MALTE: Multi-Access Live Text Editor

Draft Report

Software Engineering Project, 1DL650

Adam Inersjö Jonas Norlinder Michael Rehn Joel Westerlund

December 12, 2019

Contents

1 Introduction

MALTE is an open-source tool that runs in the browser and enables collaborative code editing. Moreover, each connected user have their own terminal on which they may compile and run the source code entered in the editor. Our tool runs in the cloud with all computationally heavy work, e.g. compilation, done on server-side. This enables the users to utilise the product at any device capable of running a modern web browser.

The goal of MALTE was to allow all developers to collaboratively edit and compile code. In order to fulfill this goal, the system was intended to have supporting functionality to make code collaboration easy. This includes access to a terminal, file explorer with common file system operations, caret synchronisation, user list and permissions management.

2 Background

To enable users to edit code simultaneously some problems with synchronization has to be solved. When several users makes modifications to, e.g. a text file, the system has to ensure that all users has the same view of what the actual state of the text file should be, without inconsistencies. In order to provide a consistent state throughout the system, we make use of data structure called replicable growable array (RGA). RGA belongs to a class of data types called conflict-free data types (CRDT). In this section we firstly introduce some terminology commonly used, we then argue for our choice of consistency model. Finally we describe how we use CRDTs and RGA to enforce this consistency model.

2.1 Terminology

A *replica* in a distributed system represents a "copy" of some shared state. Each replica can be modified with *operations*, which is data that represents what modification is made, and how it should be performed.

The *operation* are then sent to other clients which apply them to their own replicas. This can be problematic in a distributed setting since some operations are not commutative, i.e., the order of operations influence the end state of the replica. For example, insertions into a string is not commutative since $a + b \neq b + a$. Algorithms behave differently and each algorithm guarantees their own properties of the final state. What properties can be guaranteed for an algorithm is said to be the algorithms *consistency model*.

2.2 Consistency Model

All distributed systems who deal with replicated data have to deal with the problem of consistency. There are many different consistency models, of which two are especially important. Eventual Consistency (EC) is a common consistency model which allow different replicas to diverge as long as they sometime in the future converge [1].

Strong Eventual Consistency (SEC) is a specific consistency model within the EC consistency model family. This model promises that all replicas that have observed the same operations, in any given order, will converge to the same state [1] without the need of any further synchronisation.

The promises of SEC is suitable for a system as MALTE as it ensures that all operations are considered for the final state. For the file tree, however, we are content to use the relaxed EC definition. The reasoning for this is that file system operations are less frequent and consistency can be solved with simpler means instead, e.g. checking if a file actually exists before trying to rename it. This is possible thanks to the use of a central server who has authority over the file system and that can resolve any potential race conditions.

2.3 Conflict-Free Replicated Data Type

A conflict-free replicated type (CRDT) is a data structure which guarantees SEC without the need for any synchronization [2]. The simplest examples of a CRDT would be a counter with the operations *add* and *subtract*. Both the *add* and *subtract* operations are commutative, thus, it does not require synchronization to converge, since e.g. $1 + 2 + 3 = 3 + 2 + 1$.

Operations in CRDT are often distinguished between upstream operations and downstream operations. The former being an operation generated by the local replica while the latter being an operation received from a remote replica. CRDTs for group editing usually have a very good downstream complexity whereas the upstream complexity suffers, resulting in less responsive group editing [3].

3 Problem Formulation

MALTE is with no doubt a distributed system with many points of replication. The text in each file is one point of replication. Another point of replication is the file system. Work has to be done to ensure either EC or SEC for both these points of replication.

As with all openly accessible system, it has to protected by some sort of authentication. To this end, authentication and user permissions is a problem which has to be solved. User data must also be stored in some form of persistent storage.

4 Related Work

There are already similar systems for collaborative, in-browser editing of code. These systems range from simple collaborative editors, such as Firepad, to full-fledged integrated development environments (IDEs) running on the cloud, such as Visual Studio Online [4, 5]. Visual Studio Online can be accessed through a browser and extended to allow for collaborative editing in a project. REPL.IT extends the concept of a project in Visual Studio Online by allowing for classrooms where students can learn and collaborate while teachers can assist the students and grade their solutions [6].

