Concept and Architecture of a Pervasive Document Editing and Managing System

Stefania Leone
University of Zurich, Switzerland
Department of Informatics
leone@ifi.unizh.ch

Thomas B. Hodel
University of Zurich, Switzerland
Department of Informatics
hodel@ifi.unizh.ch

Harald Gall
University of Zurich, Switzerland
Department of Informatics
gall@ifi.unizh.ch

ABSTRACT
Collaborative document processing has been addressed by many approaches so far, most of which focus on document versioning and collaborative editing. We address this issue from a different angle and describe the concept and architecture of a pervasive document editing and managing system. It exploits database techniques and real-time updating for sophisticated collaboration scenarios on multiple devices. Each user is always served with up-to-date documents and can organize his work based on document meta data. For this, we present our conceptual architecture for such a system and discuss it with an example.

Categories and Subject Descriptors
C.2.4 Distributed Systems **[Computer-Communication Networks]:** Computer System Organization, Distributed Systems, *Distributed Applications*

General Terms
Management, Measurement, Documentation, Economics, Human Factors

Keywords
Pervasive Document Editing and Management System, Computer Supported Collaborative Work (CSCW), Collaborative Document

1. INTRODUCTION
Text documents are a valuable resource for virtually any enterprise and organization. Documents like papers, reports and general business documentations contain a large part of today's (business) knowledge. Documents are mostly stored in a hierarchical folder structure on file servers and it is difficult to organize them in regard to classification, versioning etc., although it is of utmost importance that users can find, retrieve and edit up-to-date versions of documents whenever they want and, in a user-friendly way.

1.1 Problem Description
With most of the commonly used word-processing applications documents can be manipulated by only one user at a time: tools for pervasive collaborative document editing and management, are rarely deployed in today's world. Despite the fact, that people strive for location- and time- independence, the importance of pervasive collaborative work, i.e. collaborative document editing and management is totally neglected. Documents could therefore be seen as a vulnerable source in today's world, which demands for an appropriate solution: The need to store, retrieve and edit these documents collaboratively anytime, everywhere and with almost every suitable device and with guaranteed mechanisms for security, consistency, availability and access control, is obvious.

In addition, word processing systems ignore the fact that the history of a text document contains crucial information for its management. Such meta data includes creation date, creator, authors, version, location-based information such as time and place when/where a user reads/edits a document and so on. Such meta data can be gathered during the documents creation process and can be used versatilely. Especially in the field of pervasive document management, meta data is of crucial importance since it offers totally new ways of organizing and classifying documents: On the one hand, the user's actual situation influences the user's objectives. Meta data could be used to give the user the best possible view on the documents, dependent of his actual information. On the other hand, as soon as the user starts to work, i.e. reads or edits a document, new meta data can be gathered in order to make the system more adaptable and in a sense to the users situation and, to offer future users a better view on the documents.

As far as we know, no system exists, that satisfies the aforementioned requirements. A very good overview about real-time communication and collaboration system is described in [7]. We therefore strive for a pervasive document editing and management system, which enables pervasive (and collaborative) document editing and management: users should be able to read and edit documents whenever, wherever, with whomever and with whatever device.

In this paper, we present collaborative database-based real-time word processing, which provides pervasive document editing and management functionality. It enables the user to work on documents collaboratively and offers sophisticated document management facility: the user is always served with up-to-date documents and can organize and manage documents on the base of meta data. Additionally document data is treated as 'first class citizen' of the database as demanded in [1].

1.2 Underlying Concepts
The concept of our pervasive document editing and management system requires an appropriate architectural foundation. Our concept and implementation are based on the TeNDaX [3] collaborative database-based document editing and management system, which enables pervasive document editing and managing.

TeNDaX is a Text Native Database eXtension. It enables the storage of text in databases in a native form so that editing text is finally represented as real-time transactions. Under the term 'text editing' we understand the following: writing and deleting text

SIGDOC'05, September 21–23, 2005, Coventry, United Kingdom.

