applied sciences

MDPI

Article

CoVSCode: A Novel Real-Time Collaborative Programming Environment for Lightweight IDE

Hongfei Fan [1], Kun Li [1], Xiangzhen Li [1], Tianyou Song [1], Wenzhe Zhang [1], Yang Shi [1,*] and Bowen Du [2,*]

1 School of Software Engineering, Tongji University, Shanghai 201804, China; fanhongfei@tongji.edu.cn (H.F.); 1553321@tongji.edu.cn (K.L.); 1552970@tongji.edu.cn (X.L.); 1551177@tongji.edu.cn (T.S.); 1551719@tongji.edu.cn (W.Z.)

2 Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK

* Correspondence: shiyang@tongji.edu.cn (Y.S.); B.Du@warwick.ac.uk (B.D.)

Received: 27 August 2019; Accepted: 24 October 2019; Published: 31 October 2019

Abstract: Real-time collaborative programming is an emerging approach that supports a team of programmers to view and edit shared source code at the same time. Each programmer can edit any part of the source code, and changes become instantly visible at other collaborating sites. With a broad range of application scenarios and benefits, real-time collaborative programming has attracted increasing interest from academia and industry. Lightweight integrated development environments (lightweight IDEs) have rapidly grown in popularity in the recent years, but there are serious problems and limitations with existing real-time collaboration support for lightweight IDEs. In this study, we contribute a novel real-time collaborative programming environment named *CoVSCode* that supports unconstrained and flexible real-time collaboration based on *Visual Studio Code*, one of the most popular and widely used lightweight IDEs. We present design objectives and rationales, the workflow and functional design from collaborating programmers' perspectives, major technical issues and solutions, and prototype implementation, as well as a set of experimental evaluations that have demonstrated the technical feasibility and good performance of the prototype system. All approaches, techniques and solutions derived in this work are generic, which can also be applied in building real-time collaborative programming environments for other lightweight IDEs.

Keywords: computer-supported cooperative work (CSCW); collaborative computing; collaborative software development; real-time collaborative programming; software development environment

1. Introduction

Software development requires effective collaboration among programmers with diverse skills and experiences. For supporting collaboration in the programming process, there are two general categories of approaches and techniques, namely *non-real-time collaborative programming* and *real-time collaborative programming* [1]. Non-real-time collaborative programming is a traditional approach that has been widely adopted and practiced in the communities for a long time, which is commonly supported by version control systems such as *Git* [2]. It is considered to be a kind of non-real-time collaboration because each programmer's modification on the source code copy will be kept private until the updated source code is uploaded to the repository and merged into other programmers' local source code copies. In contrast, real-time collaborative programming is an emerging approach that supports a team of programmers to view and edit a set of shared source code documents at the same time [3]. During a real-time collaboration session, each programmer can edit any part of the source code, and changes become instantly visible at other collaborating sites. Unlike non-real-time collaborative programming, concurrent changes made by several programmers can be automatically integrated in real-time collaboration processes without manual conflict resolution.

Real-time collaborative programming is an important supplement to non-real-time collaborative programming, which provides benefits in a broad range of scenarios. Firstly, several programmers may need to edit one source code file that cannot be further decomposed due to certain reasons. It is a well-known challenge for programmers to resolve conflicts in non-real-time collaboration while they concurrently edit and commit the same file. Secondly, in agile software development processes, programmers commonly work with frequent iterations to meet the changing requirements, and source code modules may not be well defined and isolated. Under such circumstances, programmers' tasks are strongly related and interdependent, which forces them to collaborate in a closely-coupled fashion. Thirdly, real-time collaborative programming is useful for supporting and enhancing pair programming, where two (or more) programmers, even geographically dispersed, can conduct pair programming work over the Internet. In addition, this emerging technique is also beneficial in supporting collaborative learning among students in a programming course, achieving collaborative diagnoses and troubleshooting between a support engineer at a customer's site and a senior engineer at the headquarter, and many other cases. Because of its wide application scenarios and benefits, real-time collaborative programming has attracted increasing interests from both academia and industry over the past decade [1,3–15].

Lightweight integrated development environments (lightweight IDEs), such as *Sublime Text* [16], *Atom* [17] and *Visual Studio Code* [18], have rapidly grown in popularity in recent years. For example, it is notable that *Visual Studio Code* was ranked as the most popular development environment in the *Stack Overflow*'s 2019 Developer Survey [19] only four years after its initial release in 2015. Lightweight IDEs are highly customizable, and can be significantly extended with a large number of available plugins to support various development tasks and programming languages. However, there is very limited support for real-time collaboration with lightweight IDEs. For example, even with the official real-time collaboration plugins for *Visual Studio Code* and *Atom*, programmers can only collaborate in a specific pattern that does not support truly unconstrained real-time collaboration and has serious limitations. In Section 2.3, we analyze these problems and limitations in details.

Given the increasing popularity of lightweight IDEs as well as our observation of the limitations of real-time collaboration support for them, we proposed, designed and implemented a novel real-time collaborative programming environment for *Visual Studio Code*, one of the most popular and widely used lightweight IDEs. The proposed environment met three design objectives: (i) preserving full compatibility with the existing lightweight IDE; (ii) supporting real-time collaborative programming with intuitive awareness; and (iii) achieving unconstrained and flexible real-time collaboration for programmers with equal roles. In this paper, we present the system architecture, workflow and functional design, major technical issues and solutions, prototype implementation, and a set of experimental evaluations. The approaches, techniques and solutions are generic, which can also be applied to building real-time collaborative programming environments for other lightweight IDEs.

The rest of this paper is organized as follows. In Section 2, we review related studies and present problems and limitations of existing real-time collaboration support for lightweight IDEs. In Section 3, we present and explain three design objectives for the proposed environment. In Section 4, we propose the system architecture with design rationales, and present the design of workflow and functionalities from collaborating programmers' perspectives. In Section 5, we discuss major technical issues and solutions for system building. In Section 6, we demonstrate the implemented prototype system, and present experimental evaluations in detail. In Section 7, we discuss the generality of the proposed approaches and techniques, and compare this study with related work. Finally, in Section 8, we summarize this study and identify potential research issues for the future work.

2. Related Work

2.1. Real-Time Collaborative Editing

Similar to many real-time collaborative applications in other domains, one of the fundamental techniques for supporting real-time collaborative programming is the generic real-time collaborative editing technique, which allows a group of collaborators to concurrently edit a shared document at the same time [20,21]. To achieve high local responsiveness (i.e., for a user's local editing operation to take effect immediately in the local document without delay), real-time collaborative editing systems have commonly been designed with a *replicated architecture*, where the shared document is replicated at all collaborating sites [22]. Consequently, there is an essential consistency maintenance requirement that after all editing operations have been propagated and replayed at remote sites, the shared document at each site must be identical.

To illustrate the consistency maintenance problem, a simple example is presented in Figure 1 (left side). Suppose that there are two collaborators, namely *A* and *B*, who are editing the shared document at the same time. The document's initial content was "*abcdef*". At one moment, *A* issues an operation *ins*[0, "*xy*"] (to insert the text "*xy*" at the position 0 of the document), while at the same time, *B* issues an operation *del* [2,3] (to delete the text "*cde*" from the position 2 of the document). After the two operations have arrived at remote sites and been replayed, the evolved documents became inconsistent.

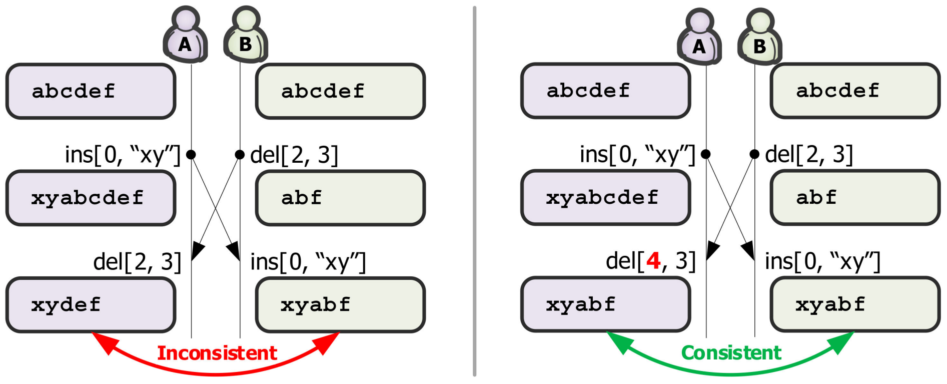


Figure 1. A simple example of operational transformation for consistency maintenance.

The illustrated problem has been well investigated within the community of computer-supported cooperative work (CSCW) over the past decades, and can be solved by the *operational transformation* (OT) technique [20,23–25]. In essence, the OT technique is capable of meeting two fundamental consistency requirements (i.e., *convergence* and *intention-preservation*) in real-time collaborative editing [21,25]. Supported by the OT technique, the inconsistency problem in the above example can be solved as illustrated in Figure 1 (right side). When the remote operation *del* [2,3] arrives at *A*'s site, it has been transformed into *del* [3,4] (i.e., the operation's position has increased from 2 to 4). Eventually, the transformed operation correctly deletes the intended text "*cde*" from the document, and achieves correct and consistent results.

It is worth noting that the example presented here is an extremely simple one, whereas numerous cases and issues exist that are far more complex. Fortunately, the CSCW community has made significant efforts in dealing with those problems, and has contributed mature and reliable OT techniques that can be utilized in building real-time collaborative applications [25,26].

2.2. Building Real-Time Collaborative Applications by Transparent Adaptation

Transparent adaptation (TA) is a generic system building approach for converting existing single-user applications into multi-user collaborative applications without knowing or changing the source code of the original applications [22]. Following the TA approach, the implemented applications can support both conventional single-user functionalities (provided by existing applications) and extra collaboration features (derived from emerging research). Technically, a TA-based collaborative application consists of an existing single-user application and an additional collaboration adaptor that invokes the programming interface provided by the single-user application. The OT technique (introduced above) plays an important role in the collaboration adaptor.

The benefits of the TA approach can be recognized from two perspectives [22]. For end-users of the collaborative applications, they can continue to use existing functionalities and user interfaces that are available in the single-user applications without learning new knowledge, while at the same time enjoying extra collaboration features. Meanwhile, researchers and software developers who are building the collaborative applications can concentrate on inventing novel collaboration techniques, and efficiently implement the applications without reinventing existing features and user interfaces.

The TA approach had been adopted and practiced in the design and implementation of a broad range of real-time collaborative applications in the past studies. Examples of TA-based collaborative applications include (but are not limited to): (i) real-time collaborative word processing applications such as *CoWord* [27] and the real-time collaboration plugin for *OpenOffice Writer* [28]; (ii) real-time collaborative 3D design tools such as *CoMaya* [29] and *Co-AutoCAD* [30]; (iii) real-time collaborative plain text editors such as *CoVim* [31]; (iv) real-time collaborative 2D graphical editing systems such as *CoWebDraw* [32]; and (v) real-time collaborative web-based editors such as *Codox Wave* [33]. In this study, we continue to follow the TA approach for building a real-time collaborative programming environment based on an existing single-user lightweight IDE, which will be discussed in Sections 4.1 and 5.1 in detail.

2.3. Problems and Limitations of Existing Real-Time Collaboration Support for Lightweight IDEs

As introduced above, lightweight IDEs have rapidly grown in popularity in the recent years. However, to the best of our knowledge, there is very limited support for real-time collaboration with lightweight IDEs. The two official real-time collaboration plugins for *Visual Studio Code* and *Atom*, namely *Visual Studio Live Share* [34] and *Teletype for Atom* [35], are the most sophisticated and mature so far, but they only support a specific collaboration pattern, with major problems and limitations.

Both *Live Share* and *Teletype* have been designed with only one specific collaboration pattern, which can be named as a *host-participator* pattern: one programmer acts as the *host*, while other programmers act as *participators*. The complete source code copy of the collaboration session is only maintained at the host's local workspace. Whenever a group of programmers start to collaborate, there must be one host who prepares the base of the source code at the local workspace, initializes the collaboration session and then invites others to join. During the collaboration session, each participator may be able to view the source code tree, but the content of any concrete source code file is only transmitted when the participator opens that file. Specifically, with *Teletype*, this is even more constrained—only the host can open/close files, whereas participators can only view and edit files that are currently opened at the host's site. Moreover, for both *Live Share* and *Teletype*, even after a source code file has been transmitted to the participator's site, it only resides in the memory (but not on the local disk). Such architectural and functional design leads to the following problems and limitations:

1. Because each source code file is only transmitted from the host to the participator on-demand, the network latency greatly affects the user experience at the participator's site, and when a file is opened, it takes a while to show up, which depends highly on the network condition.
2. The host must preserve an active network connection at all times during the collaboration session. Whenever the host encounters an issue and temporarily loses the network connection, the whole session is terminated, and all ongoing work made by the participators (that have not been transmitted to the host yet) will be lost.
3. At each participator's site, the source code files are incomplete and reside in the memory only, and as a result, there is very limited language support (because many IDE features rely on cross-file or even project-wide knowledge, such as the "*go to definition*" feature).

4. The debugging feature is commonly not supported at the participators' sites because of the incomplete and memory-residing source code files. Although *Live Share* is claimed to support collaborative debugging, the program execution actually takes place at the host's site during the debugging process, and no participator can independently debug or test the program without interfering with the host's ongoing work.

In addition to the major limitations presented above, there are also various minor issues with the two official real-time collaboration plugins. For example, both of them require Internet connections with servers hosted in the cloud, and thus are not suitable for internal development teams with high security concerns.

3. Design Objectives and Rationales

Given the popularity of lightweight IDEs together with the problems and limitations of existing real-time collaboration support (as presented above), we proposed, designed and implemented a novel real-time collaborative programming environment for a lightweight IDE. Firstly, the following three design objectives are proposed, which serve as high-level guidance for architectural and functional design, technical research and prototype implementation.

3.1. Design Objective A: Preserving Full Compatibility with the Existing Lightweight IDE

Existing lightweight IDEs (such as *Visual Studio Code* and *Sublime Text*) have been widely used in the communities, and provide sophisticated features and user interfaces covering a variety of software development tasks. For example, they commonly provide functionalities and features on source code editing, intelligent assistance, language-specific support, debugging, version control and so on.

The proposed environment should be fully compatible with the existing single-user lightweight IDE, in the sense that it must fully preserve all existing functionalities and features, user interfaces and workflows for traditional single-user programming and non-real-time collaboration. With such compatibility, programmers would effectively reuse their knowledge, skills and experience in a familiar environment, while enjoying extra real-time collaboration functionalities derived from this research.

3.2. Design Objective B: Supporting Real-Time Collaborative Programming with Intuitive Awareness

While preserving compatibility with single-user programming and non-real-time collaboration in the existing lightweight IDE, the proposed environment would support additional functionalities and features for real-time collaboration. Within a real-time collaborative programming session, a group of programmers can concurrently view and edit a shared collection of source code directories and files (i.e., the source code tree) as well as the contents of the source code files. Each programmer's change will be made instantly visible to other collaborators in real-time, and the consistency of the source code copies will be automatically maintained by the system.

During a real-time collaboration session, the proposed environment should ensure high local responsiveness at all sites, in the sense that each local editing operation performed on the source code must immediately take effect without noticeable delay (i.e., they must be as responsive as the local operations in traditional single-user programming without real-time collaboration).

In addition, the proposed environment would also support effective and intuitive awareness among programmers in a real-time collaboration session, which assists programmers in preventing potential conflicts in the closely-coupled collaborative work.

3.3. Design Objective C: Achieving Unconstrained and Flexible Real-Time Collaboration for Programmers with Equal Roles

As discussed in Section 2.3, the two official real-time collaboration plugins support the *host-participator* pattern only, which has serious limitations in meeting diverse collaboration needs. In this study, the proposed environment must support unconstrained and flexible real-time collaboration where collaborating programmers' roles are exactly equal. Given a collaborative team and a shared software project, each team member can flexibly access the project and edit the shared source code at any preferred time. Programmers may work individually at different times or collaboratively at the same time based on their collaboration needs.

Under all circumstances, the proposed environment ensures that: (i) whenever a programmer accesses the source code of the shared software project, the programmer always retrieves the latest version and continuously receives real-time updates from other active collaborators in the session; and (ii) whenever a programmer edits the source code, the change is immediately visible to other real-time collaborators, and also available to any incoming collaborator later.

4. System Architecture, Workflow, and Functional Design

In this section, we firstly describe the system architecture and its design rationales, and then present the design of the workflow and major system functionalities from the perspective of end-users (i.e., collaborating programmers).

4.1. System Architecture

Following the design objectives above, we herein present the overall system architecture, as illustrated in Figure 2. The proposed environment contains a server and several clients (as many as are needed), which are connected by communication networks. Each client will be used by one collaborator, while the server can be deployed internally within an organization or hosted via a cloud service.

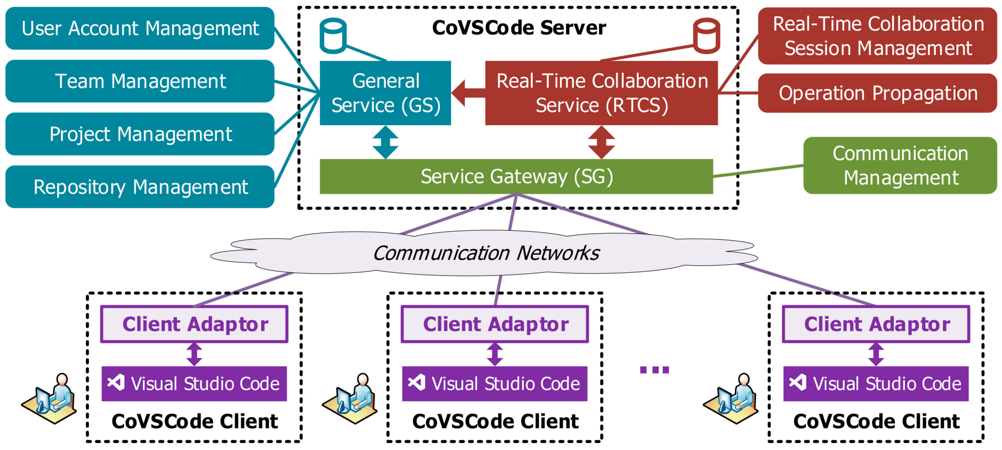


Figure 2. The *CoVSCode* system architecture.

To achieve *Design Objective A* (Section 3.1), we follow the TA approach (Section 2.2), and transparently convert the existing *Visual Studio Code* (referred to as *VS Code* hereafter) IDE into a novel real-time collaborative programming environment named *CoVSCode* without knowing or changing the source code of the original *VS Code* IDE. Each client consists of an existing *VS Code* and an additional adaptor that runs as a plugin of the underlying IDE. This ensures that all traditional single-user programming functionalities (e.g., source code editing, compilation, debugging, coding assistance) and user interfaces will be fully preserved. Since *VS Code* can be used for programming in almost every major programming language (such as Python, C/C++, C#, Java, Go, JavaScript, TypeScript, and many more) [36], the proposed *CoVSCode* would also be capable of supporting real-time collaborative programming in those major programming languages by nature. In addition, all extra functionalities of real-time collaboration will be embedded in the adaptor, which invokes various application programming interfaces (APIs) provided by the underlying IDE (but does not alter the IDE).

In this study, we have selected *VS Code* based on the following considerations. Firstly, *VS Code* has been designed with great extensibility by nature [37], and almost every part of the IDE can be customized and enhanced by utilizing the *Extension API* (i.e., programming interfaces provided by the IDE), which meets the fundamental requirements of this work. Secondly, *VS Code* has been widely used by a large number of developers in the communities [19], which implies the potential value of this work, as well as the opportunities for usability study and evaluation in the future. It is worth noting that all approaches and techniques derived in this work are generic, and can also be applied in building real-time collaborative programming environments for other lightweight IDEs of similar natures (such as *Atom* and *Sublime Text*), as long as the selected IDE provides necessary API support. In Section 5.1, we will describe what necessary API support is required.

To achieve *Design Objective B* (Section 3.2), major real-time collaboration mechanisms and functionalities will be implemented and distributed in the adaptor of each client. Particularly, to meet the requirement of high local responsiveness, the *replicated architecture* must be adopted, in which the source code copy is replicated at the local workspace of each site. This ensures that all local editing operations can take effect immediately without communicating with the server. In addition, to maintain source code consistency at all sites, certain mechanisms and techniques will be adopted and implemented in the adaptor. Under the current architecture, each client is directly connected to the server, and consequently, all editing operations are relayed by the server to other collaborating sites, which preserves the cause-effect relationships among editing operations.

The server contains (i) a *real-time collaboration service* (RTCS) component, (ii) a *general service* (GS) component, and (iii) a *service gateway* (SG) component which supports communications with clients. The GS component is responsible for managing user accounts, collaborative teams, software projects, and the corresponding source code repositories. The RTCS component is responsible for supporting real-time collaboration, including session management and operation propagation. In particular, to meet *Design Objective C* (Section 3.3), the RTCS always maintains the latest source code copy during a real-time collaboration session. When the session begins, the RTCS retrieves a copy of the source code from the GS. During the session, whenever an editing operation is relayed by the RTCS, it is also applied to the source code copy maintained at the RTCS component. When the session terminates, the latest source code will be sent back to the repository maintained by the GS.

4.2. Workflow and Functional Design

In this section, we propose a general workflow for a team of programmers to work in a real-time collaborative manner, and present the design of major system functionalities for each phase of the process.

4.2.1. Preparation of Real-Time Collaboration: User Accounts, Teams, and Projects

To use the proposed environment, a programmer should firstly obtain a user account, and then join (or create) a team. Each team may work on one or several software projects. The system provides functionalities for user account management (such as user registration, login authentication), team management (managing memberships for each team), and project management (such as project creation/deletion, managing relationships between teams and projects, and managing source code repositories for the projects). The system ensures that the source code copy stored in the server-side repository always represents the latest source code version for the project.

When a programmer (a registered user) logins into the system, he/she can select a team (if the programmer currently belongs to several teams). Consequently, the programmer may view the list of projects managed by the team and select an existing project or create a new project. When creating a new project, the programmer is allowed to upload an initial version of the source code (serving as the basis of the team's collaborative work).

4.2.2. Initializing a Real-Time Collaboration Session

When a programmer selects a project to work on (that is not currently being edited by anyone else), the current source code copy (including directories and files) will be downloaded from the server to the local workspace of the client. In the meantime, this triggers the initialization of a real-time collaboration session.

When there is only one programmer in the session, the programming work is almost the same as in a traditional single-user programming work, and the only difference is that whenever an editing operation is being performed, the change is also transmitted to the server and applied to the source code copy at the server-side. This mechanism ensures that when another member joins the session later, he/she can directly retrieve the latest copy of the source code from the server. Under all circumstances, the server always keeps a latest copy of the source code for the team, provided that all editing operations have arrived at the server.

4.2.3. Joining a Real-Time Collaboration Session

From a collaborating programmer's perspective, joining a session is the same as initializing a session. Whenever a programmer selects a project to work on, if there is an active ongoing session corresponding to that project, the server will dynamically add the current programmer to the session and transmit the latest copy of the source code to the programmer's client. After the programmer joins, all programmers in the session will resume collaborative work based on the same version of the source code. To maintain the source code consistency and ensure a smooth transition during the joining process, a distributed mechanism will be designed and deployed (to be presented in Section 5.4). In addition, as a collaboration awareness feature, whenever a programmer joins, all existing programmers in the session will be notified in the user interface.

4.2.4. Working in a Real-Time Collaboration Session

In a session with two or more programmers, each programmer edits the local source code in the same way as the traditional single-user programming. The client system automatically detects each fine-grained editing operation and immediately propagates it to all other active collaborators via the server. In the meantime, the client also maintains a live communication with the server for receiving real-time editing operations from other sites. Whenever a remote editing operation arrives, it will be replayed in the local source code copy. As a result, multiple programmers can concurrently edit any part of the shared source code, while changes are instantly visible and integrated at all sites.

Precisely, there are two levels of real-time collaboration sessions. Firstly, the abovementioned session can be regarded as a *project-level session*. Secondly, within the project-level session, if any source code file is being viewed or edited by two or more programmers at the same time, then there is a corresponding *file-level session* (which manages real-time editing operations on the contents of this particular file). Correspondingly, programmers may issue two types of editing operations: (i) the *project-level editing operation*, which creates, removes or renames a source code item (i.e. a file or directory within the source code tree) in the project; and (ii) the *file-level editing operation*, which edits the contents of a source code file. The timeliness of real-time operation propagation is designed as follows.

Firstly, when a programmer issues a project-level editing operation, it will be immediately transmitted to other sites within the same project-level session. To deal with possible conflicts among concurrent project-level editing operations, certain mechanisms and techniques will be adopted and deployed (to be presented in Section 5.3.2).

Secondly, when several programmers are viewing or editing the same file concurrently, operations performed by each programmer will immediately be transmitted to other sites within the same file-level session. However, only when this file is being saved by a programmer within the file-level session is the saved version transmitted to other programmers within the project-level session. The design rationale is that programmers outside the file-level session may not be interested in the latest content of a file's working copy before saving, and such propagation-on-saving feature also helps to improve system performance and reduce usage of communication bandwidth. To deal with possible conflicts among concurrent file-level editing operations and maintain the consistency of source code copies, certain mechanisms and techniques will be adopted and deployed (to be discussed in Section 5.3.1).

4.2.5. Leaving and Terminating a Real-Time Collaboration Session

A collaborating programmer may leave the ongoing session at any time. When the programmer closes a source code file in the editor, he/she leaves the file-level session. When the programmer closes the entire project, he/she leaves the project-level session. Similar to the joining process in Section 4.2.3, the leaving process will also be controlled by certain mechanisms that ensure the smooth transition and the consistency of source code copies. In addition, whenever a programmer leaves a session, all remaining programmers will be notified.

When the last active programmer leaves a project-level session, the entire session is terminated. Upon termination, the latest source code is maintained at the server-side, which will serve as the basis of programming work when the next session is initialized.

5. Major Technical Issues and Solutions

In this section, we present several major technical issues and solutions involved in system design and prototype implementation.

5.1. Utilizing Programming Interfaces (APIs) of the Underlying IDE

Following the TA approach (Section 2.2), the client adaptor invokes programming interfaces (APIs) provided by the underlying IDE without knowing or changing its source code. To implement the proposed real-time collaboration functionalities and features, the system requires a collection of APIs, which are listed in Table 1. The left column of the table gives generic descriptions of the API requirements, while the right column presents the concrete *VS Code* APIs that have been utilized in our prototype implementation.

Table 1. Major APIs required for implementing real-time collaboration functionalities.

Generic API Requirement		Concrete *VS Code* API (Descriptions Can Be Found at [38])
(a)	Detecting editing operations on source code directories and files (i.e., source code tree)	■ vscode.workspace.createFileSystemWatcher (globPattern: GlobPattern, ignoreCreateEvents?: Boolean, ignoreChangeEvents?: Boolean, ignoreDeleteEvents?: Boolean): FileSystemWatcher ■ FileSystemWatcher.onDidCreate: Event<Uri> ■ FileSystemWatcher.onDidDelete: Event<Uri> ■ FileSystemWatcher.onDidChange: Event<Uri>
(b)	Detecting editing operations on source code file contents	■ vscode.workspace.onDidChangeTextDocument: Event<TextDocumentChangeEvent>
(c)	Detecting file saving operations	■ vscode.workspace.onDidSaveTextDocument: Event<TextDocument>
(f)	Retrieving the positions of editing cursors	■ vscode.window.onDidChangeTextEditorSelection: Event<TextEditorSelectionChangeEvent>
(g)	Retrieving the highlight ranges in source code	■ vscode.window.onDidChangeTextEditorSelection: Event<TextEditorSelectionChangeEvent>
(h)	Setting the positions of editing cursors	■ vscode.textEditor.setDecorations(decorationType: TextEditorDecorationType, rangesOrOptions: Range[]|DecorationOptions[]): void
(i)	Setting the highlight ranges in source code	■ vscode.textEditor.setDecorations(decorationType: TextEditorDecorationType, rangesOrOptions: Range[]|DecorationOptions[]): void
(j)	Implementing other collaboration-related user interface elements	■ vscode.window.createWebviewPanel(viewType: string, title: string, showOptions: ViewColumn|{preserveFocus: boolean, viewColumn: ViewColumn}, options?: WebviewPanelOptions & WebviewOptions): WebviewPanel ■ vscode.window.showInformationMessage(message: string, options: MessageOptions, ... items: string[]): Thenable<string|undefined> ■ vscode.window.showErrorMessage(message: string, options: MessageOptions, ... items: string[]): Thenable<string|undefined>

5.2. Client System Architecture and Major Components

Given the overall system architecture (Section 4.1), the *CoVSCode* client adaptor plays a critical role in supporting real-time collaboration. As illustrated in Figure 3, the client adaptor has been designed with eight major functional components. Some of the components invoke APIs provided by the three modules in the underlying *VS Code* IDE, which are indicated by the labeled arrows (each letter refers to a corresponding item in Table 1 above).

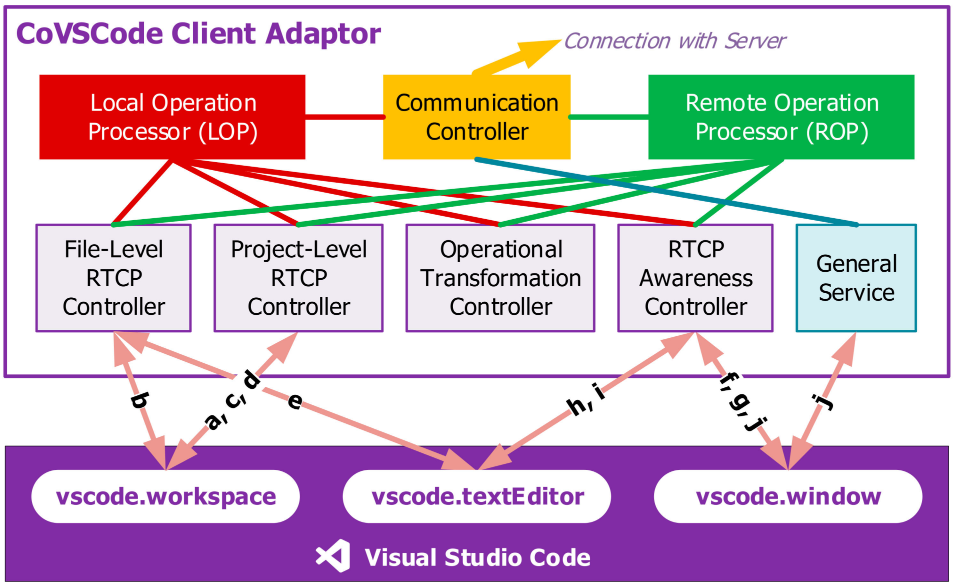

CoVSCode Client Adaptor
Connection with Server
Local Operation Processor (LOP)
Communication Controller
Remote Operation Processor (ROP)
File-Level RTCP Controller
Project-Level RTCP Controller
Operational Transformation Controller
RTCP Awareness Controller
General Service
b
a, c, d
e
h, i
f, g, j
j
vscode.workspace
vscode.textEditor
vscode.window
Visual Studio Code

Figure 3. The *CoVSCode* client system architecture and major functional components.

The *File-Level RTCP Controller* (the term RTCP represents "real-time collaborative programming" in this figure) detects local operations and replays remote operations in source code editors for supporting file-level sessions. The *Project-Level RTCP Controller* detects local operations and replays remote operations in the IDE's project explorer (i.e., the source code tree) for supporting project-level sessions. The *RTCP Awareness Controller* retrieves the local programmer's editing cursor and/or highlight range, displays all remote programmers' editing cursors and/or highlight ranges, and delivers notification messages for supporting real-time collaboration awareness features (visually presented in Section 6.1.3). The *Operational Transformation Controller* is the key to supporting consistency maintenance of the source code, with the OT technique as the cornerstone (see Section 2.1). Technically, each local editing operation will be processed by the controller, and each remote editing operation will be transformed by the controller before its replay.

The *Local Operation Processor* (LOP) integrates four controllers, and is dedicated to detecting and propagating local editing operations performed by the local user in the source code. When the user opens a source code file in the editor, the component registers a specific listener with the source code editor in the initialization procedure. Consequently, whenever the local user issues an editing operation in the programming process, the listener captures the changes that have occurred in the source code. It is worth noting that one editing operation may result in one or many changes, depending on the actual situations. The changes are then processed by the operational transformation controller, and transmitted to the server for propagation to all collaborating sites within the same session.

The *Remote Operation Processor* (ROP) also integrates the four controllers, and is dedicated to applying remote editing operations performed by remote users to the local source code. Each remote editing operation that arrives is firstly processed by the operational transformation controller, and then replayed in the source code file, which immediately becomes visible in the user interface.

Given each source code file that is being edited, there will be a pair of parallel threads that correspond to the LOP and ROP, as illustrated in Figure 4. The two processors involve some shared components, such as several *VS Code* modules (for invoking APIs) and the operational transformation controller (for processing and transforming operations). Certain mechanisms (e.g., mutex, semaphore) are needed to control the parallel execution of both threads and serialize concurrent access to the shared components in the system.

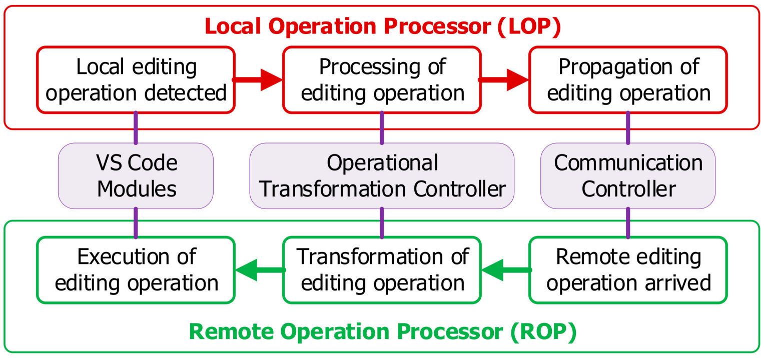

Local Operation Processor (LOP)
Local editing operation detected
Processing of editing operation
Propagation of editing operation
VS Code Modules
Operational Transformation Controller
Communication Controller
Execution of editing operation
Transformation of editing operation
Remote editing operation arrived
Remote Operation Processor (ROP)

Figure 4. Local operation processor (LOP) and remote operation processor (ROP).

5.3. *Consistency Maintenance of Replicated Source Code Copies*

As mentioned above, to achieve high local responsiveness in real-time collaboration sessions (as specified in *Design Objective B*), the system has been designed with a replicated architecture, and each collaborating site must hold a distributed copy of the source code. Consequently, any local editing operation performed can be immediately executed on the local replica, and then be propagated to remote sites within the same session. One major technical issue here is the consistency maintenance of the source code (i.e., after all editing operations have been propagated and executed at all sites, the distributed source code copies must be identical). For the two levels of real-time collaboration sessions (file-level sessions and project-level sessions), certain consistency maintenance mechanisms and techniques have been adopted and deployed as follows.

5.3.1. Consistency Maintenance in File-Level Sessions

In a file-level session, programmers concurrently edit the content of a shared source code file. Technically, a source code file is a string of textual characters with a single linear address space, and each character can be addressed by a positional value. There are two primitive editing operations that can be executed:

- *insertString* [p, s]: to insert a string s at the position p of the source code file.
- *deleteString* [p, l]: to delete a string starting from the position p with the length l.

All complex editing operations will result in one or several primitive editing operations defined above. For example, replacing several lines of the source code by pasting texts may lead to a *deleteString* operation and an *insertString* operation. The registered listener (item *b* in Table 1) will automatically detect the textual changes in the source code file and produce these primitive operations for further processing.

As discussed in Section 2.1, concurrent editing operations may lead to inconsistent document states if they are directly applied at remote sites. The operational transformation controller in the client adaptor is dedicated to file-level source code consistency maintenance, which has embedded mature OT mechanisms and techniques that were derived in prior works [21,23,25]. Whenever a local editing operation is detected and presented by the registered listener, it will be processed by the operational transformation controller, and then transmitted to the server. Whenever a remote editing operation arrives, it will be processed by the operational transformation controller, and then applied in the source code file.

5.3.2. Consistency Maintenance in Project-Level Sessions

Similar to file-level editing operations, when two project-level editing operations target source code items in one directory or in hierarchically-related directories, they may also conflict with each other and result in inconsistent source code trees. To support consistency maintenance in project-level sessions, we have adopted conflict resolution and consistency maintenance techniques that have been proposed and widely applied in prior studies [39–41].

In essence, three primitive project-level editing operations can be performed on the source code tree of a project:

- *createSourceCodeItem* (*n*, *parent*): to create a new source code item *n* in the directory *parent*.
- *removeSourceCodeItem* (*n*, *parent*): to remove an existing source code item *n* from the directory *parent*.
- *renameSourceCodeItem* (*on*, *nn*, *parent*): to rename an existing source code item from *on* to *nn* in the directory *parent*.

The registered listener (item *a* in Table 1) captures changes in the source code tree and produces primitive operations as defined above. Consequently, based on the conflict/compatible relationships among project-level editing operations, the operational transformation mechanism (deployed in the operational transformation controller) processes local and remote editing operations in a similar fashion to that in file-level sessions, and ensures source code consistency.

5.4. Smooth Transition in Real-Time Collaboration Sessions

As specified in *Design Objective C*, the system should support unconstrained and flexible real-time collaboration where programmers can join and leave a real-time collaboration session at any time. When an ongoing session accepts a new member, the server should deliver the latest source code to the member, and meanwhile, ensure the smooth transition of the session. After the new member joins, all session members should continue their work based on a consistent copy of the source code. To achieve the smooth transition in a reliable way, the following mechanism is deployed:

Client When a client requests to join an ongoing real-time collaboration session, the client sends a *REQUEST* message to the server.

Server When the *REQUEST* message arrives, the server broadcasts an *INIT* message to all existing clients in the session.

Client Upon receiving an *INIT* message from the server, the client temporarily prevents new local editing operations, and sends back an *OK* message to the server.

Server After receiving *OK* messages from all existing clients, the server further broadcasts a *COMPLETE* message to all clients, which signifies the completion of the procedure.

Client When the *COMPLETE* message arrives, a client resumes editing work immediately. Particularly, for the incoming client, the latest source code copy will be transmitted from the server to the client, following the *COMPLETE* message.

A graphical example is presented in Figure 5, which illustrates how the above mechanism works when a programmer (*D*) joins a session with three existing programmers (*A*, *B* and *C*).

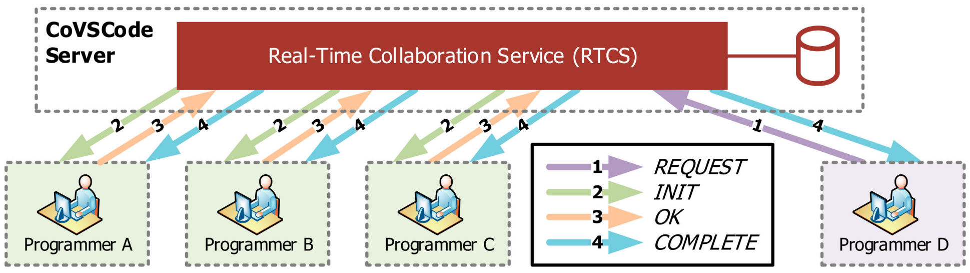


Figure 5. A graphical example illustrating the transition when a member joins the session.

In a real-time collaboration session, each client-server communication is based on a WebSocket connection, which is essentially a FIFO communication channel. When all clients in the session have received the *COMPLETE* messages, the entire session reaches a temporary state where (i) there is no message being transmitted, and (ii) all clients have executed the same editing operations, holding consistent source code copies. Eventually, the incoming client receives the latest source code, and all clients continue programming work based on a consistent source code copy as if a fresh session has been started. In addition, when a client leaves a session, the mechanism is relatively simple—the client simply sends a leaving request to the server, and the server removes the client from the member list. For both joining and leaving, notification messages will be sent to all existing clients.

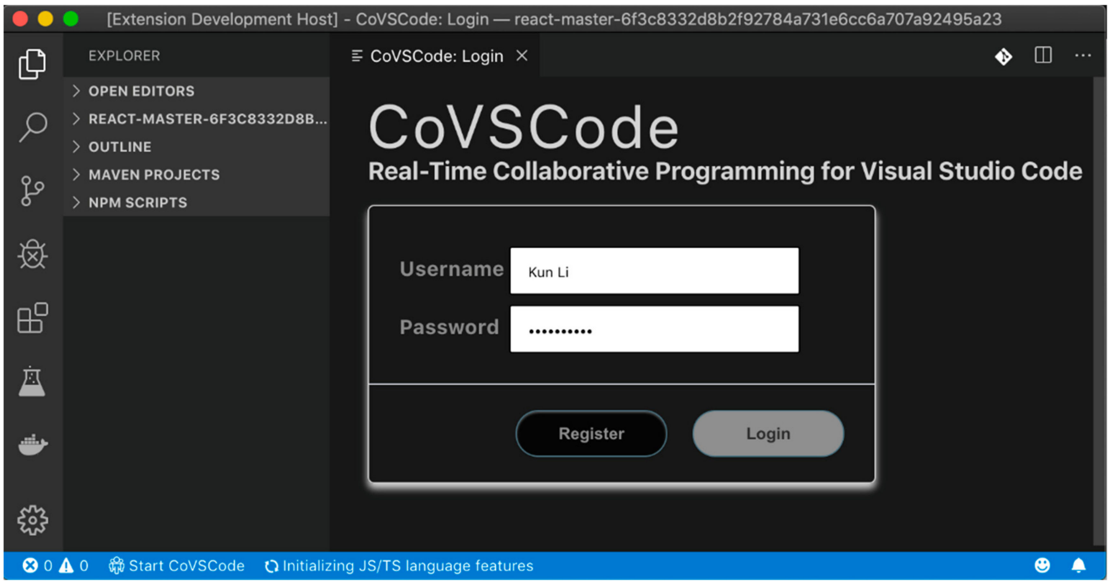
6. Prototype Implementation and Evaluations

Following the system architecture in Section 4.1, the workflow and functional design in Section 4.2, and various techniques proposed and devised in Section 5, we have successfully implemented the *CoVSCode* prototype. In this section, we demonstrate the prototype system and present preliminary user evaluations along with a comprehensive set of experimental evaluations.

6.1. Major User Interfaces of CoVSCode Prototype System

6.1.1. Login and Initialization: Preparation of Real-Time Collaboration

Figure 6 presents the *CoVSCode* client UI for a programmer to start real-time collaborative programming. Upon clicking the "*Start CoVSCode*" button in the status bar, a login panel is displayed in the IDE. The programmer may login into the system with an existing username and password, or simply register as a new user.


[Extension Development Host] - CoVSCode: Login — react-master-6f3c8332d8b2f92784a731e6cc6a707a92495a23

CoVSCode

Real-Time Collaborative Programming for Visual Studio Code

Username: Kun Li

Password: ••••••••••

Figure 6. UI snapshot of the *CoVSCode* client's login panel.

As presented in Figure 7, upon successful authentication, the programmer may create a team, join a team, or select an existing team that he/she has already joined or created, from the list on the left side. When a team has been selected, the programmer can further select a project that belongs to the team (or create a new project) from the list displayed on the right side.

When a new project is created, the programmer will be asked to specify a folder on the local disk, which contains the initial base of the source code for real-time collaboration. When an existing project is selected, the programmer will be asked to specify a local folder for hosting the project, and the latest source code copy will then be downloaded from the server to the specified local folder. Upon completion of the initialization process, the programmer starts to edit the source code in a real-time collaboration session.

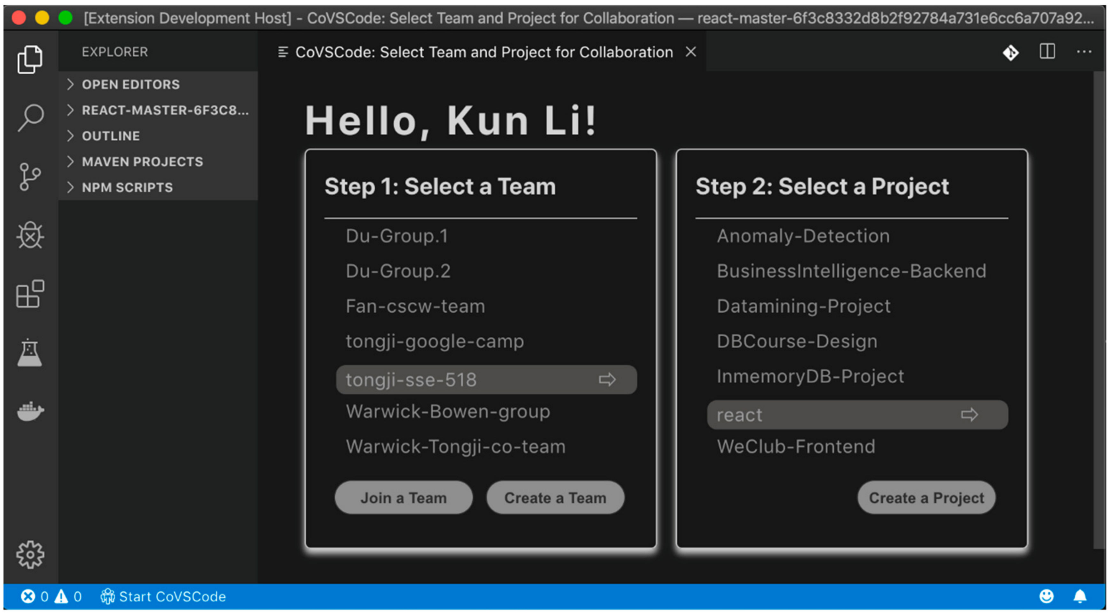


Figure 7. UI snapshot of the *CoVSCode* client's initialization panel.

6.1.2. Real-Time Collaborative Programming

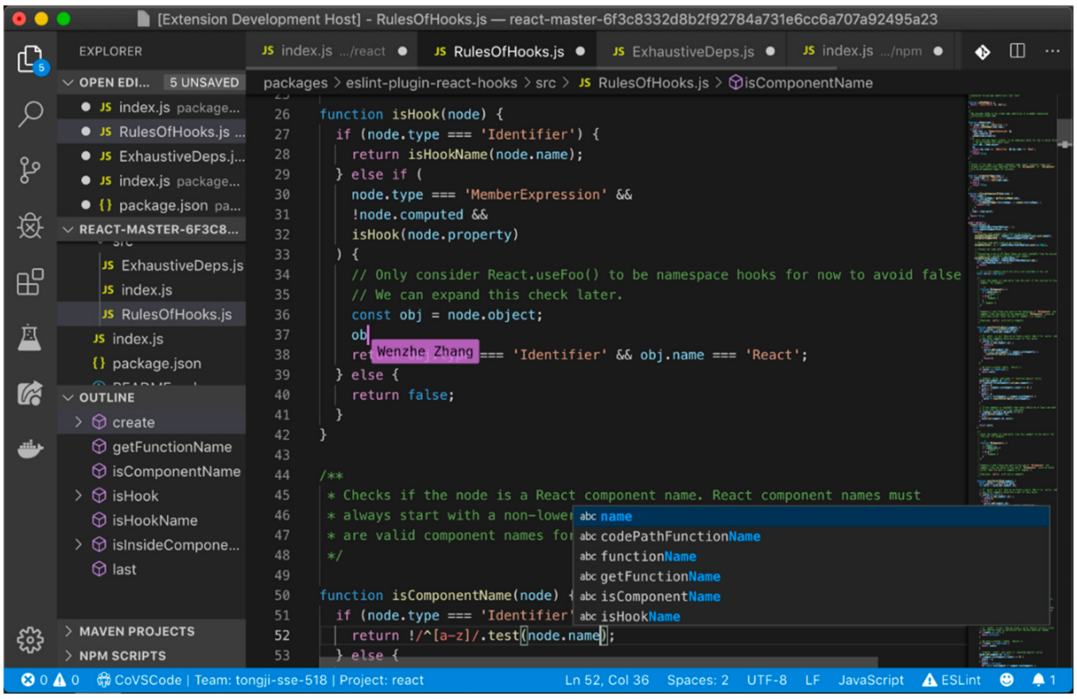
Following the initialization process in the previous step, the programmer starts to work in the real-time collaboration session. Figure 8 presents the UI snapshots of two *CoVSCode* clients used by two programmers, *Kun Li* (Site A, the upper figure) and *Wenzhe Zhang* (Site B, the lower figure), in a real-time collaboration session. At this particular moment, Site A (*Kun Li*) is editing Line 52 of the source code file, while Site B (*Wenzhe Zhang*) is editing Line 37 of the same file. Both of them can see each other's work in real-time.

As presented, the client user interface looks the same as the original single-user *VS Code* IDE, and all existing functionalities have been fully preserved. The document explorer on the left looks similar to that in the original *VS Code*, but real-time collaboration functionalities have been integrated: whenever the programmer creates, deletes or renames any source code item in the explorer, the operation is immediately transmitted to remote sites; and conversely, remote programmers' changes in the source code tree can also be replayed in the local explorer in real-time.

Similarly, the IDE's built-in source code editor has also been equipped with additional real-time collaboration features. During the programming process, each local operation is instantly detected and propagated to remote sites, while remote operations are also replayed and viewed in a real-time fashion. The source code consistency is always maintained by the system, provided that all operations have arrived at remote sites.

6.1.3. Collaboration Awareness and Notification

Collaboration awareness is an important feature in real-time collaborative programming that enables distributed programmers to learn where other users are and what they are doing. In closely-coupled real-time collaboration sessions, awareness features help programmers prevent potential conflicts, and improve mutual understanding about the relationships among concurrent programming tasks. In addition, awareness information must be intuitive for programmers to receive with minimal effort, and should not interrupt programmers' ongoing work.



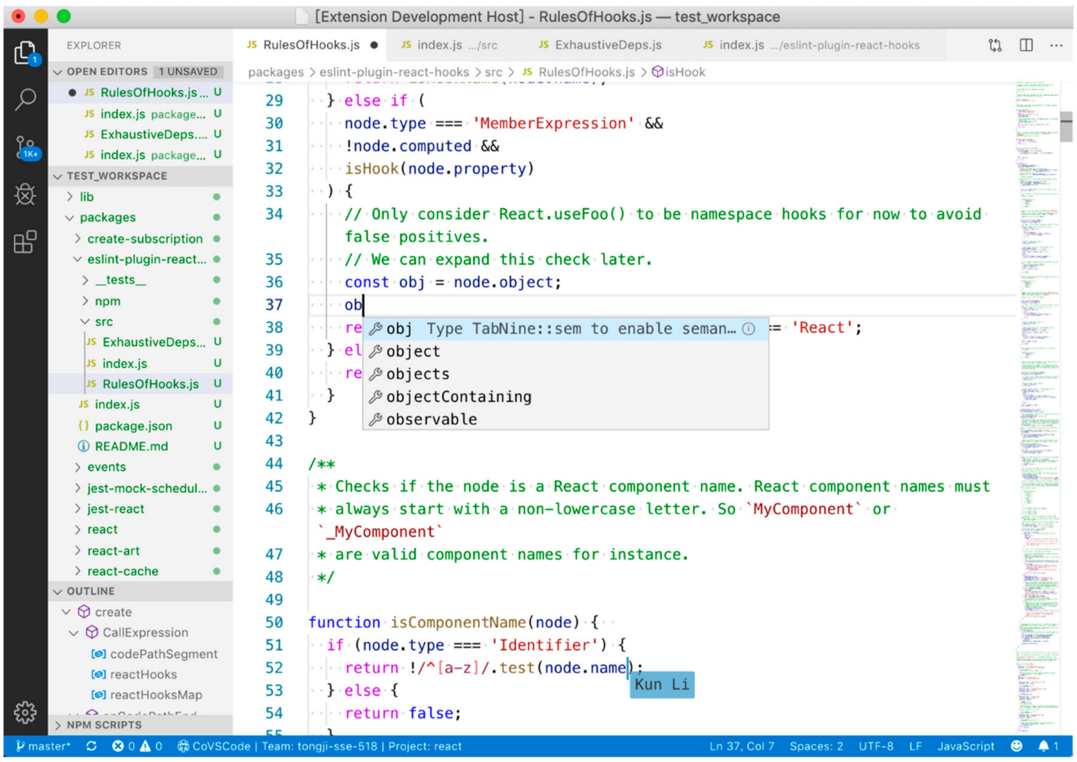
[Extension Development Host] - RulesOfHooks.js — react-master-6f3c8332d8b2f92784a731e6cc6a707a92495a23

packages > eslint-plugin-react-hooks > src > RulesOfHooks.js > isComponentName

```
function isHook(node) {
  if (node.type === 'Identifier') {
    return isHookName(node.name);
  } else if (
    node.type === 'MemberExpression' &&
    !node.computed &&
    isHook(node.property)
  ) {
    // Only consider React.useFoo() to be namespace hooks for now to avoid false
    // We can expand this check later.
    const obj = node.object;
    ob
    ret Wenzhe Zhang === 'Identifier' && obj.name === 'React';
  } else {
    return false;
  }
}

/**
 * Checks if the node is a React component name. React component names must
 * always start with a non-lower  abc name
 * are valid component names for  abc codePathFunctionName
 */                               abc functionName
                                  abc getFunctionName
function isComponentName(node) {  abc isComponentName
  if (node.type === 'Identifier'  abc isHookName
    return !/^[a-z]/.test(node.name);
  } else {
```

0 0 CoVSCode | Team: tongji-sse-518 | Project: react Ln 52, Col 36 Spaces: 2 UTF-8 LF JavaScript ESLint 1

(Site A)



[Extension Development Host] - RulesOfHooks.js — test_workspace

packages > eslint-plugin-react-hooks > src > RulesOfHooks.js > isHook

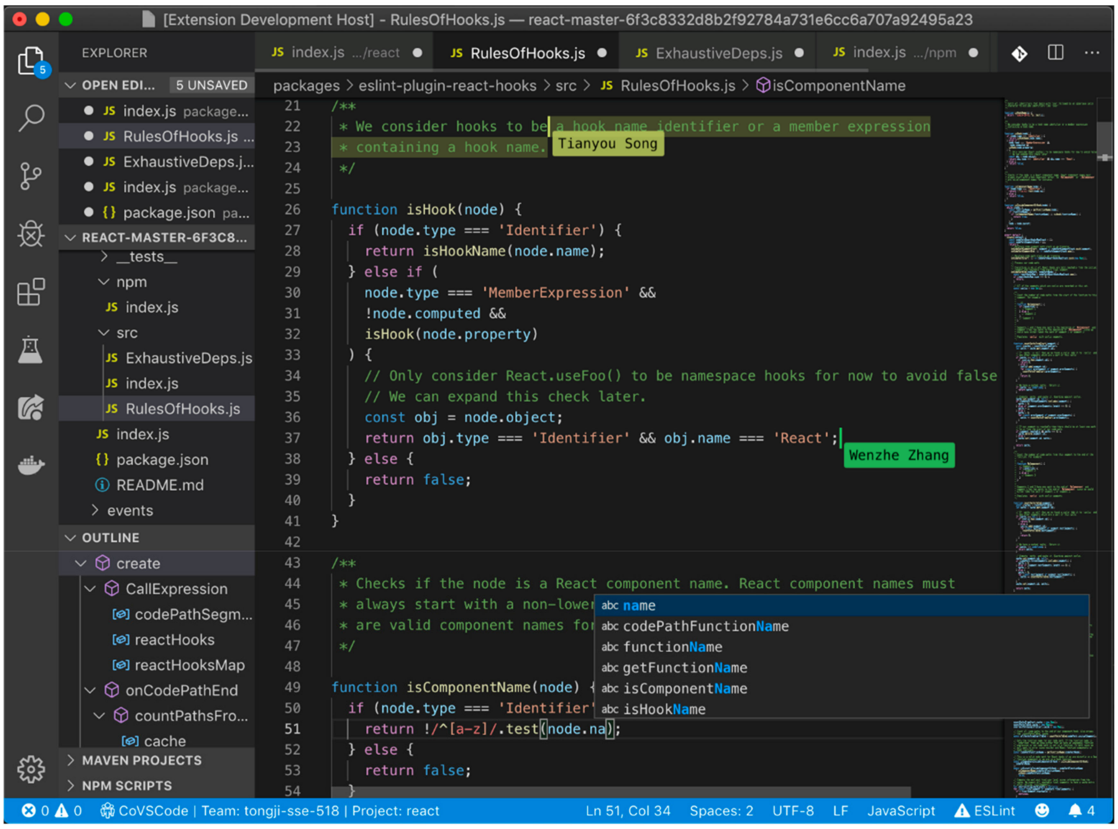
```
  } else if (
    node.type === 'MemberExpression' &&
    !node.computed &&
    isHook(node.property)
  ) {
    // Only consider React.useFoo() to be namespace hooks for now to avoid
    false positives.
    // We can expand this check later.
    const obj = node.object;
    ob
    re obj  Type TabNine::sem to enable seman... == 'React';
  } el object
    re objects
  }    objectContaining
}      observable

/**
 * Checks if the node is a React component name. React component names must
 * always start with a non-lowercase letter. So `MyComponent` or
 `_MyComponent`
 * are valid component names for instance.
 */

function isComponentName(node) {
  if (node.type === 'Identifier') {
    return !/^[a-z]/.test(node.name); Kun Li
  } else {
    return false;
```

master* 0 0 CoVSCode | Team: tongji-sse-518 | Project: react Ln 37, Col 7 Spaces: 2 UTF-8 LF JavaScript 1

(Site B)

Figure 8. UI snapshots of two *CoVSCode* clients in a real-time collaboration session.

Figure 9 illustrates the collaboration awareness features implemented in the *CoVSCode* client. In a real-time collaboration session, a unique color is assigned to each programmer and used to indicate the programmer's position (cursor) in the source code at remote sites. In addition, when a programmer selects a region of source code text, the highlighted region will also be displayed at all remote sites, which is particularly useful in a real-time online discussion over telephones or other communication tools. In this figure, it can be intuitively observed that the local programmer is editing Line 51; one remote programmer is editing Line 37; and another remote programmer is highlighting the source code comments in Lines 22 and 23.



```
21  /**
22   * We consider hooks to be a hook name identifier or a member expression
23   * containing a hook name.
24   */
25
26  function isHook(node) {
27    if (node.type === 'Identifier') {
28      return isHookName(node.name);
29    } else if (
30      node.type === 'MemberExpression' &&
31      !node.computed &&
32      isHook(node.property)
33    ) {
34      // Only consider React.useFoo() to be namespace hooks for now to avoid false
35      // We can expand this check later.
36      const obj = node.object;
37      return obj.type === 'Identifier' && obj.name === 'React';
38    } else {
39      return false;
40    }
41  }
42
43  /**
44   * Checks if the node is a React component name. React component names must
45   * always start with a non-lower
46   * are valid component names for
47   */
48
49  function isComponentName(node) {
50    if (node.type === 'Identifier'
51      return !/^[a-z]/.test(node.na);
52    } else {
53      return false;
54    }
```

Figure 9. UI snapshot of the *CoVSCode* client in a real-time collaboration session with three programmers.

Moreover, notification messages will be displayed in the bottom right corner of the IDE to show the currently active members when a programmer joins a session, and to indicate other collaborators joining and leaving during an ongoing session.

6.2. Preliminary User Evaluations

The *CoVSCode* prototype implementation validated the feasibility of the proposed approach and techniques. A group of programmers participated in preliminary user evaluations on the prototype system (Figure 10 illustrates a picture taken in the experiment), and provided positive feedback on several key issues related to real-time collaboration features and performance. Firstly, the programmers reported that all existing functionalities and UIs of the *VS Code* IDE were fully preserved without change, and the local responsiveness of editing operations was good (i.e., the same as that of the original IDE without real-time collaboration functionalities). Secondly, remote editing operations, remote cursors and remote highlights could be performed and displayed in real-time (without noticeable delay) in a LAN environment. Thirdly, when a programmer created a new project, the uploading of the initial source code copy took some time depending on the source code size, but even for large-size projects, it commonly took less than five seconds. Fourthly, when a new member joined an existing session, the transition would be completed in a timely manner (less than one second), and only the new member would encounter a slight delay for source code downloading during the initialization process (depending on the project size, and commonly less than five seconds).

Figure 10. A picture taken in a laboratory at the University of Warwick, where three programmers are participating in the user evaluation of the *CoVSCode* prototype.

During the user evaluations, we also discussed several usability issues with the programmers and received positive feedback. Firstly, the programmers reported that the prototype was easy to use, because the real-time collaboration features had been smoothly embedded in the original IDE without modification to existing features and user interfaces. Secondly, the programmers reported that the prototype was effective and useful in supporting real-time collaborative programming, and that they could use the prototype for diverse collaboration needs. Thirdly, the programmers reported that they were satisfied with the real-time collaboration features. Fourthly, the programmers found the user interfaces intuitive and helpful for supporting real-time collaborative programming, especially the colored-highlighting-based collaboration awareness features (see Section 6.1.3), which had effectively assisted them in being aware of collaborators' editing locations, understanding relationships among collaborative work, and preventing potential conflicts accordingly.

6.3. Experimental Evaluations

In addition to preliminary user evaluations, we further conducted a set of experimental evaluations on several key aspects critical to system performance and user experience. In each experiment, there were several collaborating programmers using the *CoVSCode* prototype system to conduct programming work based on an existing software project. To reflect real-world scenarios as much as possible, we selected six existing open-source projects from GitHub with diversity in project sizes and programming languages. Table 2 presents the names of the six projects, as well as the selected branch and commit, the size of the source code copy, and the number of source code items for each project.

Table 2. Six open-source software projects selected for experimental evaluations.

Project	Branch	Commit	Size (Bytes)	Items
PythonCrawler	master	d0a29c2deb36d9986321f38f97ff352c1d589135	46,935	28
koa	master	2c86b10feafd868ebd071dda3a222e6f51972b5d	357,477	103
flask	master	33db5351088f58219d0d13f1a2efa7a85880eb55	1,485,367	255
sharedb	master	f01cf0e53fa0347dc27232dc248de517914f395e	1,994,093	111
python-twitter	master	fd3479836aba62cecc479a8e1ec85c43ae48e8f8	13,617,305	261
react	master	6f3c8332d8b2f92784a731e6cc6a707a92495a23	12,025,260	1360

6.3.1. Performance of Local and Remote Editing Operation Processing

In a real-time collaboration session, whenever a local editing operation is performed, it will be detected (by the listener), processed (by the operational transformation controller) and propagated to the server; and whenever a remote editing operation arrives, it will be transformed and replayed. The processing times of these local and remote editing operations are key factors of user experience. To this end, we conducted 24 sets of experiments with two and three collaborating programmers (the most common scenarios) working on the six selected projects, respectively. During each experiment, we asked the participating programmers to concurrently edit the same source code file for at least five minutes, measured the processing times of at least 200 local or remote editing operations during the process, and finally calculated the average values. In terms of configuration, the experimental PC was equipped with an Intel Core i7 @ 2.2 GHz processor and 16 GB of RAM, and the operating system was macOS 10.14.6 (18G87).

Table 3 presents the experimental results, including the average processing times of local editing operations in the two-collaborator sessions (2C sessions) and three-collaborator sessions (3C sessions), and the average processing times of remote editing operations in the 2C and 3C sessions. It can be observed that the processing of local editing operations was very efficient (less than one millisecond for each operation), which demonstrates the high local responsiveness of the client system. The processing of remote editing operations was also efficient, which contributes to the instant replay of remote editing operations, and also implies that the duration from an operation's generation (at its original site) to its remote replay (at a remote site) mainly depends on its transmission over the network, rather than its processing upon arrival.

Table 3. Average processing times of local and remote editing operations in two-collaborator sessions (2C sessions) and three-collaborator sessions (3C sessions).

Project	**Local Editing Operation**		**Remote Editing Operation**	
	2C Session	**3C Session**	**2C Session**	**3C Session**
PythonCrawler	802 µs	803 µs	2.262 ms	0.833 ms
koa	876 µs	864 µs	4.650 ms	4.803 ms
flask	859 µs	817 µs	6.095 ms	5.940 ms
sharedb	852 µs	905 µs	2.801 ms	2.830 ms
python-twitter	853 µs	836 µs	8.080 ms	5.869 ms
react	867 µs	834 µs	2.044 ms	4.294 ms

6.3.2. Performance of Project Creation and Session Joining

Whenever a new project is created, the initialization process involves uploading of the initial source code copy. Whenever a member joins an ongoing real-time collaboration session, the joining process involves downloading of the source code copy (for the new member only). These processing times are important to the user experience. To this end, we conducted 12 sets of experiments to measure the average processing times of project creation and session joining based on the six selected projects, and the experimental results are presented in Table 4. The experimental PC's configuration is the same as that in Section 6.3.1 (i.e., Intel Core i7 @ 2.2 GHz processor, 16 GB of RAM, macOS 10.14.6 operating system).

It can be observed that even for large-size projects, the time costs for project creation and session joining were still acceptable. More importantly, such time cost for joining a session was only incurred at the new member's client, as the source code of the project is directly fetched from the server without involving other existing members.

Table 4. Average processing times of project creation (involving source code copy upload) and session joining (involving source code copy download).

Project	Size (Bytes)	Items	Project Creation	Session Joining
PythonCrawler	46,935	28	85.869 ms	75.698 ms
koa	357,477	103	274.540 ms	220.318 ms
flask	1,485,367	255	734.111 ms	599.690 ms
sharedb	1,994,093	111	777.637 ms	559.712 ms
python-twitter	13,617,305	261	3166.465 ms	3867.070 ms
react	12,025,260	1360	3923.432 ms	2480.144 ms

6.3.3. CPU Utilization and Memory Usage

In addition, two sets of experiments were conducted to measure the CPU utilization and memory usage of the *CoVSCode* client and the *CoVSCode* server, respectively. For both experiments, the experimental PC was equipped with an Intel Core i7 @ 2.2 GHz processor, 16 GB of RAM and the macOS 10.14.6 (18G87) operating system.

Firstly, we monitored the *CoVSCode* client system's CPU utilization and memory usage for a five-minute duration (with one sample collected per second), in which the programmer was conducting real-time collaborative programming work in a session. The experimental results are illustrated in Figure 11. As observed, the system's resource consumption was satisfactory in both aspects. Compared with the original *VS Code* IDE, the implemented real-time collaboration plugin (i.e., the client adaptor) did not incur significant extra cost in either the CPU utilization or the memory usage.

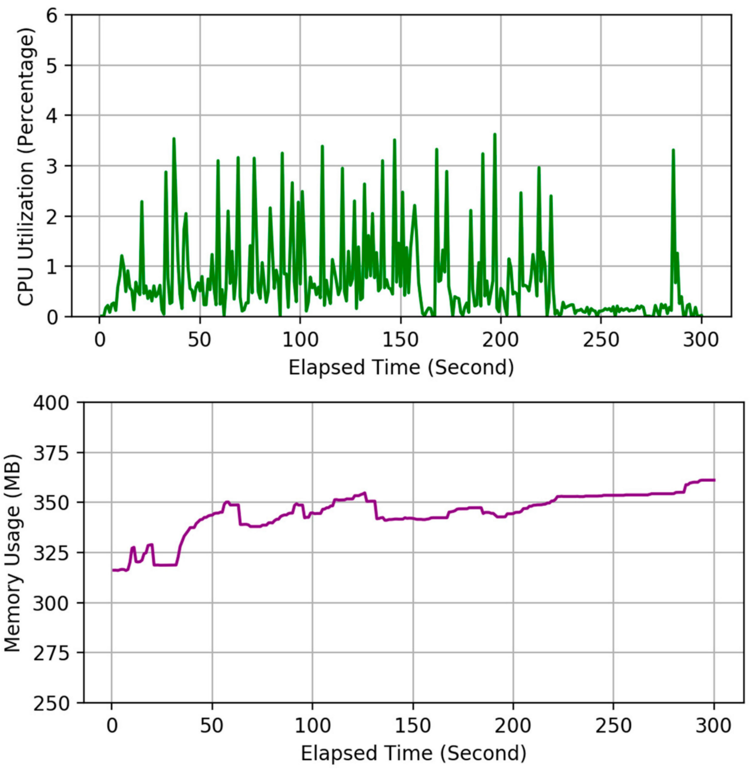


Figure 11. CPU utilization and memory usage of the *CoVSCode* client in a five-minute duration of a real-time collaboration session.

In addition, since *Live Share* is the official real-time collaboration plugin of *VS Code*, for the sake of comparison, we also monitored the CPU utilization and memory usage of the *VS Code* with *Live Share* in a five-minute duration (with one sample collected per second), in which the programmer was conducting real-time collaborative programming work in a session. The experimental results are illustrated in Figure 12. As observed, the system's resource consumption was also satisfactory, although it was slightly higher than that of *CoVSCode* presented above.

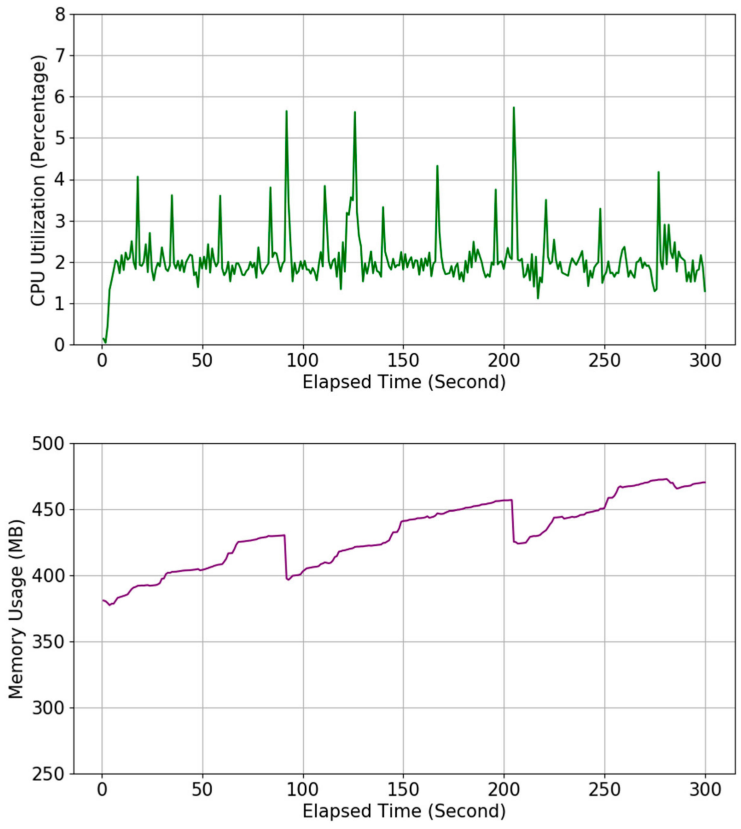


Figure 12. CPU utilization and memory usage of the *VS Code with Live Share* in a five-minute duration of a real-time collaboration session.

Secondly, we monitored the *CoVSCode* server system's CPU utilization and memory usage for a five-minute duration (with one sample collected per second), in which there was an active session with three collaborating programmers. The experimental results are illustrated in Figure 13. As observed, the resource consumption of the server system was satisfactorily economic, and one critical enabling factor was the design of the system architecture (see Section 4.1), which ensures the server to maintain only lightweight data (e.g., session members) and tasks (e.g., operation relaying), whereas most heavyweight jobs (such as local operation detection and remote operation replay) are distributed at each client. This also implies the system's good scalability for hosting large numbers of concurrent real-time collaboration sessions and active collaborating programmers.

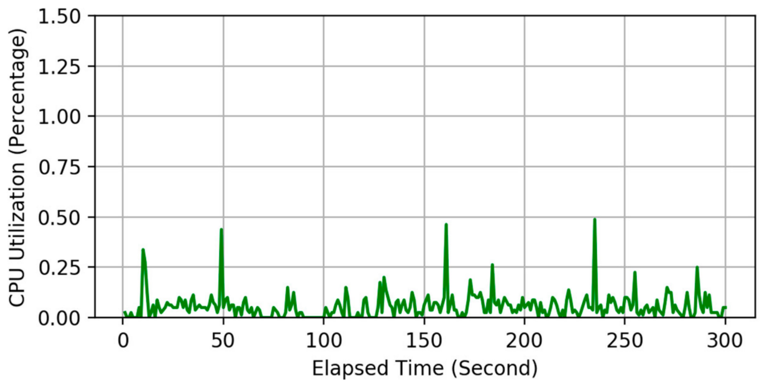


Figure 13. *Cont.*

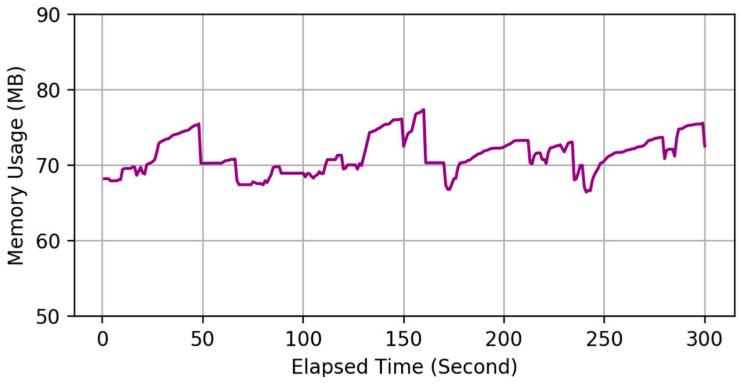


Figure 13. CPU utilization and memory usage of the *CoVSCode* server in a five-minute duration of a real-time collaboration session.

7. Discussion and Comparisons

In this study, we proposed, designed, and implemented a novel real-time collaborative programming environment for *VS Code*, one of the most popular and widely-used lightweight IDEs. It is worth pointing out that all approaches and techniques contributed in this work are generic, which can also be applied in building real-time collaborative programming environments for other lightweight IDEs such as *Atom* and *Sublime Text*. The generality of this work lies in several aspects. First of all, the three high-level design objectives and rationales (Section 3) were proposed in a generic manner, and are not specific to any concrete IDE—they could be used for guiding the design of any real-time collaborative programming environment. Secondly, in terms of the overall system architecture (Section 4.1), the underlying *VS Code* can be simply replaced by another IDE while all other components remain unchanged, and therefore the whole architecture is generic for supporting any real-time collaborative programming environment. Thirdly, the workflow and functional design (Section 4.2) was proposed in a generic way that conceptually defines how a programmer interacts with the environment, without binding to any specific IDE. Last but not least, all supporting techniques, implementation approaches, and user interface design could be straightforwardly reused for building any real-time collaborative programming environment. For example, the client system architecture (Section 5.2) could be reused with any IDE simply by replacing the underlying *VS Code*; the design of LOP and ROP (Section 5.2) is compatible with any IDE; the consistency maintenance techniques (Section 5.3) and session management techniques (Section 5.4) are reusable for any similar environment; and the UI design and collaboration awareness features (Section 6.1) are also applicable to any real-time collaborative programming environment. The only technical requirement in implementing another system of a similar nature is to investigate the APIs provided by the selected IDE, as well as to utilize the APIs following the descriptions in Table 1. This is usually not a challenge for most of the popular lightweight IDEs, as they have been designed for supporting extensions by nature, which commonly provide rich collections of APIs.

We compare this study with related works as follows.

Firstly, compared with other real-time collaborative editing systems contributed in prior works such as *CoWord* [27], *CoMaya* [29], *Co-AutoCAD* [30], *CoWebDraw* [32] and *Codox Wave* [33], this study focuses on supporting real-time collaboration for software development with lightweight IDEs, and has extended the application domains of the TA approach. Moreover, several technical issues involved in this study were not existing in those prior works. For example, the *CoVSCode* system manages two kinds of real-time collaboration sessions to support the simultaneous editing of several source code files and the source code tree, whereas in prior works there was usually a single shared document that was collaboratively edited.

Secondly, in the domain of real-time collaborative programming, one major advantage of the *CoVSCode* prototype compared with other research prototypes that have been contributed in the recent years (such as *CodePilot* [14], *CodeR* [9] and *CoRED* [6]) is its full compatibility with an existing popular and widely-used IDE (i.e., the *VS Code* IDE). Given the large user community of *VS Code*, there exists great potential for real-world impacts of this work, as well as opportunities for user studies with larger scopes in the future work.

Thirdly, compared with *VS Code*'s official real-time collaboration plugin (*Live Share*), this work establishes a significant difference, with several major advantages as follows:

1. *Live Share* supports programmers to collaborate in the host-participator pattern only, with several problems and limitations (as presented in Section 2.3), whereas *CoVSCode* supports unconstrained and flexible real-time collaboration patterns—programmers may work in the host-participator pattern (e.g., in pair programming sessions), and they may also work with precisely equal roles (i.e., everyone is the host) for meeting diverse collaboration needs (e.g., in agile software development processes, in collaborative learning, and in collaborative troubleshooting). For any software project maintained in the *CoVSCode* environment, each collaborating programmer (with permission on the project) can flexibly work on the project at any preferred time. Whenever a programmer starts to work in a day, the latest source code copy can be fetched, regardless of other collaborators' statuses.
2. With *Live Share*, only the host has the complete source code, and a source code file will be transmitted from the host to a participator only when the participator opens that file, the design of which affects the user experience (i.e., whenever the participator opens a file, it takes a while to show). Since every site has the complete source code with *CoVSCode*, whenever a programmer opens a file, it is immediately displayed and ready for editing, the same exact experience as in traditional single-user programming.
3. With *Live Share*, the host must be online at all times, and whenever the host loses the network connection for any reason, the whole session will be terminated, with all ongoing works at the participators' sites lost. With *CoVSCode*, every site has the complete source code, and each editing operation is saved at the server before further propagation. Therefore, there is no single point of failure in the session—whenever a client temporarily loses its connection with the server, no other collaborator is affected.
4. With *Live Share*, a participator cannot use IDE features that rely on cross-file code analysis (such as the "go to definition" feature) since they are restricted by the incomplete and memory-residing source code files, and no independent compilation, testing or debugging can be conducted. In contrast, with *CoVSCode*, each programmer has the complete source code copy on the local disk (which is exactly the same as in single-user programming), and thus enjoys all existing IDE functionalities and features such as language support, compilation and debugging.
5. The *Live Share* environment relies on Internet-based services hosted by Microsoft, which may not be suitable for teams with high security concerns. In contrast, the *CoVSCode* server can be simply hosted in a private cloud or inside an organization without Internet access, which is more suitable for meeting security requirements.

8. Conclusions

Software development requires effective collaboration among programmers with diverse skills and experience. In contrast to the conventional non-real-time collaborative programming which is supported by version control systems, real-time collaborative programming is an emerging approach that enables a team of programmers to view and edit shared source code documents at the same time. During a real-time collaboration session, each programmer's changes on the source code are instantly visible and integrated at other collaborating sites. Because of its wide range of application scenarios and benefits, this emerging technique has attracted increasing interests from both academia and industry. Lightweight integrated development environments (lightweight IDEs), such as *Visual Studio Code* and *Sublime Text*, have rapidly grown in popularity in the recent years due to their excellent customizability and extensibility. However, there are serious problems and limitations with existing real-time collaboration support for lightweight IDEs. For example, the two official real-time collaboration plugins (i.e., *Visual Studio Live Share* and *Teletype for Atom*) are the most sophisticated so far, but they only support the *host-participator* collaboration pattern with several limitations, and cannot meet diverse collaboration needs.

Motivated by the above observations, we have proposed, designed and implemented a novel real-time collaborative programming environment named *CoVSCode* that supports unconstrained and flexible real-time collaboration based on *Visual Studio Code*, one of the most popular and widely-used lightweight IDEs. We proposed three design objectives for the proposed environment, and then designed the overall workflow and system functions from collaborating programmers' perspectives. We also discussed major technical issues and solutions involved in system building and prototype implementation. Based on the proposed approaches and devised techniques, we implemented the *CoVSCode* prototype, and conducted a set of experimental evaluations that demonstrated the technical feasibility and satisfactory performance of the system in several aspects. In addition to the concrete *CoVSCode* prototype, all approaches, techniques and solutions contributed in this study are generic, which can also be re-applied to building real-time collaborative programming environments for other lightweight IDEs.

We are continuously working in the research domain of real-time collaborative programming environments, and several potential issues have been identified for future work. Firstly, we plan to propose and contribute more innovative features for assisting programmers in conflict prevention and collaboration awareness during real-time collaboration sessions. Secondly, we plan to design and implement a real-time collaborative programming environment for another lightweight IDE (such as *Sublime Text* or *Atom*) which will further test the generality of the approaches and techniques derived in this study. Thirdly, the *CoVSCode* prototype will be continuously developed and improved, with more in-depth real-world user evaluations.

Author Contributions: Conceptualization, H.F., B.D. and Y.S.; methodology, H.F. and Y.S.; software, K.L. and X.L.; validation, K.L., T.S. and W.Z.; data curation, K.L., T.S. and W.Z.; writing—original draft preparation, H.F. and B.D.; writing—review and editing, H.F., B.D. and Y.S.; visualization, H.F., B.D. and K.L.; funding acquisition, H.F. and Y.S.

Funding: This research was funded by the National Natural Science Foundation of China (No. 61702374, No. 61772371, and No. 61672128), the Shanghai Sailing Program (No. 17YF1420500), the National Key Research and Development Program of China (No. 2016YFB1000805, No. 2018YFB0505000, and No. 2018YFC0830406), the National Critical Science and Technology Infrastructure Program (National Seafloor Scientific Observatory, Tongji University), and the Fundamental Research Funds for the Central Universities. The APC was funded by the National Natural Science Foundation of China and the Shanghai Sailing Program.

Acknowledgments: The authors wish to thank Dongyi He, Zheng Zhao, and Yifei Bai from the School of Software Engineering, Tongji University, for their effort and support in prototype implementation.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. 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. In Proceedings of the 17th ACM International Conference on Supporting Group Work—Group'12, Sanibel Island, FL, USA, 27–31 October 2012; pp. 107–116.
2. Git. Available online: https://git-scm.com (accessed on 28 October 2019).
3. 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), Wuhan, China, 23–25 May 2012; pp. 94–101.

4. Goldman, M.; Little, G.; Miller, R.C. Collabode: Collaborative coding in the browser. In Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering—CHASE'11, Waikiki, HI, USA, 21 May 2011; pp. 65–68.
5. Goldman, M.; Little, G.; Miller, R.C. Real-time collaborative coding in a web IDE. In Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology—UIST'11, Santa Barbara, CA, USA, 16–19 October 2011; pp. 155–164.
6. Lautamäki, J.; Nieminen, A.; Koskinen, J.; Aho, T.; Mikkonen, T.; Englund, M. CoRED: Browser-based Collaborative Real-time Editor for Java web applications. In Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work—CSCW'12, Seattle, WA, USA, 11–15 February 2012; pp. 1307–1316.
7. Feldman, M.S. *CodeSync: A Collaborative Coding Environment for Novice Web Developers*; Wellesley College: Wellesley, MA, USA, 2014.
8. Guo, P.J.; White, J.; Zanelatto, R. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Atlanta, GA, USA, 18–22 October 2015; pp. 79–87.
9. Kurniawan, A.; Kurniawan, A.; Soesanto, C.; Wijaya, J.E.C. CodeR: Real-time Code Editor Application for Collaborative Programming. *Procedia Comput. Sci.* **2015**, *59*, 510–519. [CrossRef]
10. Rantala, M.R.J. Real-Time Collaborative Coding-Technical and Group Work Challenges. Master's Thesis, Tampere University, Tampere, Finland, 2015.
11. 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—CHI'17, Denver, CO, USA, 6–11 May 2017; pp. 6220–6231.
12. Deng, X. Group Collaboration with App Inventor. Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2017.
13. Fan, H.; Gao, J.; Zhu, H.; Liu, Q.; Shi, Y.; Sun, C. Balancing Conflict Prevention and Concurrent Work in Real-Time Collaborative Programming. In Proceedings of the 12th Chinese Conference on Computer Supported Cooperative Work and Social Computing—ChineseCSCW'17, Chongqing, China, 22–23 September 2017; pp. 217–220.
14. Warner, J.; Guo, P.J. CodePilot: Scaffolding End-to-End Collaborative Software Development for Novice Programmers. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems-CHI'17, Denver, CO, USA, 6–11 May 2017; pp. 1136–1141.
15. 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.
16. Sublime Text. Available online: https://www.sublimetext.com (accessed on 28 October 2019).
17. Atom. Available online: https://atom.io (accessed on 28 October 2019).
18. Visual Studio Code. Available online: https://code.visualstudio.com (accessed on 28 October 2019).
19. Stack Overflow Developer Survey Results 2019. Available online: https://insights.stackoverflow.com/survey/2019 (accessed on 22 August 2019).
20. Sun, C.; Ellis, C. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work—CSCW'98, Seattle, WA, USA, 14–18 November 1998; pp. 59–68.
21. Sun, C.; Jia, X.; Zhang, Y.; Yang, Y.; Chen, D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput. Hum. Interact.* **1998**, *5*, 63–108. [CrossRef]
22. 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.* **2006**, *13*, 531–582. [CrossRef]
23. Sun, D.; Sun, C. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Trans. Parallel Distrib. Syst.* **2009**, *20*, 1454–1470. [CrossRef]
24. Xu, Y.; Sun, C.; Li, M. Achieving Convergence in Operational Transformation: Conditions, Mechanisms and Systems. In Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work Social Computing—CSCW'14, Baltimore, MD, USA, 15–19 February 2014; pp. 505–518.
25. Sun, C. OT FAQ: Operational Transformation Frequently Asked Questions and Answers. Available online: https://www.ntu.edu.sg/home/czsun/projects/otfaq/ (accessed on 22 August 2019).

26. Sun, C. Reflections on cOllaborative Editing Research: From Academic Curiosity to Real-World Application. In Proceedings of the 2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD), Wellington, New Zealand, 26–28 April 2017; pp. 10–17.
27. Xia, S.; Sun, D.; Sun, C.; Chen, D.; Shen, H. Leveraging Single-User Applications for Multi-User Collaboration: The Coword Approach. In Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work—CSCW'04, Chicago, IL, USA, 6–10 November 2004; pp. 162–171.
28. Shen, H.; Sun, C.; Zhou, S. Leveraging Single-user OpenOffice Writer for Collaboration by Transparent Adaptation. In Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007), Qingdao, China, 30 July–1 August 2007; pp. 15–20.
29. Agustina, A.; Liu, F.; Xia, S.; Shen, H.; Sun, C. CoMaya: Incorporating Advanced Collaboration Capabilities Into 3d Digital Media Design Tools. In Proceedings of the ACM 2008 Conference on Computer Supported Cooperative Work—CSCW'08, San Diego, CA, USA, 8–12 November 2008; pp. 5–8.
30. Gao, L.; Lu, T. Achieving Transparent and Real-time Collaboration in Co-AutoCAD Application. *JUCS-J. Univers. Comput. Sci.* **2011**, *17*, 1887–1912.
31. Cho, B.; Ng, A.; Sun, C. CoVim: Incorporating Real-Time Collaboration Capabilities Into Comprehensive Text Editors. In Proceedings of the 2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD), Wellington, New Zealand, 26–28 April 2017; pp. 192–197.
32. Gao, L.; Gao, D.; Xiong, N.; Lee, C. CoWebDraw: A real-time collaborative graphical editing system supporting multi-clients based on HTML5. *Multimed Tools Appl.* **2018**, *77*, 5067–5082. [CrossRef]
33. Codox Wave. Available online: https://www.codox.io (accessed on 28 October 2019).
34. Visual Studio Live Share. Available online: https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsliveshare (accessed on 28 October 2019).
35. Teletype for Atom. Available online: https://teletype.atom.io (accessed on 28 October 2019).
36. Visual Studio Code: Programming Languages. Available online: https://code.visualstudio.com/docs/languages/overview (accessed on 28 October 2019).
37. Visual Studio Code Extension API. Available online: https://code.visualstudio.com/api (accessed on 18 August 2019).
38. VS Code API. Available online: https://code.visualstudio.com/api/references/vscode-api (accessed on 28 October 2019).
39. Agustina, A.; Sun, C.; Xu, D. Operational Transformation for Dependency Conflict Resolution in Real-Time Collaborative 3D Design Systems. In Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work—CSCW'12, Seattle, WA, USA, 11–15 February 2012; pp. 1401–1410.
40. Sun, C.; Wen, H.; Fan, H. Operational Transformation for Orthogonal Conflict Resolution in Real-Time Collaborative 2d Editing Systems. In Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work—CSCW'12, Seattle, WA, USA, 11–15 February 2012; pp. 1391–1400.
41. 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'16, Sanibel Island, FL, USA, 13–16 November 2016; pp. 61–70.