UNIVERSITY OF CALIFORNIA
SANTA CRUZ

THE DESIGN AND IMPLEMENTATION OF KEY APPLICATIONS FOR A LIVE, COLLABORATIVE ONLINE ENVIRONMENT

A dissertation submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Ethan Jonathan Papp

June 2015

The Dissertation of
Ethan Jonathan Papp is approved:

Professor Jose Renau, Chair

Professor Jishen Zhao

David Munday, PhD

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

The Design and Implementation of Key Applications for a Live, Collaborative Online Environment

by

Ethan Jonathan Papp

Modern micro-architectural simulation makes it difficult for concurrent users to collaborate on the work that they are doing in real time. This not only decreases efficiency, but also requires extra, error prone work to synchronize ideas and produce desired simulation or programming results. These problems require a solution of a collaborative work environment where users can study micro-architectural simulation, and at the same time, share live results with other users in a collaborative environment.

Traditionally, a non-collaborative alternative would require the user store their own copy of local files, manually manage access to different project directories and ultimately be responsible for security and what part of the work they share with themselves on different workstations or with other users. In a collaborative simulation environment, users can expect to share results of simulation, write collaborative code, share files, keep private versions of their own files, and seamlessly contribute to remote revision repositories, all while maintaining collaboration with their peers. Such a collaborative environment introduces interesting security and file revision synchronization problems as a result of abstracting the simulation and collaboration to a remote server, shared by all users of the collaboration environment.

This work focuses on the implementation of applications and features to make such a live, collaborative environment possible. Key features include:

- A strong client-server separation to eliminate risks of polluting files the user would otherwise not normally have access to.
- Designing and implementing a revision control flow that seamlessly integrates GIT and operational transform (OT) with the notion of live collaboration.
- Designing and implementing two key applications of the LiveOS: a terminal and a code search utility.

Acknowledgments

Jose Renau

For my education, guidance and funding.

David Munday, PhD and Professor Jishen Zhao

For their continued feedback and valuable thesis direction.

Sina Hassani

For pioneering the LiveOS and putting up with my continuous bickering.

Chapter 1

Introduction

> Nature is pleased with simplicity. And nature is no dummy.
>
> Isaac Newton

Cloud computing is a recent buzz word that, when practiced properly, has the capability of completely changing the standard developer's, designer's, debugger's and researcher's experience and work flow. The nature of cloud computing attracts users because of its inherent ability to keep users and content synchronized. Leveraging the power of the Internet, cloud computing also fosters collaboration as one of its methods of keeping content synchronized. Various fields in computer engineering are moving towards more collaboration and interactiveness which ultimately means shorter wait intervals when compiling code or sharing content and ideas. The problem is that, at this point, most environments that support cloud computing or interactiveness are specifically aimed at solving one problem and solve that problem extremely well. It is then obvious that any use other than the original intention is not worth the learning

curve, minor nuances or otherwise undesired changes in scenery; reverting back to the all so typical non-collaborative development work flow on a single desktop computer is *easier*. With this in mind, a collaborative cloud environment, designed with the intention of mimicking a typical desktop work flow *in addition to* offering the user an uncomplicated, novel way to collaborate in real time with other developers, designers, debuggers and researchers alike seems to solve the most common issues in current cloud infrastructures.

1.1 Collaboration In The Cloud

The LiveOS is a novel cloud collaboration tool that focuses on integrating functionality with efficiency. The LiveOS features a web based micro-architecture simulator, a collaborative code editor with operational transformation integration capable of recompiling various recognized programming languages on the fly, powerful and simplistic file and user control, and the ability to sandbox developer applications in a self contained, virtual operating system that runs in the cloud.

The LiveOS is also a collaborative workspace that lives in the cloud. It offers users a complete operating system experience. The LiveOS couples the power of an operating system with the simplicity of a website. As a result, users can log into the LiveOS and instantly familiarize themselves with the environment. The LiveOS includes features such as collaborative code editing, a complete file management system, micro-architecture simulation and live LaTeX document recompilation to see changes as they are typed. These features were implemented to allow programmers, designers, researchers and students the ability to accomplish collaborative

tasks in a shorter amount of time.

Using the LiveOS as a single user brings simplicity and efficiency to that user through the web browser, offloading any processing, be it micro-architecture simulation or code recompilation, to a powerful back end server. Using the LiveOS as part of a collaborative project however yields even more advantages. With an active Internet connection, any LiveOS user can take advantage of all LiveOS features. They can create new projects together and share files instantly. Using the collaborative code editor, many users can program and edit source code simultaneously, within the same document. With the added benefit of being platform agnostic, the LiveOS can be launched from anywhere a web browser can be launched.

1.2 Creating The Best User Experience

To keep the LiveOS as useful as possible, as we think of new collaborative ideas, we design a realistic use case that compliments our existing set of applications and then implement it into the LiveOS. For example, as a user's LiveOS project grows in the number of files they own, proper organization and security become critical. With many users and many files, it becomes easy for files to be in a quasi lost or unaccounted for state. Managing who is editing what file as well as who should be able to see changes to LiveOS files also becomes unmanageable. In terms of revision control, in a collaborative environment, owners of files become abstracted and a solution is needed. In this work, we focus on the design and implementation of key features which mitigate the aforementioned problems, allowing the LiveOS to inch ever closer to being the best user experience.

Chapter 2

Related Work

Black holes are where God divided by zero.

Albert Einstein

2.1 `Docker`

Docker is a cross platform development tool that can launch sandbox-like virtual machines in just a few milliseconds. These sandbox-like virtual machines are called *containers* and are completely cofigurable by the user. This tool is extremely powerful in that it is completely self contained and that it can launch Linux virtual machines for individual application development with little overhead. A developer can use a Docker container to launch an application, and in doing so, the application will live, run and generate files all within the container.

Docker is used within the LiveOS to separate applications from running directly on the LiveOS server. Separating applications increases security and control with respect to the

host operating system, whether the developer is testing new LiveOS application code or the application is production ready; docker containers are suited for debugging purposes as well as long term application use. Any application that launches new processes or relies on file system operations can be launched within a docker container and be completely self contained. The user or designer can inject commands into the docker and even expose ports on the docker container to communicate with it from the hose operating system.

2.2 Revision Control

2.2.1 GIT

GIT revision control is a popular choice for developers to control their working copies and backups of their code. The GIT infrastructure starts with a users remote repository. This repository typically holds the user's code files, binaries, READMEs etc. When the user wants to make an edit to the code, they must copy the entire remote repository to their workstation, creating a local repository. After the user copies the repository, they can modify and edit the files as normal. When they are ready to share their code changes with other users who also have access to the same repository, GIT allows the user fine grained control over what files are backed up, shared, kept local or thrown away with respect to what is stored on the remote repository. GIT revision control is extremely powerful because of the standardized GIT commands. Any user who knows the GIT commands can now manipulate and work with any repository they have access to. Refer to reference [5] for a complete explanation of GIT.

GIT within the LiveOS plays an important role. The desire to work with pre-existing,

external remote repositories with the LiveOS means that previously, users were required to manually copy the remote repository files into the LiveOS file system before beginning their collaborative work. The problem with this approach lies with revision control. Once a user has modified local repository files within the LiveOS, how do they merge or contribute their changes back to the remote repository? To solve this problem, some of our work focused on integrating GIT into the LiveOS such that users can seamlessly contribute to the remote repositories as well as solving some interesting issues that arose as a result of collaboratively contributing content while synchronizing with remote GIT repositories.

2.3 Operational Transformation, or OT

Originally proposed in 1989 [10], operational transformation was designed for multiple users to have access to the same workspace and collaboratively contribute to a single document in real time. That means that users would be able to add content while, and at the same time, all other users of the same document can see content changes instantaneously as well as make their own edits. This type of document creation is vastly different from the standard development model of contributing changes to content, synchronizing those changes, and then publishing those changes to all other users of the same content. The standard development model requires the other users to discretely obtain information from all other collaborators in real time, using some sort of revision control, such as GIT, perforce or simply manually copying the files of interest. Operational transformation focuses on the seamless fluidity of contributing content with other users in real time.

Within the LiveOS, collaboration is paramount. As such, integrating operational transformation integration was important so as to bring increased functionality over the standard document creation and modification paradigm. Some of our work focused on increasing the flexibility of OT within the LiveOS, bringing functionality such as undoing changes per character per person per file. Such functionality inherently causes scalability problems and we attempt to solve those problems with our implementation of OT revision control in Section 8.

2.4 `Code search`

Code search is a grep-like search utility that is written in the `Go` programming language [7] and composed of two main components, `csearch` and `cindex`. Originally proposed in 2006 [7], code search was aimed at being an extremely fast searching tool for Google's large code base. Using an indexing tool called `cindex`, code search can launch search queries with regular expressions against a pre-populated index file which is composed of word trigrams from the files the user is interested in searching. Code search was originally designed as a utility for Google, letting employees search through many files of source code quickly.

Within the LiveOS, code search plays an identical role. By running `cindex` with a list of directories or files to include in the index, the LiveOS can logically generate index files for users to search against. In our work, we propose a back end utility that can run the `cindex` utility in an efficient manner, in the background and without user interaction. After the index has been generated or regenerated after file changes, running `csearch` through a front end user application with or without a file regular expression will search through the previously

generated index file and return relevant search results as matched in the index file. In Section 6, we explain our implementation of the code search utility in the LiveOS.

2.5 `Ag` or `Ack-grep` or `The Silver Searcher`

The `Ag` utility is a code searching tool similar to the common `Ack` and `grep` command line searching tools [1]. Feeding `Ag` a search query followed by the directory to search returns results much quicker than other searching utilities like `Ack`. In fact, `Ag` is an order of magnitude faster than `Ack` [1] and as such, is an ideal, large scale code searching tool. Using `pthreads`, `mmap`'d files for extremely fast virtual memory access and the `Boyer-Moore strstr` string searching algorithm [1], the `Ag` tool is able to return string matches in the searched directories muck quicker than other searching utilities.

Within the LiveOS code search application, certain queries require specifically formatted output. The `Ag` utility is able to output the format and as such, is integrated into the LiveOS as a piggyback utility to the code search application. See Section 6.3.2.2 for an explanation of how the `csearch` and `Ag` utilities work together.

2.6 Other Collaborative Environments

2.6.1 Etherpad

2.6.1.1 What it is

Etherpad is a collaborative code editor allowing many users to write code in real time with other users. As a user modifies one section of code within some file named `file1`, another user, or many other users, can modify code in the same `file1`. There is also the ability to chat within Etherpad, allowing the users to easily share ideas and then implement them collaboratively in real time [2].

2.6.1.2 How its different

Etherpad is a self contained code editor that allows multiple users to collaborate on a single file in real time. Within the LiveOS, a collaborative code editor modified specifically for the LiveOS, codemirror, is capable of allowing multiple users to collaborate on a single file in real time, in addition to supporting many programming languages and on the fly recompilation for some languages. The LiveOS includes a chat application as well, letting users communicate ideas and then put them into practice using the collaborative code editor.

2.6.2 Google Docs

2.6.2.1 What it is

Google docs is a document editing platform that allows users to use a web browser to concurrently modify and edit a document in real time with other users. Conflicts between

different user versions or revisions of any document are no longer an issue, as is with a typical revision control flow such as GIT, SVN or perforce. As users change a document, it is synchronized to a remote Google repository in real time, allowing all users to see all of the changes happening on a single document, as it happens. Because Google Docs knows all changes from all users at any given time, any user can also undo any changes they have made or view an older version of the document. When a user wants to capture a version of the document, they can export the document to their local file directory for off-line editing. In addition, an owner of a document can selectively assign different permissions to different users, namely read-only or read-write [9].

2.6.2.2 How its different

Google Docs uses `OT` [6] to allow many concurrent user to edit the same file and see each others changes in real time. Google Docs does not, however, allow users to collaboratively edit code in various recognized programming languages and have that code recompile on the fly for output within seconds. Although Google Docs uses `OT` and all users can see live changes being made by other users, because Google Docs only allows users to edit and collaborate on syntactically unrecognized content, it falls short of anything other than a collaborative word processor.

Within the LiveOS, a code editor application with integrated `OT` allows collaboration in plain English as well as syntactically recognized programming languages for collaborative word processing *and* programming. See Section 2.3 as well as chapter 8 for an in-depth explanation of how the LiveOS takes advantage of OT to improve the user experience.

2.6.3 Cloud9

2.6.3.1 What it is

Cloud9 is an on-demand software testing service. It is a service that runs in the cloud and allows many users to collaboratively program and test code. Contrary to normal software testing flows, the collaborative software is tested against a remote set of scalable compute clouds, increasing software revision efficiency. Using parallel symbolic execution, a technique that can explore all feasible execution paths in a program, Cloud9 is able to balance execution workload on large clusters of computers, aggregating memory and CPU resources [8].

2.6.3.2 How its different

Cloud9 increases work flow efficiency by allowing many users to collaborate on a single document in real time. It also further increases efficiency by aggregating resources for compilation and running of applications. Cloud9 does not however provide a complete collaborative experience. The LiveOS picks up where Cloud9 falls short, providing users with operating system-like features such as the ability to chat with other users, a detailed file hierarchy, a full terminal application giving users explicit control of their work environment and live, on the fly collaborative programming, distributed resource, recompilation.

2.6.4 Koding

2.6.4.1 What it is

Koding is a web based integrated development environment (IDE) that allows users to experiment with virtual machines, sudo level terminal access and collaborative programming. Users can collaborate in real time programming in languages like Ruby, Go, Java, NodeJS, PHP, C++, C, Perl, Python to develop production ready code or can experiment with Docker integration, allowing full sudo access to a cloud based Linux operating system. Koding offers collaborative programming, root access to cloud based virtual machines and a white board for sharing ideas [3].

2.6.4.2 How its different

Koding is an excellent web based IDE for collaboratively programming and deploying code. The collaborative environment is focused on giving users full control of their web environment to develop code. As a result, Koding lacks focus on features that promote more complete seamless work flow, including the notion of project and file organization, seamless back end GIT integration, and an increased sense of security and simplicity. It is features like these which make the LiveOS a more complete operating system and user experience in the cloud.

2.6.5 Nitrous.IO

2.6.5.1 What it is

Nitrous.IO is yet another web based IDE capable of collaborative programming and utilizing Docker integration, file searching and a built in terminal application. Nitrous.IO is aimed at helping many users synchronize their concurrent work without using a pre-existing revision control scheme. Nitrous.IO contains many useful collaborative features and allows the users virtually all of the control they would need to mimic the work flow of a typical desktop workstation. [4].

2.6.5.2 How its different

Nitrous.IO is a well rounded web based cloud computing collaborative environment. Nitrous.IO does not however *abstract* most cumbersome functionality away from the user. Instead of making the work flow more efficient in terms of keystrokes and file management, Nitrous.IO has taken the same local desktop functionality and put it inside of a web browser. In contrast, the LiveOS has integrated the same functionality, but improved the use cases in terms of efficiency. Nitrous.IO, although supporting GIT revision control, sticks to the standard GIT paradigm, requiring users to handle revisions, code changes and synchronizations manually. LiveOS takes this manual flow and improves upon it to give the user a seamless experience, letting the back end logic handle the manual GIT control flow.

Chapter 3

LiveOS

Necessity is the mother of invention.

Plato

3.1 What is LiveOS?

In recent years, programming productivity and efficiency have benefited from common features like Google Docs and revision control schemes such as Github. As collaboration continues to drive the increase in productivity, novel methods for contributing changes and working together with other programmers are needed.

The LiveOS is aimed at being a complete suite of operating system like applications, all within the web browser, fostering collaboration and efficiency. In opposition to a typical local development flow, users looking for a collaborative work flow can, with an Internet connection, log into the LiveOS from any workstation and converse, manage files, contribute to files and subsequently to remote GIT repositories, in real time with other collaborators. In addition, the

LiveOS includes a micro-architectural simulation tool named ESESC. Combining collaboration with architectural simulation means that instead of using a personal workstation to simulate, gather and analyze micro-architecture results, manage source code files, edit code, seamlessly back up the contributed content and share the results of your simulations with other users in real time, we propose the LiveOS. A user can use the LiveOS to mimic the flow of work on their traditional workstation, with the added benefit of being able to collaborate with other users in real time. Multiple users can be logged in to the LiveOS and as a result, they can see and share in real time: code, project files, configuration docs, latex papers and simulation results.

3.1.1 Design Philosophy

The LiveOS is designed with security, simplicity and scalability as its first priorities.

3.1.1.1 Security

Security plays a large role in the development and use of the LiveOS. As the LiveOS continues to have new applications developed, improper intentional and unintentional access to LiveOS core functionality is a large concern. However, after the integration of a core utility named Docker, the LiveOS applications can live, run, generate files, and be freely destructive. This is because Docker is the main application that allows the LiveOS's code base to continue to grow while staying secure. Docker allows launching and developing applications inside of a lightweight virtual machine. If a vulnerability goes unnoticed or an application is experimentally destructive, the LiveOS server and file system are left unharmed. Docker virtual machines, or containers, are self-contained and self sufficient. The user can hook into the container and

control the processes that run, the packages that are installed, or even continue to develop their application and test with it in real time, all while being within a virtual machine that is within a web browser.

