

# E-Breaker: Flexible, Distributed Environment for Collaborative Authoring

Nelson Baloian<sup>1</sup>, Francisco Claude<sup>3</sup>, Roberto Konow<sup>2</sup>, Sebastian Kreft<sup>1</sup>  
<sup>1</sup>*Department of Computer Science, Universidad de Chile, Santiago, Chile*  
*{nbaloian, skreft}@dcc.uchile.cl*

<sup>2</sup>*Facultad de Ingeniería, Universidad Diego Portales, Santiago, Chile*  
*roberto.konow@mail.udp.cl*

<sup>3</sup>*David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada*  
*fclaude@cs.uwaterloo.ca*

## Abstract

*This paper presents a system called E-Breaker for supporting small and medium size group authoring of any kind of documents following a regular structure. The system supports a decentralized model of development, thus not requiring a central repository. A set of rules for content ownership maintains the synchronization of the work among all members of the developing team which can work on- or off-line. It allows fine-grained locking of documents' content.*

**Keywords:** CSCW, Collaborative Authoring.

## 1. Introduction

In the last years, the traditional working style of people depending on computing resources to do their work has dramatically changed due to the influence of the recent development of mobile computing devices and wireless networks. The concept of “workstation” is being less used and today it is common to find people working anywhere, anytime not necessarily attached to a specific location or time of the day. This working style has been named as “nomadic computing” by some authors and according to [1] the future of the personal computing is on cell phone-like computing devices. According to [2] the number of people working out of an office has grown by 35% since the year 2000. In the past, it was common that the working computer would be fixed at the working place like the office, and working teams would meet on a regular basis in that working place. This facilitated the use of central repositories to support the collaborative work synchronization since they will have access to it. Nowadays people do not work creating documents and/or programs on a computer attached to a certain physical place. Consequently, meetings for coordinating work frequently do not take place in a predetermined place, or at predetermined time. We believe that there are many unexplored applications of

collaborative work in nomadic systems. This opens a wide and exciting area of research, in our case, focused in the distributed confection of structured documents. For example, consider the task of writing an article: Three or four researchers are discussing about a new idea they would like to present as a paper. After that, they open their laptops and begin to write the outline of the document, where they explain briefly different segments of the document. After that, they assign different sections of the document to each author, and they start working on the task of writing. A wireless network may be available, allowing them to work in a synchronous way, in other case, they can work asynchronously through email or using a pen-drive. Subsequently they separate and work independently. The members, or maybe only a few of them, may meet again; new members who join the project will need to merge the work. We propose that a system suitable for this paradigm should meet the following requirements:

**Work on a peer-to-peer architecture without having a central repository:** As we want to support people who may start a new development without previous preparation, a central repository may not be always available for all members at that moment. Because of this, every member of the developing group should have a copy of the project, as updated as possible, even when working alone.

**Allow synchronous and asynchronous collaborative working:** Of course the system should support the synchronous collaboration work when two or more users are on line, providing adequate tools. But it should also allow synchronizing the work with other participants which are off-line in the best possible way, and provide mechanisms for merging the code developed off line.

**Allow the inclusion of new unforeseen participants:** Because the system is aimed to support flexible and

changing teams, there should be a way to include unforeseen participants and assign them tasks. However, the system should avoid an uncontrolled explosion of participants and maintain a certain order in the versioning of the document.

**Allow fine grained locking of a document:** In a less formal and flexible working team everyone may have access to the working documents and be able to modify them. However, to synchronize the documents copies of all participants in a full peer to peer environment, where there is no central repository may be a complex task, if we do not want to introduce too restrictive rules about who has the lock of a document is a complex task to accomplish in a fully distributed environment where there is no central server.

Some authors have already pointed out to the necessity of not having a centralized repository to coordinate the work of a software developing team [5], while others also have stressed the necessity of having a fine grained, logical oriented locking of the code [6]. These requirements can also be applied to the collaborative authoring of any document which has a certain structure. The decentralized model is certainly the most flexible and suitable model for these requirements. However, there is still no system which meets all the requirements mentioned. Developing such a system represents a challenge of high complexity, in the design and in its implementation. In this work we will present an extension of Code-breaker [14], which is was designed for supporting exclusively small and medium size software development based on an extreme programming principle, meeting the requirements mentioned above. The new system presented in this paper extends the same idea to other interesting problems in the field of collaborative authoring of documents in general. The main idea behind this work is to present a new scheme for the exchange and synchronization of well structured documents.

