Managing XML Versions and Replicas in a P2P Context

Deise de Brum Saccol1[1,2], Nina Edelweiss[2], Renata de Matos Galante[2,4], Carlo Zaniolo[3]
[2]*Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS)*
Av. Bento Gonçalves, 9500, Porto Alegre, RS, Brazil
{deise, nina, galante}@inf.ufrgs.br
[3]*Computer Science Department – University of California (UCLA)*
405 Hilgard Avenue, Los Angeles, CA, United States
zaniolo@cs.ucla.edu

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. We also show that the proposed environment can also be used for plagiarism detection, web page ranking, and software clone identification.

1. Introduction

P2P systems refer to a class of applications that use distributed resources to perform tasks in a decentralized context. Each participant acts both as client and server, providing access to resources through direct and decentralized communication [1]. Their usability is mainly dependent on techniques used to find and retrieve results. The results quality may be measured by metrics such as the result set size, query satisfaction, and processing time [2].

However, searching for resources in P2P systems must deal with two important issues: the first is the existence of replicas and the second is the presence of multiple versions of a resource. Replicas (i.e. multiple representations) are important for performance optimization: when the user poses a query then the results must be returned from peers that best satisfy performance and fast response time requirements. To take advantage of resource replication it is necessary to detect these replicas; otherwise, redundant results at a high processing cost are returned to the user.

The second problem arises from the evolving behavior of some resources, which is a fundamental aspect in persistent information systems. This feature is even more evident in XML domain, with frequent structure and content changes. The evolution aspect must be managed to allow historical analysis for dynamic resources.

The applications of the version concept are many and diverse, for instance the management of the co-authoring software, as studied in [3]. However, past approaches focus on centralized management and truly P2P distributed management still represents a difficult challenge. In P2P systems, versioning techniques must consider that versions and replicas may be spread over several peers. In such context, detecting duplicates and versions is mainly useful for query optimization. To address this issue, our paper proposes *DetVX*, an environment for the detection, management and querying of XML replicas and versions.

The main contributions of this paper are:

- A brief environment specification to detect, manage and query replicas and versions in a P2P environment;
- A replica and linear version detection mechanism based on *hash* functions and document similarity;
- A temporal XML model, based on *diff* algorithms and timestamps, for representing versioned resources and supporting basic temporal queries.

The paper is organized as follows: Section 2 presents related works. Section 3 briefly describes the proposed environment. Section 4 discusses the replica and version manager module; a similarity function is presented for version detection in content and structural evolutions. Query capabilities are presented in Section 5. Section 6 highlights other applications that may use our mechanism. Section 7 presents conclusions and future work.

[1] This work has been partially supported by CNPq under grant No. 142396/2004-4, Capes under grant No. 1451/06-5, PERXML under grant No. 475.743/2004-0 and DIGITEX - CTInfo under grant No. 550.845/2005-4.

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

2. Related Work

There has been some recent works on temporal XML models [5][6], extensions to its query languages [7], temporal libraries [8] and version control [4]. However, version control systems model files as text line sequences, storing the last version and using reverse editing scripts to retrieve previous versions [21]. These systems do not preserve the logic structure of the original file and do not support complex queries, and thus are inadequate to support XML versions. These gaps are addressed in some works, such as [9][10] and [11][12], respectively.

Previous works focus on version management rather than version detection (i.e. the creation of a new version from an old one). 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. Moreover, existent replica detection proposals focus on identifying multiple representations of the same object in the real world [13], which may have content or structure differences. However, our work considers a replica as an identical copy of a XML file.

To address this issue, we propose a detection mechanism based on file similarity. There is some research on change detection that can be used as a basis for measuring similarity. Some approaches use *diff* algorithms to detect differences between files [14][15]. 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 tree using basic operations [16][17].

Diff algorithms can be used to detect differences and, in a certain way, a similarity value between files. However, *diff* results are a delta script with no semantic information regarding the similarity between documents. Also, the tree edit distance results do not contain valuable information related to the similarity level that could be used to detect resource versions. Our work focuses on this gap and proposes an environment for detecting and managing replicas and versions of XML documents in a P2P context.