3.1.1.2 Scalability

The LiveOS is built with a client-server paradigm, leveraging the power of NodeJS to develop client applications that run within a web browser but can offload work, information and load balance on the back end server, running NodeJS.

3.1.1.3 Simplicity

Simplicity within the LiveOS is a large concern, with similar web based operating systems being overly complicated and difficult to understand. The LiveOS introduces the user to all applications, features and settings from one main menu in the top left corner of the LiveOS home screen. Clicking on this menu gives the user options including:

- Apps
- File Manager
- People
- Settings
- Projects
- Logout

Figure 3.1: LiveOS main menu options

From these six menu options shown in Figure 3.1, all of the LiveOS features are available to the user. Clicking on `Apps` shows the user a list of all 5 currently available LiveOS applications (refer to Section 3.3.1 for application information). Clicking `File Manager`, `People` or `Settings` will open a separate application window, showing a file hierarchy, a list of users currently online, or the LiveOS settings menu, respectively. The `Projects` and `Logout` options will take the user to a different screen entirely, allowing them to select a different project to work within or to log out of the LiveOS, respectively.

Simplifying the process to launch applications and to use the interactive functions of the LiveOS, such as keyboard shortcuts and individual project profiles, allows users to begin efficiently accomplishing collaborative code edits or contributing to a project with extremely minimal ramp up time.

3.2 Why LiveOS?

The motivation behind the LiveOS is increased collaboration in addition to simplifying the flow of micro-architectural simulation by way of novel simulation methods and collaborative user applications. The notion of increased ability to simulate and gather results comes directly from the live simulator and supporting collaborative file managing and file editing applications.

3.2.1 Collaboration

A live collaboration code editor named *codemirror* is a centralized editor within the LiveOS, suited to support editing and creating content from simple word processing tasks to on the fly recompiled LaTeX documents, to editing content in your favorite programming language. This editor supports operational transformation (OT) to allow truly *live* collaboration with other LiveOS users. In addition, the LiveOS furnishes the user with a chat application to synchronize ideas with other users before implementing them in the code editor. Another notion of collaboration within the LiveOS includes a user-managed file hierarchy, organized into dedicated LiveOS *projects*. Each project can consist of one or many users who can control the file permissions they grant other users within that project, or other projects.

3.2.2 Feedback

Live feedback is essential to a fluid, live, collaborative environment. The LiveOS provides live feedback to its users in a couple of different ways.

1. Live LaTeX recompilation

2. Live ESESC recompilation based on configuration changes

3. Instant code and file search results using the `code search` application, even with thousands of files

4. On the fly source code syntax highlighting as well as live source code recompilation for supported programming languages

3.2.3 Cloud-like

The LiveOS gains much of its appeal as a collaborative environment by leveraging a back end server which launches and supports a front end web page with which clients can connect. The notion of allowing clients to connect to a front end web page that communicates with a cluster of servers running a NodeJS environment to compile code, run different application and in general, develop code, resembles much of what we call today `cloud computing`. The LiveOS employs a set of servers to execute the back end file and task processing which lends way to easily synchronizing front end user collaboration. This, in essence, gives users a front end operating-system-like user interface to accomplish typical server farm tasks, all while being extremely usable.

3.3 General Organization of LiveOS

The LiveOS model is composed of clients and one or many back end servers. The server is based on a standard Arch Linux operating system with Nodejs running on top of it.

The Nodejs framework hosts the LiveOS file system and all applications that need back end communication. Web sockets allow and manage the client server connection. Each application in the LiveOS includes client side and host side code. The client side code is run on the client computer and handles events such as rendering the view, handling data returned from the server and keeping the individual client state separate from other clients and connections.

The server handles socket connections to each client. Over the sockets, client information is sent, usually separated by application type. For example, a terminal application will use a web socket to send data only relevant to itself to the back end server. On the back end server, a terminal application server is running, which catches and interprets the client terminal application messages. For a single user connection to the LiveOS, a maximum of 7 different web sockets can be active for that one user at any time. One web socket per application handles the majority of that application's communication. In the presence of many users, each application socket connection must be unique, so as to ensure separate application sessions. For example, if ten users are using the terminal application, the client and server must have a way to differentiate between which instance of the terminal to send information to. This same requirement applies to all LiveOS applications.

Each client begins with a unique socket connection to the server. The user can then launch applications.

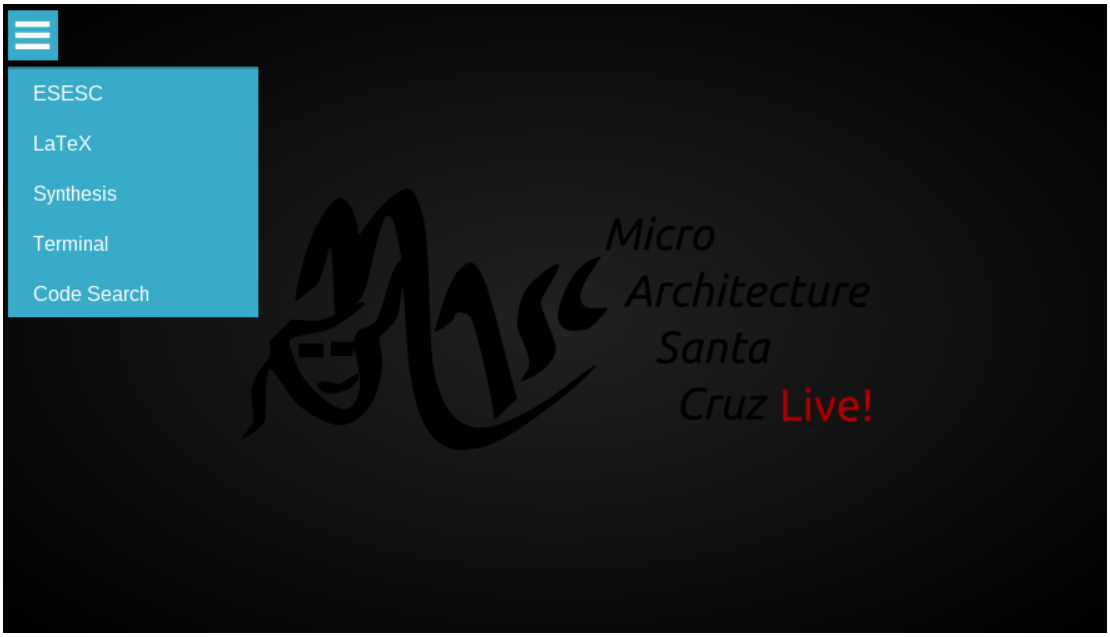


Figure 3.2: LiveOS main menu applications

3.3.1 LiveOS Applications

ESESC ESESC is a fast micro-architecture simulator designed to simulate and verify processor architecture. The LiveOS includes, as a standalone application, a *live* version of the ESESC simulator, allowing users to simulate micro-architecture and receive accurate feedback within seconds. Users can also share results in real time with other LiveOS users.

LaTeX LaTeX integration into the LiveOS allows users to collaboratively write LaTeX documents with other LiveOS users. In addition, LaTeX documents can be recompiled in real time, allowing users to view the up to date compiled LaTeX document as they write.

Synthesis Synthesis in the LiveOS allows users to simulate FPGA designs with on the fly recompilation.

Terminal The terminal application is meant to give the LiveOS user complete control over their environment. The LiveOS terminal emulates a full Linux terminal, giving the user sudo access within a completely self contained Linux environment. The LiveOS supplies the user with private, shared and administrative terminal variants, each with different file access and sharing options.

Code search Potentially having thousands of files, LiveOS users need a method for quickly searching through their many files for a file to edit or a piece of code to extract. The code search application is a single application with eyes into all of the user's files. The user can use code search to filter through large amounts of files and find a single line of code within seconds.

File Manager The LiveOS presents the user with a file manager application to keep track of files and to increase organization. The file manager behaves like most operating systems file management hierarchies and supports adding, deleting, exporting, renaming and moving files.

People Within LiveOS projects, users have access to an application that shows all other users currently using the LiveOS. Within this application, users can also chat with each other and grant or remove access to projects to other users.

Settings The settings application within the LiveOS contains options like the default editor selection and editor theme selections. In addition, the settings application allows users to set shortcuts for hot-key actions as well as store GIT credentials for cloning repositories within the LiveOS.

Projects The projects page of the LiveOS allows users to enter an existing project or create a new one. Projects encapsulate user files as well as establish the notion of separate work spaces. See below in Figure 3.3 for a general organization of the functionalities described in this work.

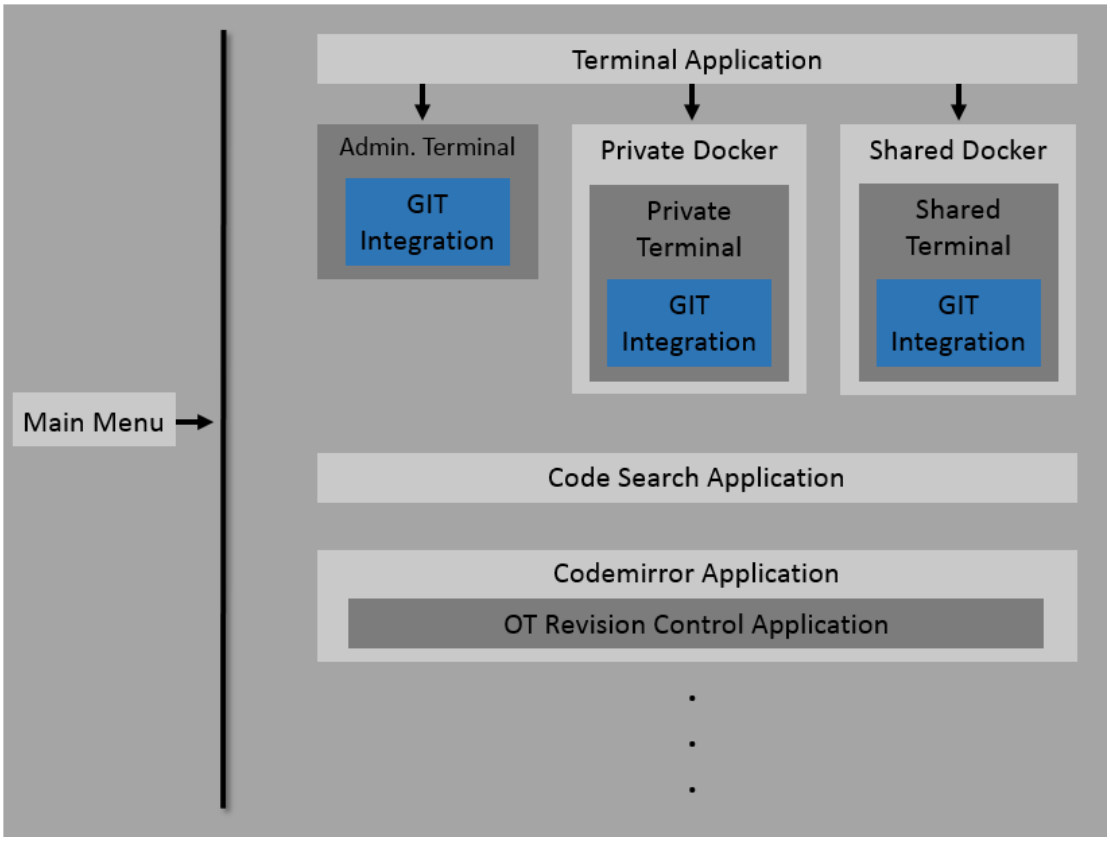


Figure 3.3: General LiveOS organization

3.3.2 Application Design Principals

3.3.2.1 Front End Design

While building the front end of a LiveOS application, we focused on the following design principals:

- Keep it as simple as possible.

- Avoid any unnecessary clutter in the user's view.
- Keep all necessary application options within the application's menu.
- Abide by the LiveOS's color schemes.
- Default to the codemirror application for all text and content editing tasks.
- No 'save' buttons.

Following these principals helps keep the LiveOS client side code compact and lightweight.

3.3.2.2 Back End Design

While developing the back end of a LiveOS application, we focused on security and scalability in the form of:

- Create a new NodeJS application server per application to avoid blocking execution.
- Use docker where possible to keep potentially harmful applications from interacting directly with the LiveOS server.
- Use encrypted TCP connections to communicate with dockerized applications for increased security.

The above guidelines will help maintain LiveOS usability, scalability and security.

Chapter 4

LiveOS Module Developed: Docker Integration

If you can't explain it to a six year old,

you don't understand it yourself.

Albert Einstein

4.1 Introduction: Docker in the LiveOS

Docker is a platform that allows users of the LiveOS to have the freedom to develop, compile, manage and manipulate content on almost any operating system, within just a few milliseconds. Docker's ability to spawn lightweight virtual machines almost immediately and subsequently destroy them in the same amount of time, make it ideal for an application to be built around it or exist completely inside the Docker instance, leveraging the powerful Docker layered file system to manage what would normally be a large amount of overhead (launch-

ing an operating system, loading project files). Because Dockers are an extremely lightweight virtual machine, once launched, an application built within Docker is essentially free to be as destructive as it wants as there is total separation between the Docker virtual machine operating system and and the LiveOS operating system (and the file system that is running the LiveOS environment itself). For example, most applications within the LiveOS launch new processes on the server that is hosting the environment. File operations, user operations and mistakes, malicious behavior or not, can potentially cause detrimental damage to the LiveOS host operating system. For these reasons, we integrated Docker into the LiveOS so as to supply LiveOS applications with a sandbox to compile applications, launch processes, work securely and maintain file separation; it is the infrastructure backbone that the LiveOS relies on to maintain user, process and application separation, be it for security or reliability.

Naturally, applications of the LiveOS benefit from Docker integration. With Docker integration, the designer has complete control of what information, if any, is sent from the LiveOS server to an application running within a docker container(a *dockerized* application). Additionally, they can control what information makes it out of the docker and into the LiveOS server. Some examples of communication include file transfers from the docker operating system files to LiveOS files in addition to control messages that are sent when the dockerized application needs information from the server to complete a request (see Section 5.4.1). The ESESC application also uses Docker to compartmentalize the compilation and runtime operations.

4.2 Why Docker Is Needed

Docker plays two main roles within the LiveOS, and these roles are best described in terms of the Terminal application. This is because the terminal application is currently the most substantially integrated with Docker. Described in Chapter 5.1, the Terminal application leverages Docker to accomplish reliability and security.

4.2.1 Environment Reliability

By compartmentalizing applications, the LiveOS as a whole is much less likely to crash or experience reliability problems related to individual applications once those applications are launched and running within a docker container. The application is, as a whole, wrapped and contained by the container it is running within. Because communication into and out of the container is handled by the application designer, dockerized applications become a sandbox for the user to work and experiment within.

4.2.2 Environment Security

Once implemented, an application launched within a Docker container is, by default, severed from all outside communication. From this point, it is completely up to the designer to decide what ports and or information are available to the application within the container. By completely severing communication to the application, back doors and application vulnerability are no longer an issue.

4.3 Design

The design of Docker containers within the LiveOS is best explained through our implementation of Docker with the terminal application. Currently, Docker integration is being used for two applications: Terminal and ESESC. The Terminal application within the LiveOS was an excellent candidate to begin Docker integration with because of its potential security issues when exposing it to a LiveOS user. Originally, a terminal in the LiveOS was launched via spawning a TTY process directly on the LiveOS back end server (see Section 5.3.2.1). Because allowing any LiveOS user a direct connection a process running on the LiveOS server is extremely dangerous, we decided to compartmentalize the individual TTY processes using Docker.

4.3.1 Launching Dockers

Currently, Docker within the LiveOS handles the task of compartmentalizing the terminal application. This means that when a user launches a docker terminal within the LiveOS, instead of launching a terminal process directly on the LiveOS server, the terminal process must be launched inside of a docker that is launched by the LiveOS server.

4.3.1.1 `docker run`

The `docker run` command is used to launch a docker container. At the time of launching the LiveOS, all docker containers must be launched using the `docker run` command. This command handles running the docker as well as any associated setup that is required. For example, at the time of launch, we specify the docker container to run, the path

to the LiveOS file mounts and the read-write permissions for those file mounts. Immediately after running the docker container, there are usually some specific commands that need to be run within that instance of the docker container using the `docker exec` command.

4.3.1.2 `docker exec`

The `docker exec` command is used to execute a command within an already running docker container. After starting a container, there are commands that need to be executed which accomplish creating user profiles, setting user permissions, creating symbolic links for the LiveOS file mount points, setting environment variables for that specific instance of the docker container only, and finally switching the default user account of the operating system within the docker to be the newly created user. After these commands have been run, the docker is ready to be exposed to the user and a terminal window is spawned, allowing a front end connection into the running docker container.