Other related software that experiences synchronisation issues are collaborative productivity tools such as Google Docs and Figma, all of which either uses Operational Transformations or Conflict-Free Replicated Data Structures to handle synchronisation issues [7–9]. As to our knowledge there are no open-source projects that provides the functionality we will implement.

5 Requirements

For the system to be considered complete, we considered the following requirements to be necessary

- Concurrent editing of files
- User and permission management
- Shell/terminal access
- File explorer with basic file system operations

Concurrent editing of files requires that two or more users should be able to edit the same file concurrently, with the guarantees of SEC, i.e. replicas are allowed to diverge as long as the converge when they have received all operations.

User and permission management requires the concept of a user. The users should be able to configure who has access to the project through the use of a UI. The system also needs to be able to authenticate users.

Shell/terminal access requires that a user have access to a shell. This shell should be able to receive input from the user and display output from commands. The shell access should be personal and not shared between users.

Finally, the file explorer should have a graphical representation of the project's file structure. The graphical interface should also be able to perform all basic file system operations, i.e. creation removal and renaming of a file or folder.

6 Technical Documentation

This section describes all information needed to understand the system on a technical level. This includes how the different parts of the system communicates, how different types of data is stored and instructions of how to run and edit the system.

6.1 API Specification

The project communicates mainly over a WebSocket connection, but partly also over HTTP. A full description of the API specification can be found in the Appendix.

6.2 Concurrent Editing

Replicable Growable Array

To enable concurrent editing we have chosen to implement the Replicable Growable Array (RGA). The RGA is basically a linked list where each node contains either on or more characters, which together represents text. For example, Figure 1 show how the text "HELLO" could be stored in a linked list.

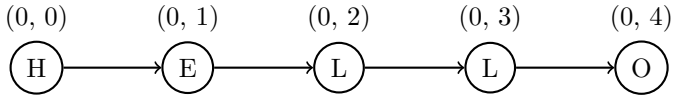

Figure 1: The basics of the RGA is a linked list with one character in each node.

Above every node in Figure 1 there is a tuple. This tuple is the node's identifier, or id. The first number in the tuple is the id of the user who inserted the node. The second number is the upstream vector clock value at the time the node was inserted. In our system we simplified the RGA structure to use Lamport timestamps instead of a vector clock. A node, or its id, is said to precede another node's id iff $clock_1 < clock_2$ or $clock_1 = clock_2 \wedge user\_id_1 < user\_id_2$ [3]. We use $id_1 \prec id_2$ to denote that id_1 precedes id_2 [3].

The RGA supports two operations: insertions and deletions. An insert operation contains two pieces of information, a reference node and the node that should be inserted. When applying the insertion, we will start by looking up the reference node. Then, we traverse the linked list until we find a node that precedes the new node; we will insert the new node before the preceding node. By following this rule, concurrent operations will be ordered across all replicas in a deterministic matter, no matter the order of operations.

The insertion process is illustrated in Figure 2 where two concurrent insertions are made by user 1 and user 2. Both users want to concurrently insert a node to the right of the letter E. We have the following precedence order $(0, 2) \prec (1, 5) \prec (2, 5)$. By following the insertion rules specified above, we will end up with the final state shown in the last row of Figure 2, irregardless of the order of operations.

A delete operation, on the other hand, contains only a reference id. This node, however, cannot simply be removed from the RGA, since concurrent insertions might reference the node that is being removed. Instead, the concept of tombstones are used. A tombstone is a boolean flag, `true` or `false`, which if set to `true` would denote that this character has been removed and should not be visible. A concurrent insertion and deletion is shown in Figure 3. No matter the order of operations, we will end up with the state in the final row, thanks to the fact of never removing the node, only flagging it as a tombstone.

The fact that nodes are never removed means that the document will only increase in size, never shrink is problematic since creating upstream operations have a time complexity of $\mathcal{O}(n)$ where n is the document size, i.e. number of nodes. As a result, the RGA will become slower as time goes on, even if the amount of "useful" data stays the same.