We limit the scope to documents that can be mapped to an LALR grammar. We state that solving the problem of collaborative work for this kind of documents opens a wide and interesting range of applications to real life situations. Consider the general scenario of the example stated above of writing an article. Many authors, distributed around the world, have to interact and generate a document. They have to share their contributions, and sometimes even integrate their writing into foreign systems. Our solution supports different conflict resolutions schemes, collaborative working, and even the possibility of working with different platforms, such as the case when an author writes in Latex and other in Open Office, waiving the painful process of transforming the source documents.

## 2. State of the Art

Back in the late 80's and early 90's when the Internet was rapidly expanding, there was a great interest in the distributed systems. It was then predicted that such systems will be the dominant technology for the synchronous collaborative work in the future [7]. We can nowadays confirm those predictions and add that these system have also deeply influenced the working style in all fields, Of course, computer system programming being was one of the first, and many systems have been developed since very early. We can classify those systems in two categories according to the aspect they stress with their support.

### 2.1 Versioning management systems

In the 1990's perhaps the most used tool for collaborative work synchronization was created, CVS, [3] initiating a wave of development of tools supporting Version Management. CVS problems are well known [8]: it uses a centralized model, a central data repository and only few operations or commands which can be executed off-line. This makes this structure really unsuitable for synchronous collaborative programming development. All developers need access to the central server for almost all operations. Today, there is a whole family of CVS-like tools: GNU-Arch, Subversion, CSSC, PVCS, etc. These applications are frequently used in the Open Source community and also in large business environments. All of them follow the same schema: one central repository, and file-level permissions. (Check in, Check out). These tools are used for Version Management in mid to large software development projects with many programmers involved.

### 2.2 Collaborative development environments

One of the first approaches to the implementation of collaborative development environments is the Orwell system [9]. This system allows the Smalltalk programmers to develop programs using a common library. An interesting aspect of this system is that it organizes the developing system code in methods and classes instead of files, thus using a more logical approach to present the code. Another Collaborative Environment that follow the same idea of the Orwell system is Tukan [10]. This synchronous distributed team programming environment for Smalltalk claims to solve the problems that Extreme Programming teams have. Tukan incorporates a version management system and adds awareness information, communication channels and synchronous collaboration mechanisms. It also provides a shared code repository with a distributed version management and the code integration can be made in a centralized or decentralized way. The IBM Rational ClearCase System [11] provides real time support for collaboration between developers located anywhere on the Internet. It uses a central server, that manages users permissions and differences between the source code

versions. The server has also support for multiple repository server deployments for large-scale enterprise teams. Another tool to which supports the collaborative editing of source code is the Collab add-on for the Netbeans 5.0 [12]. This add-on allows the NetBeans users to edit files collaboratively, share files and provides space to communicate with other E-breaker Organization, Roles and Ownership

In order to allow the synchronization of the document being developed among the members of the group in an asynchronous scenario, E-Breaker imposes that any existing piece of document in any of the participants' computer should be "owned" by someone. A E-Breaker collaborative document development project starts with one person defining the project and others joining it. Each new member including the one who created the project has to register an e-mail address and receives a digital signature.

### 3. Rules for document source ownership

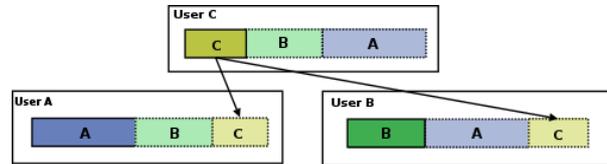
All members can develop new writings which are owned by him/her. Other members will receive the document's source and can use, modify, and even share it with others, but the only "official" version can be distributed or approved by the owner. In this way, there will be always a "current final version" of the entire document which will be the sum of all the code pieces each participant owns. In order to allow users to delegate their work, they can pass the ownership of the code among each other. Figure 1 and Figure 2 show an example how ownership of the document source may develop during a project involving three collaborators.

