several features that must be considered for being a version. The function is not restricted to a specific application and can be adapted to consider other similarity features for different scenarios. Moreover, each feature can be differently weighted, which turns our proposal into a more flexible approach. A detection mechanism based on classifiers is presented, which eliminates the hard task of threshold definition.

The version detection problem is not a new issue. Neither is the use of *Naïve Bayesian* classifiers to categorize documents. However, the use of this technique for solving the mentioned problem requires the variable definition (i.e., attributes or features) that describe each category, which usually is a hard domain-dependent task for version detection. In the response of these requirements, we proposed a similarity function and used a classification technique, which provides an accurate solution for an aged problem. The experiments produced very good results, over 90% for recall and precision rates [27].

The current state is as follows. We have already implemented *XVersion*, a tool for representing and querying document history. Basic retrieval capabilities have been implemented, allowing simple temporal queries over the historical representation. As future work, we are going to incorporate the detection mechanism in the environment. The completion of the detection mechanism will allow us to measure improvements on selected testbeds**,** including JXTA. Also, we are going to apply the mining method for document classification using testing sets derived from a particular application domain (probably from *Wikipedia*). Therefore, we will also optimize the similarity function, the weights used and the mining method. The good results are expected to be maintained in different scenarios. We are also going to explore other mining and classification techniques, such as *support vector machines*, and to extend the temporal capabilities already implemented in *XVersion* tool.

References

[1] Aberer, K.; Hauswirth, M.. An Overview on Peer-to-Peer Information Systems. In: Workshop on Distributed Data and Structures, Paris, France, 2002.

[2] Conradi, R.; Westfechtel, B.. Version Models for Software Config. Management. ACM Comput. Surv., 30(2):232–282, 1998.

[3] Chen, X., Francia, B., Li, M., McKinnon, B., Seker, A.. Shared information and program plagiarism detection. IEEE Transactions on Information Theory, v. 50, n. 7, p-1545-1551, 2004.

[4] Baeza-Yates, R., Castillo, C.. Relating Web Characteristics with Link based Web Page Ranking. In: Intl. Symposium on String Processing and Information Retrieval, Laguna de San Rafael, Chile, 2001.

[5] Ducasse, S., Niertrasz, O., Rieger, M.. On the effectiveness of clone detection by string matching. Journal of Software Maintenance and Evolution: Research and Practice, v. 18, n. 1, p. 37-58, 2006.

[6] Yang, B., Garcia-Molina, H.. Efficient Search in Peer-to-peer Networks. In: Intl. Conf. on Distributed Computing Systems, Vienna, Austria, 2002.

[7] Chien, S-Y., Tsotras, V. J., Zaniolo, C.. XML Document Versioning. SIGMOD Records, v. 30, n. 3, Sept., 2001.

[8] Ozsoyoglu, G. and Snodgrass, R.. Temporal and Real-Time Databases: A Survey. IEEE Trans. on Knowledge and Data Engineering, 7, 513-532, 1995.

[9] Katz, R.; Chang, E.. Managing Change in a Computer-Aided Design Database. In: Intl. Conf. on Very Large Data Bases, Brighton, England, 1987.
[10] Chawathe, S.S.; Abiteboul, S.; Widom, J.. Managing Historical Semistructured Data. Theory and practice of object systems, v.5, n.3, 143-162, 1999.
[11] CVS: Concurrent Versions System. Available at: http://www.nongnu.org/cvs.
[12] Chien, S.; Tsotras, V.; Zaniolo, C.; Zhang, D.. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In: Intl. Conf. on Web Information Systems Engineering, Kyoto, Japan, 2001.
[13] Wang, F.; Zaniolo, C.. An XML-Based Approach to Publishing and Querying the History of Databases. World Wide Web: Internet and Web Information Systems, v.8, n.3, 233-259, 2005.
[14] Cobena, G., Abiteboul, S.; Marian, A.. Detecting Changes in XML Documents. In: Intl. Conf. on Data Engineering, San Jose, USA, 2002.
[15] Chawathe, S. S.. Comparing Hierarchical Data in External Memory. In: Intl. Conf. on Very Large Data Bases (VLDB), Edinburgh, Scotland, 1999.
[16] Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys 33, 31–88, 2001.
[17] Guth, G.J.: Surname spellings and computerized record linkage. Historical Methods Newsletter 10, 10–19, 1976.
[18] Wan, X.; Yang, J.. Using Proportional Transportation Similarity with Learned Element Semantics for XML Document Clustering. In: Intl. Conf. on World Wide Web, Edinburgh, Scotland, 2006.
[19] Baeza-Yates, R.A.; Ribeiro-Neto, B.A.. Modern Information Retrieval. ACM Press / Addison-Wesley, 1999.
[20] Flesca, S. e Pugliese, A.. Fast Detection of XML Structural Similarity. IEEE Transactions on Knowledge and Data Engineering, 17, 160-175, 2005.
[21] Bilenko, M.; Mooney, R.; Cohen, W.; Ravikumar, P.; Fienberg, S.. Adaptive Name Matching in Information Integration. IEEE Intelligent Systems, [S.l.], v.18, n.5, p.16–23, September/October 2003.
[22] Dorneles, C. F.; Heuser, C. A.; Lima, A. E. N.; Silva, A. S.; Moura, E. S.. Measuring similarity between collections of values. In: ACM Intl. Workshop on Web Information and Data Management, Washington, DC, 2004.
[23] Langley, P.; Iba, W.; Thompson, K.. An analysis of Bayesian classifiers. In: National Conference on Artificial Intelligence, San Jose, USA, 1992.
[24] Wang. Y.; Hodges, J.; Tang, B.. Classification of Web Documents Using a Naive Bayes Method. In: IEEE Intl. Conf. on Tools with Artificial Intelligence. IEEE Computer Society Washington, DC, USA, 2003.
[25] Saccol, D.B.; Edelweiss, N.; Galante, R.M.. Detecting, Managing and Querying Replicas and Versions in a Peer-to-Peer Environment. In: IEEE TCSC Doctoral Symposium, in conjunction with the IEEE Intl. Symposium on Cluster Computing and the Grid, Rio de Janeiro, Brazil, 2007.
[26] Gong, L.. JXTA: A Network Programming Environment. IEEE Internet Computing, 5(3):88–95, May/June 2001.
[27] Saccol, D. B.; Edelweiss, N.; Galante, R. M.; Zaniolo, C.. XML Version Detection. In: ACM Symposium on Document Engineering, Winnipeg, Canada, 2007 (to appear).
[28] Saccol, D. B.; Giacomel, F. S.; Galante, R. M.; Edelweiss, Nina.. Grouping and Querying XML Document Versions in a P2P Environment (in Portuguese). In: XML: Aplicações e Tecnologias Associadas, Portugal, 2007.
[29] Cobena, G.; Abiteboul, S.; Marian, A.. Detecting Changes in XML Documents. In: Intl. Conf. on Data Engineering, San Jose, USA, 2002.
[30] Saccol, D. B.; Edelweiss, N.; Galante, R. M.; Zaniolo, C.. Managing XML Versions and Replicas in a P2P Context. In: Intl. Conf. on Software Engineering and Knowledge Engineering, Boston, USA, 2007.
[31] Qizx/Open library. Available at: http://www.xfra.net/qizxopen