To counter this problem, we will make use of the fact that a central server is used. When all clients have closed a specific file, the system will throw away the RGA structure and save a plain text file to the file system. When a client requests the file again, the RGA structure will be re-created without the

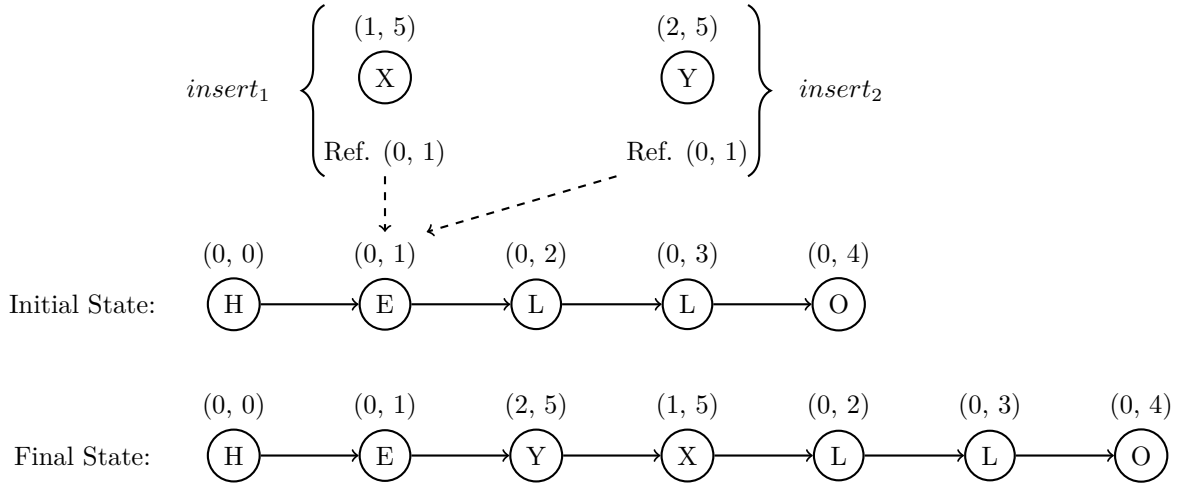

Figure 2: Two concurrent insertions, $insert_1$ and $insert_2$ modify the linked list shown in the first row in a deterministic manner to end up with the final state shown in the last row, no matter the interleaving of the insertions. The id specified next to "Ref." is the insertions reference id.

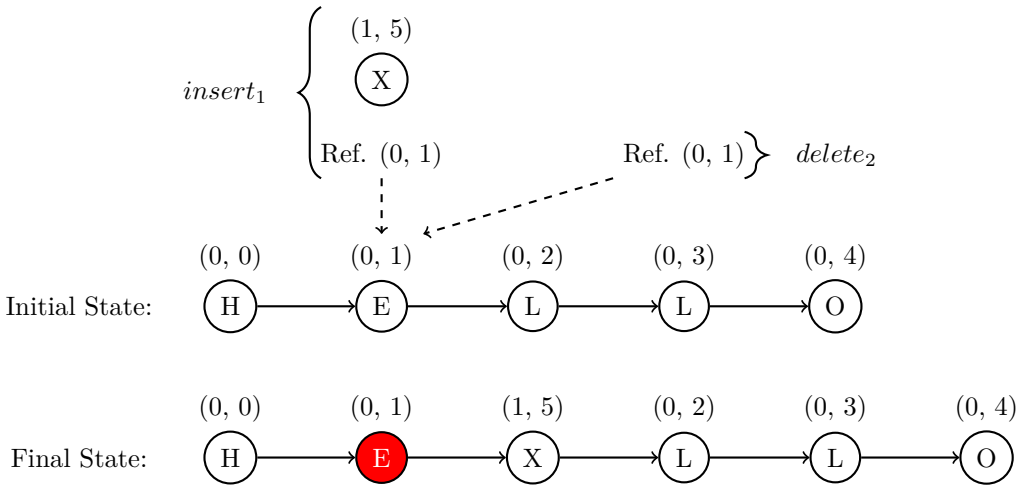

Figure 3: Two concurrent operations, one insertion and one deletion. The coloured node signify that this node has been tombstoned, i.e., has had it's tombstone flag set to true.

tombstoned nodes. Important to note is that this method unfortunately weakens the system's tolerance to network partitioning – operations from temporarily unavailable client might have to be thrown away.

Operation Mapping

Since the main focus of the project was the collaborative nature of editing code, we chose to use an existing front-end code editor, called Monaco instead of creating our own. The downside of not having full control of the internals of the front-end editor is greatly compensated by the fact that we did not have to spend the entire project implementing an editor, but could instead use an existing one that was well-tested and documented.