4.3.2 Mounting LiveOS File Directories Within Docker

Docker containers within the LiveOS are helpful largely in part of their ability to mount external file directories. This means that as we launch different types of docker containers, we can mount LiveOS files and directories as either read only or read write based on the nature of the container. As explained in Section 5.3.2, the terminals within the LiveOS require read only access to the LiveOS project files. As such, the docker terminals that are exposed to the user within the LiveOS have restricted access to collaborative files for the sake of organization and security. Users cannot edit LiveOS files within their dockerized terminal, and this

limits the amount of changes that users can make on the LiveOS server itself. With the docker terminal containing the necessary project files as read only, users can collaborate on files, editing them by using an application other than the terminal or copying the file using the `_git` commands (see Section 5.4.1.3) so that a read write version of the file is available to the local docker file system.

4.3.3 Handling Communication Into the Docker Containers

Docker containers inherently lack open communication to other dockers and the back end LiveOS server. This is to maintain as much separation as possible to minimize disruptions to the server that is hosting the LiveOS. To be able to communicate inside of a docker container while also keeping the user of a terminal separated from the back end files and functionality, we have utilized a library developed by another member of the MASC group. This secure TCP connection library allows a connection to be opened from within the docker container to a specific port on the LiveOS server. With this connection, we can accomplish communication into and out of the docker container by way of creating our own terminal utility. This utility, launched as an executable, allows communication of specific files and revision control messages to the LiveOS server for handling. This utility is available to any LiveOS user, allowing them to move files and control content generated in the docker container without giving them complete access to back end LiveOS server functions. See Section 7.3.4 for an example of this communication.

4.3.4 LiveOS Docker Containers

The LiveOS leverages a few different styles of docker containers to accomplish reliability and security. Below are explanations of each type of docker currently used within the LiveOS.

4.3.4.1 Private Docker

The private docker is a docker container that handles individual user code development, experimentation or compilation. Each user of the LiveOS is given their own private docker to work within. Being private, no other users have access to the files mounted within or generated within a private docker. The private docker is currently only accessible through a terminal front end (more information in Section 5.3.2.2). When launching a terminal from the LiveOS menu, the user has the option to open their private docker. Once inside, they are working within a full Arch Linux distribution with a layered file system. Any files that they generate will stay private and will persist across closing the terminal, restarting the LiveOS or refreshing the web page. If a user's private docker becomes unusable for any reason, they can restart their private docker and instantly have a new, untouched Arch Linux operating system container to use.

At the time of launch, there is one user account created within the docker that matches the name of the LiveOS user. This user account has super user access to reign within their private docker.

4.3.4.2 Shared Project Docker

The shared project dockers are launched per project within the LiveOS. As projects are created, new dockers are created as well. This gives the ability for users to work collaboratively within a shared terminal, which runs inside of the shared docker. Currently, a shared terminal is the only way to access the shared project docker. The shared docker terminals are accessible to any LiveOS user with access to that specific project. Users can work within the shared project docker terminal and collaboratively see other user files within the same project docker. The idea is to work with other users on common sets of files within a project.

At the time of launch, there is one *project* user that is created within the shared project docker. When users connect to the shared project docker terminal, they are working as this project user. Being collaborative, all users of a shared project docker terminal have super user access.

4.3.4.3 RW_git_docker

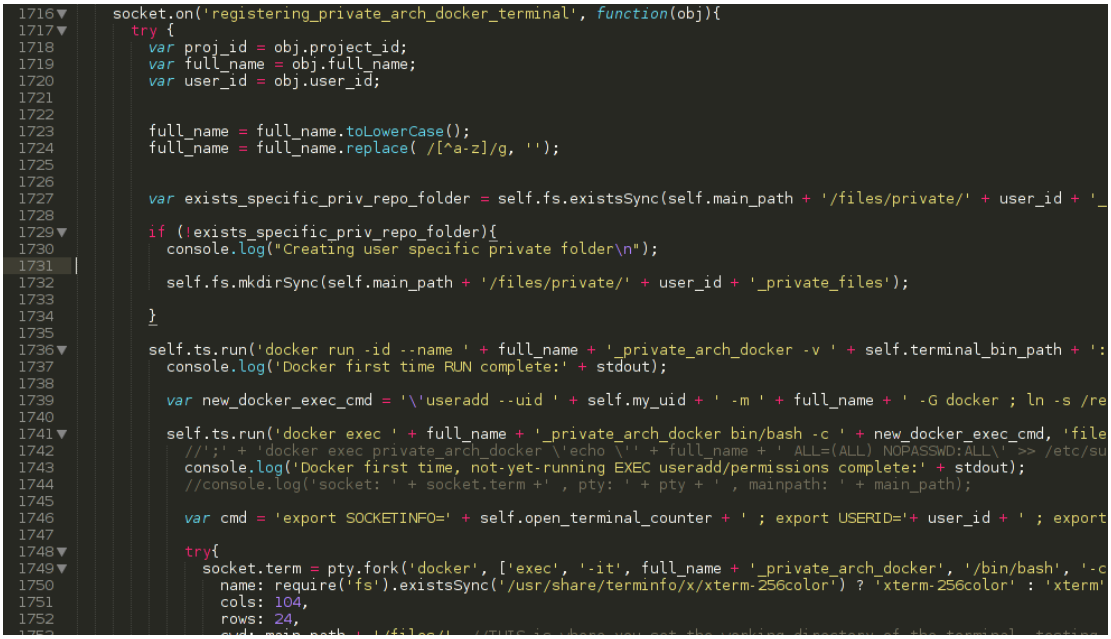
The `RW_git_docker` is spawned concurrently with the LiveOS and serves as the private and shared dockers middleman for LiveOS file system operations. This means that while the private and shared dockers mount LiveOS file directories as read only, the `RW_git_docker` mounts them as read write, allowing it to carry out operations that require writing to the LiveOS file system.

The `RW_git_docker` has write access to the project directories within the LiveOS. As such, when doing file write operations, the user account created within the `RW_git_docker` must write files to the LiveOS file directories with the same permissions as the user account who

has launched the LiveOS itself. In order to avoid said file permission issues when writing new files into the LiveOS file directories, we assign the user account within the `RW_git_docker` to have the same UID (user identification) as the user who launched the LiveOS. This mitigates any permission issues when writing or reading files that have been obtained through this docker. Because this docker has read write access to the LiveOS file system, no user can directly touch or manipulate the `RW_git_docker`.

4.4 Sample Use

See below for some examples of how the LiveOS is using Docker to launch the terminal application.



```
1716  socket.on('registering_private_arch_docker_terminal', function(obj){
1717    try {
1718      var proj_id = obj.project_id;
1719      var full_name = obj.full_name;
1720      var user_id = obj.user_id;
1721
1722
1723      full_name = full_name.toLowerCase();
1724      full_name = full_name.replace( /[^a-z]/g, '');
1725
1726
1727      var exists_specific_priv_repo_folder = self.fs.existsSync(self.main_path + '/files/private/' + user_id + '_
1728
1729      if (!exists_specific_priv_repo_folder){
1730        console.log("Creating user specific private folder\n");
1731
1732        self.fs.mkdirSync(self.main_path + '/files/private/' + user_id + '_private_files');
1733
1734      }
1735
1736      self.ts.run('docker run -id --name ' + full_name + '_private_arch_docker -v ' + self.terminal_bin_path + ':
1737        console.log('Docker first time RUN complete:' + stdout);
1738
1739        var new_docker_exec_cmd = '\'useradd --uid ' + self.my_uid + ' -m ' + full_name + ' -G docker ; ln -s /re
1740
1741        self.ts.run('docker exec ' + full_name + '_private_arch_docker bin/bash -c ' + new_docker_exec_cmd, 'file
1742          //';' + 'docker exec private_arch_docker \'echo \'' + full_name + ' ALL=(ALL) NOPASSWD:ALL\' >> /etc/su
1743          console.log('Docker first time, not-yet-running EXEC useradd/permissions complete:' + stdout);
1744          //console.log('socket: ' + socket.term +' , pty: ' + pty + ' , mainpath: ' + main_path);
1745
1746          var cmd = 'export SOCKETINFO=' + self.open_terminal_counter + ' ; export USERID='+ user_id + ' ; export
1747
1748          try{
1749            socket.term = pty.fork('docker', ['exec', '-it', full_name + '_private_arch_docker', '/bin/bash', '-c
1750              name: require('fs').existsSync('/usr/share/terminfo/x/xterm-256color') ? 'xterm-256color' : 'xterm'
1751              cols: 104,
1752              rows: 24,
```

Figure 4.1: Source code example of launching the terminal application within a docker

In Figure 4.1, source code lines `1716` through `1735` prepare string commands that

will be used to initialize and customize the intended application to be launched within the docker container. Source code on lines `1736`, `1741` and `1749` use the prepared string commands within a `docker run` and two `docker exec` commands respectively.

The code on line `1736` spawns the docker container using a `docker run` command. Because this docker is being launched for a single user only, the docker is given a name to match the user's name. Other initialization commands happen within the `docker run` command such as specifying file mount points, access permissions to those file mounts and finally, among others, specifying the operating system image for the docker to have. Line `1741` executes a few housekeeping commands *within* the already running docker that was launched using line `1736`. The commands executed within the running docker consist of changing the new Linux user's UID to match that of the creator of the LiveOS directory on the server. This allows users created within docker environments to have the proper access to files that reside within the LiveOS file system that were mounted into the docker container when it was launched. The code on line `1741` also creates symbolic links from the mount point of the LiveOS files into the docker to the new users home directory within the docker. This makes it so that all files that should be mounted to the docker upon launch are easily accessed in the users home folder.

Line `1749` is the last step of the process and handles `fork`ing a new TTY process on the LiveOS server and passing that process into the already running docker container using the `docker exec` command.

Chapter 5

LiveOS Application Built: Terminal

> Give me six hours to chop down a tree
> and I will spend the first four sharpening
> the axe.
>
> Abraham Lincoln

5.1 Introduction: A Terminal in the LiveOS

The nature of the LiveOS warrants application and back end control at a granular level, allowing users to control processes, see debug output and modify applications as they are running. Live simulation and collaborative programming, in addition to live Latex compilation and the detailed infrastructure of the LiveOS back end in general lend way to a familiar method to make sense of a complicated work flow. This normally means having the ability to monitor LiveOS applications as well as edit their runtime behavior by way of using a terminal. Because users will not necessarily have direct back end access into the Arch Linux server to complete

application runtime modifications, we implemented a terminal to use within the web browser.

The goal of the terminal application is to give the user native, familiar access to well known control functions within the LiveOS. After all, the LiveOS is built on a Linux back end, therefore control over the environment is almost identical to understanding control of a standard Linux operating system.

Having a terminal application within the LiveOS also presents interesting security and control concerns, which inspired multiple implementations of the LiveOS terminal, each implementation being governed by who the user is and what they are intending to do. In this section, we will describe the terminal application in terms of the front end client code and the back end server code.

5.2 Why a Terminal Is Needed

The terminal application on traditional Linux operating systems allows the user full control of files, signals, systems, users, etc. and is a necessary utility for an efficient Linux work flow. To bring more functionality and usability to the LiveOS, we implemented a terminal application that uses a Javascript library called `pty.js`, and emulates a full Linux terminal. This gives the user full control of the LiveOS. For example, a terminal within the LiveOS can touch any of the running processes on the server as well as spawn any new processes the user might want to create. A typical design flow when researching architectural design by way of simulation includes specifying a configuration, simulating and viewing output. When that flow is abstracted away from the user in the form of a web-based simulation tool, the user loses the

ability to examine problems in compilation or in simulation because the process is running and unattended on the back end server. To solve this problem, hooks into the back end processes are accomplished within the LiveOS via the terminal application.

5.3 Design

The design of the terminal application is logically split into a general front end and back end design, the front end including the client UI view and functions such as capturing keystrokes and parsing commands, and the back end design which handles the keystrokes captured by the front end and greatly varies by the type of terminal that has been launched.

5.3.1 Front End Design

Because a web browser does not know what a terminal is, the LiveOS must create a window with the appearance of a terminal and seamlessly handle back end processing and communication. Upon launching a terminal, the client's web browser displays a terminal window, which is a black rectangular box, including a prompt similar to what they would see on a standard Linux install, along with a blinking cursor. When a user types, the client application code must capture the keystrokes and send them via web sockets to the server. Client Javascript code is able to detect focus within the terminal window that was launched, and upon pushing a key on the keyboard, a Javascript event handler captures which key that was pressed and sends the result to the back end terminal server. See Figure 5.1 for an example of writing text into a terminal window, seeing the characters appear in the terminal window as well as running a

command and seeing its output.

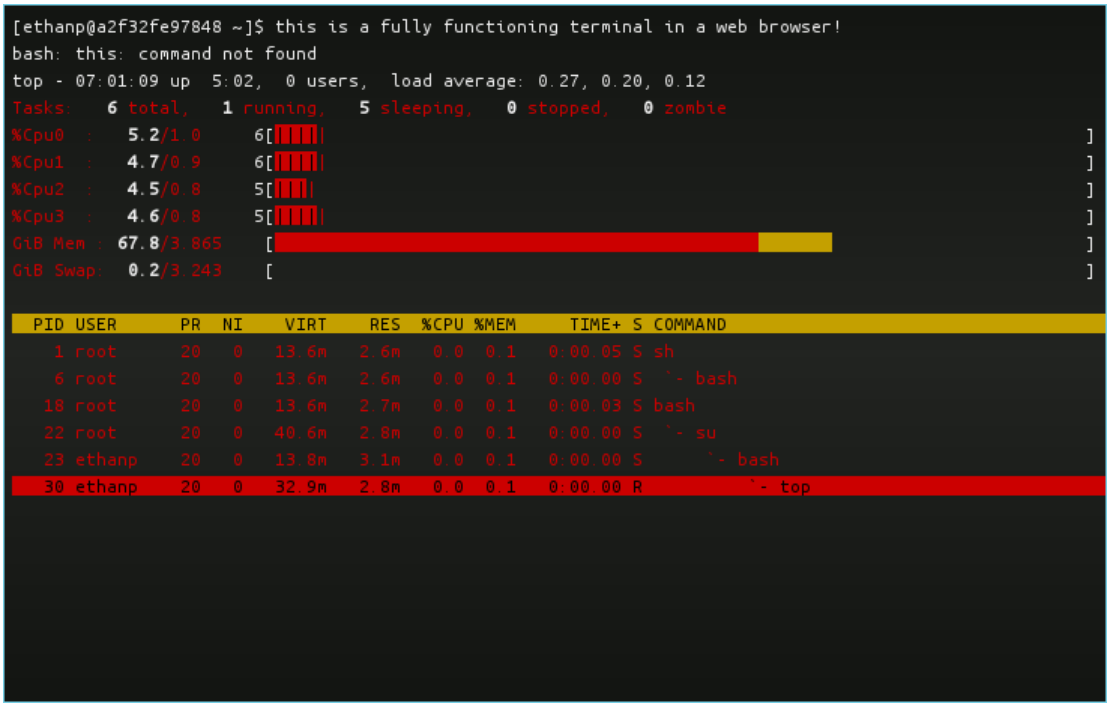


Figure 5.1: LiveOS terminal application UI

In addition, the terminal window menu has functionality built in to handle the GIT integration (more on this in Section 7). In the menu, an option to clone a git repository presents the user with a pop up window which contains the input fields for the type of authentication the user desires to use to clone the repository, the clone URL and an input box for specifying the newly cloned repository branch name. A new branch is required each time a new repository is cloned for the sake of keeping the back end GIT hierarchy organized as well as avoiding accidental disruptions to the master branch of the repository.

5.3.2 Back End Design

In general, when the server receives the keystrokes that are sent from the client front end code, the terminal server is required to handle and process what those characters are, and then supply the client with the correct response. The back end terminal server implements a Javascript library called pty.js. This library receives captured characters typed into the terminal window running on the client machine and runs the data through a teletypewriter (TTY) process and captures the output. Using this library, the back end captures the TTYs output, parses it for command sequences that are important to the terminal server, and sends the result to the client web page view, completing the flow of using the LiveOS terminal.

For example, typing `'ls' + enter` will, under normal circumstances in a Linux terminal, print the contents of the current working directory. In the LiveOS, typing the same sequence would first send a message containing the letter `l` to the server via web sockets. The web socket connection between the client front end and terminal server back end handle all communication between the client and server. The server, with the capability to run a TTY process, feeds the letter `l` into a TTY that it spawns on the Linux server, and receives the Linux server's standard output(in this case, the letter `l` would be returned by the Linux TTY). That output would then be parsed and sent to the client to be displayed. The same process would happen with the letter `s` and finally when the user hits `enter`, the Linux TTY executes the `ls` command (this is because when a TTY sees the `enter` character, it executes the text that has been enteres on the command line). The output of the `ls` command is then sent back to the terminal window within the LiveOS. It is this process that allows a user of the LiveOS to have

a similar amount of control as a standard Linux install.

When running commands that involve a large amount of output, for instance the `find / mycache.cpp` command, the web socket can potentially saturate because of the large volume of output being sent from the terminal server to the client web page terminal view. To solve this problem, we implemented server side data flow control which can dynamically assess the socket load and drop messages which would otherwise stall the terminal application completely.