(characters), copying & pasting text, defining text layout & structure, inserting notes, setting access rights, defining business processes, inserting tables, pictures, and so on i.e. all the actions regularly carried out by word processing users. With 'real-time transaction' we mean that editing text (e.g. writing a character/word) invokes one or several database transactions so that everything, which is typed appears within the editor as soon as these objects are stored persistently. Instead of creating files and storing them in a file system, the content and all of the meta data belonging to the documents is stored in a special way in the database, which enables very fast real-time transactions for all editing tasks [2].

The database schema and the above-mentioned transactions are created in such a way that everything can be done within a multi-user environment, as is usual done by database technology. As a consequence, many of the achievements (with respect to data organization and querying, recovery, integrity and security enforcement, multi-user operation, distribution management, uniform tool access, etc.) are now, by means of this approach, also available for word processing.

2. APPROACH

Our pervasive editing and management system is based on the above-mentioned database-based TeNDaX approach, where document data is stored natively in the database and supports pervasive collaborative text editing and document management.

We define the pervasive document editing and management system, as a system, where documents can easily be accessed and manipulated everywhere (within the network), anytime (independently of the number of users working on the same document) and with any device (desktop, notebook, PDA, mobile phone etc.).

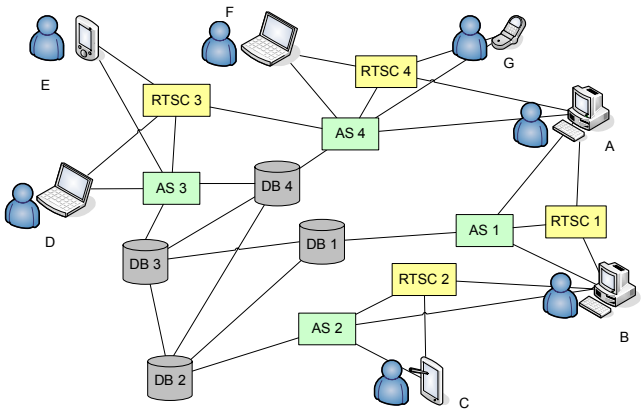

Figure 1. TeNDaX Application Architecture

In contrast to documents stored locally on the hard drive or on a file server, our system automatically serves the user with the up-to-date version of a document and changes done on the document are stored persistently in the database and immediately propagated to all clients who are working on the same document. Additionally, meta data gathered during the whole document creation process enables sophisticated document management. With the TeXt SQL API as abstract interface, this approach can be used by any tool and for any device.

The system is built on the following components (see Figure 1): An editor in Java implements the presentation layer (A-G in Figure 1). The aim of this layer is the integration in a well-known word-processing application such as OpenOffice.

The business logic layer represents the interface between the database and the word-processing application. It consists of the following three components: The application server (marked as AS 1-4 in Figure 1) enables text editing within the database environment and takes care of awareness, security, document management etc., all within a collaborative, real-time and multi-user environment. The real-time server component (marked as RTSC 1-4 in Figure 1) is responsible for the propagation of information, i.e. updates between all of the connected editors.

The storage engine (data layer) primarily stores the content of documents as well as all related meta data within the database Databases can be distributed in a peer-to-peer network (DB 1-4 in Figure 1)..

In the following, we will briefly present the database schema, the editor and the real-time server component as well as the concept of dynamic folders, which enables sophisticated document management on the basis of meta data.

2.1 Application Architecture

A database-based real-time collaborative editor allows the same document to be opened and edited simultaneously on the same computer or over a network of several computers and mobile devices. All concurrency issues, as well as message propagation, are solved within this approach, while multiple instances of the same document are being opened [3]. Each insert or delete action is a database transaction and as such, is immediately stored persistently in the database and propagated to all clients working on the same document.

2.1.1 Database Schema

As it was mentioned earlier that text is stored in a native way. Each character of a text document is stored as a single object in the database [3]. When storing text in such a native form, the performance of the employed database system is of crucial importance. The concept and performance issues of such a text database are described in [3], collaborative layouting in [2], dynamic collaborative business processes within documents in [5], the text editing creation time meta data model in [6] and the relation to XML databases in [7].