The use of an externally developed front-end editor meant that a bridge had to be implemented to convert the data that the editor provided into operations that could be applied to the RGA structure and sent between client and server. This process is shown in Figure 4. The Monaco editor exposes methods of receiving changes that has occurred in a code buffer, for example the addition or removal of a string of text. This single operation, which can reference text with more than one character, is mapped into multiple Internal Operations, each only able to represent the addition or removal of a single character. The reason

for this mapping is that the RGA structure described above is only working with single characters, since a user may want to insert new characters in the middle of previously written text. The Internal Operations each have a position relative to the text document which indicates where in the document the operation should be applied. The last step of the mapping of operations is to convert each Internal Operation into a single RGA Operation which can later be applied to the RGA structure. The RGA Operations also represent the insertion or deletion of single characters, but in contrast to Internal Operations the RGA Operations have a reference that indicates where in the RGA structure the operations will be applied.

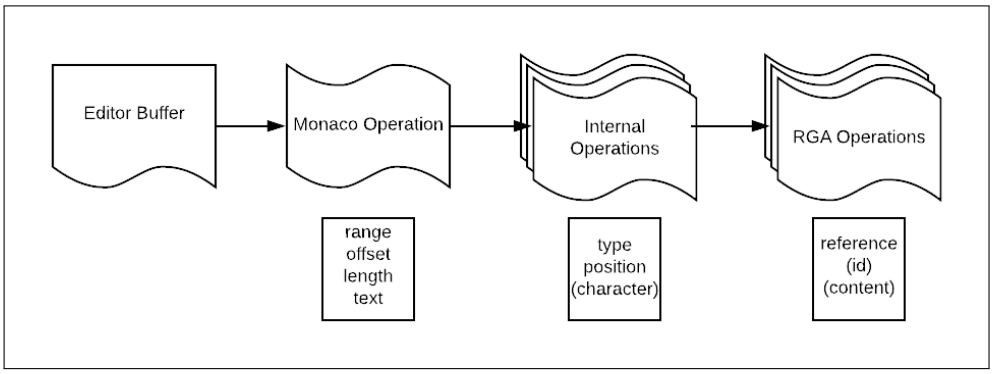

Figure 4: Mapping of Monaco editor operations to RGA operations

After the creation of RGA Operations is complete, they are applied to the local RGA structure of the client making a change in the code buffer. This synchronizes the state of Monaco's code buffer and the internal RGA structure. The same operations are afterwards sent to the server to be applied there, and afterwards broadcasted to all other clients editing in the same buffer. On the side of the receiving client, each incoming RGA Operation is directly converted into a Monaco Operation, skipping the intermediate step of Internal Operations.

To easier see where other users are editing in a file, *caret synchronization* has been added. The synchronization of user carets occur in a similar fashion as the synchronization of operations. The caret position of a user is fetched from Monaco and its position is translated into a reference into the RGA. This caret is then sent to the server and forwarded to other users where the RGA reference is re-translated into a Monaco position.

6.3 Shell Access

To give each user a good development experience we provide an in-browser shell which is piped to a shell on the server. We fork a pseudo-terminal using node-pty. On Windows we spawn an instance of Powershell, on other operating systems we spawn an instance of the Bash shell. We pipe data from/to the shell to the client using WebSocket. The client uses the project Xterm.js as a terminal emulator, which is a tool to render and generate input compatible with the aforementioned pseudo-terminal.

Our current implementation spawns one pseudo-terminal per user using the same underlying user on the host. An advantage with this is that each user can do whatever they like independent of any other user and do some work simultaneously on the file system. A major disadvantage with this approach two users would not be able to compile the same files in parallel.

6.4 File Explorer

MALTE provides a file explorer with common file system operations, e.g. create/remove/rename files and folders. At first look it seems like this could be solved with a similar approach to that of concurrent editing, explained in subsection 6.2. However, since the user has shell access, they can create and modify files from the terminal. If the file explorer only reflected changes with the built-in commands, it would quickly be out-of-date after, e.g., a compilation.

To ensure consistency between the file explorer and the actual file system, file watching was used. File watching is the concept of being able to register a handler for whenever something on the file system

has changed. In MALTE, chokidar was used to provide file watching. Whenever the handler is called, MALTE will query the file system and broadcast an up-to-date state of the file system to all clients.