#### 3.1 Exceptions to the Rules

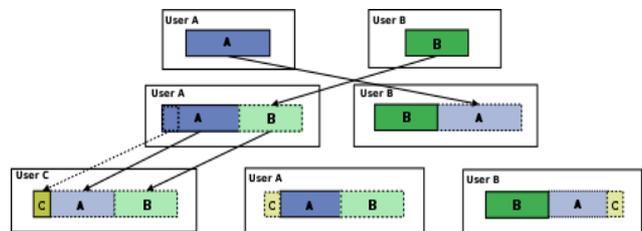
It is important to maintain the rights of the owner of the code and the order of, so an alternative should the project itself in order to avoid an uncontrolled explosion of versions. It is also known that in many projects it is sometimes impossible to maintain and respect every rule because of the emergence of unforeseen situations exist for bypassing the rules in exceptional cases. For example, it could happen that a certain user cannot work on the project anymore and that he is not reachable to ask him to delegate the work to other users. In this case there are two mechanisms that can be applied and the two coexist giving more flexibility to the system. The first one is that a user can ask the rest of the team to approve or reject by voting a petition for becoming the owner of a certain code piece that is owned by a third member of the team and/or to force the acceptance of a given modification.

By automatically locking the inherited classes of a locked class, i.e., the user that owns a specific class, owns by default the subclasses that extend it, a better control of the whole system is achieved. For example, a class that has been implemented to fit a small set of requirements and is not completely defined could have

many changes in their implementation issues, the data representation, and many similar details. This functionality ensures that the users that try to inherit from such classes must have the permission from the owner of the parent class, preventing inconsistencies



**Figure 1:** Colors show ownership of the code: blue for user A, green for user B and yellow for user C. In the first row, A and B start a new project writing both a part of the code. In the second, they merge their works and keep the ownership. In the third row, C joins the project and A grants ownership rights to part of the code.



**Figure 2:** User C works on the part of the owner code and distributes it to A and B with the new code included

It is certain that having temporary code or avoiding modifications completely is not possible, but this option of the system allows giving a little more control to the process and as it is based on the rules defined for the system, they are still flexible enough to support a more relaxed working style. This same example can be used for collaborative writing instead of programming classes just by segmenting the document in sections and sub sections

#### 3.3 Synchronizing the work

Synchronization must be possible when working synchronously as well as asynchronously. When working synchronously the information about changes of any type is sent to all connected participants. When a latecomer joins a working session with one or more other participants, their records are compared to update information about changes. Only code changes which are issued by the owner of the code are forcibly exchanged so there is no conflict about which is the latest version, since the owner issues a correlative number when its code is ready to be distributed. This number is also used to check if the change has been incorporated already. When an owner wants to publish a new version of a code a file with an XML content containing metadata and data for the code is generated and signed with his digital signature. The same is done

for distributing information about changes to the code ownership and new members.

In order to support the fact that some participants could be seldom on-line simultaneously with the rest of the group or that various subgroups do not meet each other frequently E-Breaker offers an asynchronous mechanism based on the use of e-mail. The XML files with the changes are sent to all email addresses of the project. Users can download them and process them off-line.

### 3.4 Assigning Roles

E-Breaker is aimed to support more a flat project structure in which every participant has the same rights and responsibilities. However, sometimes even in small projects there may be a need for having a certain hierarchy in order to maintain the synchronization among the participants. E-Breaker introduces two mechanisms which allow this with flexibility. The first one is, when a user is created it may or not receive the right of accepting new participants for the project. The number of participants which is allowed to invite can be also be specified. This rule helps to keep the control about the number of participants in the project. The second one is about receiving the ownership of a code. A user may receive or not the permission of passing the ownership of a code to a third one. This may be used to assign responsibilities to certain members of the team which they will not be able to avoid by granting rights to another member. With these simple two rules it is possible to assign administrative roles to certain people.

## 4. Documents Architecture

Our synchronization method applies to document types which can be described by a LALR grammar. Some examples are Java files or a limited version of a text document. The idea of applying this to text documents is very interesting, since we can synchronize documents written in Latex and Open Office for example, the only limitation is that the document format is limited and that the editor used should implement the merging method. Every file processed generates an XML file; this XML represents the abstract parse tree of the LALR grammar. The representation is direct but has some issues when synchronizing. The main problem is for example if we have the grammar shown in table 1.

If the functions 1 and 2 have not been modified, the only change is that the third function has been added. If we watch the parse tree, function2 has been shifted one level below and function3 uses its place in the tree. To solve this problem, we consider every non-terminal symbol that is used to describe list of components that are in the same level and we mark them as not able to be represented. By doing this, we have all the functions of the example in the same level of the tree, and the synchronization is easier, because we have look for a match in the same level for both trees. In the example,

every non-terminal symbols that should not be printed are marked with a “\*”. In the formal definition of the grammar we just have to add a binary vector which describes which non-terminal symbols should be printed.

Table 1 A problematic grammar

*CompilationUnit→CompilationUnit Class Enum 
Class→ClassName ListMinUnits
*ListMinUnits→varDeclaration   Method   StaticBlock 
varDeclaration→Modifiers Type Id [= Expression]
Method →MethodName Modifiers ParamList ReturnType StatementsBlock
StaticBlock→static StatementsBlock

The main problem with this approach is that if we have two versions of a Java file, for example:

```
class Example {
    function1() {...}
    function2() {...}
}
```

*Version 1*

```
class Example {
    function1() {...}
    function3() {...}
    function2() {...}
}
```

*Version 2*

The file generated is an XML file in which every tag represents a printable element of the grammar. We add three fields to every node: key, date and owner. Those fields are used for the synchronization, to maintain versions and historic information. Owner specifies the user which owns the node or the component represented by it. The date field stores the time of the last modification to that component, and the key stores a hash function which is used to identify changes during the synchronization, using this key we can skip from synchronizing complete branches of the tree.

An example of a XML file generated from a Java file is shown in table 2 and the resulting XML file, parsing this code with the same grammar already shown in table 3.

Table 2: An example of the XML generated from the Java

class Complex { double r,i; public Complex(double r, double i) { this.r = r;
---

```

this.i = i;
}
...
}

```

**Table 3:** The generated XML code

```

<?xml version='1.0' encoding='UTF-8'?>
<javaxml>
<class name="Complex">
<source>
<field type="double">
<var name="r">
</var>
<var name="i">
</var>
</field>
<method name="Complex" public="true">
<parameters>
<parameter>double r</parameter>
<parameter>double i</parameter>
</parameters>
<code><![CDATA[ {
this.r = r;
this.i = i;
}
]]></code>
...
</source>
</class>
</javaxml>

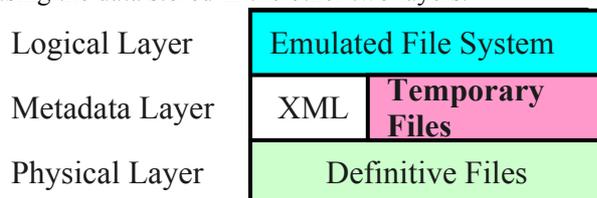
```

## 5. The System architecture

In this chapter we will describe the physical implementation of E-breaker. First we will describe the file architecture and then the software architecture.

### 5.1 The file Architecture

For implementing the logical management of the code as described in chapter 3 e-Breaker uses three logical layer file system architecture as seen in figure 3. The bottom layer is the physical layer, containing the accepted Java files. The middle layer is the meta data layer containing data for access management and presentation of the code. The upper layer is the logical file system which implements the emulated file system using the data stored in the other two layers.



**Figure 3:** The three layer architecture of e-Breaker

D-Files: This layer contains the files with content that is accepted by their owner. It is used to create distributions of the software, giving an alternative to

build a patched version also, including code that has not been accepted yet.

Temporary Files: Those are the copy made for every file containing modifications which are still not approved by the owner of the code.

XML Files: There is an XML for every file containing the information about the owner, permissions and information needed for the merging phase.

The Emulated File System: Is the logical layer that manages the logical access to the physical files and presents the information about which part of the code is owned by which user and whether the local code has been approved or released by the owner. For this, it uses the information stored in the XML files. It also implements a transparent file system for the user merging the temporary files with the accepted ones when corresponds.

This file organization allows users to manage their own versions of every file, but without losing the real branch of the whole document being written. The system should always have a copy of the “real files”, that is, the files containing code accepted by the owner. The reason for having a XML file for every file in the system is to simplify the merging phase every time a user has the chance to synchronize his working copy. The merging of the code, including the detailed and complex permission system of the system is almost impossible without any other information and very uncomfortable if this information is stored in the source itself.

### 5.2 The software Architecture

The software architecture of E-Breaker is a typical three-layer architecture. The **Awareness layer** implements the interface between the application and the graphical interface (which is optional, since the current version of E-breaker is implemented as a set of commands). The **Core layer** implements the logic of the application. The **Communication layer** implements the communication with other running instances of E-Breaker. The Core layer includes the Project Admin module which implements the administration of an E-Breaker project. The User Admin module implements the user administration tasks. The Code Admin module implements the code administration which includes mainly maintaining the three-layer files system with the source code, the temporal information and the meta-data. The Sync module implements the synchronization of the information which includes receiving a new version of the project of publishing the own version.

The Peer Detection module detects other currently online and announces to other users the presence of this instance. For this it uses uses the JXTA™ [13] technology, which provides libraries and several APIs to make the implementation of peer-to-peer networks more reliable. E-Breaker uses this technology to discover the participants of the developing team in the LAN and to establish a connection between them. JXTA also allows the system to be extended for many

users, so that they can be connected from anywhere in the Internet, even though firewalls.

The Send/Receive module is in charge of sending and receiving synchronization information to/from other users. It communicates with the Core module in order to pass the information received from other users in order to synchronize the files and receives the information to send to other users.

## 6. Conclusions

With the system presented in this document it should be possible to support a collaborative document creation, giving the opportunity to the small to medium-size working team to use a tool that is flexible enough to work without having troubles because of a complicated tool. The simplicity behind this idea gives the tool a real chance to be competitive in the market.

The authors have been engaged in developing software for medium-size enterprises, with their own small size developing Software Company. The problem and opportunity of these development teams is that they are not really tied to a fixed working place. It is very common that small companies work without a common physical place and in many cases without a common working schedule. This causes that often a member of the team is not able to work for a fixed period of time. The roles also change very dynamically within the project with people getting in and out of the project during the development. The existing tools are unable to maintain the order needed in this situation. They mostly consist of separated tools for the development and the administration. This imposes an extra human effort for keeping the order of the developing process with the consequent resource consumption in a situation where it cannot be accepted, because it is too expensive compared to the size of the project being developed.

The rules that the system implements about ownership of the code or document segments for controlling the coordination of the participant's work also support this fact and add more flexibility, so that the user can create a project that works under the rules that are most similar to the way his/her team really works.

## References

- [1] Schümmer,T. ,Schümmer,J. : Support for Distributed Teams in eXtreme Programming,In eXtreme Programming Examined, edited by Succi, Giancarlo, Marchesi, Michele , Addison Wesley, 2001.
- [2] Bowen,S.,Maurer,F. : Designing a Distributed Software Development Support System Using a Peer-to-Peer Architecture, 26th Int. Comp. Software and Apps. Conf. (COMPSAC 2002), pp. 1087-1092, 2002.
- [3] Berliner,B : CVS II:Parallelizing Software Development, 1989.
- [4] SourceForge,<http://www.vasoftware.com> , last visited on 14 February 2008.
- [5] Van der Hoek, A. , Heimbigner,D. , Wolf, A.L. : A generic, peer-to-peer repository for distributed configuration management, icse, pp. 308, 18th International Conference on Software Engineering (ICSE'96), 1996.
- [6] Magnusson,B. ,Asklund,U.,Minör,S. : Fine-grained revision control for collaborative software development, Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering, pp. 33 – 41, 1993.
- [7] Xu1,B., Lian,W., Gao,Q. : A General Framework for Constructing Application Cooperating System in Wind, ACM SIGSOFT Software Engineering Notes ,Volume 28 , Issue 2 (March 2003), pp. 15
- [8] Neary,D. : Subversion - a better CVS, <http://www.linux.ie/articles/subversion/> last visited on 13 February 2008
- [9] Thomas,D. , Johnson,K. : Orwel, a configuration management system for team programming, Conference on Object Oriented Programming Systems Languages and Applications, pp. 135 – 141, 1988.
- [10] Schümmer,T. ,Schümmer,J. : TUKAN: A Team Environment for Software Implementation. OOPSLA'99 Companion. OOPSLA '99, Denver, CO, pp. 35-36, 1999.
- [11] IBM Rational ClearCase, Integrated SCM for Rational Developer products and Eclipse, <ftp://ftp.software.ibm.com/software/rational/web/whitepapers/int-scm-rad-eclipse.pdf>, White papers of IBM, December 2004
- [12] Netbeans, Sun Microsystems, <http://www.netbeans.org>, last visited on 13 February 2008.
- [13] JXTA Technology: Creating Connected Communities, Sun Microsystems, <https://jxta.dev.java.net/>, last visited on 13 February 2008.
- [14] N. Baloian, R. Konow, F. Claude, M. Matsumoto: A decentralized and flexible tool supporting extreme programming software development. Procs. CRIWG'06, Medina del Campo, Spain, September 2006, LNCS 4154, 2006, pp. 179-186.