Figure 2 depicts the core database schema. By connecting a client to the database, a *Session* instance is created. One important attribute of the Session is the *DocumentSession*. This attribute refers to *DocumentSession* instances, which administrates all opened documents. For each opened document, a *DocumentSession* instance is created. The *DocumentSession* is important for the real-time server component, which, in case of a

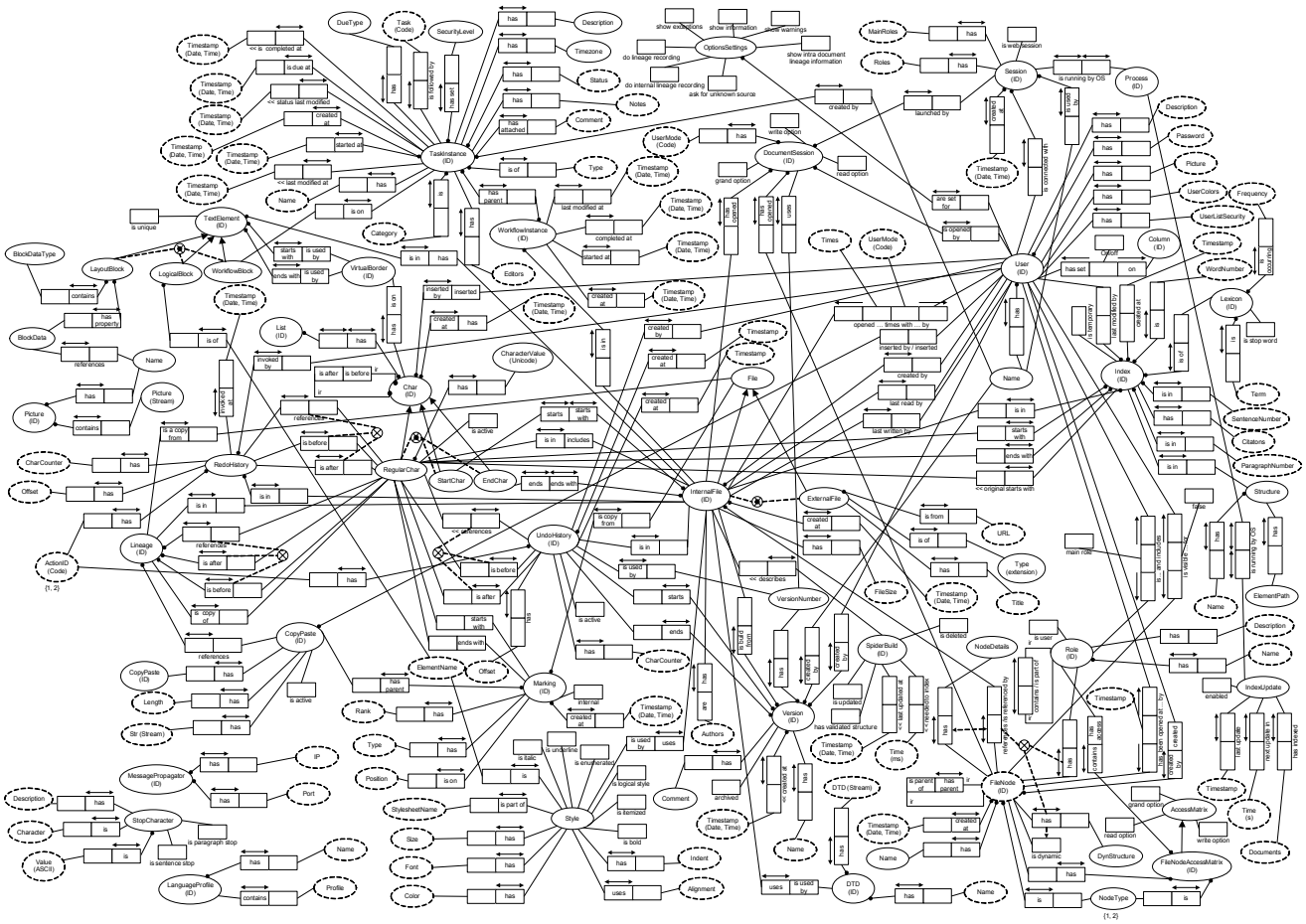

Figure 2. TeNDaX Database Schema (Object Role Modeling Diagram)

change on a document done by a client, is responsible for sending update information to all the clients working on the same document. The *DocumentId* in the class *DocumentSession* points to a *FileNode* instance, and corresponds to the ID of the opened document. Instances of the class *FileNode* either represent a folder node or a document node. The folder node corresponds to a folder of a file system and the document node to that of a file. Instances of the class *Char* represent the characters of a document. The value of a character is stored in the attribute *CharacterValue*. The sequence is defined by the attributes *After* and *Before* of the class *Char*. Particular instances of *Char* mark the beginning and the end of a document. The methods *InsertChars* and *RemoveChars* are used to add and delete characters.

2.1.2 Editor

As seen above, each document is natively stored in the database. Our editor does not have a replica of one part of the native text database in the sense of database replicas. Instead, it has a so-called image as its replica. Even if several authors edit the same text at the same time, they work on one unique document at all times. The system guarantees this unique view.

Editing a document involves a number of steps: first, getting the required information out of the image, secondly, invoking the corresponding methods within the database, thirdly, changing the image, and fourthly, informing all other clients about the changes.

2.1.3 Real-Time Server Component

The real-time server component is responsible for the real-time propagation of any changes on a document done within an editor to all the editors who are working or have opened the same document.

When an editor connects to the application server, which in turn connects to the database, the database also establishes a connection to the real-time server component (if there isn't already a connection). The database system informs the real-time server component about each new editor session (session), which the real-time server component administrates in his *SessionManager*. Then, the editor as well connects to the real-time server component. The real-time server component adds the editor socket to the client's data structure in the *SessionManager* and is then ready to communicate.

Each time a change on a document from an editor is persistently stored in the database, the database sends a message to the real-time server component, which in turns, sends the changes to all the

editors working on the same document. Therefore, a special communication protocol is used: the update protocol.

Update Protocol

The real-time server component uses the update protocol to communicate with the database and the editors. Messages are sent from the database to the real-time server component, which sends the messages to the affected editors. The update protocol consists of different message types. Messages consist of two packages: package one contains information for the real-time server component whereas package two is passed to the editors and contains the update information, as depicted in Figure 3.

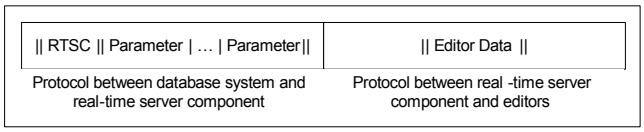

Figure 3. Update Protocol

In the following, two message types are presented:

```
||u|sessionId,...,sessionId|||editor data||
```

u: update message, sessionId: Id of the client session

With this message type the real-time server component sends the editor data package to all editors specified in the sessionId list.

```
||ud|fileId|||editor data||
```

ud: update document message, fileId: Id of the file

With this message type, the real-time server component sends the editor data to all editors who have opened the document with the indicated file-Id.

Class Model

Figure 4 depicts the class model as well as the environment of the real-time server component. The environment consists mainly of the editor and the database, but any other client application that could make use of the real-time server component can connect.

ConnectionListener: This class is responsible for the connection to the clients, i.e. to the database and the editors. Depending on the connection type (database or editor) the connection is passed to an *EditorWorker* instance or *DatabaseMessageWorker* instance respectively.

EditorWorker: This class manages the connections of type 'editor'. The connection (a socket and its input and output stream) is stored in the *SessionManager*.

SessionManager: This class is similar to an 'in-memory database': all editor session information, e.g. the editor sockets, which editor has opened which document etc. are stored within this data structure.

DatabaseMessageWorker: This class is responsible for the connections of type 'database'. At run-time, only one connection exists for each database. Update messages from the database are sent to the *DatabaseMessageWorker* and, with the help of additional information from the *SessionManager*, sent to the corresponding clients.

ServiceClass: This class offers a set of methods for reading, writing and logging messages.

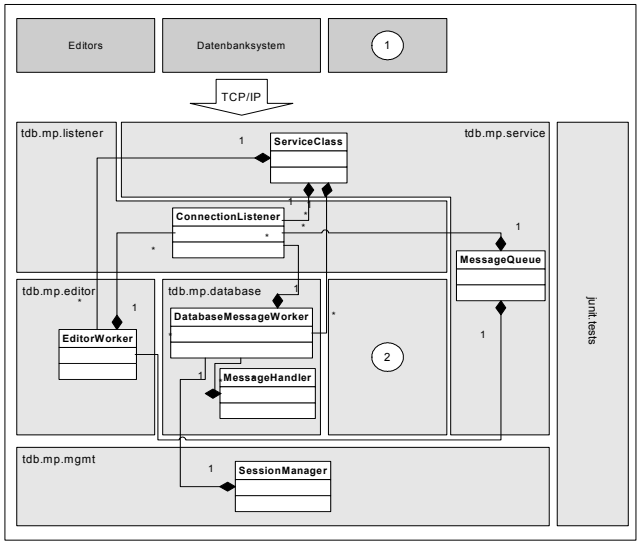

Figure 4. Real-Time Server Component Class Diagram

2.1.4 Dynamic Folders

As mentioned above, every editing action invoked by a user is immediately transferred to the database. At the same time, more information about the current transaction is gathered.

As all information is stored in the database, one character can hold a multitude of information, which can later be used for the retrieval of documents. Meta data is collected at character level, from document structure (layout, workflow, template, semantics, security, workflow and notes), on the level of a document section and on the level of the whole document [6].

All of the above-mentioned meta data is crucial information for creating content and knowledge out of word processing documents.

This meta data can be used to create an alternative storage system for documents. In any case, it is not an easy task to change users' familiarity to the well known hierarchical file system. This is also the main reason why we do not completely disregard the classical file system, but rather enhance it. Folders which correspond to the classical hierarchical file system will be called "static folders". Folders where the documents are organized according to meta data, will be called "dynamic folders". As all information is stored in the database, the file system, too, is based on the database.

The dynamic folders build up sub-trees, which are guided by the meta data selected by the user. Thus, the first step in using a dynamic folder is the definition of how it should be built. For each level of a dynamic folder, exactly one meta data item is used to. The following example illustrates the steps which have to be taken in order to define a dynamic folder, and the meta data which should be used.

As a first step, the meta data which will be used for the dynamic folder must be chosen (see Table 1): The sequence of the meta data influences the structure of the folder. Furthermore, for each meta data used, restrictions and granularity must be defined by the user; if no restrictions are defined, all accessible documents are listed. The granularity therefore influences the number of sub-folders which will be created for the partitioning of the documents.

As the user enters the tree structure of the dynamic folder, he can navigate through the branches to arrive at the document(s) he is looking for. The directory names indicate which meta data determines the content of the sub-folder in question. At each level, the documents, which have so far been found to match the meta data, can be inspected.

Table 1. Defining dynamic folders (example)

Level	Meta data	Restrictions	Granularity
1	*Creator*	Only show documents which have been created by the users "Leone" or "Hodel" or "Gall"	One folder per creator
2	*Current location*	Only show documents which where read at my current location	One folder per task status
3	*Authors*	Only show documents where at least 40% was written by user 'Leone'	Each 20% one folder

Ad hoc changes of granularity and restrictions are possible in order to maximize search comfort for the user. It is possible to predefine dynamic folders for frequent use, e.g. a location-based folder, as well as to create and modify dynamic folders on an ad hoc basis. Furthermore, the content of such dynamic folders can change from one second to another, depending on the changes made by other users at that moment.

3. VALIDATION

The proposed architecture is validated on the example of a character insertion. Insert operations are the mostly used operations in a (collaborative) editing system. The character insertion is based on the TeNDaX Insert Algorithm which is formally described in the following. The algorithm is simplified for this purpose.

3.1 Insert Characters Algorithm

The symbol *c* stands for the object "character", *p* stands for the previous character, *n* stands for the next character of a character object c and the symbol *l* stands for a list of character objects.

$$c = character$$

$$p = previous\ character$$

$$n = next\ character$$

$$l = list\ of\ characters$$

The symbol c_1 stands for the first character in the list *l*, c_i stands for a character in the list *l* at the position *i*, whereas *i* is a value between 1 and the length of the list *l*, and c_n stands for the last character in the list *l*.

$$c_1 = first\ character\ in\ list\ l$$

$$c_i = character\ at\ position\ i\ in\ list\ l$$

$$c_n = last\ character\ in\ list\ l$$

The symbol β stands for the special character that marks the beginning of a document and ε stands for the special character that marks the end of a document.

$$\beta = beginning\ of\ document$$

$$\varepsilon = end\ of\ document$$

The function *startTA* starts a transaction.

$$startTA = start\ transaction$$

The function *commitTA* commits a transaction that was started.

$$commitTA = commit\ transaction$$

The function *checkWriteAccess* checks if the write access for a document session *s* is granted.

$$checkWriteAccess(s) = check\ if\ write\ access\ for\ document\ session\ s\ is\ granted$$

The function lock acquires an exclusive lock for a character *c* and returns 1 for a success and 0 for no success.

$$lock(c) = acquire\ the\ lock\ for\ character\ c$$

$$success : return\ 1,\ no\ success : return\ 0$$

The function *releaseLocks* releases all locks that a transaction has acquired so far.

$$releaseLocks = release\ all\ locks$$

The function *getPrevious* returns the previous character and *getNext* returns the next character of a character *c*.

$$getPrevious(c) = return\ previous\ character\ of\ character\ c$$

$$getNext(c) = return\ next\ character\ of\ character\ c$$

The function *linkBefore* links a preceding character *p* with a succeeding character *x* and the function *linkAfter* links a succeeding character *n* with a preceding character *y*.

$$linkBefore(p,x) = link\ character\ p\ to\ character\ x$$

$$linkAfter(n,y) = link\ character\ n\ to\ character\ y$$

The function *updateString* links a character *p* with the first character c_1 of a character list *l* and a character *n* with the last character c_n of a character list *l*

$$updateString(l, p, n) = linkBefore(p\ c_1) \wedge linkAfter(n, c_n)$$

The function *insertChar* inserts a character *c* in the table *Char* with the fields *After* set to a character *p* and *Before* set to a character *n*.

$$insertChar(c, p, n) = linkAfter(c,p) \wedge linkBefore(c,n) \wedge linkBefore(p,c) \wedge linkAfter(n,c)$$

The function *checkPreceding* determines the previous character's CharacterValue of a character *c* and if the previous character's status is active.

$$checkPreceding(c) = return\ status\ and\ CharacterValue\ of\ the\ previous\ character$$

The function *checkSucceeding* determines the next character's CharacterValue of a character *c* and if the next character's status is active.

checkSucceeding(c) = return status and CharacterValue of the next character

The function *checkCharValue* determines the CharacterValue of a character *c.*

checkCharValue(c) = return CharacterValue of character c

The function *sendUpdate* sends an update message (*UpdateMessage*) from the database to the real-time server component.

sendUpdate(UpdateMessage)

The function *Read* is used in the real-time server component to read the *UpdateMessage*.

Read(UpdateInformationMessage)

The function AllocatEditors checks on the base of the *UpdateMessage* and the *SessionManager*, which editors have to be informed.

AllocateEditors(UpdateInformationMessage, SessionManager) = returns the affected editors

The function *SendMessage(EditorData)* sends the editor part of the *UpdateMessage* to the editors

SendMessage(EditorData)

In TeNDaX, the Insert Algorithm is implemented in the class method *InsertChars* of the class *Char* which is depicted in Figure 2. The relevant parameters for the definitions beneath, are introduced in the following list:

- nextCharacterOID: OID of the character situated next to the string to be inserted
- previousCharacterOID: OID of the character situated previously to the string to be inserted
- characterOIDs (List): List of character which have to be inserted

Thus, the insertion of characters can be defined stepwise as follows:

Start a transaction.

startTA

Select the character that is situated before the character that follows the string to be inserted.

$$getPrevious(nextCharacterOID) = PrevChar(prevCharOID) \Leftarrow \Pi_{After}\, \vartheta_{OID = nextCharacterOID}(Char))$$

Acquire the lock for the character that is situated in the document before the character that follows the string which shall be inserted.

lock(prevCharId)

At this time the list characterOIDs contains the characters c_1 to c_n that shall be inserted.

$$characterOIDs=\{ c_1, \ldots, c_n \}$$

Each character of the string is inserted at the appropriate position by linking the preceding and the succeeding character to it.

For each character c_i of characterOIDs:

insertChar(c_i, p, n)

Whereas $c_i \in \{ c_1, \ldots, c_n \}$

Check if the preceding and succeeding characters are active or if it is the beginning or the end of the document.

$$checkPreceding(prevCharOID) = IsOK(IsActive, CharacterValue) \Leftarrow \Pi_{IsActive,\, CharacterValue}\, (\vartheta_{OID = nextCharacterOID}(Char))$$

$$checkSucceeding(nextCharacterOID) = IsOK(IsActive, CharacterValue) \Leftarrow \Pi_{IsActive,\, CharacterValue}\, (\vartheta_{OID = nextCharacterOID}(Char))$$

Update characters before and after the string to be inserted.

updateString(characterOIDs, prevCharOID, nextCharacterOID)

Release all locks and commit Transaction.

releaseLocks

commitTA

Send update information to the real-time server component

sendUpdate(UpdatenMessage)

Read update message and inform affected editors of the change

Read(UpdateMessage)

Allocate Editors(UpdateMessage, SessionManager)

SendMessage(EditorData)

3.2 Insert Characters Example

Figure 1 gives a snapshot the system, i.e. of its architecture: four databases are distributed over a peer-to-peer network. Each database is connected to an application server (AS) and each application server is connected to a real-time server component (RTSC). Editors are connected to one or more real-time server components and to the corresponding databases.

Considering that editor A (connected to database 1 and 4) and editor B (connected to database 1 and 2) are working on the same document stored in database 1. Editor B now inserts a character into this document. The insert operation is passed to application server 1, which in turns, passes it to the database 1, where an insert operation is invoked; the characters are inserted according to the algorithm discussed in the previous section. After the insertion, database 1 sends an update message (according to the update protocol discussed before) to real-time server component 1 (via AS 1). RTCS 1 combines the received update information with the information in his SessionManager and sends the editor data to the affected editors, in this case to editor A and B, where the changes are immediately shown.

Occurring collaboration conflicts are solved and described in [3].

4. SUMMARY

With the approach presented in this paper and the implemented prototype, we offer real-time collaborative editing and management of documents stored in a special way in a database. With this approach we provide security, consistency and availability of documents and consequently offer pervasive document editing and management. Pervasive document editing and management is enabled due to the proposed architecture with the embedded real-

time server component, which propagates changes to a document immediately and consequently offers up-to-date documents. Document editing and managing is consequently enabled anywhere, anytime and with any device.

The above-descried system is implemented in a running prototype. The system will be tested soon in line with a student workshop next autumn.

REFERENCES

[1] Abiteboul, S., Agrawal, R., et al.: “The Lowell Database Research Self Assessment.” Massachusetts, USA, 2003.

[2] Hodel, T. B., Businger, D., and Dittrich, K. R.: “Supporting Collaborative Layouting in Word Processing.” IEEE International Conference on Cooperative Information Systems (CoopIS), Larnaca, Cyprus, IEEE, 2004.

[3] Hodel, T. B. and Dittrich, K. R.: "Concept and prototype of a collaborative business process environment for document processing." Data & Knowledge Engineering 52, Special Issue: Collaborative Business Process Technologies(1): 61-120, 2005.

[4] Hodel, T. B., Dubacher, M., and Dittrich, K. R.: “Using Database Management Systems for Collaborative Text Editing.” ACM European Conference of Computer-supported Cooperative Work (ECSCW CEW 2003), Helsinki, Finland, 2003.

[5] Hodel, T. B., Gall, H., and Dittrich, K. R.: “Dynamic Collaborative Business Processes within Documents.” ACM Special Interest Group on Design of Communication (SIGDOC) , Memphis, USA, 2004.

[6] Hodel, T. B., R. Hacmac, and Dittrich, K. R.: “Using Text Editing Creation Time Meta Data for Document Management.” Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, Springer Lecture Notes, 2005.

[7] Hodel, T. B., Specker, F. and Dittrich, K. R.: “Embedded SOAP Server on the Operating System Level for Ad-hoc Automatic Real-Time Bidirectional Communication.” Information Resources Management Association (IRMA), San Diego, USA, 2005.

[8] O’Kelly, P.: “Revolution in Real-Time Communication and Collaboration: For Real This Time.” Application Strategies: In-Depth Research Report. Burton Group, 2005.