6.5 User Account Management

As we wanted to be able to display currently logged in users, enable synchronisation of individual caret positions for each user and displaying their username, we decided to force each user to log in before access is gained to the system. In order to minimize our implementation for user account management – e.g. requiring forms to be filled out, databases to be populated – we decided to use GitHub's OAuth API. Thus, to sign in to MALTE you sign in using your GitHub credentials. As with many OAuth API's, the first time you sign-in you are asked to authorise exposure of some personal information connected to your GitHub account that MALTE request's to use.

MALTE limits the requested data from GitHub to only the basics needed for authentication and cosmetics: read-only access to user profile data and e-mail address. This integration allows our system to fetch a user's avatar and unique GitHub username. This allows our system to display a small picture of each logged in user at the top right of the screen providing a nice user experience and situational awareness. The data that we store from GitHub is the following, defined as a Typescript interface

```
interface User {
  login: string;         // The GitHub user name
  id: number;            // Unique GitHub id
  avatar_url: string;    // URL to the user's avatar
  url: string;           // URL to the user's GitHub page
}
```

The downside of this solution is that it requires users of our system to have a GitHub account. It is however free to register a GitHub account, thus it would be of no greater effort than to create an account locally on our server. Furthermore, the users already in possession of a GitHub account would experience minimal effort in getting started using MALTE – a reasonable expectation given the target audience of developers.

6.6 User Permissions and Sessions

To prevent access to the server by anyone with a GitHub login we store approved users in a database and only the GitHub logins that match the users approved in advance are allowed access. In the current version we have made it so that the first user to access the server will automatically get approved in order to allow utilisation of the system. That user can then via the user interface add and remove approved users in the database. In order to prevent the situation where all approved users are removed from the database (i.e. deadlocking the system) we have implemented a restriction on the backend preventing a user from removing their own username from the database. The case of two separate user's that concurrently removes each other from the list of approved users, will not be a problem: as soon as a user is removed from the list of approved users, they will be unauthenticated, thus hindering the second user from removing the first.

The current implementation is not ideal as there exist a possibility that someone sets up a server and immediately gets hijacked by another user who logs in to the system first. This is considered an edge case and can be solved by the project owner by manually editing the database of pre-approved users. A more ideal solution though would be to have a meta server spawning project servers enabling the set up of user permissions before the project server is launched.

6.7 Development Instructions

The project is structured within a monorepo, as can be seen in Figure 5. The project is split into four seperate packages: `rga`, `malte-common`, `frontend` and `backend`; `frontend` is a web application that contains the client code and user interface; `backend` contains code for the central server that all clients are connected to; `rga` contains the data structure described in subsection 6.2 and is consumed by both `backend` and `frontend`; `malte-common` contains common functionality and type definitions that is shared

between both backend and frontend. The package `rga` and `malte-common` **must** be built before starting or building `frontend` or `backend`.

All four packages can be configured to rebuild automatically when a file has been changed to make development easier, thus we recommend that you firstly start `rga` and `malte-common`, using `npm run build:watch` and then run `frontend` and `backend` using `npm start`. Then the whole project will recompile as soon as any file have been changed on disk.

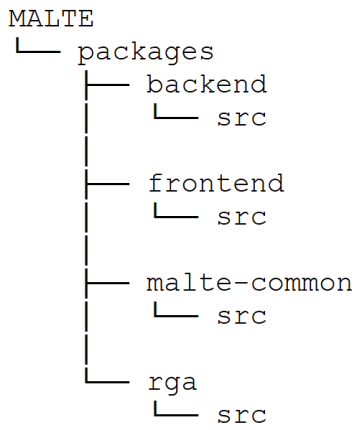

Figure 5: File structure

6.8 Dependencies

You need to have Node 12 or greater, MongoDB and for node-pty you need to run `apt install -y make python build-essential` on Linux or `npm install -global -production windows-build-tools` on Windows. Windows also requires that you install Windows SDK (Desktop C++ Apps will suffice). `MONGODB_URI` default to `mongodb://localhost:27017/malte`, so a local development won't require one to change that environment variable.

Another dependency is that you need to register an OAuth application on Github and provide the keys in the environment variables `GH_CLIENT_ID` and `GH_CLIENT_SECRET`.

6.9 Tests

All functions that are easily testable should have tests. In each package one can run `npm run test` to run the test suite.