After implementing the terminal as described above, we expanded the terminal application to also run within a docker container. Currently, there are two types of terminals implemented in the LiveOS.

- Terminals running on the server directly (Administrative terminal).
- Terminals running within a docker container (Private and Shared docker terminals).

5.3.2.1 Administrative Terminal Design

Because the administrative terminal is not launched through a docker container, the terminal back end can simply launch a TTY process and display the typical standard output to the user within a window on the LiveOS web page. With direct hooks into the Arch Linux back end server, it is extremely effective in diagnosing LiveOS problems and controlling internal LiveOS functionality; the ability to gain superuser access is also available. For these reasons, the administrative terminal within the LiveOS is only intended for use by a LiveOS administrator.

5.3.2.2 Private Docker Terminal Design

The private docker terminal is a terminal that is launched within a docker container and presented to the user as a self contained ecosystem, housing the user's private files. This means all of the files the the user creates and edits within the private docker terminal are only visible and editable by them.

When the LiveOS is first launched, a private docker container, without the terminal GUI, is launched in the background per user of the LiveOS. The docker container is assigned a name based on the user is it created for and the terminal prompt is set to match the user's name; this helps the user know that they are working within their private docker container. In addition, a unique user account is generated within the docker container, and finally, the user's private file mount point is given to the container. When the user requests to open their private docker terminal application, a terminal window is spawned in the user's web page. At this point, the private docker terminal is fully controllable and the user's files and actions are contained and isolated from the back end server as well as all other users. The files generated in a LiveOS file mount point within the private docker terminal are persistent and they can be accessed after exiting the private docker terminal (or a total server outage). The files that are generated within the users local docker home path however are volatile and will be lost only if the LiveOS server is completely powered off. This sort of temporary file storage is useful for building large projects, primarily because the user has control of if and when these local docker files are deleted. In the event that a user no longer needs their local docker files or wishes to reset their specific docker configuration, they can kill and restart their docker. After the restart,

the container will only contain persistent files within the mount point in the docker, but not any previously generated or edited local docker files.

The private docker terminal serves as an isolated playground for the user to clone repositories, design, edit and compile or work on their own set of files.

5.3.2.3 Shared Docker Terminal Design

For collaborative tasks, the shared docker terminal is available. This terminal, also running within a docker container, focuses on the ability to collaborate on a single set of files with other users. This terminal is also completely isolated from the LiveOS server itself.

At the time of the LiveOS launch, a shared docker is launched per project in the background (a project may contain many GIT repositories or other files). As users join the shared docker, they are each spawned a separate view of a terminal window, all pointing at the same set of mounted files. This ensures all changes that project users make are reflected back to one central mount point, allowing all users working with a specific shared docker to create, view or edit changes in real time. As local docker files are created and added, each user of the shared docker terminal can see and use the files.

The shared docker terminal also solves an interesting problem: GIT permissions across multiple users. When a shared terminal user clones a GIT repository, to stay completely collaborative, all repository features become available to any other user of that same shared docker terminal, including synchronizing changes that any user has made upstream to the remote repository. This is accomplished by storing the credentials of the user who *first* clones a given repository within the shared docker terminal. Any subsequent user who then works

with that repository and requires remote authentication to publish their changes will do so using the credentials that were stored when the repository was cloned. This method of handling the security of a single shared repository clone abstracts the need for individual authentication and allows any user with project access to contribute to a repository. It should be mentioned that as a result of handling authentication in this way, tracking specific user commits becomes more difficult, as the commit will always appear as originating from the shared repository owner. To combat this issue, individual user commits to any shared GIT repository are tracked by an automatic message inserted into the commit message. This message specifies which LiveOS user has committed the change.

5.4 Terminal Accessories

5.4.1 Custom Control Commands

Because the LiveOS file permissions require any file write commands to function differently than they would in a standard Linux terminal, we created a custom terminal binary which is invoked via the command line. Allowing users within a private or shared docker terminal full control over writing files could potentially be dangerous, so we implemented a binary that wraps powerful background functionality into a simple set of static and non-configurable user commands.

The commands which invoke the binary are: `_kill`, `_git`, `_edit`. These commands accomplish helpful tasks within the terminal such as listing and killing the available dockers (private or shared), initiating GIT operations with local docker files, and bringing files

that are local to the docker container into the LiveOS for saving or editing, respectively.

5.4.1.1 `_kill` command

When using the `_kill` command, the user of an administrative terminal has the ability to shutdown and restart any docker, private or shared. This is helpful because if the state of a docker becomes unusable or local docker files are causing problems, an administrator can fix the problem by restarting the docker. Because of the nature of Docker, any docker that is restarted will lose its locally generated files.

When using the `_kill` command within a private or shared docker, users have the ability to restart their own docker (private) or the shared docker they are working in (shared). They do not have access to other user's private or shared dockers.

5.4.1.2 `_edit` command

The `_edit` command implements docker to LiveOS file sharing functionality. When in a private or shared docker terminal, a user has the ability to create local files within the docker. If a user desires some local file to exist outside of the docker container, they can invoke the `_edit` command, which will copy the local docker file into the LiveOS file system. Once the file is copied, it will persist and be treated as a normal LiveOS project file. Users can then open the file in the collaborative code editor and see all changes.

5.4.1.3 _git command

The _git command brings specific GIT functions, which would otherwise require file write access, to the private and shared docker terminals. When in either of these terminals, the _git command exposes the following functions:

- _git branch
- _git branch <branch name>
- _git branch -d <branch name>
- _git checkout -b <new name>
- _git checkout <branch name>
- _git sync

_git branch Any private or shared docker terminal user can use the _git branch command to see the available branches/which branch of the repository they are currently working on. They can also use the _git branch -d command to delete any local branch they have created. Finally, any user can use the _git branch <branch name> command to create a new local branch of the repository to work in.

_git checkout Any private or shared docker terminal user can use the _git checkout command to either checkout an existing branch to begin working or create a new branch and check it out to being working.

_git sync The sync command allows any user of a private or shared docker terminal the ability to (in order): commit their local changes made to the repository, pull the most recent changes from the remote origin, and push their changes back to the origin. This command is a powerful way to integrate git functionality within private and shared dockers without allowing users full control of file writing operations. See Section 7.3.4 for a more in-depth explanation.

5.5 Sample Use

5.5.1 Administrative Terminal

The administrative terminal is intended for use by administrators of the LiveOS. Any user who spawns an administrative terminal has the ability to destroy files of any LiveOS user in addition to core LiveOS source code files. For this reason, the administrative terminal can only be launched with a specific set of secret commands.

When in the LiveOS, press the keys `left shift + left control + s` together to bring up the LiveOS secret prompt. At the prompt, type `term` and hit enter. This set of commands launches the administrative terminal. From this point, the administrative terminal can be used to: kill rogue processes, install LiveOS dependencies, develop LiveOS code, manipulate other user files etc. The administrator has full control over the LiveOS.

5.5.2 Private Docker Terminal

The private docker terminal is a terminal users can work inside to keep all of the generated output files, configuration files, even complete repositories private. It is ideal for pro-

totyping code and features that are not accessible to any other LiveOS user. As an example, to launch a private terminal, clone a new repository and begin working in that repository privately:

1. Navigate to LiveOS menu and select `Apps` → `Terminal` → `Private Terminal`
2. Navigate to the menu within the private docker terminal and select `Clone Repo`
3. Select authentication method, enter the URL to clone and a new branch name
4. Change directory into the `PrivateFiles` folder within the private docker terminal
5. Begin work in the terminal, using the supported GIT 7 commands when desired

A simple alternative to launching a private docker terminal through the menu would be to use the keys `alt + t`.

5.5.3 Shared Docker Terminal

The shared docker terminal is most useful when the user desires to view and modify files that other users also have access to. The supported GIT commands (see Section 7) also work for shared docker terminals; the authentication and individual user commits are handled seamlessly on the back end. The process to launch and use the shared docker terminal is similar to the private docker terminal:

1. Navigate to LiveOS menu and select `Apps` → `Terminal` → `Shared Terminal`
2. Navigate to the menu within the private docker terminal and select `Clone Repo`
3. Select authentication method, enter the URL to clone and a new branch name

4. Change directory into the `SharedFiles` folder within the shared docker terminal

5. Begin work in the terminal, using the supported GIT 7 commands when desired

Chapter 6

LiveOS Application Built: Code Search

> The mind that opens to a new idea never returns to its original size.
>
> Albert Einstein

6.1 Introduction: All Of Your Code, Immediately Accessible

Working within the LiveOS, a user with many thousands of files faces the challenge of quickly seeking individual files, even finding specific keywords within their many thousands of lines of code or file content. Typical operating systems solve this problem with a global search utility. This search utility ideally has the ability to search through all of the user's files in a speedy fashion at anytime throughout a normal work flow. The LiveOS is no exception, potentially containing many thousands of files for one project. For this reason, we implemented a standalone application that is quickly accessible via keyboard shortcuts or menu selections and has the ability to immediately navigate the user to a specific line of code they are looking

for in just a few seconds, ultimately increasing the efficiency of LiveOS file management and file access.

6.2 Why Code Search Is Needed

The LiveOS potentially contains many cloned GIT repositories, each with a large number of working files of which the user might be interested in accessing or editing quickly. The first method of accessing these files would be to use the LiveOS's file manager. If the user knows where to find the files they are looking for, this utility works great. However, when the user knows the filename, code snippet, code comment or any other file name or file content specifier, but not necessarily the location or the name of the file or content itself, a utility such as code search is extremely helpful.

Consisting of an easy to use front end user interface with hooks directly into the LiveOS's collaborative code editor, codemirror, and an intelligent back end server, the code search application extends the efficiency and usability of the LiveOS. The code search application's ability to search many thousands of files quickly is also a notable feature of the application. Code search utilizes an indexing program called `csearch`, and we combined it with `the_silver_searcher` or otherwise called `Ag` (short for Ack-grep). These two utilities, paired with intelligent logic on the back end, allow code search to supply the user with instant results for any query of the LiveOS project file hierarchy.

6.3 Design

The code search application is split into a front client application and a back end code search server. The general flow is as follows below, in Figure 6.1.

Figure 6.1: Code search overall design flow

When the user launches the application, a plain black rectangular application window is launched, containing one text input area and some hints for how to use the search. Upon typing into the text area, and following a static timeout period after the user stops typing search characters, the characters are sent to the code search server through web sockets. The server catches the set of characters, uses the `csearch` and `Ag` utility to run a search of the characters typed against a preexisting index file which the server has generated based on the users project files, and sends the file results to the client application window for viewing. At this point, the client application parses the returned results, intelligently separates the results based on file name, line number, and the content that was returned and shows the results to the client. In addition, the returned

results are selectable for quick editing in the collaborative code editor.

6.3.1 Front End Design

The front end application is responsible for capturing initial user input as well as parsing query results sent from the code search server and making those results interactive, such that a mouse click or keyboard interaction within the application will open a resultant file for collaborative editing. In addition, the front end has the ability to trigger a LiveOS file system re-index.

Typing a search term or advanced search query will return results into the main code search application window. The results will be arranged such that the filename is printed first, followed by the line number on which the query match was found, followed by the content of the matching query line, with the matched portion of the content highlighted, as well as two lines of contextual content above and below the matched query result. At this point, the results are arranged in the code search application window based on which file has been most recently edited, and if no files have been recently edited within five minutes, the files are arranged in ascending alphabetical order.

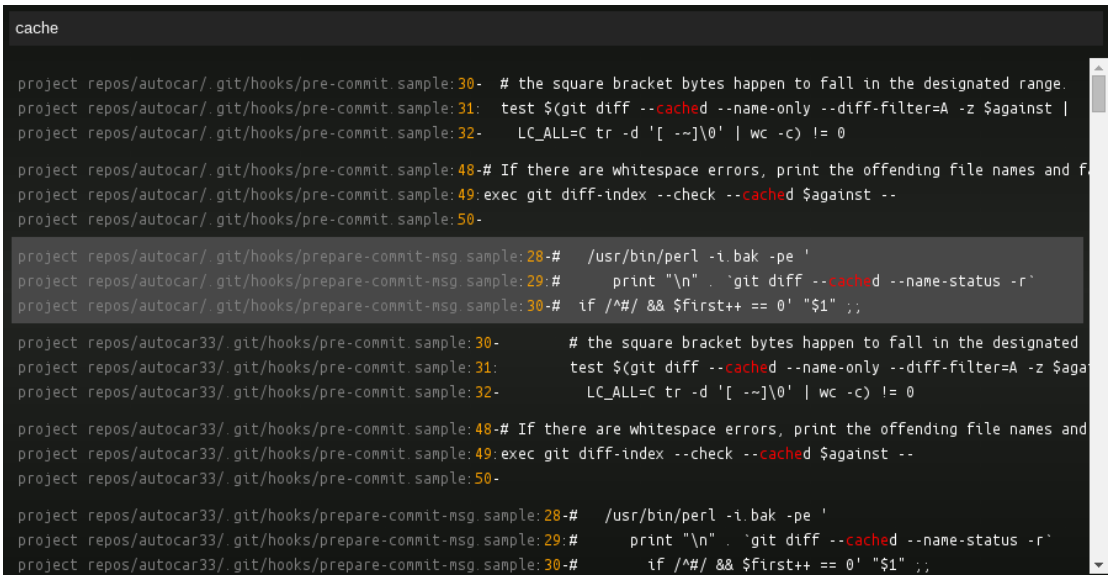


Figure 6.2: Code search front end client view with results

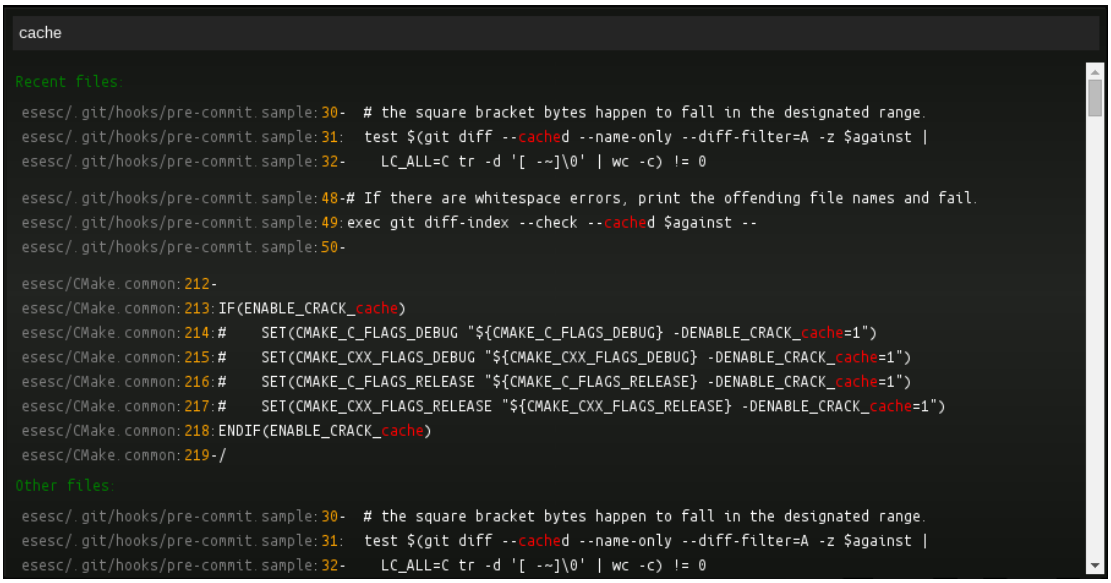


Figure 6.3: Code search front end client view with recently edited results

Clicking on, or using the arrow selection keys on the keyboard, will focus the results area of the code search application window, as seen in Figure 6.2. The user can scroll the results using the `up` and `down` arrow keys, or use the mouse to scroll and select results. Hitting the

`enter` key on a selected result will close the code search window and open the selected file, at the proper line number, in the collaborative code editing application. Alternate options to open selected results include double clicking the result or hitting the `shift + enter` keys. These two alternate methods will keep the code search window open in addition to opening the collaborative code editor for the selected result. In Figure 6.3, two recently edited files are given priority over other search results. It should be mentioned that if the user closes the code search application, upon reopening, their previous query and results will automatically populate the code search window.

6.3.1.1 Use Modes

To begin a search in the LiveOS using code search, launch the application and type into the auto-focused, text-enterable search area. The front end code search application displays searching hints before typing into the search box. There are a few search modes supported by the code search application.