Many applications may use version detection mechanisms. For plagiarism detection, comparing file checksums is enough for detecting exact replicas, but insufficient for partial copies [22][23]. 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 [25]. At last, the software clone problem that arises during the development of systems may have a negative impact on their maintenance [24]. The proposed mechanism can help to detect such clones.

3. DetVX Environment

DetVX is an environment for detecting and managing replicas and versions of XML documents in a P2P context [26]. *DetVX* is based on a super peer architecture [19]. Super peers are responsible for receiving the query and resending it to aggregated peers and other super peers. Peers must (re)connect in super peers in order to share their files. Shared XML files are related to a knowledge domain, used as a peer grouping criterion in super peers. An ontology is used to represent the knowledge domain [18]. Super peers are managed by the administrative super peer, as depicted in Figure 1.

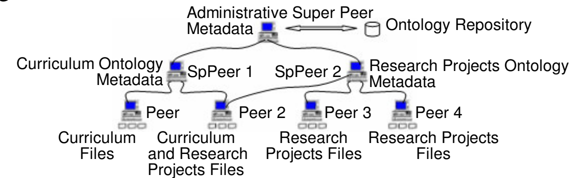

Fig. 1. *DetVX* Environment

Files may be duplicated or versioned over the super peer network. To provide the functionalities for replica and version detection, this work proposes the following modules. The *peer manager* is responsible for (re)connecting peers and periodically verifying modifications in shared files. The *ontology manager* maintains the ontology repository and associates ontologies to super peers. The *replica and version manager* identifies and manages document replica and versions. The *query processor* is responsible for verifying the query domain and rotating queries to peers. Metadata play a fundamental role and are detailed in Section 3.1. In this paper, we do not detail the peer and ontology managers. More details may be found in [26].

3.1 Document and Metadata Representation

The term *file* refers to a physical representation stored in a peer; *document* refers to the representation of an object in the real world. In other words, one document can be stored as many files, either because it is replicated or versioned. A file has a registering and a modification time. The modification time is considered to define the file order over time. Local (`fileID`) and global identifiers (`GFID`) are used to identify a file in a peer and in a specific location in the network, respectively [26]. Documents also have identifiers (`docID`) and they are used to identify versions and replicas of the same object in the real world.

To manage identifiers and other relevant information, the approach relies on the extensive use of metadata. Metadata are represented as XML files and classified in two levels, as shown in Figure 1. In this paper, only super peer metadata are considered. Super peer metadata basically specify the available versions and replicas in a specific super peer (`superPeerId`), and the corresponding timestamps for each element (`timeStart: TS, timeEnd: TE`) that is found in certain file (`fileID`) in a peer (`peerID`), as shown in Listing 1.

```
1  <Metadata superPeerId="SP1">
2    <document docID="D1" fileID="F7" HDoc="YES">
3      <version versionID="1" peerID="P1" registeringTime="10/10/2005"
4         modificationTime="08/08/2004" duplicate="no"
5         hashResult="d49622ddab3733549e54749755fd52b5">
6        <element name="author" TS="08/08/2004" TE="10/15/2004"/>
7        <element name="address" TS="08/08/2004" TE="10/15/2004"/></version>
8      <version versionID="2" peerID="P2" registeringTime="11/20/2005"
9         modificationTime="10/16/2004" duplicate="yes"
10        hashResult="7c00bb062edc60fa548729a3d55c04fd">
11       <locationDuplicate>Peer 3</locationDuplicate>…</version>
12 </Metadata>
```

Listing 1. Super Peer Metadata

Each element has two timestamps inferred from the modification time of the file in which the element is contained. Super peer metadata information is updated whenever a new file is registered into a peer and is extensively used during querying process.

4. Replica and Version Manager

This module is responsible for detecting replicas and versions and representing the history in a new structure, called *H-Doc* file.

4.1 Detection Mechanism

To solve the detection problem, a first approach is to look for replicas and versions in the local peer. If they are not found, the detection is executed in the next peer of the super peer network. The detection mechanism is executed whenever a file is registered or updated in a peer. When a file is removed, only the metadata need to be updated. A peer modification checking service is responsible for periodically watching the peer and notifying its super peer whenever a change is detected.