6.10 Code Style

We use Prettier for styling and ESlint for code checks. ESlint is configured with their recommended configuration as well as checking for the default Prettier configuration. You can run all checks by running the command `npm run lint` in all packages.

7 Quality Control

We are utilizing modern techniques to achieve quality control: unit tests, regression testing with continuous integration and continuous deployment, linting tools and code review.

7.1 Tests

We have implemented tests for any part that were easily testable. Most importantly, the data structure which is the heart of this application, RGA, have a full test suite to verify proper implementation, e.g. that replicas converge to a consistent state. Some parts that consisted of many or only side-effects were

opted out from formalized testing. Those functions were instead verified by just using and trying to break it, this included mostly the frontend. TODO: See appendix X for a list of all functions that have tests.

7.2 Continuous Integration and Deployment

Continuous integration (CI) where implemented by using GitHub Actions. It will perform static code analysis on all packages and run each package's regression test suite, except for `malte-common` since it contains no tests. We also utilized branch protection rules on the main branch, such that no feature branch could be merged to the main branch until all CI tests passed.

To further help with increase the code quality we also implemented continuous deployment using Heroku. Each time a commit were made on the main branch it triggered a Docker build on Heroku, which built the entire system from scratch. The workflow were: (1) commit to main branch, (2) Heroku gets triggered, (3) Heroku checkouts the latest commit on main branch, (4) Heroku will build and deploy according to the Dockerfile in the source code project root. If the build failed, we received an email and then we could inspect the logs to uncover why it failed. This helped tremendously with product quality and we uncovered several bugs that only occurred on a full build from scratch.

7.3 Code Review

While developing we used the GitFlow workflow, a branching model for Git that is well-suited for collaboration in a team. This accommodated us well and was easily integrated into another requirement we had while checking in code: code reviews.

Each feature branch, bugfix branch or hotfix branch could never be checked into the main branch without first being reviewed by at least one other team member. There were two benefits for this. First, it established a baseline for code quality, since no one wanted to accept poorly written code into the main branch. Secondly, it increased the collective knowledge of the system within the group.

8 User Documentation

8.1 Installation Deployment

To use the system, there are two approaches. Either you are invited to a running instance of MALTE, or you can host an instance of MALTE yourself. If you are invited to a running instance of MALTE, all you have to do is visit the URL of the server, sign-in with GitHub, and start editing files.

If you wish to host an instance of MALTE yourself, the preferred way is to use the provided docker image, malte-uu@hub.docker.com. Pay attention to the environment variables: these need to be configured for the system to work properly. An example command to run the app yourself would be

```
docker run -p 4000:4000 malte-uu:dev
```

and then visit the url *localhost:4000*. Alternatively, since the system is open source, you can access and build it from the source code directly, accessible at the GitHub repository rehnarama/MALTE.

8.2 Configuration

The following environment variables exists and may be changed whenever needed. Backend:

- `MONGODB_URI`: Url to MongoDB (default: mongodb://localhost:27017/malte)
- `REACT_APP_BACKEND_URL`: (default: http://localhost:4000). The backend URL. All requests and WebSocket connections will be established with this URL. e.g. `REACT_APP_BACKEND_URL`=http://192.168.124.5:4000. N.B. that if no protocol is defined, http is assumed by most browsers.
- `REACT_APP_FRONTEND_URL`: (default: http://localhost:3000).
  E.g. REACT_APP_FRONTEND_URL=http://192.168.124.5:3000. N.B. that if no protocol is defined, http is assumed by most browsers.

- `GH_CLIENT_ID`: (no default). The GitHub OAuth app client id. This can be generated at GitHub
- `GH_CLIENT_SECRET`: (no default). The GitHub OAuth app client secret. This can be generated at GitHub
- `PROJECT_DIRECTORY` - name of the workspace. Will create folder in `/home/$PROJECT_USERNAME/$PROJECT_USERNAME`. Will use random folder in `/tmp` (and Windows equivalent) if this is empty.

Frontend:

- `REACT_APP_BACKEND_URL`: (default: http://localhost:4000 during development). The backend URL. All requests and WebSocket connections will be established with this URL. e.g. `REACT_APP_BACKEND_URL=http://192.168.124.5:4000`

8.3 First-Time Setup

MALTE doesn't allow any user to login to a project. The first time someone authenticate, anyone will be approved and stored as an authenticated user in the database. