Supporting Cross-Platform Real-Time Collaborative Programming: Architecture, Techniques, and Prototype System

Yifan Ma[1], Zichao Yang[2], Brian Chiu[1], Yiteng Zhang[1], Jinfeng Jiang[1], Bowen Du[3(✉)], and Hongfei Fan[1(✉)]

[1] School of Software Engineering, Tongji University, Shanghai, China
{mtage,1850250,1852137,1751047,fanhongfei}@tongji.edu.cn
[2] Institute of Software, Chinese Academy of Sciences, Beijing, China
yangzichao21@otcaix.iscas.ac.cn
[3] Department of Computer Science, University of Warwick, Coventry, UK
B.Du@warwick.ac.uk

Abstract. Real-time collaborative programming supports a group of programmers to edit shared source code concurrently across geographically-distributed sites and collaborate in a closely-coupled fashion. There exists a number of problems and limitations for this emerging approach to be applied in real-world scenarios, and two critical issues are the lack of support on cross-platform collaboration and multi-level consistency maintenance. In this study, we have proposed, designed and implemented a novel Cross-Platform Real-time Collaborative Framework (CP-ROOF), and meanwhile achieved conflict resolution of multi-level editing operations. Based on the proposed framework, we have successfully implemented two collaboration clients that have realized cross-platform real-time collaboration over Eclipse and IntelliJ IDEA, two of the most popular Java programming environments. In this paper, we present design objectives and rationales, workflow and functional design, CP-ROOF's architecture and components, and major technical issues and solutions. Preliminary user evaluations and performance experiments have demonstrated the feasibility of the framework and the satisfactory performance of the prototype systems in a wide range of scenarios.

Keywords: Real-time collaborative programming · Cross-platform collaboration · Multi-level consistency maintenance · Operational transformation

1 Introduction

Software development requires effective collaboration among programmers with diverse skills and expertise. In general, there are two categories of approaches

H. Gao and X. Wang (Eds.): CollaborateCom 2021, LNICST 407, pp. 124–143, 2021.
https://doi.org/10.1007/978-3-030-92638-0_8

in supporting collaboration during the programming process, namely *non-real-time collaborative programming* and *real-time collaborative programming* [13,14]. Non-real-time collaborative programming is a traditional and mature approach that has been widely applied in the industry, which is always based on version control systems such as *Git* [1]. Programmers edit source code in their private workspaces and manually merge other programmers' work when necessary. In contrast, real-time collaborative programming supports a group of programmers to view and edit the shared source code at the same time, while changes performed by collaborators are transmitted and merged instantly [11]. Operation conflicts caused by concurrent editing are resolved automatically, which ensures the consistency of distributed source code after all remote editing operations have been replayed locally.

Real-time collaborative programming is beneficial in various scenarios. As presented in [11,13,21], such novel approach achieves closely-coupled collaboration in agile software development, supports distributed pair programming, enables remote diagnoses and troubleshooting, and many more. As an emerging approach, *real-time collaborative programming* has attracted increasing interests from both academia and industry in recent years [9,11,13,15,23].

There exists a variety of problems and limitations with existing real-time collaborative programming environments. One critical issue is the lack of support for cross-platform collaboration. Most existing real-time collaborative programming tools have been designed for single environments only. For example, *Code With Me* [2] provides real-time collaboration features on IntelliJ IDEA only, and *CoVSCode* [13] supports real-time collaboration on Visual Studio Code only. Detailed analysis on the limitations of existing real-time programming environments will be presented in Sect. 2.

To address the above mentioned issue in real-time collaborative programming, we propose, design and implement a novel Cross-Platform Real-time Collaborative Programming Framework (CP-ROOF), as well as two client prototype systems based on the CP-ROOF. The framework has been proposed with generic approaches and design. Based on CP-ROOF, specific collaboration clients on different platforms can be designed and implemented with little effort. In this study, we have implemented two client prototypes based on CP-ROOF, namely CoIDEA and CoEclipse, which have enabled real-time collaboration over IntelliJ IDEA and Eclipse, two of the most popular Java IDEs. We present the design objectives, workflow and functional design, system architecture and components, major technical issues and solutions, and a set of experimental evaluations.

The rest of this paper is organized as follows. Firstly, in Sect. 2, we review related studies, and present problems and limitations on existing real-time collaboration environments and tools. In Sect. 3, we introduce and explain three design objectives for the proposed solution. In Sect. 4, we present the design of collaboration functionalities and the proposed framework in detail. In Sect. 5, we discuss major technical issues and solutions in implementing the framework. In Sect. 6, we demonstrate the prototype system, and present performance evaluations. Finally, we summarize this study and identify potential issues for future work in Sect. 7.

2 Related Work

Real-time collaborative programming benefits programmers a lot in multiple scenarios. There exist several research prototypes and preliminary products for supporting real-time collaborative programming, such as *CoEclipse* [11], *CoVSCode* [13], *Teletype for Atom* [6] and *Code With Me* [2]. However, none of these systems has been widely applied in real-world software industry, because there exists a variety of problems and limitations with existing real-time collaborative programming techniques. In this study, we aim to address two critical issues among them, which are discussed as follows.

Firstly, existing tools and environments have been designed for supporting single programming environments, and none of them supports cross-platform collaboration. For example, *Code With Me* [2] supports real-time collaboration with IntelliJ IDEA only; *CoEclipse* [11] supports Eclipse only; *Teletype for Atom* [6] provides real-time collaboration features for Atom only; and *CoVSCode* [13] supports real-time collaboration on *Visual Studio Code* only. *Saros* [3] was claimed to support real-time collaboration for both IntelliJ IDEA and Eclipse, but its versions for the two platforms are not compatible with each other, and the version for IntelliJ IDEA (Saros/I) is restricted to two-participant sessions. In conclusion, none of these systems really permits programmers to freely collaborate in an unconstrained manner using different IDEs. The lack of cross-platform support impedes the application of real-time collaborative programming.