The replica detection mechanism aims to verify if a file is a copy of any other file stored in any peer belonging to the same super peer network. In our work, a duplicate (or replica) is defined as an identical copy of a XML file. The replica detection is done by comparing the file hash result with all 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 aims to verify if a modified file is a version of any other file stored in any peer belonging to the same super peer network. Since this work assumes the linear versioning approach, this activity will compare the candidate file only with the last file versions available in the super peer network.

There are two types of evolution that are considered:

- Content: <x>A St, 7</x> ⇒ <x>B St, 8</x>
- Structure and content: <x>A St, 7</x> ⇒ <y>B St</y> <z>8</z>

In this proposal, version detection is based on file similarity. The general idea is that two files with high similarity are considered two versions of the same document; two different documents, otherwise.

Let's first consider the content evolution type.

4.1.1 Content Evolution

Suppose two files, *f1* and *f2*, shown in Listing 2.

```
<employee>                               <employee>
  <name>Marcos</name>                      <name>Marcos</name>
  <hiringDt>10/10/03</hiringDt>            <hiringDt>10/10/03</hiringDt>
  <job>engineer</job>                      <job>manager</job>
  <salary>3700</salary>                    <salary>4900</salary>
  <address>7 St</address>                  <address>7 St</address>
  <phone>65982541</phone>                  <phone>65982541</phone>
</employee>                              </employee>
```

Listing 2. XML Files

In order to evaluate the similarity between these files, some features are observed:

- ***Diff* results:** the root element in both files has six child elements. Using a *diff* algorithm, the differences between the files are detected. As Listing 3 shows, the content of the elements *salary* and *job* do not match in the second file. In other words, 67% of the original elements kept unchanged in the second file.

The assumption here is the following: the bigger percentage of matched elements, the bigger chance the files are versions of the same document.

```
<delta> <Deleted update="yes"  pos="0:0:3:0">3700</Deleted>
        <Deleted update="yes" pos="0:0:2:0">engineer</Deleted>
        <Inserted update="yes"  pos="0:0:2:0">manager</Inserted>
        <Inserted update="yes" pos="0:0:3:0">4900</Inserted> </delta>
```

Listing 3. *Diff* result[2] for files *f1* and *f2*

- **Matched and unmatched elements:** We consider the term *matched* to refer to an element that has the same content in both files (for example, *name*); *unmatched*, otherwise (for example, *salary*). Let's take a look at the unmatched elements *salary* and *job*. Using a (combination of) string similarity function(s), we calculate a value that demonstrates how similar the unmatched elements are. The more similar the respective unmatched elements, the bigger chance the files are versions of the same document.

- **Element change relevance:** Another important issue is the relevance of individual changes. Some domain concepts can change more frequently than others. Let's suppose that we have an *address* element. Two different addresses can easily refer to the same person; however, two different birthdates suggest that we are analyzing two different objects in the real world. In other words, the change relevance is differently weighted for different concepts. We assume different weights, such as *high* (1), *medium* (0.5) and *low* (0). The average of weighted relevances is used to calculate file similarity. The smaller change relevance they present, the bigger chance the files are versions of the same document.

Based on the previous discussions, 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 $w_x, w_{x+1},…w_y$ as positive factors and $w_z, w_{z+1},…w_q$ as negative factors, we assume that $w_x + w_{x+1} +...+w_y = 1$ and $0<= w_z + w_{z+1} +...+w_q <= 1$.

In our approach, three features are considered to produce the following content evolution 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 (defined by the system administrator). *P* and *S* factors (w_1 and w_2, respectively) are positive values (the greater these values, the more similar the files) and *R* factor (w_3) is a negative value (the smaller this value, the less relevance the change and the more similar the files). The factors ($w_1, w_2, ..., w_n$) must be defined based on the importance of the three features in

[2] We are currently using *XyDiff* implementation [14], but the architecture allows changing to other *diff* algorithms.

specific applications/domains and recall/precision measures [30].

The intervals of the defined variables are defined as: $\{P|P \in [0,1]\}$, $\{S|S \in [0,1]\}$, $\{R|R \in [0,1]$. Analyzing the minimum e maximum values of *P*, *S* and *R*, and the sum restrictions for positive and negative factors, we conclude that the similarity function produces a value *simC* that ranges from -1 to 1, i.e. $\{simC|simC \in [-1, 1]\}$.