1. Search string with no spaces, e.g. `cache` or `uint32_var`
2. Search string with spaces, e.g. `'L1 cache model'` or `'uint_32_var = 0xDEADBEEF'`, NOTE: the single quotes are required
3. Search string with spaces or no spaces with a *fuzzy* file regular expression specifier, separated by a space, e.g. `uint32_var .cpp` or `'uint32_var = 0xDEADBEEF' procesr.cp`
4. Search `--help` with string specifier, separated by space, e.g. `--help mongo`

In Example 1, searching for either `cache` or `uint32_var` will return results into the code search window with either the file name or file content containing or exactly matching the query; the same applies for Example 2. In Example 3, the search is augmented with a utility called `FZF`, or otherwise known as `fuzzy file search`, which allows the user to search with a regular expression as a filter for the returned results. When searching with a fuzzy file regular expression, the user has the ability to get extremely specific with the type of results that are returned. Looking at Example 3, searching for `uint32_var .cpp` will filter the standard `csearch` and `Ag` results by using the `FZF` utility and only showing results from files that include a `.cpp` in their name. In addition, the fuzzy part of the regular expression search allows the user to misspell, or completely forget characters of a regular expression specifier, and still get relevant results. For example, searching for `'uint32_var = 0xDEADBEEF' procesr.cp` will only return results that contain the string within the single quotes, from files that *roughly* match the string literal `'procesr.cp'`. Most notably, the first results returned would be content that contains the string within the single quotes in the file named `processor.cpp`, with similarly named files following. Finally, in Example 4, specifying `--help` as the first search argument, followed by a space and then the users search query, the code search application will only return results returned from searching the help and README files of the LiveOS. Clicking on any result will open the README file for the LiveOS, allowing the user to view and scroll throughout the file.

6.3.2 Back End Design

Code search heavily relies on the back end logic to properly generate up to date index files and intelligently run the users keystrokes through the `csearch` and `Ag` utilities. Efficiency is key here so as to minimize computation overhead yet still return results to the user in a timely manner.

6.3.2.1 Indexing Logic

The `csearch` utility is packaged with an indexing tool called `cindex`. This indexing utility is run as its own process on the LiveOS server. To run `cindex` as a standalone utility, the user must specify the path(s) of files they wish to create an index for, as well as give the index a name. Upon indexing completion, `csearch` is able to reference the uniquely named index file and provide extremely specific or extremely broad search results, depending on the cindex file referenced. Because users could potentially be adding, editing or removing files at any time, and because the `cindex` utility is computationally expensive, re-indexing using the `cindex` utility has the possibility of becoming a large computational overhead for the server. To combat this, we first implemented a smart indexing scheme that would capture user changes to files, file additions and file deletions.

Using a lightweight feature of the Nodejs framework called `node watch`, we initially launched a recursive watch of all user files in the LiveOS. This allowed the code search back end server to be notified whenever a project file was saved, modified, added or deleted. My original goal was to always have the most up to date file changes and file content available to the code search application. This meant editing, erasing or adding any file within the LiveOS

would trigger a re-index, albeit a lightweight re-index because the delta change is ideally small, and the code search query results would then reflect up to date file changes and additions. With the information available by using a recursive `node watch`, we implemented functionality to re-index files based on the information from `node watch`. Because the delta re-indexes were so lightweight, this implementation worked great for one to two concurrent LiveOS users. With two to three users, each with many large GIT repositories, the re-indexing overhead becomes noticeable and impactful to LiveOS performance.

As such, this implementation was not scalable because of the overhead associated with watching every user file within the LiveOS when the total file count is large. In addition, delta re-indexes per user per file change was not scalable, and also proved to be unnecessary. To combat these two problems, we completely redesigned the auto-indexing logic, making it much more simple. The indexing logic for `cindex` now runs once every 24 hours for each index file. That means that any file that is added, erased or modified will reflect changes in the code search application after 24 hours. This keeps overhead for the code search application exceptionally low and acts as a nightly cleanup re-indexing. If any user makes a change and immediately requires an up to date index file to search against, we implemented a manual re-index option, located within the menu of the code search application.

6.3.2.2 `csearch` and `Ag`

The `csearch` utility is able to search files much faster than other search utilities for the main reason of using a pre-existing index file that is built from the files that are to be searched; in the case of the LiveOS, files within each project directory. This utility is essential

to the speed and usability of the code search application. An index file per project (section 6.3.2.1) that is generated at periodic intervals becomes an invisible file that sits in each LiveOS project directory. The idea is that users can only search and edit files within the current project they are working on. For example, if a user has two different projects within the LiveOS, each with different versions of the same GIT repository clone, they certainly do not want to be referencing a `csearch` index file that includes files (and potentially editing those files) outside of their current project directory scope.

Once the index file is generated and stored in its proper location, the `csearch` utility is free to reference it. When the user has entered their search query, the message is sent via web sockets to the back end, where the code search server first parses the query. The query possibilities include those mentioned in Section 6.3.1.1. Regardless of the search query, `csearch` and `Ag` will process the user input for every query, with the exception being the `FZF` utility, which is only applied to the search query if the user specifies a regular expression along with the search.

A `csearch` command will run as a process on the back end code search server and will produce standard output in the form of alphabetized file names and line numbers next to those file names, denoting what line of the listed file the result was found. `Csearch` is able to include the content found at the specific line number within the filename results, however controlling the style in which the content is returned via `csearch` is difficult.

At this point, the `Ag` utility takes control. We pass the filenames into the `Ag` utility and, running as a separate process, `Ag` returns the filename, the line number, and the content at the line number, along with 2 lines of context above and below the result. Because `Ag` is run on

a very small subset of files, less than 40 files at a time, it adds very little overhead to the total search time, with the trade off being more configurable results.

The filenames, line numbers, and content are each packaged into their own Javascript objects and sent back to the client. The results are split this way to allow easier manipulation in terms of color coding the output and organization in the client application window.

6.3.2.3 Integration with OT

In order to show the user prioritized results based on the most recently edited files, at the time of launch, the code search application server opens a connection with the LiveOS file manager. The reason is as follows: the file manager is notified every time a file within the LiveOS is changed based on the OT integration. OT handles multiple concurrent users editing a file, and the file manager handles saving those changes as they happen. When the file manager triggers a save of a file, a message is sent to the code search server containing the file name and edit time. The code search server stores these files and times in a queue, and keeps them there for a minimum of five minutes. Each time the same file is edited, the five minute counter is reset for that file. The reason for this is so that if any user initiates a search that contains results from any recently edited files (within five minutes), it is most likely the case that the user is again wishing to access the recently edited file, so any results from recently edited files are prioritized within the code search client application window by being displayed before all other query results.

6.4 Sample Use

A typical code search use case begins by launching the code search application at any time throughout programming. Upon launch, the cursor is automatically focused to the search box, and the user can begin typing a search string. Once the code search server completes the search, the user can examine the results, and navigate either via keyboard arrow keys or using the mouse. Pressing the `enter` key or double clicking with the mouse will open the selected result for collaborative editing. See Figure 6.4 for an example.

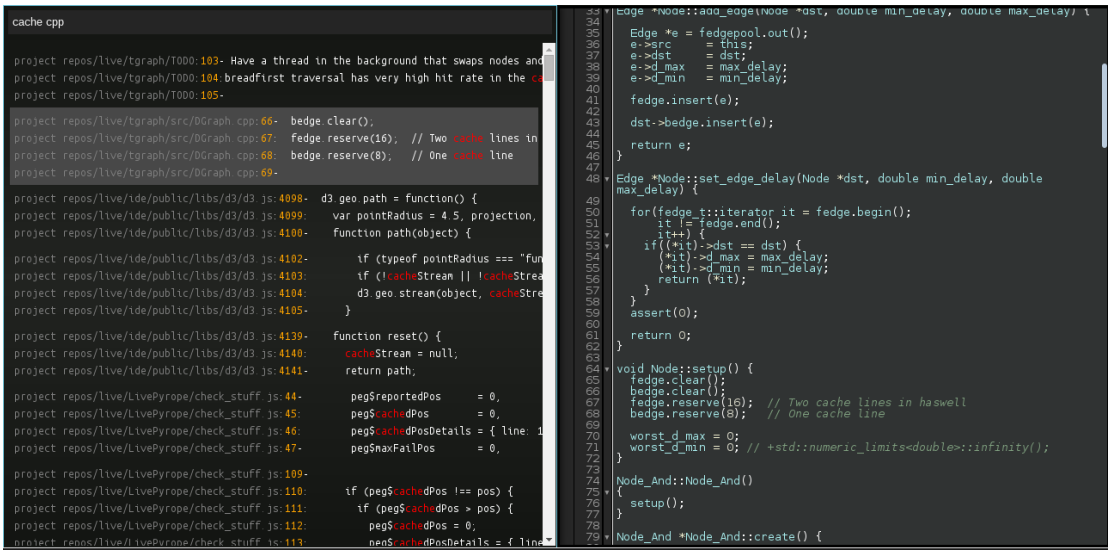


Figure 6.4: Code search results with collaborative `codemirror` code editing application

Chapter 7

LiveOS Module Developed: GIT Integration

If a machine is expected to be infallible,

it cannot also be intelligent.

Alan Turing

7.1 Introduction: Repositories In The LiveOS

Currently, many of the MASC research lab's files reside on remote GIT repositories. For the LiveOS to be an efficient and truly collaborative environment, it makes the most sense to have a seamless back end connection with remote GIT repositories. The nature of the LiveOS promotes collaboration and ease of use, so for this reason, we implemented back end functionality that abstracts the typical GIT work flow away from the user and built it into the LiveOS environment.

7.2 Why GIT Integration Is Needed

The typical work flow experienced by a user outside the LiveOS involves programming or otherwise generating content within files that the user, sometime later, will create revisions of and store in a remote location. Although the work flow becomes highly collaborative and efficient when working in the LiveOS, many users can be working on local files at any time, and these local files will eventually require the same cumbersome revision control as a user working outside of the LiveOS.

In addition, at scale, the LiveOS can potentially have an unlimited number of users and files. With the notion of projects within the LiveOS, individual files and complete repositories can potentially belong to many thousands of users, or just a few. This project based model, combined with collaborative editing breeds the need for a more autonomous file revision control mechanism.

To combat this issue, we brought the functionality of GIT revision control to the back end of the LiveOS and abstracted the GIT use cases that we expose to the LiveOS user. This allows the increased volume of changes per file due to collaboration to be captured in a central and remote location seamlessly.

7.3 Design

GIT integration was an important goal for the LiveOS and as such was integrated with close attention paid to seamless, fluid control. Separate of the LiveOS, acquiring new files to work with from GIT is typically handled by the user in the form of a `git clone` command.

Requiring write access to the local working directory, the `git clone` command writes files to the current working directory and the user can begin using the files.

Within the LiveOS however, the file directory structure complicates this model. In addition, users of the LiveOS should not necessarily have complete access to all of the GIT commands for the reason of maintaining file organization across many collaborators. Currently, using GIT within the LiveOS is handled individually based on the type of terminal (see Section 5) the user is working in.

7.3.1 Front End: Storing Authentication Credentials

Before using all of the GIT features within the LiveOS, the user must store their credentials in the LiveOS database. This allows easy, quick control using either the GIT clone menu or the `_git` commands within a terminal.

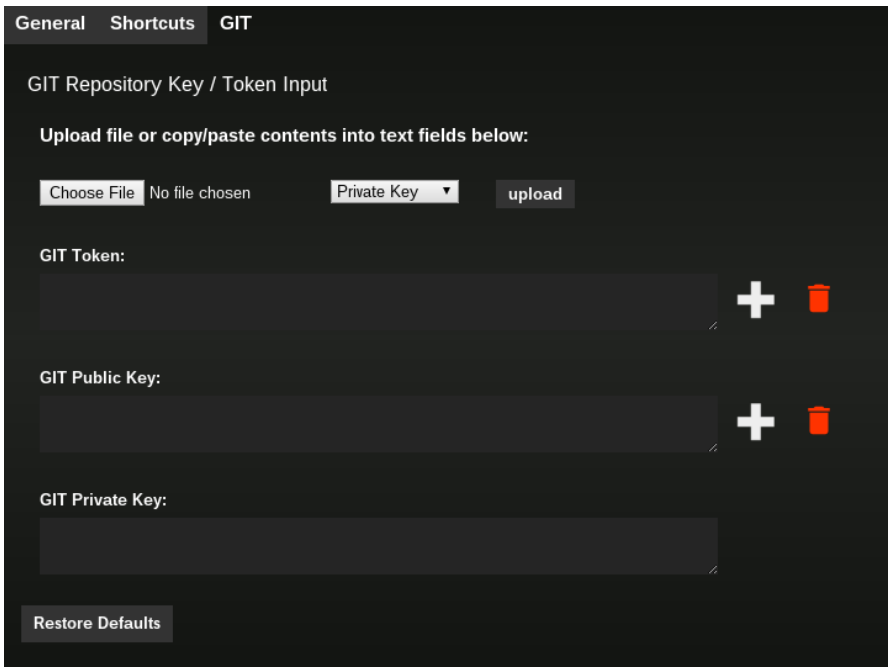


Figure 7.1: LiveOS settings menu, GIT tab

A user can upload either their GIT issued OAuth2 token or if they plan to use public/private key authentication, they can upload their public and private key pair. Just below the upload option in Figure 7.1, there are three text boxes, allowing the user to copy and paste the contents of either of the previous three authentication types, similar to `github.com`.

7.3.2 Front End: Acquiring New Files

Users working within any of the three types of terminals are presented with an extremely similar user interface to obtain new GIT repositories.

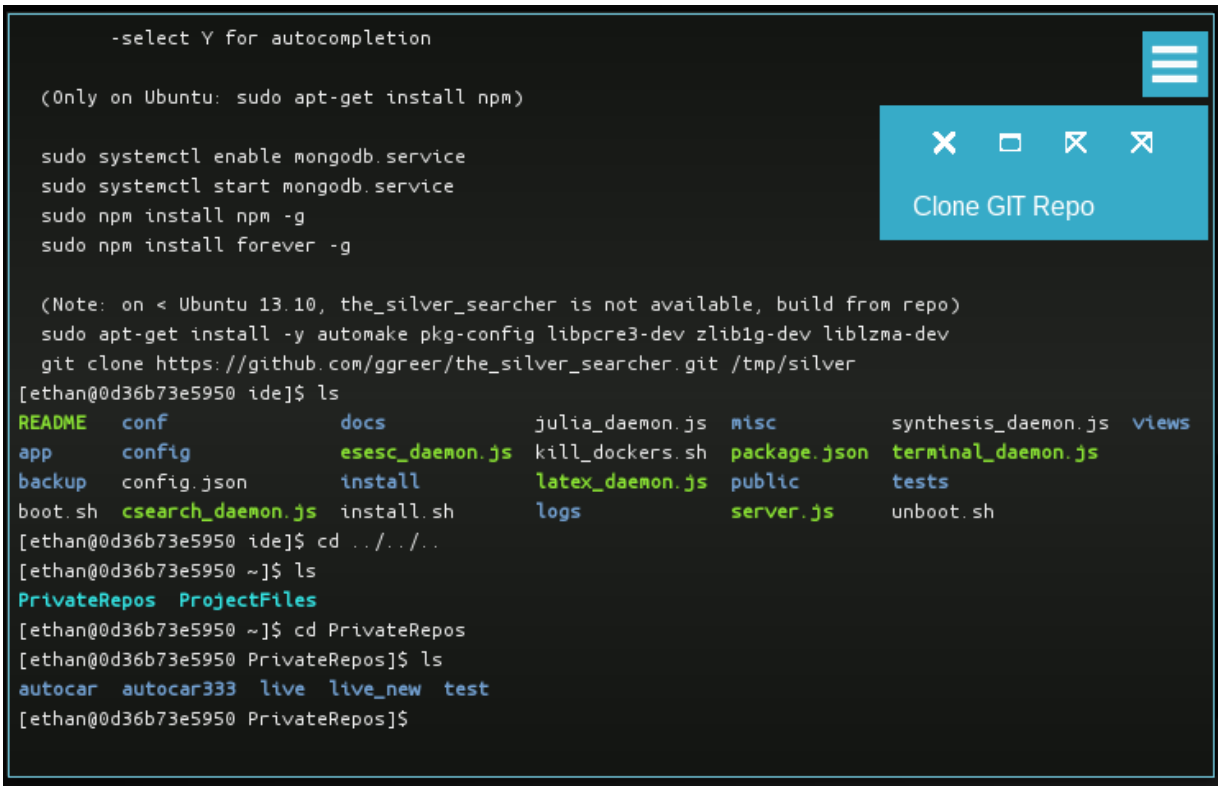


Figure 7.2: LiveOS private docker terminal menu options

Seen in Figure 7.2, clicking on the menu of the terminal will show a `Clone GIT Repo` option. Clicking this option will cause a new pop-up to appear. Shown in Figure 7.3, This pop-

up contains the standard options, in GUI form, that a user would use to clone a GIT repository. The typical flow involves entering the required type of authentication, be it a GIT user name and password, a GIT generated user token, or a SSH public and private key pair. This requires the user to know the security type of the repository they are trying to clone beforehand. In addition, the URL to clone as well as a new branch name are required at the time of a repository clone. These two fields are particularly important when considering multiple clones of the same repository. If the user has already cloned a repository, they will be prompted to give a new name to the folder that the cloned files will be written to. This allows users within the LiveOS to have multiple working copies of a single repository.

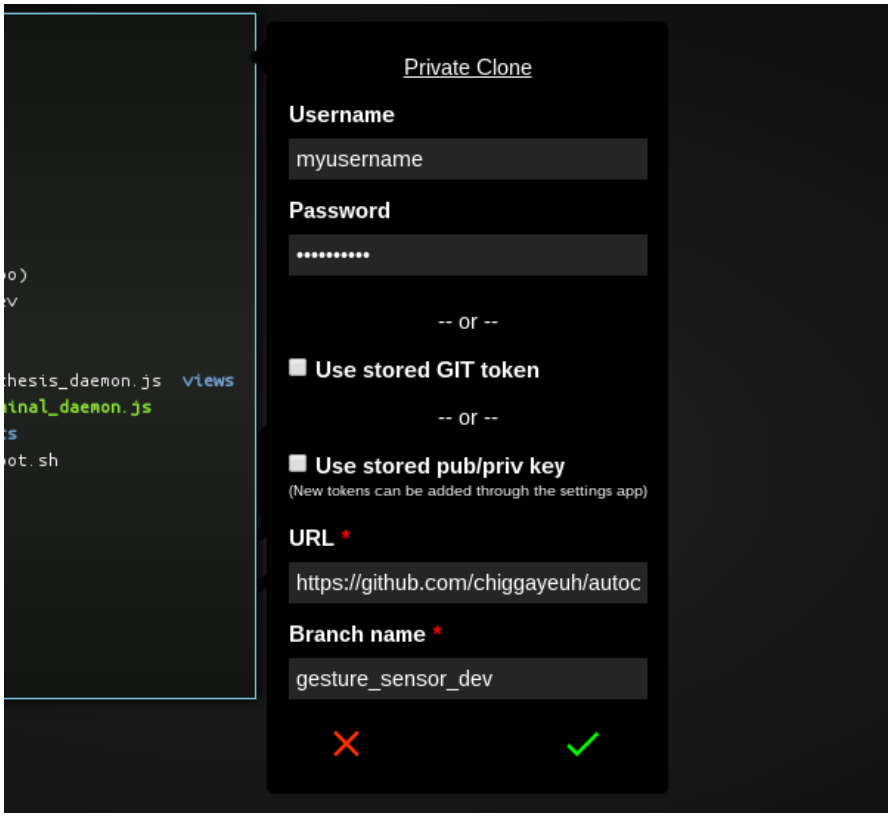
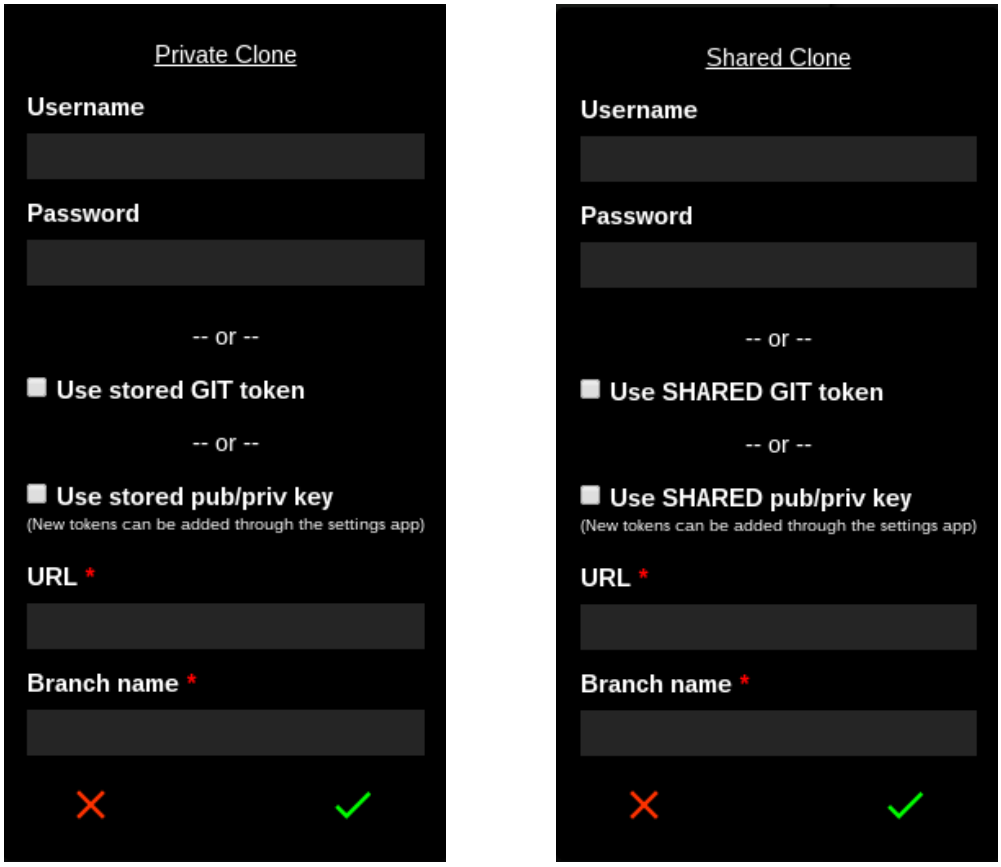


Figure 7.3: LiveOS private docker terminal clone menu

After the required fields are completed, hitting the green check mark will begin the clone, while the red ‘X’ will cancel the operation. The only visual difference to the user on the front end is the verbiage of the GIT token and public/private key check box labels. Shown in Figure 7.4, the differences in labels clarify what type of authentication will be used based on the type of terminal the user is within (more on this in Section 7.3.3).



(a) Private docker GIT clone menu (b) Shared docker GIT clone menu

Figure 7.4: GIT private and shared project docker differences

The major differences and heavy lifting of the repository cloning and other GIT file operations happen on the back end.

7.3.3 Back End: Acquiring and Using New Files

The back end GIT integration currently runs within the back end terminal server. It is programmed this way for convenience; the terminal is the place where the user initiates a new GIT repository clone, therefore putting the control for GIT integration within the terminal server allows direct communication between the client terminal window and the back end functionality. Different control flows are executed based on the terminal the user is within: administrative, private docker or shared project docker.

7.3.3.1 Administrative Terminal

If a user is working within an administrative terminal, the back end commands that would be executed to clone a GIT repository are extremely similar to cloning a GIT repository outside of the LiveOS. This is because the administrative terminal acts almost identical to a standard Linux terminal, being a representative *emulated* version of a full terminal, running on the LiveOS server. For this reason, the administrative terminal within the LiveOS does not need or have a menu to clone a GIT repository. Instead, to clone a GIT repository, the administrative terminal user can use the standard GIT commands to obtain, modify and contribute to GIT remote repositories.

The commands that are run on the terminal server to make this clone happen are, in order:

1. `cd new_repository_directory` //move to the directory that the user is currently working in

2. `git clone https://username:password@git_server_address.git` //execute the clone

3. `cd git_remote_server_address` //move into the newly cloned folder

4. `git checkout -b my_new_branch_name` //make a new branch to work in

This example shows the flow for cloning a repository from a public GIT repository using the administrative terminal and user name and password authentication. Being that this clone is happening within an administrative terminal, the flow is relatively simple. The server process executing the clone operation has unlimited write access to any directory that the user is working in. As such, the flow to clone a repository within an administrative terminal only involves moving into the correct directory and executing the `clone` and `checkout` commands there.

7.3.3.2 Private Docker Terminal

The process for cloning a GIT repository into a private docker terminal is significantly different. The difference is due to, for security reasons, docker only having read access to the LiveOS file mounts (see Section 4.3.2), as well as docker intentionally having extremely limited connection back to the LiveOS server.

Inside of a private docker terminal, the user sees a `Clone GIT Repo` menu option. Fields allowing the user to specify the type of authentication to use as well as the URL to clone from and a new branch name appear as shown in Figure 7.3. This menu abstracts the standard GIt flow presented to the user and allows the LiveOS to have complete control of the

file operations. As such, there are complications with the operations that have to take place as a result of a user trying to clone a GIT repository into a read only mount point within the private docker. The process of finding the current working directory and cloning files into that directory will not work for two reasons: the current working directory listed within the docker is relative to the docker container's file system and completely separate from the LiveOS's file system, and the container's file system path is relative to the LiveOS file mount point that you feed into the docker, which is defined as read only to the docker.

To mitigate this problem, we designed the `Clone GIT Repo` option within a private docker terminal to use special file directories as well as a separate docker container with proper write permissions. Firstly, because the private docker terminal aims at keeping its contents completely separate from all other users and projects, the working directory of a users private docker terminal always references a private files directory, unique to the users ID. This means that any GIT operations within the private docker terminal reference the user's unique private working directory. Secondly, because the users unique private files folder is within the read only mount point of the private docker terminal, the problem of writing cloned repositories to a read only folder still exists.

Enter the `RW_git_docker`, which stands for the read-write git docker. The `RW_git_docker` is a special docker instance that is spawned at the time of the LiveOS launch, and it is powerful because it mounts the LiveOS's private and shared directories as read-write (see Section 4.3.4.3 for more information). One of the duties of the `RW_git_docker` is to translate GIT operations within the private docker terminal into operations that itself will execute instead. Because both the private docker terminal and the `RW_git_docker` mount

the same file directories (but with different permissions), the private docker terminal can receive the `git clone` request from the user, send the object containing the necessary information including the type of terminal, the user, the current project and the type of authentication to the terminal server, and then pass that information to the `RW_git_docker` to complete. The `RW_git_docker` then writes the files to the users unique private file folder at which point the private docker terminal can access the newly cloned files as read only.

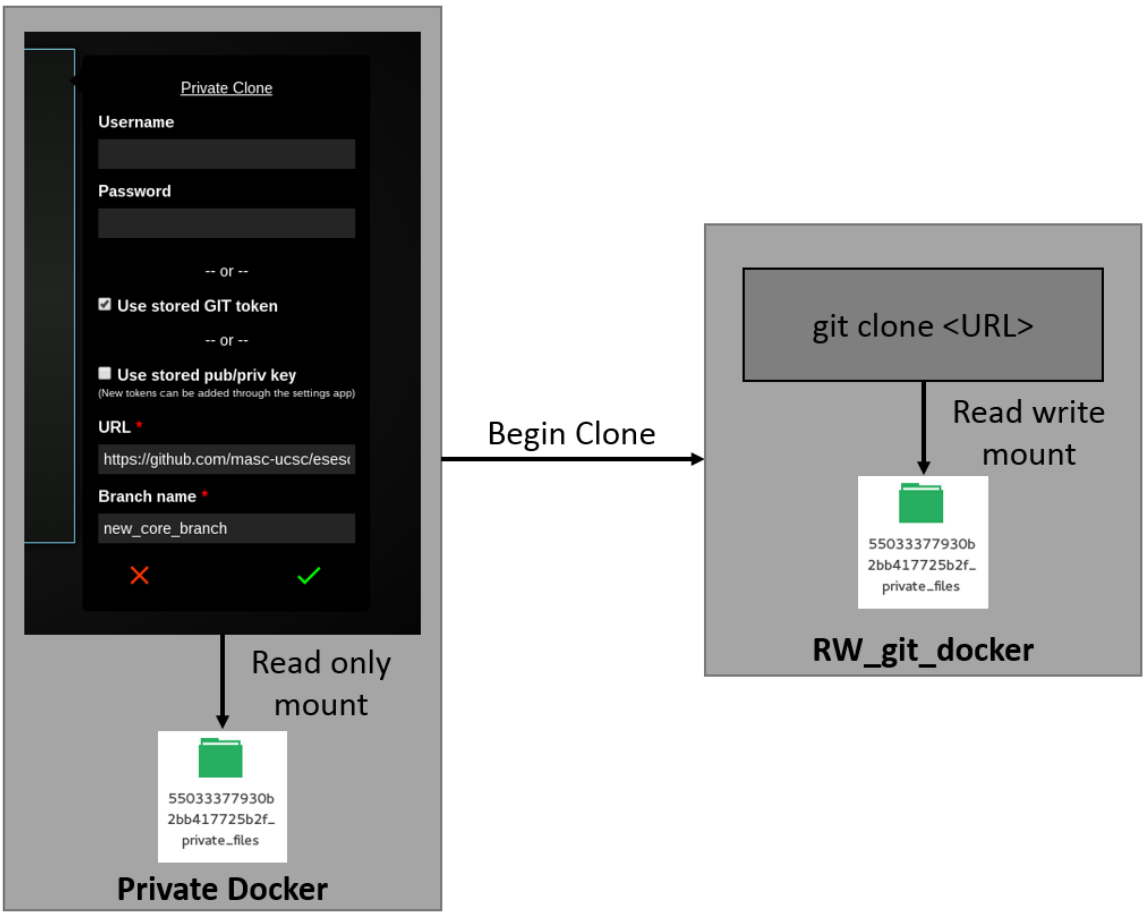


Figure 7.5: LiveOS private docker terminal GIT clone using token authentication

The commands that are run on the terminal server to carry out Figure 7.5's operations are, in order:

1. `terminal_server.exec` //begin the process that will run the following commands in order

2. `docker exec RW_git_docker docker_command &&` //launch the RW_git_docker with the specified command

where the `docker_command` is composed of many sequential commands to carry out the remaining actions to actually clone the repository using the `RW_git_docker` instead of the users private docker. For example:

- `mongodb.find.user_git_token` //locate and create a local copy of the users stored GIT token

- `git clone https://' + '\\\<' + token + '\\\>' + '@' + url + '` //do the clone using the GIT token

- `cd ESESC` //move into the directory that was just created as a result of cloning the repository

- `git checkout -b new_core_branch` //make the required new branch within the repository

Summary: Private Docker Terminal Authentication Methods Authentication methods for cloning a GIT repository within the private docker terminal include the following. All authentication types follow the same flow as shown directly above, however with potentially different `git clone` syntax based on the authentication type.

- **User name and Password**: The user does not have the option to store their user name and password for GIT operations. Each time the user desires to clone a git repository using their user name and password, they must manually enter them into the `Clone GIT Repo` menu. These values will be passed along in the `git clone` command that the `RW_git_docker` runs.

- **GIT Token**: GIT token authentication is possible if the user has previously stored their GIT token in the settings menu (see Section 7.3.1). After checking that the users GIT token exists in the database, the `RW_git_docker` is launched and the token is passed along in the `git clone` command.

- **Public and Private Key Pair**: Public and private key authentication is possible if the user has previously stored their credentials in the settings menu. After checking that the credentials exist, a few specific steps must happen in order, each time a repository is cloned.

  - Save the user's public and private key from the database into separate text files.
  - Create a temporary user account within the `RW_git_docker` (remember the docker is a full Linux operating system).
  - Populate the temporary users `.ssh` folder with the saved public and private key.
  - Populate the temporary users `known_hosts` file with the identity of the repository the user is cloning from .
  - Run the clone command. When attempting to use public private key authentication, the `git clone` command will look into the temporary `RW_git_docker` users

`.ssh` folder and attempt to use the credentials that were placed there in the previous step.

These three authentication methods allow a user to clone GIT repositories into their private folders within the LiveOS. Another notion of using GIT repositories lies within the shared project docker terminal.

7.3.3.3 Shared Project Docker Terminal

The process for cloning a GIT repository into a shared docker terminal differs from the private docker terminal and the administrative terminal in the back end security handling. Shared project docker terminals abide by the same read only standard for LiveOS file mount points, so a similar flow for cloning a repository exists in the sense that the shared project docker terminal must call on the `RW_git_docker` to write files into the LiveOS file mount directory within the shared project docker.

After opening a shared project docker, the user has a `Clone GIT Repo` option in the menu, which when clicked, will bring up a menu for cloning a repository. The only difference in this menu when compared to the private docker terminal is shown in Section 7.4. The differences in the cloning security options are subtle yet important.