Secondly, existing prototypes only support file-level consistency maintenance of the source code, ignoring folder-level conflicts. To achieve rapid local responsiveness in the sense that a user's local editing operation can be applied in the document without noticeable delay, real-time collaborative editing systems have commonly been designed with a *replicated architecture* [19]: the shared document is replicated at all collaborating sites. Consequently, one critical issue is to ensure the consistency of the distributed documents. *Operational Transformation (OT)* [8,16–18,22] is a well-established consistency maintenance technique, where the basic idea is to transform a remote operation into a new form according to the effects of previously executed concurrent operations. OT has been widely adopted in a wide range of real-time collaborative applications [10,11,13,14,16]. Most of the existing real-time collaborative programming solutions like [3,11,13] can transform and resolve conflicts between file-level editing operations (i.e. inserting or deleting a string in a file). However, concurrent folder-level editing operations (i.e. creating, deleting and renaming folders/files) may also conflict, and most of existing solutions miss support for it. For example, suppose there are two collaborators, namely A and B. At one moment, A creates a file and B deletes the whole parent folder concurrently. Their development environment may be inconsistent after the two operations have been propagated and replayed at remote sites. B has the file created by A whereas the folder including its content is deleted in site A. *Cloud Storage Operational Transformation(CSOT)* [16] is inspiring for designing transformation functions for folder-level editing operations. However, CSOT does not achieve seamless integration between folder-level editing operations and file-level editing operations.

To address the above mentioned limitations, we propose a novel solution for supporting real-time collaborative programming which will be described in the following sections.

3 Design Objectives and Rationales

In this study, we aim to address the two major limitations of existing real-time collaborative programming environments as presented above. We firstly present three design objectives of the proposed solution in this section.

3.1 Design Objective A: Supporting Cross-Platform Real-Time Collaborative Programming

Integrated Development Environment (IDE) is an essential part in programmers' lives. Each IDE delivers its unique features and benefits, and studies have shown that most programmers are satisfied with the IDE they are using [7]. For Java programmers, IntelliJ IDEA is used most, but there are still more than 20% of the programmers using other IDEs as a result of considerations on license cost, usability and collaboration-related capabilities [4,5,7]. As explained in Sect. 2, most of existing real-time collaborative programming environments support single IDEs or editors only. To achieve real-time collaboration in programming, some programmers are forced to adapt to another IDE which they may not be familiar with.

Our proposed solution must be capable of supporting cross-platform real-time collaboration, in the sense that programmers would be enabled to use different IDEs in the same real-time collaboration session. For example, with our solution, several Java programmers would be able to use both IntelliJ IDEA and Eclipse for real-time collaboration, so that each of them may continue to use the most preferred IDE without change. The solution would employ an architecture where the essential and generic functions are designed independently, and each platform-specific collaboration adaptor (i.e. collaboration client plugin for a specific IDE) could be implemented with least effort by following the same set of interface and process.

3.2 Design Objective B: Supporting Unconstrained Multi-level Consistency Maintenance

As presented in Sect. 2, existing real-time collaborative programming environments ignore folder-level consistency maintenance. Concurrent folder-level editing operations may also conflict and cause inconsistency like file-level editing operations. Integrating file-level and folder-level editing operations is also a challenge. For example, we have to consider when a file-level operation and a folder-level conflict with each other and how to resolve it. We have to consider the mechanism to define the relationship between file-level operations and folder-level operations and coordinate them.

The proposed solution should support unconstrained multi-level consistency maintenance, in the sense that collaborators can create, delete or rename files or folders in the shared project at any time while others are editing files and/or folders concurrently. The system should be capable of identifying the context of each operation and resolving conflicts between multi-level editing operations internally and automatically.

3.3 Design Objective C: Supporting Flexible Extensibility and Reusability in Design and Implementation

In addition to addressing the two limitations mentioned in Sect. 2, the proposed framework and system should be extensible and adaptable for multi-language and multi-functional support. Firstly, the proposed framework should be language-neutral, which can be applicable for supporting real-time collaboration with any programming language. For example, the design of the general workflow should accommodate common collaboration requirements from programmers using different programming languages. Secondly, the proposed solution should be extensible for diverse collaboration functionalities. For example, it should provide interfaces for easily integrating other functionalities such as collaboration awareness support, audio and video calls, and higher-level semantic conflict prevention support like DAL [12].

4 CP-ROOF: A Novel and Generic Cross-Platform Real-Time Collaborative Programming Framework

To achieve the above mentioned design objectives, we propose a novel Cross-Platform Real-time Collaborative Programming Framework (CP-ROOF), which serves as an essential part of our solution. In this section, we firstly present the design of the generic workflow and functionalities of real-time collaboration based on CP-ROOF from end-users' perspective, and then describe the architectural design of CP-ROOF in detail, which consists of three components named CP-ROOF Core (fundamental real-time collaborative programming support), CP-ROOF Server (collaboration coordinator) and CP-ROOF Client (transparent collaboration client adaptor).

4.1 Workflow and Functional Design

Although different programmers may prefer different IDEs and tools, the collaboration workflow can be unified. We extract common elements from the collaboration processes, and design the CP-ROOF framework based on the unified workflow as requirements. CP-ROOF provides out-of-the-box support for realizing this generic workflow, while specific collaboration clients may provide additional functionalities as appropriate.

Firstly, to initialize a collaboration session, a programmer may choose to create a brand new repository (i.e. the project including files and folders that

programmers will edit collaboratively). Local folders and files in the repository will be synchronized to the server, which serve as the base version for the real-time collaboration session. When there is only him/her working in the session, the only difference from the single-user programming work is that the editing operations are also transmitted to the server to maintain the source code copy on the server. When other programmers choose to collaborate with the programmer, they can join the session by the identity. The server transmits the copy of the repository including all folders and files to each collaborator in a short time.

After collaborators join the same collaboration session, they can start to work in the real-time collaboration style. According to Design Objective B, collaborators can edit source code files, create new files or folders, and delete any file or folder in the shared repository. The client will detect every editing operation and transmit it immediately to the server through the CP-ROOF framework. The editing operation will then be propagated to all other clients within the session. Each client of CP-ROOF will receive, transform and execute remote operations in the local environment. The conflicts between multi-level operations will be resolved by CP-ROOF automatically and transparently. As illustrated in Fig. 1, each collaborator is aware of the presence of others.

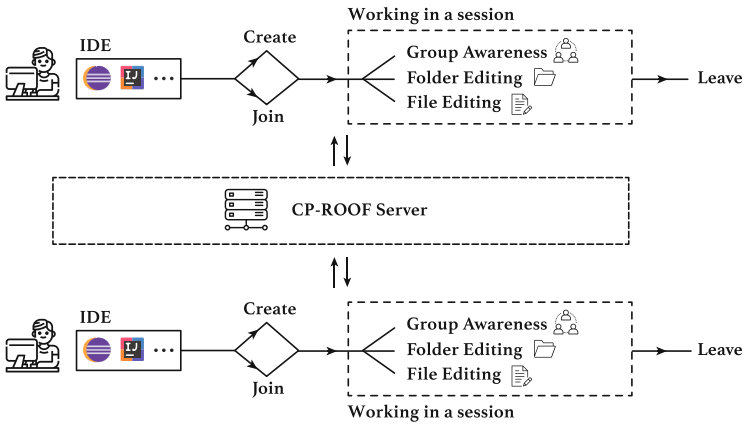

Fig. 1. The workflow design of CP-ROOF based real-time collaboration

A collaborator may leave from the session at any time, and other collaborators will be notified when a programmer leaves. CP-ROOF will maintain the latest collaboration repository even if all collaborators have left from the session. The repository will serve as the basis of programming work when the next session is initialized.

4.2 Architectural Overview of CP-ROOF

Based on the workflow and functional design above, we propose and design the Cross-Platform Real-time Collaborative Programming Framework (CP-ROOF) as follows. We will firstly present an overview with responsibilities of components and layers in this section.

As a generic cross-platform framework, CP-ROOF does not capture or apply editing operation from users directly. Instead, specific adaptors are responsible for interacting with collaborators and handling operations from users directly based on the common components provided by CP-ROOF. We have designed and implemented two specific adaptors in Java platforms named CoEclipse and CoIDEA in the form of plugins on Eclipse and IDEA respectively as illustrated in Fig. 2. Editing operations will be processed by CP-ROOF after being captured by plugins. CP-ROOF consists of three components, i.e. CP-ROOF Core, CP-ROOF Client, and CP-ROOF Server as illustrated in Fig. 3.

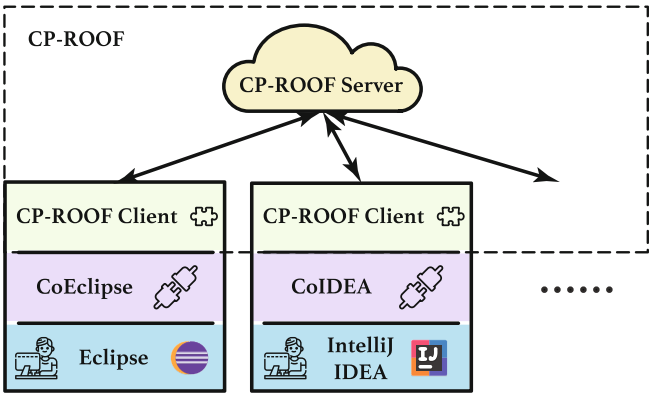

Fig. 2. Overview of cross-platform real-time collaborative programming environments

Under CP-ROOF, the input and output of each user's operation are processed by a hierarchy of layers. Each module has several layers according to its responsibility. CP-ROOF Core locates at the center of CP-ROOF as illustrated in Fig. 3. It is responsible for providing the most fundamental collaborative programming support. It includes the models of operations that users can perform, utilities and other messages transmitted between CP-ROOF Client and CP-ROOF Server. We also design and implement multi-level transforming functions for editing operations in CP-ROOF Core. The details of CP-ROOF Core will be presented in Sect. 4.3.

CP-ROOF Client provides common interfaces and classes for the design and implementation of concrete real-time collaboration adaptors. CP-ROOF Client is also responsible for controlling the transformation and execution of remote editing operations at each collaborating site.

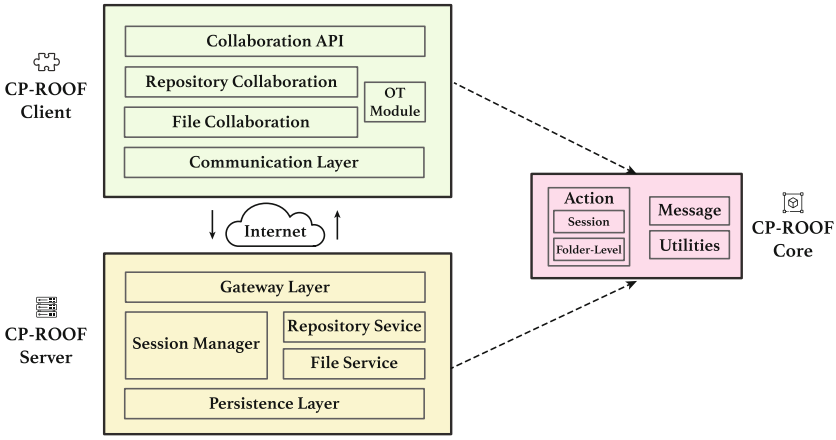

Fig. 3. CP-ROOF architecture and components

After a user chooses to create or join a real-time collaboration session, CP-ROOF Client will attempt to establish a persistent connection with the CP-ROOF Server. The connection is based on WebSocket by default and all messages and editing operations are transmitted via such persistent connection. CP-ROOF Server is responsible for managing all connections with clients on different platforms and broadcasting editing operations and messages to them. Design details and rationals of CP-ROOF Server and CP-ROOF Client will be presented in Sect. 4.4 and Sect. 4.5, respectively.

4.3 CP-ROOF Core: Fundamental Real-Time Collaborative Programming Support

According to Design Objective A, the proposed framework should support common collaboration operations in cross-platform real-time collaborative programming. To devise the framework, it is necessary to formally define what operations users can perform on a shared repository. A replicated repository in real-time collaborative programming can be described as a hierarchical file tree. After a user performs an operation of joining or creating a collaboration session, there are four folder-level operations that users can perform on the replicated repository:

1. $CR(p, T_p)$: to create a subtree T_p with a pathname p. A T_p can be a single file or a folder node.
2. $DL(p)$: to delete the subtree rooted with node p.
3. $RN(p, q)$: to change the name of node p to q.
4. $UP(p, d)$: to update the content of a file node(p) with file-level operation d.

Each $UP(p, d)$ embeds a specific file-level editing operation. There are two string-wise file-level operations that users can perform to update the content of a file:

1. $Insert(p, s)$: to insert string s at the position of p.
2. $Delete(lower, upper)$: to delete the string from $lower$ to $upper$.