To calculate *P*, we use a function *calcP* that returns the percentage of matched elements based on the *diff* result. *S* is calculated by using a (combination of) string similarity function(s) *(StrSim())* and it is defined as the average of unmatched elements (*ue*) similarity values. The function is defined in more details as follows:

$$simC(f1,f2) = w_1 * calcP(diff(f1,f2)) + w_2 * \frac{\sum_{x=1}^{t} StrSim(ue1_x, ue2_x)}{t} - w_3 * \frac{\sum_{x=1}^{t} R(ue_x)}{t}$$

As depicted in Figure 2(a), the similarity function values are not uniformly distributed. To uniformly distribute the values, we sort and map the *m* similarity function results into *n* classes. The mapping, represented in a transformation table, categorizes *m/n* members in each class. Since we have 100 different similarity values, this transformation generates *0.01*m* members in each class.

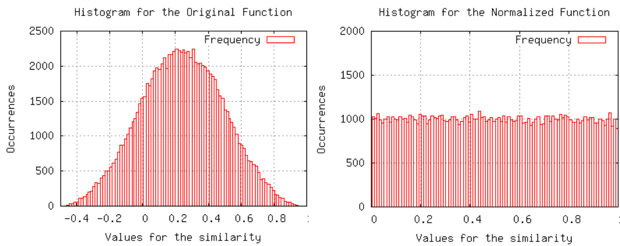

Fig. 2. Similarity Function Values (a) and (b)

Figure 2(b) shows the distribution of the mapped uniform transformation. We generated 1.000.000 values according to the original similarity function, using 0.5, 0.5 and -0.5 as the weight values, and grouped them into 100 classes. These classes were mapped to values $\in [0,1]$, in order to uniformly distribute the function values. To ensure that the mapping is correct, we generated more 100.000 values and mapped them to this table.

After producing the similarity values, a threshold is used to detect versions based on them. The threshold generation is an ongoing work and it is not detailed in this paper. Further study is still needed to assess which threshold is better respect to precision and recall.

4.1.2 Structure and Content Evolution

Suppose two files, *f3* and *f4*, shown in Listing 4.

```
<employee>
  <name>Marcos</name>
  <hiringDt>10/10/03</hiringDt>
  <job>engineer</job>
  <salary>3700</salary>
  <phone>65982541</phone>
</employee>
```

```
<employee>
  <name>Marcos</name>
  <salary>4500</salary>
  <address>7 St</address>
  <phone>65982541</phone>
</employee>
```

Listing 4.XML Files

In order to evaluate the similarity between these files, the discussions about *diff results* and *element change relevance* in the last sub-section are still valid. Another feature is also observed:

- **Added and removed elements:** using a *diff* algorithm, the differences between the files are detected. Analyzing the files and the *diff* results, we can see that the *f4* has added one element (*address*) and has removed two elements (*job* and *hiringDt*). Let's refer *added* to the elements in the first situation and *deleted* to the elements in the second situation. These concepts are similar to the ideas presented in [29], which consider *plus*, *minus* and *common* elements for measuring similarity between a document and a DTD.

We consider the term *matched* to refer to an element that has the same structure and content in both files (for example, *name* and *phone*); *unmatched*, for those elements that the content has changed (for example, *salary*). Similar to the ideas presented for the content evolution, the following features are considered to produce the structure evolution similarity function:

$$simE(f3,f4) = simC(f3,f4) + w_4*A + w_5*D$$

Where: *simC* is the content similarity value, *A* is the percentage of added elements and *D* is the percentage of deleted elements. *A* and *D* factors (w_4 and w_5, respectively) are negative values (the smaller these values, the more similar the files).

The intervals of the defined variables are defined as: $\{A|A \in [0,1]$, $\{D|D \in [0,1]$. Analyzing the minimum e maximum values of *simC*, A, *D*, and the sum restrictions for positive and negative factors, we conclude that the similarity function produces a value *simE* that $\in [-3, 2]$.

To calculate *A*, we use a function *calcA* that returns the percentage of added elements, based on the *diff* result. To calculate *D*, we use a function *calcD* that returns the percentage of removed elements, based on the *diff* result.

$$simE(f3,f4) = simC(f3,f4) - w_4*calcA(diff(f3,f4)) - w_5*calcD(diff(f3,f4))$$

The similarity values are not uniformly distributed. Similarly, the process detailed in the previous section is applied on the results to uniform these values. Also, the threshold process presented in Section 4.1.1 is still valid.

Whenever a new version or replica is detected, the timestamps described in the super peer metadata need to be updated. Metadata updating is described in [26].

4.2 A Consolidated Historical Representation

After detecting the versions, the system stores them in a new physical file, which contains the entire history of a document. The document history is named *consolidated historical representation* and represented in *H-Doc* files. *H-Doc* files are stored in the respective super peer where the original versions are registered. Timestamps are responsible for validating data in specific versions. *H-Doc* representations are generated only for frequently accessed and evolved files. The goal is to provide faster query processing for queries that ask historical retrieval.

The *H-Doc* generation process is detailed in [27]. Listing 6 shows the *H-Doc* file generated for Listing 4. Consider that *f3* and *f4* have *01/01/2004* and *01/01/2005 as m*odification times, respectively.

```
<employee TS=”01/01/2004 TE=NOW”>
        <name TS=”01/01/2004” TE=”NOW”>Marcos</name>
        <hiringDt TS=01/01/2004 TE=”12/31/2004”>10/10/03</hiringDt>
        <job TS=”01/01/2004” TE=”12/31/2004”>engineer</job>
        <salary TS=”01/01/2004” TE=”12/31/2004”>3700</salary>
        <salary TS=”01/01/2005” TE=”NOW”>4500</salary>
        <phone TS=”01/01/2004” TE=”NOW”>65982541</phone>
        <address TS=”01/01/2005” TE=”NOW”>7 St</address>
<employee>
```

Listing 5. *H-Doc File*

In *DetVX* environment, the generation of the *H-Doc* file is done by *XVersion* tool, a currently implementation work [27], based on *diff* algorithms and timestamps.

5. Query Processor

After detecting replicas and versions, temporal queries may be posed on the original files located in the peers or on the historical representation stored in the super peers.

5.1 Querying the Original Files

To evaluate which files must be accessed to answer a query, our approach relies on metadata described in Section 3.1. The query submission works as follows: the user poses a query in a specific peer (named *querying peer*). This query belongs to a specific domain. Looking at the super peer metadata, it is possible to see how to access the history or versions of an element or document.

Considering the super peer metadata described in Listing 1, some temporal retrieving examples are described below:

1. Retrieve the version v_i of an element e_j – for instance, get the first version (`versionID="1"`, line 3) of the element *author* (`element name="author"`, line 6). By searching the version number represented in metadata, the system can verify that the first version of the queried element is found in *peer* 1 (`peerID="P1"`, line 3) located at *super peer* 1 (`superPeerId="SP1"`, line 1). Thus, the system must access this file and return the results.
2. 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 (`versionID`) of the element *address* (`element name="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"`, line 2). In this case, the system can access this file in the super peer, as described in the next section.

5.2 Querying the H-Doc File

Consider a document D as a *n-tuple* $D = (root, e_1, e_2,..., e_n)$ and an element e in this document as a 3-tuple $E=(TS, TE, <content>)$, where TS and TE denote the timestamps. Temporal restrictions are applied based on a specific date x or on an interval x and y ($x<y$). Some temporal clauses are:

1. *Select_Before (E, x)*: returns the elements e that are valid in *H-Doc* file before x (elements whose $TS < x$);
2. *Select_After (E, x)*: returns the elements e that are valid in *H-Doc* file after x (elements whose $TE>x$);
3. *Select_Between (E, x, y)*: returns the elements e that are valid in *H-Doc* file between x and y (elements whose $TS<=y$ and $TE>=x$);
4. *Select_Now (E)*: returns the elements e that are valid in *H-Doc* file in current time (elements whose $TE=now$);

The same clauses are defined for retrieving entire documents, such as *Select_Before (D, x)*, *Select_After (D, x)* and others. Query capabilities based on *XQuery* language [20] have been implemented in our tool named *XVersion.* This tool generates the *H-Doc* document and allows basic temporal queries over the historical file. More details about *XVersion* may be found in [27].

6. Other Applications

This paper focuses on version and replica detection problem in P2P systems. Although this is the motivating scenario for our system and experiments, we expect that our proposal can be used in other applications, such as:

▪ **Web page ranking**: ranking methods usually involve the location and frequency of keywords in a web page. Search engines verify if the searched keywords appear close to the page top (headline or in the first few paragraphs). Frequency is also considered by analyzing how often keywords appear in relation to other words in a web page [25]. Another factor that may be considered for ranking is the incoming link degree (i.e. the number of links that point out to a page p). However, new p versions may have a small incoming link degree, mainly because of the pages that were pointing to p are not aware of the new version. In such context, version and replica detection may be useful for ranking new versions even if they have low incoming degrees.

▪ **Plagiarism detection:** Digital files may be easily copied, either partially or completely. One way to detect plagiarism is by comparing file checksums, which is simple and suffices for reliably detecting exact copies. However, detecting partial copies is more complicated [22]. By using the mechanism proposed in this paper, similar files are identified. The threshold definition must be in accordance to such application. For instance, partial copies must be identified with a low threshold, whereas complete copies must be detected with a higher threshold.

▪ **Software clone identification**: replicated code can arise during the development and evolution of software systems and it has a negative impact on their maintenance. The detection gets difficult mainly because of small differences, such as reformatting, code and variable name changes [24]. Existent detection mechanisms usually rely on the use of a parser, but this approach is dependent on the programming language syntax. The classical plain-text representation of code is convenient for programmers but requires parsing to uncover the deep structure of the program. Representing code in a structured format, as XML documents [28], permits easy specification of numerous software-engineering analyses by leveraging on the abundance of XML tools and techniques. In this context, the proposed mechanism may be used for software clone detection.

7. Concluding Remarks

This paper focused on detection and management of XML replicas and versions in P2P contexts. The relevance of such problem is quite evident in many scenarios, such as plagiarism detection, web page ranking, software clone identification, assuring link permanence in Web documents, and enhancing search in P2P systems. To increase efficiency and effectiveness in such systems, this paper briefly described the proposed architecture and functionalities of the *DetVX* environment.

We have proposed a simple structure for representing metadata which can be used for managing and querying the available files. A document similarity function used as the basic idea in the detection mechanism was also described. The proposal requires no intervention by the user. The user is only requested to update the document and register the file; the system detects prior versions or duplicates, generates identifiers and manages all the related metadata.

The current state of the project 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 *DetVX* environment. The completion of the detection mechanism will allow us to measure improvements on selected testbeds**,** including JXTA [31]. Results will be presented in the conference.

References

1. Aberer, K. and Hauswirth, M.. An Overview on Peer-to-Peer Information Systems. Workshop on Distributed Data and Structures, Paris, France, 2002.
2. Yang, B. and Garcia-Molina, H.. Efficient Search in Peer-to-Peer Networks. In: Proceeding of the Intl. Conf. on Distributed Computing Systems, Vienna, Austria, 2002.
3. Westfechtel, B., Munch, B. P., and Conradi, R. A Layered Architecture for Uniform Version Management. IEEE Trans. Software Eng., 27(12):1111–1133, 2001.
4. Chien, S-Y., Tsotras, V. J., Zaniolo, C. (2001). XML Document Versioning. SIGMOD Records, Vol. 30 Number 3, Sept.
5. Su, H., Kramer, D., Chen, L., Claypool, K. T., Rundensteinrer, E. A.. XEM: Managing the Evolution of XML Documents. Proc. of 11th Intl. Work. on Res. Issues in Data Engineering, Heidelberg, 2001.
6. Grandi, F. and Mandreoli, F.. The Valid Web: an XML/XSL Infrastructure for Temporal Management of Web Documents. Proc. of Advances in Information Systems, 2000.
7. Gao, D. and Snodgrass, R.T.. Temporal Slicing in the Evaluation of XML Queries. Proc. of Very Large Database Systems, 2004.
8. Wang, F. and Zaniolo, C.. Representing and Querying the Evolution of Databases and their Schemas in XML. In Workshop on Web Engineering, SEKE, San Francisco, USA, 2003.
9. Chien, S.; Tsotras, V.; Zaniolo, C. and Zhang, D.. Storing and Querying Multiversion XML Documents using Durable Node Numbers. Proc. of the 2nd Intl. Conf. on Web Information Systems Engineering, 1, 232-241, vol.1, 2001.
10. Grandi, F., Mandreoli, F., Tiberio, P.. Temporal Modeling and Management of Normative Documents in XML Format. Data & Knowledge Engineering, v. 54, n. 3, p. 327-354, Sept., 2005.
11. Vagena, Z. and Tsotras, V.. Path-Expression Queries over Multiversion XML Documents. Proc. of Intl. Workshop on the Web and Databases, 49-54, 2003.
12. Wang, F. and Zaniolo, C.. An XML-Based Approach to Publishing and Querying the History of Databases. World Wide Web: Internet and Web Information Systems, 2005.
13. Weis, M. and Naumann, F.. Detecting Duplicates in Complex XML Data. Proc. of the 22nd Intl. Conf. on Data Engineering, 2006.
14. Cobena, G., Abiteboul, S. and Marian, A.. Detecting Changes in XML Documents. Proc. of 18th Intl. Conf. on Data Engineering, 41-52, 2002.
15. Wang, Y., DeWitt, D. J., Cai, J. (2003). X-Diff: An Effective Change Detection Algorithm for XML Documents. Intl. Conf. on Data Engineering, 519-530.
16. Chawathe, S.S.. Comparing Hierarchical Data in External Memory. Proc. of the 25th Intl. Conf. on Very Large Data Bases, Morgan Kaufmann Publishers Inc., 90-101, 1999.
17. Wan, X. and Yang, J.. Using Proportional Transportation Similarity with Learned Element Semantics for XML Document Clustering. WWW '06: Proc. of the 15th Intl. Conf. on World Wide Web, ACM Press , 961-962, 2006.
18. Peres, A., Lopes, M., Corcho, O.. Ontological Engineering: with Examples from the Areas of knowledge Management, e-Commerce and Semantic Web. Springer, 1st edition, 2004.
19. Schollmeier, R.. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architetures and Applications. Proc. of the 1st Intl. Conference on Peer-to-Peer Computing, 27-29, Linköping, Sweden. IEEE Computer Society 2001.
20. XQuery 1.0: An XML Query Language. W3C Proposed Recommendation. Available at: http://www.w3.org/TR/xquery.
21. CVS: Concurrent Versions System. Available at: http://www.nongnu.org/cvs.
22. Schleimer, S., Wilkerson, D., Aiken, A.. Winnowing: Local Algorithms for Document Fingerprinting. Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, San Diego, California, p. 76-85, 2003.
23. 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.
24. 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.
25. Baeza-Yates, R., Castillo, C.. Relating Web Characteristics with Link based Web Page Ranking. Proc. of the 8th Intl. Symposium on String Processing and Information Retrieval, 2001.
26. Saccol, D.B., Edelweiss, N., Galante, R.M.. Detecting, Managing and Querying Replicas and Versions in a Peer-to-Peer Environment. In: 1st IEEE TCSC Doctoral Symposium, in conjunction with the 7th IEEE Intl. Symposium on Cluster Computing and the Grid, Rio de Janeiro, 2007 (to appear).
27. Saccol, D. B.; Giacomel, F. S.; Galante, R. M.; Edelweiss, Nina.. Grouping and Querying XML Document Versions in a Peer-to-Peer Environemnt (in Portuguese). In: Actas do XATA-XML: Aplicações e Tecnologias Associadas, Lisboa, 2007.
28. Badros, G. J.. JavaML: A Markup Language for Java Source Code. In Proc. of the 9th Intl. Conf. on the World Wide Web, Amsterdam, 2000.
29. Bertino, E., Guerrini G., Mesiti, M.. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. Information Systems, v. 29, n. 1, Special issue on web data integration, p. 23-46, 2004.
30. Baeza-Yates, R.A., Ribeiro-Neto, B.. A. Modern Information Retrieval. ACM Press / Addison-Wesley, 1999.
31. Gong, L.. JXTA: A Network Programming Environment. IEEE Internet Computing, 5(3):88–95, May/June 2001.