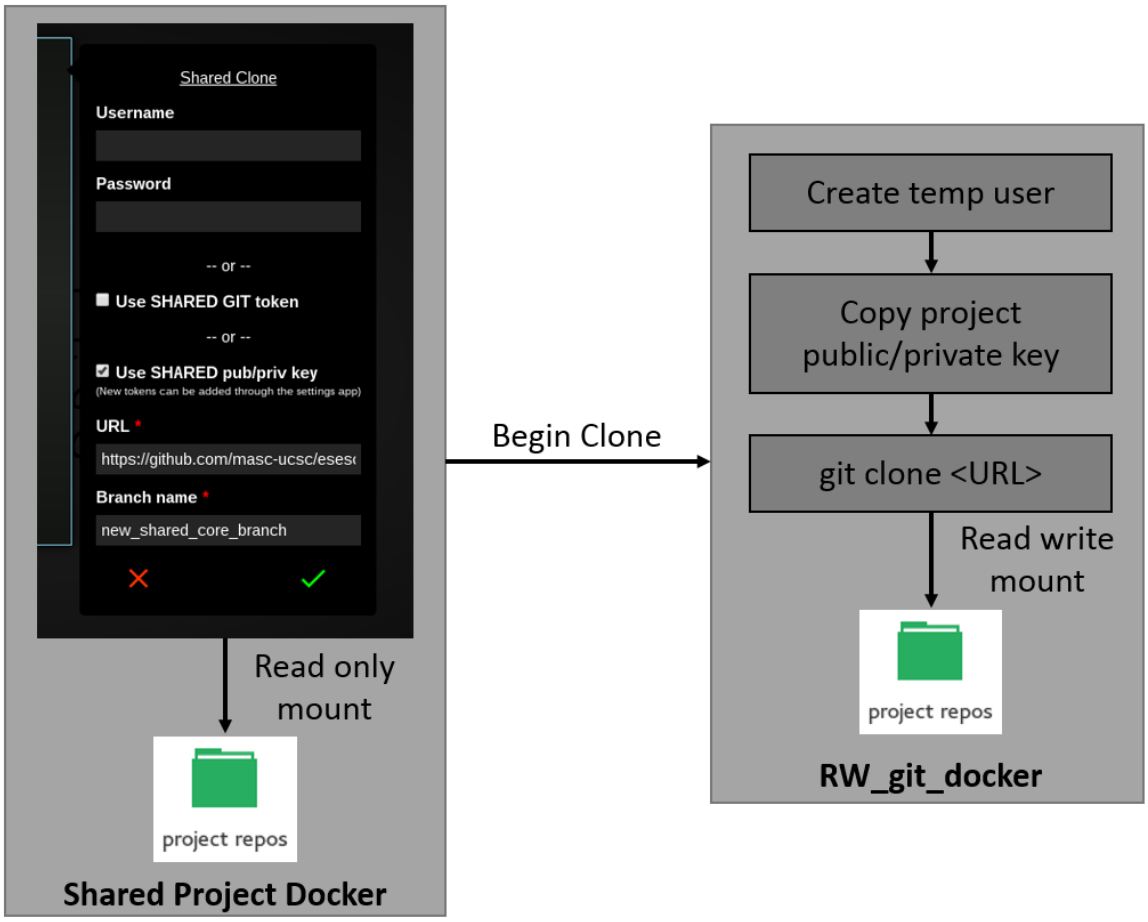


Figure 7.6: LiveOS shared project docker terminal GIT clone using public/private key pair authentication

In Figure 7.6, the user is specifying to clone a GIT repository within the shared project docker with the *shared* public/private key pair. The shared public/private key pair within a shared project docker is essential to the proper operation of collaboration within GIT repositories. When a new project is created, a new shared project docker is launched along with it. At this point, the project has no cloned repositories. Once the owner or another user clone the first repository within that shared project docker, their security credentials (for example their public/private key pair) for that specific clone/type of authentication get stored as the shared

project dockers authentication for all future clones of that security type. This establishes the notion of an *owner* of a LiveOS project, in terms of authentication. What this means is that the first user (call them `user1`) to clone a GIT repository with their public/private key pair will have that pair stored for all future repository clones within that project and of that security type. This allows any subsequent user to clone any repository that `user1` has access to. The same logic applies to using GIT tokens within the shared project docker.

Summary: Shared Project Docker Terminal Authentication Methods Authentication methods for cloning a GIT repository within the shared project docker terminal include the following:

- **User name and Password**: The user does not have the option to store their user name and password for GIT operations. Each time the user desires to clone a git repository using their user name and password, they must manually enter them into the `Clone GIT Repo` menu. These values will be passed along in the `git clone` command that the `RW_git_docker` runs. Because the user name and password do not get stored, there is no notion of an *owner* for user name and password authentication within the shared project docker. Users can then only contribute to the remote repository if they have access.

- **GIT Token**: GIT token authentication is possible if either a user working in the project has previously used their GIT token to clone a repository(the token is shared and reused) or if the user has their own GIT token stored and is available to become the project's GIT token. After checking that either of the previous two options are true, the `RW_git_docker` is launched and the token is passed along in the `git clone` command.

- **Public and Private Key Pair**: Public and private key authentication is possible if either a user associated with the project has cloned a repository using their public/private key pair (thus storing it for use by all project members) or the user has previously stored their credentials in the settings menu, and they are available to become the project's public/private key pair. After checking that either of the previous two exist, a few specific steps must happen in order:

  - Save the project's public and private key from the database into separate text files.
  - Create a temporary user account within the `RW_git_docker` (remember the docker is a full Linux operating system).
  - Populate the temporary users `.ssh` folder with the saved project public and private key.
  - Populate the temporary users `known_hosts` file with the identity of the repository the user is cloning from.
  - Run the clone command. When attempting to use public private key authentication, the `git clone` command will look into the temporary `RW_git_docker` users `.ssh` folder and attempt to use the credentials that were placed there in the previous step.

These three authentication methods allow a user to clone GIT repositories into the specific project folder within the LiveOS.

7.3.4 Back End: Contributing to Repositories Using the _git Command

After cloning a repository, the user of a private or shared project docker terminal most likely would like to do something with those files within the LiveOS and contribute changes back to the remote repository. As described in Section 5.4.1.3, the user has the ability to manipulate files within a GIT repository using the `_git` command within a private or shared docker terminal. The different `_git` commands include:

- `_git branch`
- `_git branch <branch name>`
- `_git branch -d <branch name>`
- `_git checkout -b <new name>`
- `_git checkout <branch name>`
- `_git sync`

The general flow involves invoking one of the `_git` commands within either a private or shared project docker terminal. At this point, the command references a symbolically linked binary that we have created and placed inside each of the private and shared docker terminals. The invoked binary sends a message from within the terminal to the server with the GIT parameters the user has specified, along with things like the type of terminal the message is being sent from, the current path, the user ID, the project id and any files specified when the binary was called. Because the docker terminals do not have write access to their LiveOS file mount

folder, the terminal server itself then launches a process, moves into the correct directory as per the message sent from the user's terminal, and executes the GIT command.

Because the process of running these commands involves sending a message with parameters to the terminal server which then ultimately runs the process, the `_git sync` command easily wraps three GIT commands into one user command. When the user launches the `_git sync` command, they are telling the server they would like to commit their local changes that they have made in the docker container, pull the most recent changes from the remote repository and finally, push the merged contents back up to the remote repository. This allows the user to contribute changes to the repository, even in a read only environment. In the event of a `_git sync` failure, the user is alerted of the error and error logs are updated for a LiveOS administrator to review.

Chapter 8

LiveOS Application Built: OT Based Version Vontrol

What worries you, masters you.

John Locke

8.1 Introduction: Operational Transformation Based Revision Control

Collaborative environments allow many users to connect and edit content. Traditionally, editing content in any word processor was a single-user event. The user was responsible for any change, revision, addition or deletion. To revise the content or go back to a snapshot of the content from a certain date in the past, the user would be limited to the working history of the word processor, typically the working memory of the program, or if the user has manually

saved different revisions of the file to any non-volatile form of storage.

Live collaboration tools within the LiveOS, like the codemirror application, tie the traditional use of a word processor with the Internet to bring all single-users into a collaborative environment. Being that it is an application that users can rely on to maintain a collaborative nature with all other content editors, any single document can experience a large amount of concurrent use. This use translates to a tremendous amount of user changes and revision data, and the lines of what a *revision* and an *undo* operation actually are become blurred. Can a single user only undo their own changes to the content? Should any single user be able to undo other users' changes? Should any single user ever be able to undo all other user changes? This is an interesting problem because in traditional word processors, operation *revisions* would change the state of the document. With many collaborative users, undoing content at will can be devastating to all users' experiences.

Operational transformation allows many users to collaborate on a single document and see all changes being made in real time. In the LiveOS, the codemirror application relies on operational transformation to allow collaboration. Up until now, the codemirror application with OT integration has been plagued by a few usability issues, namely: granular OT revision control and the ability to respect the originality of collaboratively created content.

In this chapter, we explain our implementation of operational transformation based revision control within the LiveOS. This work is focused on giving the user the non-destructive ability of visualizing content with single character resolution from any time in the past, written by all, some or no other users within the LiveOS.

8.2 Why Operational Transformation Revision Control Is Needed

With the LiveOS aimed at being as useful, simple and collaborative as possible, applications that encourage many users to type, edit or contribute in any way increases the effectiveness of the LiveOS. When many users are collaborating on a single version of any content, the ability to work with, add to and edit any version of the content, in character by character deltas, is extremely important. For example, one user working with only their own original file can undo, redo, add or delete with no impact or repercussions caused to other users. This typically works fine because the user is aware of the changes they are making and can continue working with the content until they are satisfied.

With two or more users collaborating, for example in the simplest case, one user solely contributes content that is non-overlapping with the other user. Solely editing content from either user, in this case, is trivial and will not impact the other users work. Further, any changes that are made are easily traceable to the editor. In a more complex example, many users are editing different parts of the content, overlapping and non-overlapping. If any user causes negative impact to the content as a whole, a solution to revising the content to the last known working state is not immediately defined. With the current OT implementation, content revision information is stored per user and lost after the document is closed. This means that any one user cannot control or improve detrimental changes contributed by other users nor can they look back in time further than the current working session for changes.

With our improved implementation of OT revision control, the goal is to have a truly collaborative solution to creating and contributing content *in addition to* destroying or revising

content, such that the *undo*s can be undone and the *redo*s can be undone, all at a character by character resolution, ranging from the very first revision of the file up to the most recent change.

8.3 Design

The OT revision control application is designed to be launched along side of the file the user is currently editing. Further, the revision control application manipulates and reflects OT revision data for the file that is open in the code editor window within the LiveOS.

8.3.1 Database Schema Design

OT information within the LiveOS code editor application is constantly being sent between all clients who are editing any file with OT enabled and the OT server. Each individual character is being sent from clients to the OT server to ensure the proper OT functionality of keeping each instance of the OT enabled file up to date with all content that users are typing. At this point, the individual characters typed are only contained within the working memory of the OT server.

To be able to recall any revision ever made by any user for any file, we intercept each OT message containing the user, file, operation and time stamp and store it to the LiveOS's mongodb using a special OT schema that we created. With each character stored as it is typed, we can then interface our OT revision control application with the database to selectively show different combinations of events.

8.3.2 Front End Design

The front end design of the OT revision control module is split into three main views: a user selection view, a content control view and the content display itself. The goal for each of these three views is to simplify the user's control over the potentially extremely large set of revision data. Upon launching the application, the user is presented with user selection options, a time-based slider for content revision control and a simple display of the stored OT revision logs. See Figure 8.1.

Figure 8.1: OT revision control application

8.3.2.1 User Selection

In the user selection area, individual users can select which LiveOS user's OT revision content they want to see. The user has three options for the selection of which content to show for the file currently being edited or viewed: only their own OT revision content, all users revision content or a subset of users' revision content. Each button is mutually exclusive and only one can ever be selected at once.

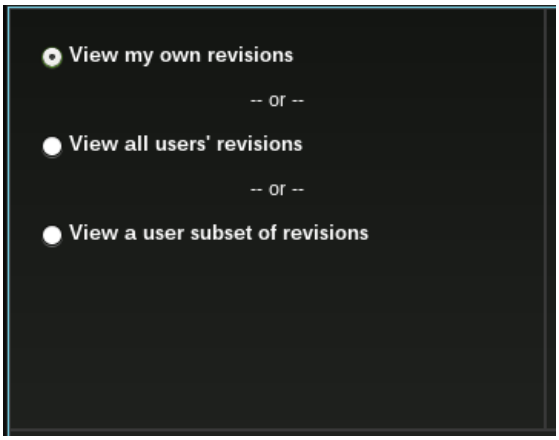


Figure 8.2: OT revision control user selection

Personal revision content Selecting this radio button simply prompts the front end client code to send a web socket message to the back end. This message contains the user ID of the user who is requesting personal revision data and the name of the file they are requesting it for, which is the file that they are currently viewing or editing.

All revision content Selecting this radio button sends a web socket message to the back end with the file name only. The idea is that the message indicates the user is requesting all available OT revision data for the given file, and filtering on a certain user is not needed.

Subset of revision content Selecting this radio button sends a web socket message to the back end, querying for a given file, all users that have edited it. Upon completion of the query, the client code parses the results and displays, with their own selection boxes, the names of which are available to see OT revision content for the given file. Upon selecting one or more of the returned options, another web socket message is sent to the back end to retrieve the selected user content.

8.3.2.2 OT Revision Content Control

In the content control area, the user can select between showing the raw OT revision logs or a file differences view (`diff` view). Below these options, a slider can be moved to control the time period of the OT revisions that are shown. The slider has two endpoints: the left endpoint represents the oldest OT revision log for the current query while the right endpoint represents the most recent OT revision log for the current query.

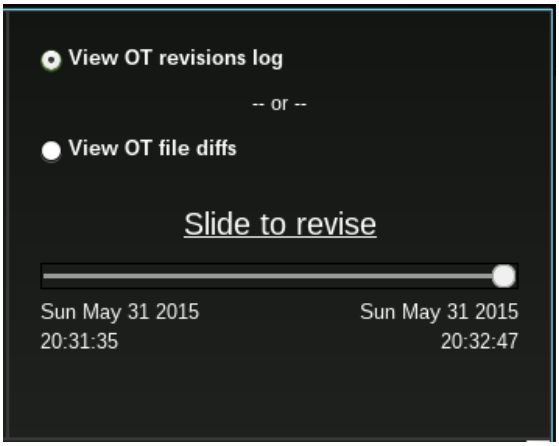


Figure 8.3: OT revision content control

In Figure 8.3, the radio buttons are mutually exclusive, meaning only one can be selected at a

time. Further, the slider's left and right endpoint date displays are dynamically updated based on the revision logs that are returned from the database query. To give the user more specific information about the current revision they are viewing, as the user hovers over or moves the slider, a pop-up containing the date is shown to the right of the slider element. Seen in Figure 8.4, this date shows what date and time the sliders position is currently representing, and thus also showing the date and time of the currently displayed results.

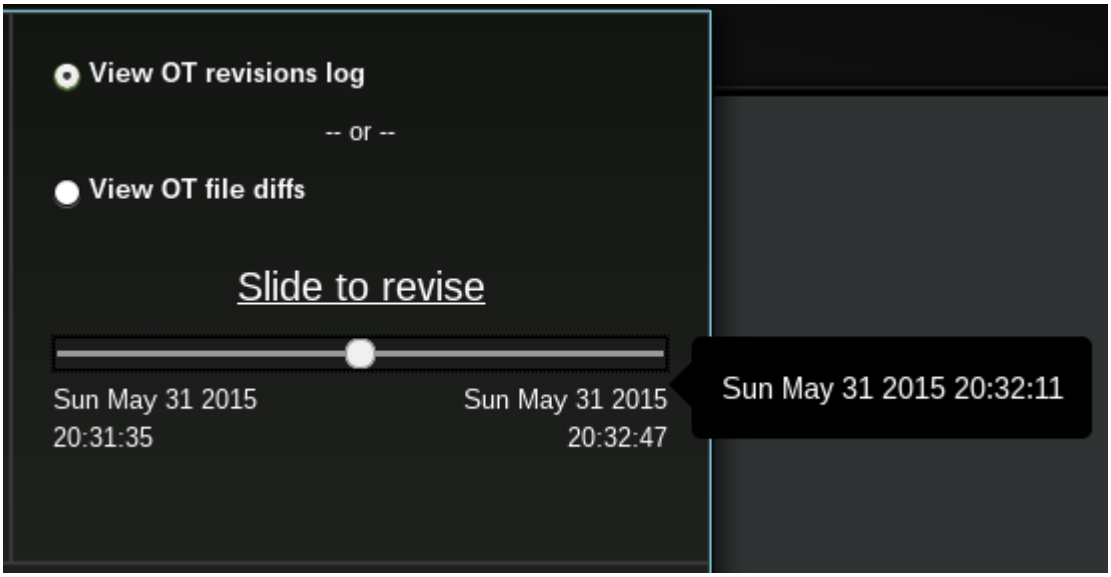


Figure 8.4: OT revision content control slider with pop-up date display

OT revision logs and `diff` view These two radio buttons simply control elements on the front end. If the OT revision logs button is selected, the content display shows a list of OT revisions logs returned from the database as text containing the date of the modification, the character modified, the position at which it was modified and name of the modifier. If the `diff` view button is selected, a new view is loaded into the content display described below in section 8.3.2.3.

OT revision slider The slider element is currently programmed to linearly space the OT revision content across its one-hundred steps. For example, if personal revisions exist from only the past 24 hours, moving the slider from its rightmost position to the middle of its travel will display, in the OT revision log view, results from 24 hours ago up to results from 12 hours ago. If in the `diff` view, the 12 hour period spanning from the current time through the past 12 hours will be removed from the content display, thereby showing the content as it would look without your personal additions or revisions, 12 hours ago. It should be noted that because the results are split linearly across the one-hundred steps of the slider, content written at a consistent rate will be manipulated more evenly as the user moves the slider.

8.3.2.3 Content Display

The content display is capable of displaying OT revision logs as well as a content viewing utility called `merge.js`. This utility is an add-on supplied by the LiveOS code editor `codemirror`.