CP-ROOF Core provides the model of the above user-generated operation. The user's operation during collaboration is divided into two categories, session-level operations and folder-level operations. All of them are sub-class of CoOperation (collaboration operation), which contains information about which collaborator issued it. File-level operations are sub-class of folder-level operations.

When a collaborator chooses to join or leave a collaboration session, a corresponding sub-type of session-level operation like InitJoinOperation will be created and transmitted to the server. Relatively, sub-types of folder-level operation like NodeCreateOperation and NodeUpdateOperation are used when collaborators are working in a session and creating a new file/folder node or editing the content of an existing file. This inheritance structure is language-neutral and highly extensible to support more diverse operations when necessary.

To provide multi-level consistency maintenance, CP-ROOF Core includes an integrated transformation function for creating, deleting, renaming and updating operations in both folder-level and file-level which will be presented in detail in Sect. 5.1.

4.4 CP-ROOF Server: Collaboration Coordinator

In real-time collaborative programming, collaborators will receive editing operations from others instantly. CP-ROOF Server is the central coordinator to receive and broadcast source code files, editing operations and various notifications. CP-ROOF Server is responsible for maintaining all files in working sessions and managing all persistent connections with clients. It will notify others about the operations when a collaborator creates, joins a session or modifies the file tree inner the working repository. The central document server works not only for notification but also for consistency maintenance. CP-ROOF Server serializes editing operations and allocates a total order which helps ensure correct transformation and execution ordering.

Each user operation will be collected by CP-ROOF Client and sent to CP-ROOF Server as a request. The request will be processed by multiple layers in CP-ROOF Server. In the gateway layer, requests are deserialized and distributed to different service layers. In real-time collaboration, collaborators may produce a mass of operations which have to be synchronized in a short time and servers have to notify others frequently. Consequently, persistent connections between clients and servers are maintained to save network overhead. Necessary files and information are stored in and read from the persistence layer which is advantageous to maintain collaboration sessions and recover from a breakdown.

4.5 CP-ROOF Client: Transparent Collaboration Client Adaptor

As a part of the cross-platform framework, CP-ROOF Client does not interact with users directly. It supplies out-of-the-box design and implementation for

supporting real-time collaborative programming. Depending on CP-ROOF Client, specific IDE plugins can support the whole workflow in Sect. 4.1 with little effort.

Based on the operations model in CP-ROOF Core, CP-ROOF Client provides collaboration API for both session-level and folder-level. Each call from collaboration plugins on different platforms will be transformed into universal formats and transmitted to CP-ROOF Server. Remote operations received from CP-ROOF Server will also be transformed meticulously and executed in the local environment. The repository collaboration layer and file collaboration layer are in charge of above transformation. And the communication layer is responsible for establishing a connection and exchanging messages with CP-ROOF Server. CP-ROOF Client uses WebSocket to connect to server by default and it is extensible to use a different protocol. Only if the protocol supports specific means of information interchange defined by CP-ROOF.

5 Major Technical Issues and Solutions

In this section, we present major technical issues and solutions involved in the design and implementation of CP-ROOF, as well as the implementation of two concrete collaboration clients based on CP-ROOF.

5.1 Multi-level Operational Transformation

Transformation Control in Multi-level Editing Operations. As presented in Sect. 2, Operational Transformation (OT) is widely used to support consistency maintenance in the replicated architecture. Studies have shown one basic strategy to apply OT is to separate the high-level transformation control algorithms from the low-level transformation functions [17,20]. Control algorithms like [20,22] are responsible for controlling the order, target and reference of transformation. To reduce the complexity, we have adopted and implemented *Context-Based Operational Transformation(COT)* [20] in CP-ROOF. As for file-level transformation functions, the design of string-wise transformation functions for individual files can be found in prior work [17].

The Context-Based Operational Transformation (COT) [20] algorithm provides efficient solutions to control transformation and execution of local and remote operations. To combine COT algorithm with cross-platform collaborative programming, we have to consider how to perceive editing operations from users and coordinate local operations and remote operations. Firstly, we have to ensure processors that handle local operations and remote operations compete orderly for the execution of COT module. Secondly, we have to ensure that the context of the editing operation being executed is the same as the state of the document that is being changed. In other words, if another thread changes the document after a transformation and before the execution of the operation, the transformation based on old context will become invalid and the execution may

modify the document wrongly. Only by ensuring these two conditions can we ensure the consistency between document replicates over collaboration sites.

For example, as presented in Fig. 4, there are two operation processors running on different threads to handle local and remote operations respectively. At one moment, the remote operation processor (ROP) receives a remote operation and the transformed operation is $Insert\ (7, \text{“}a\text{”})$ (to insert the text “a” at the position 7 of the document). Unfortunately, the local operation processor (LOP) receives a local editing event and the document is changed before execution. The transformation becomes obsolete and the execution will modify the document wrongly due to the inconsistency between the context of remote operation and local document state.

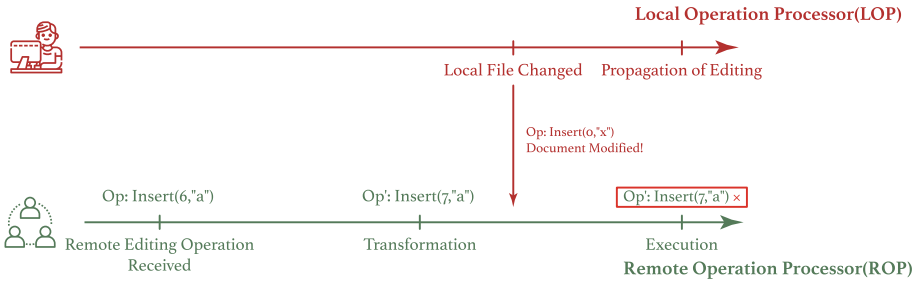

Fig. 4. An example scenario for illustrating execution competition

To ensure the conditions derived above, we have designed two strategies to maintain the consistency of document content and the structure of file tree respectively.

Firstly, when a local file-level editing operation is detected by the system, before it is applied in the document, LOP will request a lock in the OT module of CP-ROOF Client as illustrated in Fig. 5. If there is currently a remote editing operation being transformed and executed, LOP will be blocked until ROP completes the transformation and the document state is changed. The opposite is similar when ROP receives a remote editing operation. In this way, the OT module is invoked orderly, and editing operations will be transformed and executed based on the current document state strictly. Technically, the detection of local editing operation could be achieved by implementing handlers associated with particular events. For example, in Eclipse and IntelliJ IDEA IDEs, a local editing operation can be detected in the form of a keyboard press event that is about to change the document.

Secondly, things become different when it comes to folder-level editing operations. Existing IDEs provide a mass of ways to change local file tree. Collaborators may click buttons, move folder nodes by mouse or even create and delete files through file explorer. It is hard for CP-ROOF and specific client plugins to detect local editing and request the lock before local operation execution

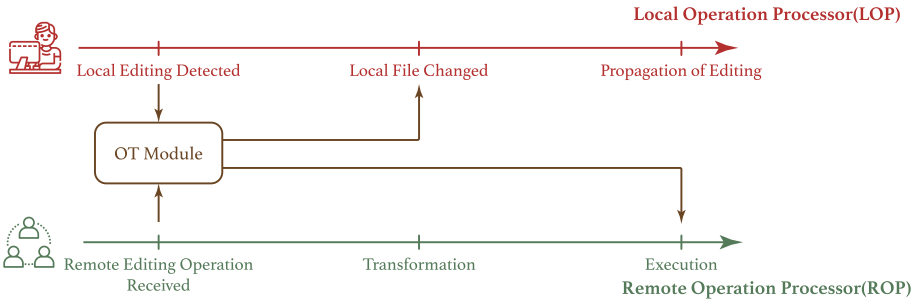

Fig. 5. Integrated processors for file-level editing

as described in file content editing. Therefore, we propose a novel coordination strategy for folder-level editing operation transformation control.

As presented in Fig. 6, ROP requests the lock when it receives a remote editing operation. But LOP will not try to acquire the lock until it detects the local file tree has been changed. If any remote editing operation is executed in a different file-tree state after transformation, a conflict will be perceived. The OT module will try to fetch the updated file-tree, transform and execute the remote operation again. We have derived that the OT module can detect the conflict in the form of *IOException* in Java. For example, if one site detects a folder-level change $DeleteFile(\text{“}src/A.java\text{”})$ (to delete the file in $src/$ named $A.java$), ROP transforms a remote operation into $Rename(\text{“}src/A.java\text{”}, \text{“}src/Action.java\text{”})$ (to rename file $src/A.java$ to $src/Action.java$) concurrently. The execution of *Rename* operation will fail due to the file deletion and an *IOException* will be thrown. The OT module will catch the exception and a new transformation process will be conducted. In this way, local and remote operations are coordinated to be transformed and executed correctly. This control process is implemented in CP-ROOF and transparent to both plugin developers and collaborators.

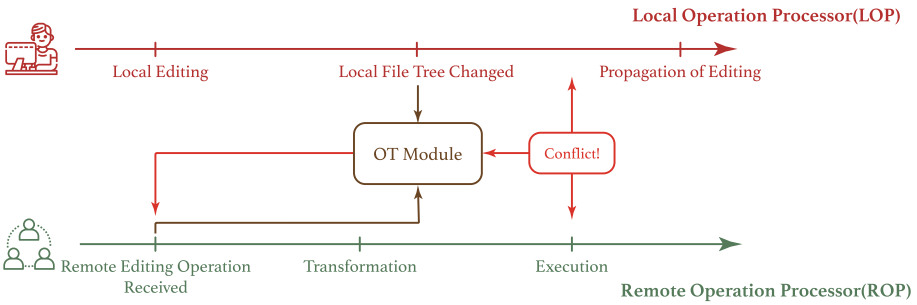

Fig. 6. Integrated processors for folder-level editing

Multi-level Operational Transformation Functions. There are four folder-level operations, $CR(p, T_p), DL(p), RN(p, q), UP(p, d)$, that users can perform on the replicated repository as presented in Sect. 4.3. Following *Cloud Storage Operational Transformation (CSOT)* [16], their relations can be defined depending on whether they produce inconsistent tree states when executed in different orders as defined in Definition 1 and Definition 2.

Definition 1 (Conflict "⊗"). *Given two operations, they are conflict only if these operations are concurrent and different execution orders result in inconsistent file-tree states.*

Definition 2 (Compatible "⊙"). *Given two operations, they are compatible only if they do not have a conflict relation.*

In order to integrate file-level and folder-level OT algorithms supporting transformation control mentioned in Sect. 5.1 and reduce the complexity of the system, we devise a new set of conflict and compatible relations between editing operations as shown in Table 1.

Table 1. Conflict and compatible relations.

	$CR(p_2, Tp_2)$	$DL(p_2)$	$RN(p_2, q_2)$	$UP(p_2, d_2)$
$CR(p_1, Tp_1)$	$\otimes \leftrightarrow (p_1 = p_2)$	$\otimes \leftrightarrow (p_2 \subset p_1)$	$\otimes \leftrightarrow (p_2 \subset p_1) \vee$ $((parent(p_1) = parent(p_2))$ $\wedge (nodename(p_1) = q_2))$	$\odot$
$DL(p_1)$		$\odot$	$\otimes \leftrightarrow (p_2 \subseteq p_1) \vee (p_1 \subset p_2)$	$\otimes \leftrightarrow (p_1 \subseteq p_2)$
$RN(p_1, q_1)$			$\otimes \leftrightarrow (p_1 \subseteq p_2) \vee (p_2 \subset p_1) \vee$ $((parent(p_1) = parent(p_2))$ $\wedge (q_1 = q_2)$	$\otimes \leftrightarrow (p_1 \subseteq p_2)$
$UP(p_1, d_1)$				$\otimes \leftrightarrow (p_1 = p_2)$

Take the conflict relation of $DL(p_1)$ and $UP(p_2, d_2)$ when $p_1 \subseteq p_2$ as an example, the combined-effect of $DL(p_1)$ and $UP(p_2, d_2)$ is that all nodes within $subtree(p_1)$(i.e. the subtree with p_1 as root) are deleted, no matter what p_2 is. The situation is similar when UP is replaced with RN and CR. This solution simplifies the complexity of the system and supports the transformation control well.

It is worth pointing out that $UP(p, d)$ embeds a file-level operation (i.e. an $Insert(p, s)$ or a $Delete(lower, upper)$) as presented in Sect. 4.3. Therefore, the conflict and compatible relationships between folder-level operations and file-level operations are inherited directly from relationships between $UP(p, d)$ and other folder-level operations.

Based on the above conflict and compatible relationship, we devise a set of new transformation functions which are different from CSOT [16]. The basic idea of designing transformation functions is to define another operation so that the transformed operation can be correctly executed and achieve document consistency in face of concurrent operations. For example, to integrate folder-level

editing operations with file-level editing operations, our solution defines UP with a nested file-level editing operation. Whenever a conflict of two UP happens, the folder-level transformation function does nothing. The transformation control algorithm takes out the internal file-level operation of UP and passes it to the file-level operational transformation functions for processing. This solution easily supports multilevel operational transformations and it performs well. It also satisfies the *Convergence Property 1 (CP1)*, which ensures the same repository is produced by executing two concurrent and mutually transformed operations in different orders. *CP1* is required when designing transformation functions combined with COT and has been discussed in detail in [20,22].

5.2 Client Design and Implementations

This section shows how the specific collaboration clients are designed and implemented based on CP-ROOF. These clients can be implemented in the form of plugins of IDEs which help programmers conduct real-time collaboration based on their preferred environments. We have implemented two clients named CoEclipse and CoIDEA as the reference implementations, based on the Eclipse and IntelliJ IDEA platforms, respectively.

With the full use of the components provided by CP-ROOF, the process of developing a new collaborative programming client simply involves three steps: (1) identifying the mechanism of listening for editing operations on a different platform, (2) implementing direct editing of local files and folders, (3) invoking the common components from CP-ROOF to receive editing events and displaying collaboration notifications. During the first step, the designer has to identify how to detect file editing operations on the specific IDE. After that, the designer has to implement classes that help the CP-ROOF Client apply remote operations locally. The designer has to follow the process of locking described in Sect. 5.1. The result of this activity is the implementation of two interfaces defined in the CP-ROOF Client. As illustrated in Fig. 7, interfaces named $ILocalFileEditor$ and $ILocalRepositoryEditor$ are the protocol between the CP-ROOF Client and collaboration plugins. During the third step, the client can receive collaboration events from remote sites and present them on the user interface finally.

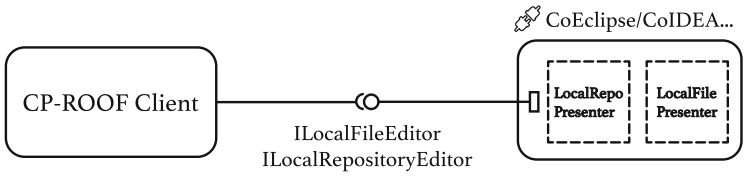

Fig. 7. Integration of CP-ROOF Client and specific IDE platform

Following the above design Steps 5.2, we have designed and implemented two prototype client systems, namely CoEclipse and CoIDEA, which are IDE plugins to support real-time collaborative programming in Eclipse and IntelliJ IDEA, respectively. They can support the whole workflow presented in Sect. 4.1. Both of them fully reuse components provided by CP-ROOF and their design follows the Model-View-Presenter pattern. To support CP-ROOF Client especially implement interfaces *ILocalFileEditor* and *ILocalRepositoryEditor*, CoEclipse and CoIDEA require a collection of plugin APIs, which depend on the concrete platform. For example, CoEclipse utilizes *IWorkspace.addResourceChangeListener (IResourceChangeListener):void* to detect folder-level editing operations on Eclipse. We will demonstrate their UI and cross-platform collaboration based on CoEclipse and CoIDEA in Sect. 6.1.

6 Experimental Evaluations

In this section, we demonstrate the cross-platform collaboration based on the proposed framework along with preliminary user evaluations and performance evaluations.

6.1 Cross-Platform Collaboration and Evaluations

Figure 8 presents CoEclipse UI for a programmer to start a real-time collaborative programming session. The user interface is similar to original Eclipse and existing single-user functionalities are preserved. Once clicking the "*Connect to Server*" button, a configuration dialog is displayed. The programmer needs to choose to create a brand new collaborative repository or join an existing session. Necessary parameters like the identity of the repository are required. Local source code will be uploaded to CP-ROOF server if a new repository is created. The initialization process is similar in CoIDEA.

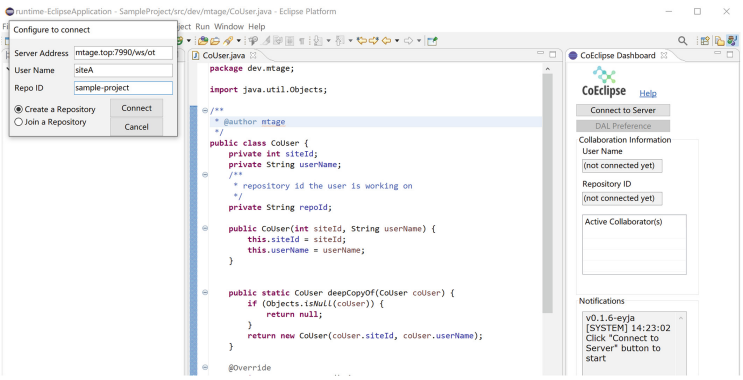

Fig. 8. UI snapshot of CoEclipse client's initialization panel

When programmers are working on the same repository in a collaboration session, all folder-level and file-level editing operations from other collaborators will be synchronized and replayed locally. As presented in Fig. 9 and Fig. 10, the collaborator using CoIDEA is working on the static method *copyOf* and the CoEclipse user is editing the constructor of class *CoUser*. The CoIDEA user is notified about the deletion of a file performed by the CoEclipse user. Both of them can see each other's work in real-time. The CoEclipse and CoIDEA prototype implementation have confirmed the feasibility of cross-platform real-time collaborative programming based on the CP-ROOF framework.

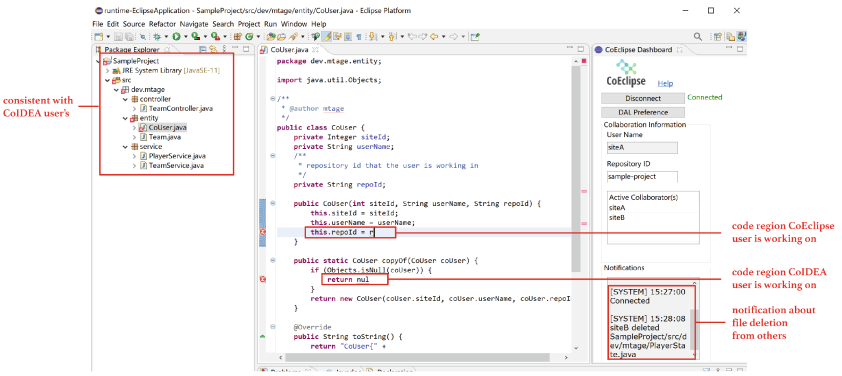

Fig. 9. UI snapshot of CoEclipse client in a real-time collaboration session

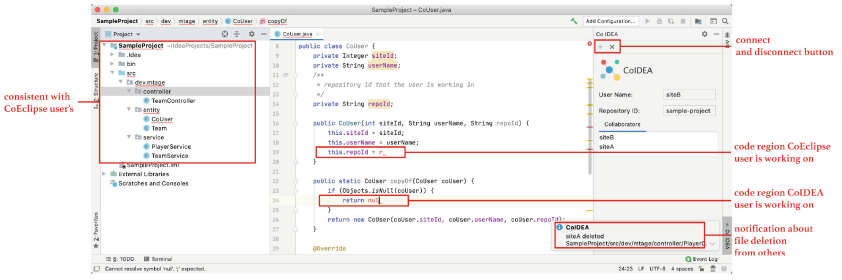

Fig. 10. UI snapshot of CoIDEA client in a real-time collaboration session

6.2 Performance of Major Procedures During Collaboration

In addition to the above user evaluations, we have conducted a set of performance evaluations on key procedures critical to user experience. Firstly, whenever a collaborator creates a new collaboration session, CP-ROOF Client will upload the initial source code copy. A replication of source code will be transmitted to local when other collaborators join an existing session. We selected several repositories to evaluate the performance of the initialization of real-time collaboration session.

The repositories were fetched from GitHub with diversity in project sizes to reflect real-world Java programming scenarios as much as possible. Table 2 presents the names, branches, and initialization times of each project. The experimental computer was equipped with an Intel Core i7@2.6 GHz processor and 64 GB of RAM, while the operating system was macOS 10.15.7. As illustrated in Table 2, even for large-size projects, the time cost for initialization was still acceptable.

Table 2 also presents the average processing times of local and remote editing operations in real-time collaboration sessions. It can be observed that local editing operations are always captured and applied instantly. The duration from an operation's generation to its remote replay mainly depends on its transmission over the network. The last column of Table 2 lists the average delay of the replaying of remote editing operations in real-time collaboration sessions over the Internet, which demonstrates the prototype's real-time performance in real-world scenarios.

Table 2. Average processing times of session creation (involving source code copy upload), session joining (involving source code copy download) and editing operations

Repository	Branches	Commit	Size	Items	Creation	Joining	Local editing	Remote editing
Halo	Master	bec10e9	2.0 MB	579	605 ms	1448 ms	1.33 ms	16.50 ms
Apollo	Master	b0173d7	6.3 MB	1214	911 ms	3307 ms	1.26 ms	20.92 ms
Lombok	Master	1a15270	6.9 MB	1946	1086 ms	4897 ms	1.28 ms	24.25 ms
Commons-lang	Master	d1e9e59	7.9 MB	472	870 ms	2151 ms	1.64 ms	25.83 ms
Netty	4.1	df53de5	18.6 MB	3109	2029 ms	6911 ms	1.36 ms	34.08 ms
Spring boot	Master	e1ad2cd	26.9 MB	7644	4461 ms	16138 ms	1.23 ms	40.83 ms

Moreover, we conducted further experiments to confirm the scalability of CP-ROOF in supporting real-time collaboration by a large number of participants. In a real-time session, whenever a remote operation is received, CP-ROOF Client will attempt to acquire the lock in OT module, and transform and replay the operation as presented in Sect. 5.1. Regardless of the network transmission delay (which is completely out of our control), the process of transformation and replaying is the most critical and time-consuming element that affects the user experience. The execution duration of such process is highly dependent on the number of sites and operations in the session, which affects the system's scalability. During our experiment, we simulated multiple sets of concurrent editing operations and measured the processing times of transformation. The most recently created version (MRCV) scheme is an effective buffering mechanism for operational transformation proposed in [20]. We have evaluated the processing times of CP-ROOF without MRCV and with MRCV respectively.

We have designed two simulation scenarios. In the first scenario, there is one local site and n remote collaborators working in the same session. Each remote site performs an editing operation concurrently. The local site receives n editing operations, transforms and replays them. Table 3 presents the processing times of

transformation of n operations on average. The percentages of each row present the percentage of remote folder-level operations. The rest are file-level operations that edit the same file concurrently.

Table 3. Times of transformation in n-collaborator sessions

Scenario A	n = 5	n = 10	n = 20	n = 30	n = 100
No MRCV (20%)	2.45 ms	5.02 ms	238.79 ms	31045.81 ms	(Timeout)
MRCV (20%)	2.33 ms	4.58 ms	8.57 ms	14.25 ms	46.61 ms
No MRCV (30%)	2.30 ms	3.33 ms	64.27 ms	8349.77 ms	(Timeout)
MRCV (30%)	1.44 ms	2.74 ms	5.31 ms	9.14 ms	44.173 ms

In the second scenario, there is a local site and a remote site working in the same session. Each site performs $n/2$ editing operations and local processing times of transformation are measured as presented in Table 4. The percentages of folder-level operations are still 20% and 30% respectively.

Table 4. Times of transforming n operations in two-collaborator sessions

Scenario B	n = 6	n = 10	n = 20	n = 30
No MRCV (20%)	2.04 ms	3.98 ms	681.57 ms	(Timeout)
MRCV (20%)	2.08 ms	2.48 ms	7.20 ms	11.33 ms
No MRCV (30%)	1.04 ms	2.97 ms	534.45 ms	(Timeout)
MRCV (30%)	1.03 ms	2.58 ms	5.05 ms	11.62 ms

It can be observed that even for a large amount of collaborators, the proposed framework can transform multilevel operations with acceptable time costs with MRCV regardless of the percentage of folder-level operations. Such time cost for transformation is only incurred when receiving remote operations, whereas the processing of local operations mainly depends on the original IDE and can be completed locally and instantly. Experimental results have demonstrated the good scalability of the CP-ROOF system in supporting large-scale collaborations by a large number of sites.

7 Conclusions and Further Work

Real-time collaborative programming is an emerging approach that enables a team of programmers to edit shared source code concurrently. However, the lack of cross-platform collaboration support and multi-level consistency maintenance impedes the real-world application of real-time collaborative programming. In

this paper, we have contributed a novel Cross-Platform Real-time Collaborative Programming Framework (CP-ROOF), which supports real-time collaboration for programmers using different IDEs. Both file-level and folder-level consistency maintenance have been supported and integrated. We presented a set of design objectives and rationales, proposed the architecture of CP-ROOF, and designed the three components of CP-ROOF (namely CP-ROOF Core, CP-ROOF Server, and CP-ROOF Client) in detail. We have also presented major technical issues and solutions in supporting the implementation of CP-ROOF. Following the CP-ROOF framework, we have successfully implemented two client prototypes named CoEclipse and CoIDEA to support cross-platform real-time collaboration over Eclipse and IntelliJ IDEA. Preliminary user evaluations have confirmed that all design objectives have been met, and performance evaluations have demonstrated the high efficiency of the implemented prototype system.

We are continuously working in the domain of real-time collaborative programming environments, and there are several issues identified for future work. Firstly, we plan to support more IDEs for real-time collaboration based on the proposed framework. During that process, we will invent more functionalities to assist programmers in collaboration awareness. Secondly, we will further improve and validate the proposed lock and transformation control to ensure consistence maintenance under complex situations in long-term collaboration. Thirdly, the CoEclipse and CoIDEA prototype systems will be continuously developed and improved, and the programs and source code will be released for the community to utilize when appropriate. Consequently, more in-depth evaluations will be conducted with more diverse scenarios.

Acknowledgment. This study has been sponsored by the National Natural Science Foundation of China (No. 62172301, No. 61772371, No. 62173248, No. 62073245, and No. 61702374), the Natural Science Foundation of Shanghai (No. 21ZR1465100), and the Fundamental Research Funds for the Central Universities.

References

1. Git. https://git-scm.com/. Accessed 12 Apr 2021
2. Code With Me: The ultimate collaborative development service by JetBrains. https://www.jetbrains.com/code-with-me/. Accessed 12 Apr 2021
3. Saros. https://www.saros-project.org/. Accessed 12 Apr 2021
4. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019#development-environments-and-tools/. Accessed 12 Apr 2021
5. Java Programming - The State of Developer Ecosystem in 2020 Infographic — JetBrains: Developer Tools for Professionals and Teams. https://www.jetbrains.com/lp/devecosystem-2020/java/. Accessed 12 Apr 2021
6. Teletype for Atom. https://teletype.atom.io/. Accessed 12 Apr 2021
7. Bergström, A.: A survey on developers' preferences in integrated development environments (2018). https://www.diva-portal.org/smash/get/diva2:1177860/FULLTEXT01.pdf
8. Cai, W., He, F., Lv, X., Cheng, Y.: A semi-transparent selective undo algorithm for multi-user collaborative editors. Front. Comput. Sci. **15**(5), 1–17 (2021). https://doi.org/10.1007/s11704-020-9518-x

9. Chen, Y., Lee, S.W., Xie, Y., Yang, Y., Lasecki, W.S., Oney, S.: Codeon: on-demand software development assistance. In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, pp. 6220–6231 (2017)
10. Cho, B., Sun, C., Ng, A.: Issues and experiences in building heterogeneous co-editing systems. Proc. ACM Hum.-Comput. Interact. **3**(GROUP) (2019). https://doi.org/10.1145/3361126
11. Fan, H., Sun, C.: Achieving integrated consistency maintenance and awareness in real-time collaborative programming environments: the CoEclipse approach. In: Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 94–101 (2012)
12. Fan, H., Zhu, H., Liu, Q., Shi, Y., Sun, C.: A novel DAL scheme with shared-locking for semantic conflict prevention in unconstrained real-time collaborative programming. IEEE Access **5**, 22566–22583 (2017)
13. Fan, H., et al.: CoVSCode: a novel real-time collaborative programming environment for lightweight IDE. Appl. Sci. **9**(21), 4642 (2019). https://www.mdpi.com/2076-3417/9/21/4642
14. Fan, H., Sun, C., Shen, H.: ATCoPE: Any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in software development, pp. 107–116 (10 2012)
15. Kurniawan, A., Soesanto, C., Wijaya, J.: CodeR: real-time code editor application for collaborative programming. Procedia Comput. Sci. **59**, 510–519 (2015)
16. Ng, A., Sun, C.: Operational transformation for real-time synchronization of shared workspace in cloud storage. In: Proceedings of the 19th International Conference on Supporting Group Work, GROUP 2016, pp. 61–70. Association for Computing Machinery, New York (2016)
17. Sun, C.: OT FAQ: Operational transformation frequently asked questions and answers. https://www3.ntu.edu.sg/scse/staff/czsun/projects/otfaq/. Accessed 12 Apr 2021
18. Sun, C., Chen, D., Jia, X.: Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems. In: Proceedings of the 21st Australasian Computer Science Conference, pp. 441–452. Citeseer (1998)
19. Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., Cai, W.: Transparent adaptation of single-user applications for multi-user real-time collaboration. ACM Trans. Comput.-Hum. Interact. (TOCHI) **13**(4), 531–582 (2006)
20. Sun, D., Sun, C.: Context-based operational transformation in distributed collaborative editing systems. IEEE Trans. Parallel Distrib. Syst. **20**(10), 1454–1470 (2009)
21. Wang, A.Y., Mittal, A., Brooks, C., Oney, S.: How data scientists use computational notebooks for real-time collaboration. Proc. ACM Hum.-Comput. Interact. **3**(CSCW), 1–30 (2019)
22. Xu, Y., Sun, C.: Conditions and patterns for achieving convergence in ot-based co-editors. IEEE Trans. Parallel Distrib. Syst. **27**(3), 695–709 (2016)
23. Zhang, J.: An Investigation of Technology Design Features for Supporting Real-Time Collaborative Programming in an Educational Environment. Master's thesis, Pennsylvania State University, State College, PA, USA (2018)