A Generic Pattern-based Design for Distributed Collaborative Editors

Asma Cherif

King Abdulaziz University, Faculty of Computing and Information Technology
Information Technology Department, Jeddah, Saudi Arabia

Abstract – With the advent of information sharing technologies, users can share and collaborate in performing common tasks through the use of distributed collaborative applications. Specifically, Distributed Collaborative Editors (DCE) are more and more used in many fields including healthcare, education, software engineering, military, etc. However, all proposed DCE frameworks focused on coordination algorithms, i.e. how to ensure data convergence, while very little attention was given to addressing the design of DCE which shall be the fundamental step for DCE frameworks construction. In this article, we analyze and design DCE as the structure of five layers comprising classifiers, relations among them, and their main properties. We contribute in two areas. First, we define a generic DCE architecture that captures organization, business and software aspects of DCE. Secondly, we propose a pattern-based design that emphasizes good design principles for the business core of DCE. This design will allow DCE researchers and developers to converge towards the same generic framework while improving the quality and performance of proposed DCE applications and tools.

Keywords – Distributed Collaborative Editors, software engineering, architectural design, GoF, design patterns.

DOI: 10.18421/TEM92-28
https://doi.org/10.18421/TEM92-28

Corresponding author: Asma Cherif,
King Abdulaziz University, FCIT, IT.
Email: acherif@kau.edu.sa

Received: 26 January 2020.
Revised: 23 April 2020.
Accepted: 30 April 2020.
Published: 27 May 2020.

© 2020 Asma Cherif; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.
The article is published with Open Access at www.temjournal.com

1. Introduction

During the last years, collaborative applications have become increasingly popular. As well-known examples of collaborative applications, we cite GoogleDocs, Overleaf and Wikipedia. Many tools have been developed to facilitate collaboration in many fields such as Computer-Supported Collaborative Learning (CSCL) that gained a broader usage in educational field as well as real time collaborative software design and engineering tools.

In particular, Distributed Collaborative Editors (DCE) allow many dispersed users to collaboratively edit and share documents thus being of relevance for communities editing reports such as researchers, engineers, health care practitioners, etc.

The research field of collaborative editing has emerged as the study of the complex interaction between many collaborators with the aim of allowing for concurrent editing of shared data. Two architectures were proposed for DCE: centralized and decentralized. To avoid the single point of failure and blocking scenarios, the distributed architecture was proposed thus allowing the deployment of DCE over P2P adhoc networks.

Early collaborative applications were essentially based on a centralized architecture responsible for synchronizing concurrent updates applied to the shared content (see Figure 1.(a)). However, a central design leads inevitably to single point of failure and reduces the availability of shared content.

In contrast, the decentralized approach (see Figure 1.(b)), has the advantage of making collaborative application scaling well, thus allowing a vast number of users to take the benefits of collaboration. Nevertheless, distributed systems rely on data replication to improve the availability of shared data and cut with blocking situations where clients should wait for the central server coordination.

Many DCE algorithms were developed to synchronize replicated data and enforce the convergence property despite concurrent updates. However, proposed solutions focused mainly on coordination frameworks and convergence...
maintenance while few efforts were dedicated to the design of DCE. We strongly believe that providing a robust design for DCE is crucial and fundamental to improve existing tools independently of the underlying network infrastructure or coordination algorithm. This design shall allow for flexibility to make DCE easier to implement, change, test, and reuse. It shall ensure performance and scalability and support replication.

We investigate these concepts to answer the following research question:

*How can DCE be modeled in a way that allows for flexibility, reuse and performance?*

Our research is primarily based on existing literature over collaborative editing (e.g. [1], [2], [3]). Precisely, our target is to focus on the following sub-questions:

1. How can the architecture of DCE be defined to capture the organization and interactions between the elements of any DCE?
2. How can the large-scale reusable design patterns be used to provide an effective and generic design of DCE?

In this work, we propose an architecture and a design model for DCE. We reuse the most known GoF patterns [4] to provide a reusable and flexible design that might be used for various kinds of DCE independently of the data structure of the elements handled by DCE algorithms.

The remainder of the paper is organized as follows. Section 2 gives a general overview on collaborative editing. Section 3 presents the primary features of DCE. Section 4 presents the DCE architectural design. Section 5 gives an overview on the core classes. Finally, Section 6 concludes.

### 2. Background

DCE represents a special kind of collaborative applications that provide collaborative editing features such as wiki pages or social media content. Therefore, they are very useful for communities of editors either in development companies, healthcare or academic organizations, etc. DCE can be used for documentation and communication in many tasks including design and engineering [5]. They have many benefits including:

- Shortening production time while increasing the quality of produced documents.
- Allowing multiple users to contribute in sharing documents quickly and easily [6].
- Enabling all users to view and modify their documents in a real time fashion.

Many distributed systems based on replication were suggested such as version control systems like Git [7]. These systems are particularly useful in applications like collective code development or collaborative documents editing. Many software solutions were proposed for word processing such as Coword [8] and P2PEdit [9], shared calendars such as DesCal [10], and collaborative drawing tools such as [11], etc.

Collaborative editing solutions are generally based on optimistic replication where the copy of the shared document is made available at each collaborating site (see Figure 1.(b)). This allows for the high availability of data, making possible the real time editing of the same document. Indeed, every user can edit his/her local copy concurrently to other users as if he/she was the only user of system.

The major concern with replication is that shared data should be kept consistent even though being modified concurrently by many users. Ensuring the consistency of replicated shared data is a challenging task. Different modifications on the shared data are received in different orders by different sites due to the network latency, churns or even the existence of malicious users.
Many collaboration algorithms were proposed in the last recent years to ensure the convergence property (e.g. [1], [2], [12]). Recently, some research efforts were directed towards applying replication-based collaborative editing over opportunistic networks (e.g. [13], [12], [14]) which represent an emergent solution that allows for establishing communication between devices even in the absence of a direct link between nodes. This may be caused by broken or damaged infrastructures. Motivated by the fact that nowadays mobile devices have been made available for everyone, this kind of networks is tending to be the next generation of wireless communication between users. Most importantly, opportunistic networks are effective in emergency situations [14]. In such situations, collaborative applications may be of relevance for distributed health care practitioners and rescue team members.

Collaborative editing requires a technique to ensure causality and convergence preservation while meeting the distribution and real time aspects of the DCE. Generally, the Operational Transformation (OT) technique is used because it achieves both causality and convergence preservation properties. Meanwhile, it increases the responsiveness [15].

OT is a synchronization technique based on optimistic replication. It aims at ensuring the convergence of replicated copies of the shared data despite the out-of-order execution of updates on different sites [1]. This approach is used in many collaborative applications. Famous applications such as Google Docs also use the OT approach to synchronize shared documents.

In OT approach, all updates performed on shared data are logged locally at every collaborating site. Thus, remote modifications can integrate easily the effect produced by concurrent updates performed on the document.

According to [1], shared data is represented by a sequence of elements that may be a XML node, a page or a paragraph, etc. depending on the granularity or the target application. More generally, any shared content including multimedia elements such as video and audio files can be edited collaboratively.

In general, two primitive updates are considered one for inserting new elements and another for deleting existing elements. Executing different updates in different orders at different sites leads to divergence situations. For instance, if we consider a text document initialized to “”hello”” and two operations $O_1 = \text{Ins}(2, a)$ and $O_2 = \text{Del}(2)$ generated simultaneously by two collaborators let site 1 and site 2 (see Figure 2.(a)), then the resulting states at site 1 and 2 are “haello” and “hllo”. The reception and execution of $O_1$ at site 2 produces the state “hallo”. Meanwhile, the reception of $O_2$ at site 1 produces the state “hello” since the effect of $O_1$ and $O_2$ previously executed at both sites were not taken into account. This leads to a different state at both sites thus the two sites diverge as shown in Figure 2.(a).

To avoid divergence, OT uses Inclusive Transformation (IT). Applying IT on the example presented above, leads to including the effect of deleting “e” at position 2 since “a” was inserted at the same position before the reception of the deletion. This would lead to increasing the position of $O_2$ thus the third character “e” is deleted instead of the second one “a”. $O_2$ is then transformed to $O_2' = \text{IT(Del}(2), \text{Ins}(2,a)) = \text{Del}(3)$. Accordingly, the final states at both sites converge to “hallo” as shown in Figure 2.(b).

In the following section, we present the functional and non-functional requirements of DCE to provide a detailed view of its features.

3. DCE Requirements and Features

In order to describe the features of DCE, we conducted a qualitative analysis study through the review of the existing DCE applications.
Generally, DCE provide an editor equipped with GUI allowing for basic features such as viewing and editing concurrently the same document. Edits are performed by many users from different sites.

On the top of this basic DCE environment, additional features such as undoing previously executed operations, locking specific parts of the document, varying the granularity of text propagation as well as text highlighting to increase the awareness of concurrent edits may be provided [16].

According to the existing DCE, the following user functional requirements should be provided:

1. The system shall allow for sharing documents.
2. The system shall allow for collaborative editing of the shared documents.
3. The system shall allow for replicating shared documents to allow for data availability.
4. The user shall be able to see concurrent updates and should know by whom these updates were performed.
5. The user shall be able to handle the disturbance of collaborators by hiding concurrent updates when needed [16].
6. The user shall be able to undo any undesired operation.
7. The system shall ensure a consistent view of the shared data (either automatically or by user interaction for conflict resolution).
8. The system shall store the history of modifications.
9. The user should be able to control the access to its shared data.
10. The user shall be able to join and leave editing group at any time.

The main features of DCE can be summarized in the basic functionality that shall be provided by any DCE tool including creating and sharing documents, editing documents, managing access rights for owned elements and visualizing concurrent updates performed by other members in a real-time fashion. The above-mentioned fundamental features are illustrated in the UML use case diagram presented in Figure 3.

As for non-functional requirements, it is necessary to enforce the following quality attributes:

1. Efficiency: to be efficient DCE shall allow for:
   - High local responsiveness: the system shall be as responsive as single-user editors [17].
   - High concurrency: users shall be able to concurrently modify the shared document at any time.
2. Security: users shall be able to assign access rights to their documents [9]. Moreover, “it is highly recommended to work collaboratively on encrypted documents”[3].
3. Transparency and ease of use: users should be able to use and learn easily the collaborative editing system. The underlying complexity of distributed system should not impact the progress of collaboration.
4. Scalability: a group shall be dynamic i.e. users may join or leave the group at any time.
5. Decentralized coordination: all concurrent updates shall be synchronized in a decentralized fashion to avoid the single point of failure.

A candidate architecture that better suites the previous requirements is presented in the following.

4. System Architecture

Based on the system requirements and features presented in Section 3, we propose the generic DCE architecture that highlights the main subsystems as well as their interactions. Indeed, based on the definition of architecture given in [18], we define the concept of DCE architecture as follows:

Definition 1 (DCE Architecture) The architecture of a distributed collaborative editing system is the set of fundamental concepts of the system in its environment comprising software elements, their interrelations and properties that highlight the flow of operations exchanged between collaborators through a network connection.

The definition stresses that the architecture of DCE consists of multiple structures that have interrelationship and allows to exchange data between collaborators through a network connection (see Figure 4.). The Business core forms the core of DCE. It is responsible for document, concurrency and access control management. Shared data should be visualized via a graphical interface allowing the user to modify, be aware of concurrent modifications and manage access rights of his/her owned objects. The updates that any collaborator perform or receive to alter the shared content should be managed via a queue.

![Figure 3. UML use case diagram for DCE](image-url)
**Figure 4. Five-Layers Architecture for DCE**

**Figure 5. UML class Diagram for DCE core**
Accordingly, we propose to organize DCE using the layered architectural pattern that is well suited for these applications. Indeed, it has the advantage to be portable and supports many platforms with minimal modifications since the modifications are localized in the layer that is subject to change. It allows for clear separation between layers which supports incremental development and future changes [19].

Figure 4. presents a five-layers architecture for DCE. The first layer encompasses classes that handle the graphical representation of DCE. The second layer is dedicated to authentication and access control management. This is useful to manage access rights to shared documents and is of relevance in opportunistic networks. Indeed, if an opportunistic network is used in the last layer, shared documents can be carried by nodes that are not members of the collaborating group. The third layer contains core classes for the coordination model. The fourth layer allows to handle the causality of remote operations by managing the queue of requests before they are forwarded to the main core. It also allows to deal with encryption and decryption to enforce security. Lastly, the fifth and the last layer handles network communications and how to send and receive operations over the network.

In the following, we discuss the main layer which is the DCE core in more detail and show how GoF patterns can be used to better meet DCE requirements.

### 5. Design of the DCE Business Core

In this section we present the UML class diagram for the main core classes. These classes involve the different algorithms required to maintain the consistency of the shared document. It provides the main basic classes for real-time collaborative editing algorithms including document, operation and transformation strategy classes. Each algorithm can be implemented by derived classes. Figure 5 shows an overview of the classes used in the DCE core layer. These classes are summarized in Table 1.

In the following, we explain our design choices for each of the classes used to model DCE.

### Table 1. An overview on the DCE core classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Class type</th>
<th>Main responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>Concrete</td>
<td>Handles the local copy of the shared document</td>
</tr>
<tr>
<td>Concrete Element</td>
<td>Concrete</td>
<td>Represents elements to be inserted or deleted from the shared document depending on the granularity</td>
</tr>
<tr>
<td>Flyweight Element Factory</td>
<td>Concrete</td>
<td>Allows to get a flyweight if it already exists otherwise add it to the factory</td>
</tr>
<tr>
<td>Flyweight Element</td>
<td>Abstract</td>
<td>Defines an interface for flyweight objects</td>
</tr>
<tr>
<td>Operation</td>
<td>Abstract</td>
<td>Defines the operation that will be performed to change the document’s state</td>
</tr>
<tr>
<td>Operation Factory</td>
<td>Concrete</td>
<td>Implements the OperationFactoryInterface and allows to create the appropriate operation given a Type parameter</td>
</tr>
<tr>
<td>Operation Factory Interface</td>
<td>Abstract</td>
<td>Defines the factory interface for operation creation</td>
</tr>
<tr>
<td>Log</td>
<td>Concrete</td>
<td>Handles the history of operations</td>
</tr>
<tr>
<td>Transformation Strategy</td>
<td>Abstract</td>
<td>Defines the transformation approach that will be used</td>
</tr>
<tr>
<td>Director</td>
<td>Concrete</td>
<td>Builds the different parts of the document’s representation</td>
</tr>
<tr>
<td>Representation Builder</td>
<td>Abstract</td>
<td>Allows to design a specific representation</td>
</tr>
<tr>
<td>Concrete Representation</td>
<td>Concrete</td>
<td>Represents the concrete class to create a specific representation</td>
</tr>
</tbody>
</table>

First, the Document class represents the shared document design. A document is composed of many elements where each element is an instance of the concreteElement class. Elements are altered by Operation objects; thus, the Document class uses the Operation class. Each document is associated to a
log that stores all operations performed on its elements. Since the number of elements may be very high in each document, we use the flyweight pattern to design elements. As a matter of fact, the flyweight creates a factory of elements and stores intrinsic state of the element inside the ConcreteElement while the FlyweightElement interface is responsible of storing the extrinsic state. This means that instead of creating new elements each time, it is possible to reuse an existing object that have the same extrinsic state then modify its intrinsic state. This is mainly used to gain processing time since the creation of Element objects can be costly. Meanwhile, many elements can be redundant in the document with same shared attributes and some few differences such as their position in the document.

The flyweight is very useful either for textual or graphical editors, where the same characters and/or shapes can be inserted at various locations in the document.

FlyweightElementFactory class represents the factory for the basic element of the document according to a fixed granularity. The advantage of using the flyweight elements is to improve the performance of the application by storing commonly used elements in the FlyweightElementFactory. This is very useful for mobile devices that require highly efficient algorithms. Stored elements are of the type ConcreteElement which contains information like the element’s id, size, and other related properties.

Operation class is used to define operations. To update a document, many operations are performed, they may have various kinds. Generally, existing solutions define two primitive operations which are insert and delete. Another operation “update” was introduced in [1]. Operations may be different from one shared data structure to another. Also, the operation signature may be subject to changes by adding some new attributes like the semantic dependency [1]. Thus, to simplify future changes in the specification of operations, we propose to use the factory method pattern in the design of the Operation class. For this, we define an abstract class Operation that will be derived by ConcreteOperation classes. And instead of creating these ConcreteOperations inside the Document class, we use the OperationFactory interface instead as well as their concrete subclasses. This pattern decouples the concrete operations from the Document class. Accordingly, all future possible modifications will be localized inside the factory classes.

To handle the transformation dynamically, we opt for the strategy pattern that allows to change the implementation of a given method dynamically at run time. This may be very useful in case we need to change the transformation algorithm. The operation delegates the transformation against another operation to the TransformationStrategy class. The latter defines the transformation algorithm that will be used. Since there are many existing transformation algorithms, we use the strategy pattern based on delegating an algorithm to another class, here TransformationStrategy to allow for dynamic change of the transformation function. This allows to switch between different implementations of transformations functions dynamically and also this design will let the transformation strategy hierarchy evolve easily in the future. This is necessary since some algorithms may be more suited than others for a given context. This design allows also to develop testing frameworks to compare between different transformation algorithms. It is worth noting that a transformation strategy should be defined for inclusive transformation and another abstract transformation strategy should be defined for exclusive transformation (a transformation function used in some coordination frameworks in conjunction with inclusive transformation).

The Log class is introduced to track operations performed on the shared document. This is mainly used in operational transformation-based algorithms. As a matter of fact, remote operations are transformed against local ones to take their effect into account before being applied to the document. There is only one log per each site. Therefore, we use the singleton pattern to enforce the existence of only a unique instance of the Log class. For this, the constructor is encapsulated into the Log class and the creation is done inside the class Log. The Log is stored into a static variable log that is accessed by client classes via a public static getter.

It is preferable to be able to produce many different representations of the shared document. For instance, a user may need to have an XML or latex representation for her textual document. To well manage these different possible representations in a flexible way, we use the builder design pattern. Therefore, we define the RepresentationBuilder class that has the role to build the document for each representation. It should be extended by as many ConcreteBuilders as the target representations. The abstract class RepresentationBuilder is composed with the Director class in such a way that switching from one representation to another can be done dynamically at run time.
6. Conclusion

Distributed Collaborative Applications (DCA) are very important applications since they allow users to concurrently edit the same text or multimedia shared document thus shortening production time while increasing the quality of produced documents. In this paper, we have proposed a pattern-based design for DCE to help designers and engineers converge toward a generic model that suites the properties of the DCE. Our design is based on GoF design patterns to ensure for flexibility and supporting future changes easily. Besides, the proposed model can be used for the development of simulation tools to test and compare existing coordination algorithms.

References