OT revision logs When viewing OT revision logs, the content display panel simply lists the parsed results from the database in text form, with some specific coloring to more easily spot important aspects of the data. Below in Figure 8.5, the OT revision logs are shown for a single user on an example file with content: `Hello World! \n\nLiveOS is great!`.

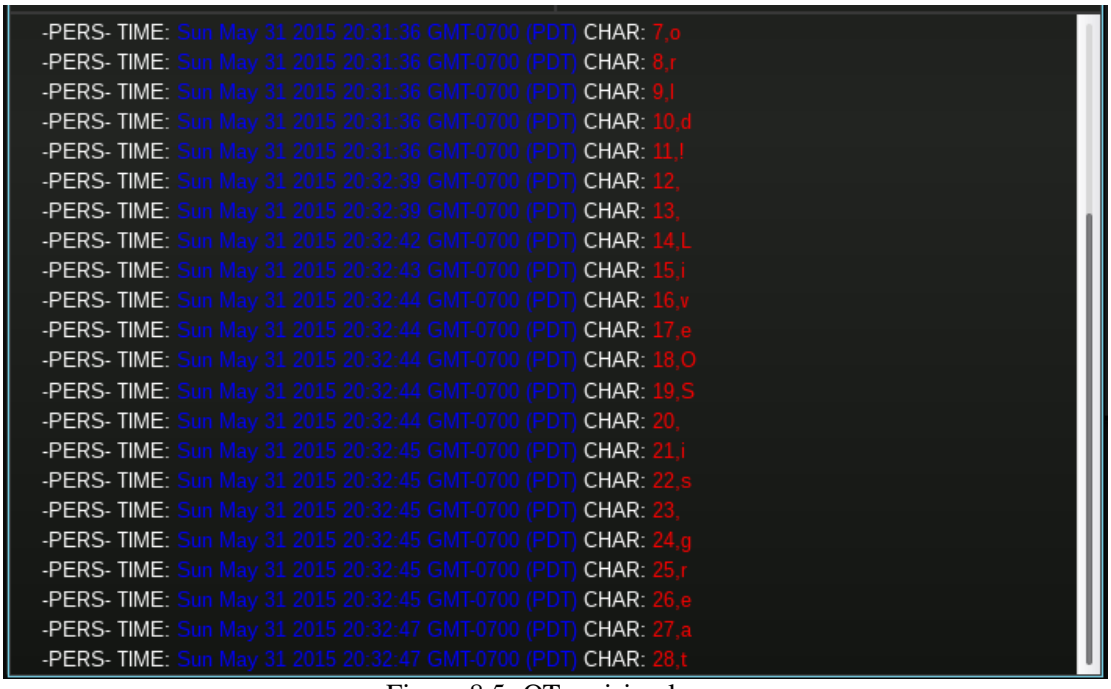


Figure 8.5: OT revision logs

The blue color represents the time the revision was stored, the red shows the character added or modified as well as the position the change happened at and finally the green color shows the user name of the modifier.

`diff` view When viewing results using the `diff` view, two side-by-side content viewing windows are shown. On the left, the content shown is based on the position of the slider. On the right, the content is static and reflects the most up to date version of the file. In `diff` view, the idea is to show the user snapshots of what the content has looked like in the past, using the time reflected in the slider and the user selection criteria. Moving the slider effectively controls the state of the left side content view.

For example, moving the slider from the furthest right to the furthest left, with the

radio box *View my own revisions* selected, character by character the user's changes will be removed from the file, until the file is either empty (this means they were the only contributor to the file) or until all of their OT revision data is removed from the file, and all that remains are contributions from other users. Note that when the slider is on the furthest right, all of the user's personal OT revisions are included in the view. This means that if they are the sole contributor to the file, there will be no differences between the left and the right view. Should differences arise as they undo certain user changes, the side-by-side content view automatically highlights differences in lines. See Figures 8.6 and 8.7 for examples. Notice too that the user can select to view or hide the content highlights as well as collapse similar parts of the content to help the user visualize content differences.

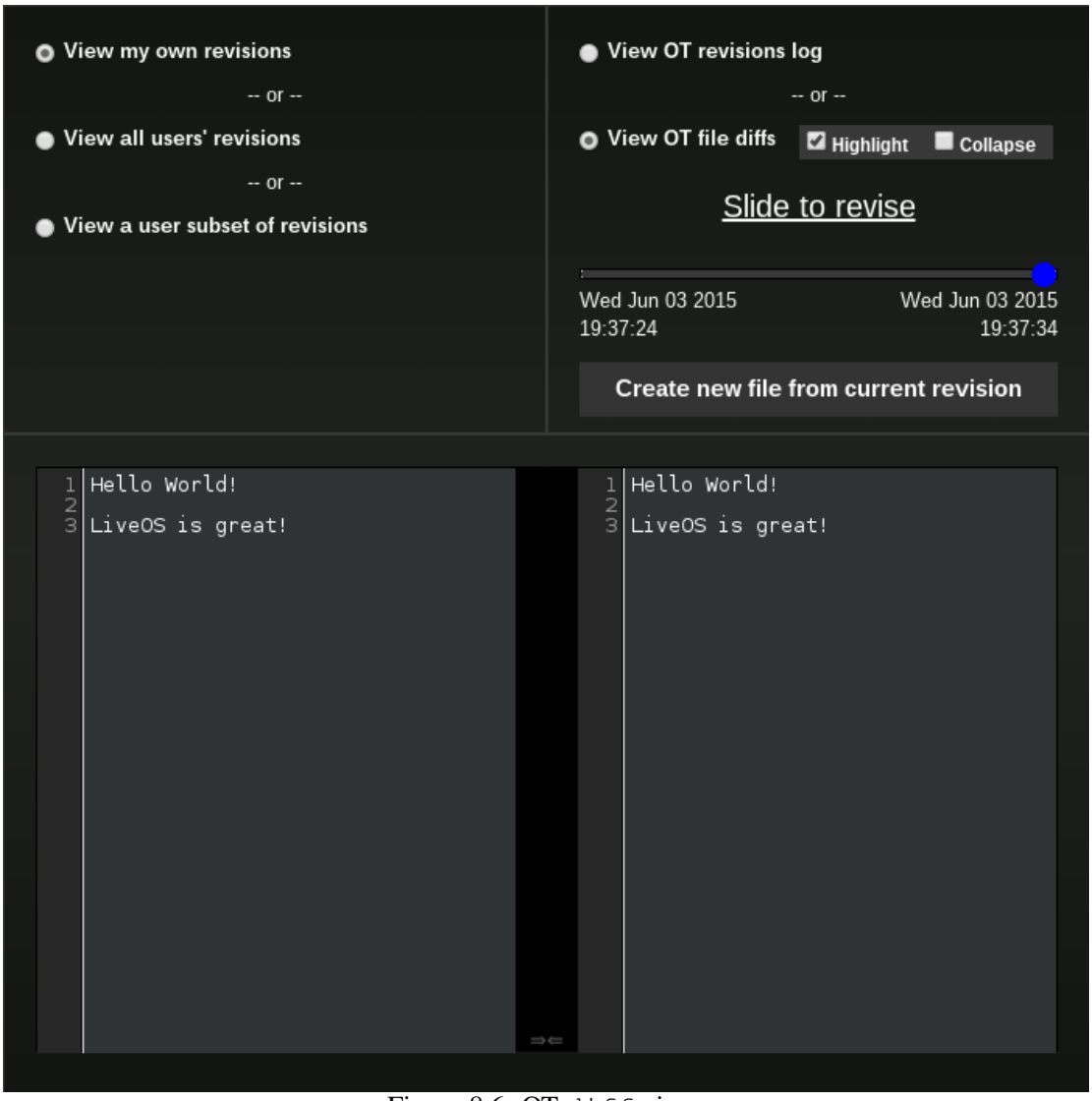


Figure 8.6: OT `diff` view

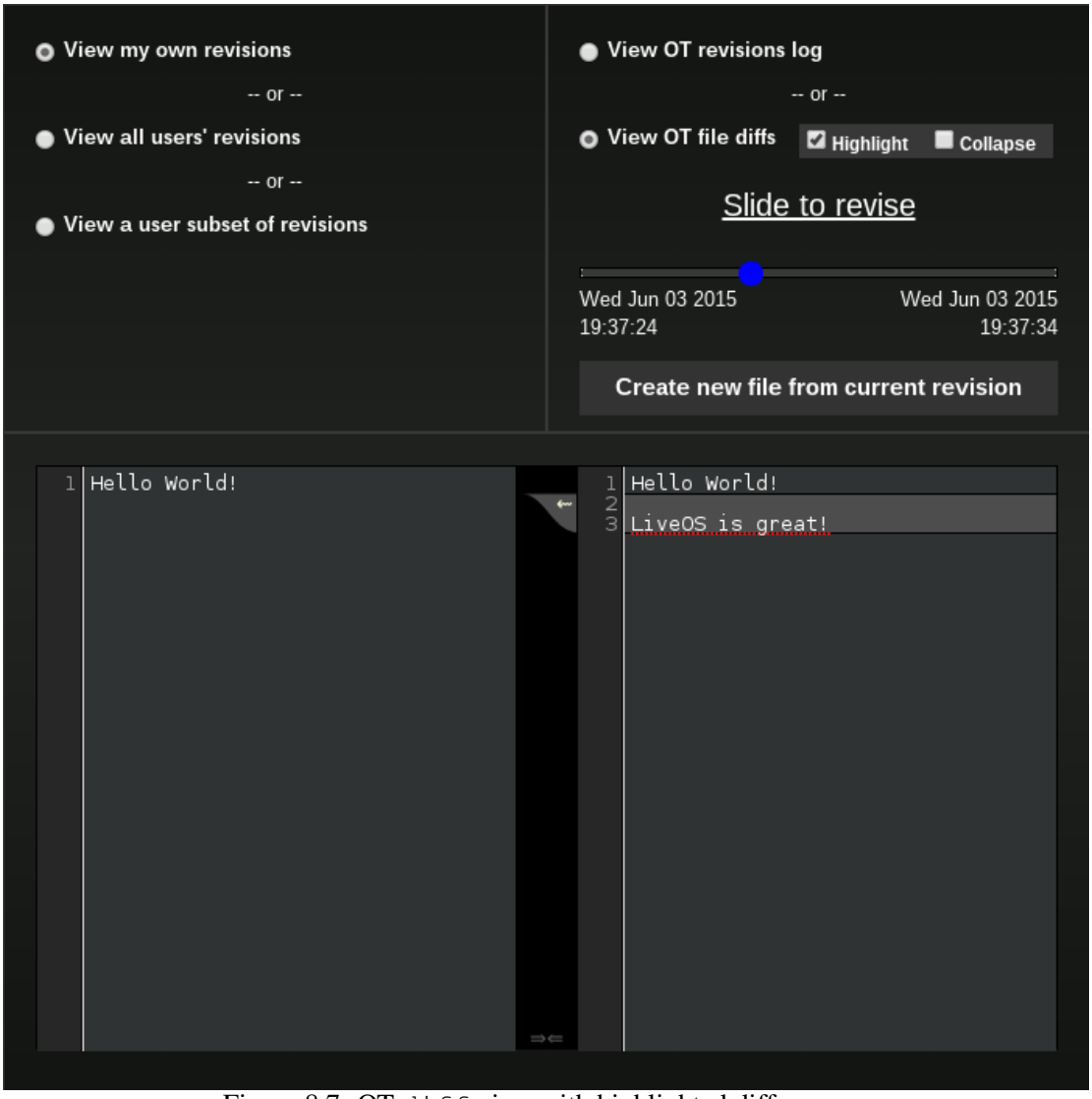


Figure 8.7: OT `diff` view with highlighted differences

8.3.3 Back End design

The back end design of the OT revision control module includes hooks into the LiveOS database as well as logic to control handling the potentially enormous amounts of database queries on large sets of data that are possible. The back end must handle three main

types of user requests: single user revisions, all user revisions and a subset of user revisions. These three types of requests resolve to different logic to parse the returned database results, extract the information needed to update the front end view and send the information back to the client. It should be mentioned that the back end for the OT settings application is the file server for the LiveOS.

8.3.3.1 Range selection

Universal to all three look up types, the slider value, ranging between 0-100, needs to be conditioned in order to be useful to any database query. On the back end, the slider value is used as a multiplier against the oldest revision date for the current file in the database as well as the difference in time between the newest and oldest revision in the database for the current file. In OT log display mode, the database can then find all records *less than* the returned time. This makes sense because in OT log display mode, the user is requesting to see all revisions *up to* the current time chosen by the slider, therefore the query must return all results ranging in time from the oldest revision *up to* the time chosen by the slider.

In the `diff` view, the user is interested in the state of the file *as it has looked* in the past according to the position of the slider. Therefore, the database query must return all results *greater than* the time specified by the slider. It is at this point that the client application can, in the `diff` view, *remove* from the current file the results returned by the query.

The exact equation is:

$$oldest\_record\_time + ((slider\_value/100) * (newest\_record\_time - oldest\_record\_time))$$

8.3.3.2 Single user OT revision control

When a user is requesting to view their own OT revisions, the back end queries the mongodb using the users ID, the file they are requesting revisions for and finally the position of the content control slider. The database query is dependent on all three pieces of information. More specifically, the users name and the file they are editing are passed as strings into the database query. Once the command is executed, the returned results are sorted by the date the revision was made.

8.3.3.3 All users OT revision control

When a user is requesting to see *all* user revisions, the query to the database is simple. The query specifies to look for any OT revision that has ever happened to the given file (respecting the range selection described in section 8.3.3.1). Returned from the query is the user who executed the operation, the time the operation happened and the operation and position itself. These results are sorted based on the date the revision was made.

8.3.3.4 Selective subset OT revision control

When a user is requesting to see a *subset* of user revisions, multiple database queries must take place. First, the back end must determine which users, if any, have made OT revisions to the current file. This is accomplished with a single database query, searching for any user ID who has made revisions on the current file. This is returned to the front end application to be displayed as selectable entities. Once the user can visually see who has made revisions to a file, they can then select which user's revisions to see. From this point, the database query becomes

similar to the others, the only exception being that the results returned will be from those users who are selected on the front end application.

Chapter 9

Conclusion and Future Work

> I am enough of an artist to draw freely upon my imagination. Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.
>
> Albert Einstein

In this work, we have discussed key additions and modifications to the LiveOS collaborative environment. This work greatly enhances the user experience of the LiveOS and allows more users to experiment with, control and publish their collaboratively generated content in a new, seamless, minimalistic way. The applications and functionalities implemented allow future LiveOS users and developers to leverage:

- A terminal within the LiveOS to debug and develop new code.
- A code search utility to quickly search through thousands of source code files for a spe-

cific string.

- A back end utility named Docker, capable of compartmentalizing applications into their own virtual operating system.
- Seamless GIT integration for increased work flow efficiency.
- OT revision control to see the state of any document at any time since creation with single character, user and file resolution.

The cumulation of these new functionalities makes the LiveOS more useful than ever. The terminal is powerful as a single entity yet portable for integration into any LiveOS application. As new files and repositories are added, the code search application will continue to provide users with instant access to any of their files. Docker and GIT integration are currently excellent platforms to leverage within the terminal application, but as the LiveOS grows, the power of Docker and GIT can be integrated into *any* future LiveOS application. Finally, as the LiveOS moves towards a more complete collaborative experience, LiveOS users will have more control than ever over their personal and shared content using the OT revision control application.

In the future, some minor improvements can further increase the usability of the LiveOS. With respect to Docker, the LiveOS currently only uses docker containers to accomplish increased security and portability via the terminal application. As more LiveOS applications become dockerized, increased server separation and application control is possible, in addition to inter-docker communication for new ways to pipe data through docker containers with minimal server overhead. In addition, GIT integration can be expanded, allowing any

LiveOS application to have front end hooks into easy to use back end GIT operations. This would allow LiveOS users to seamlessly collaborate without ever leaving the application they are working in.

With respect to the OT revision control, the application currently struggles with the increased complexity of many users visualizing content at very specific moments in time. A more efficient and accurate algorithm for viewing differences between file versions would help to properly visualize content revisions.

Bibliography

[1] A code-searching tool similar to ack, but faster. Available at http://www.github.com/ggreer/the_silver_searcher.

[2] Collaborating in really real-time. Available at http://www.etherpad.org.

[3] Develop in go, python, node, ruby, php, etc or play with docker, wordpress, django, laravel or create android, ios/iphone, html5 apps. Available at https://www.koding.com.

[4] A full-featured ide in your browser. Available at https://www.pro.nitrous.io/?l=1.

[5] git –distributed-is-the-new-centralized. Available at http://www.git-scm.com/about.

[6] Overview of google docs. Available at https://support.google.com/docs/answer/49008?hl=en.

[7] Regular expression matching with a trigram index or how google code search worked. Available at https://swtch.com/~rsc/regexp/regexp4.html.

[8] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, January 2010.

[9] Stijn Dekeyser and Richard Watson. Extending google docs to collaborate on research papers. *Toowoomba, Queensland, AU: The University of Southern Queensland, Australia*, 23:2008, 2006.

[10] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.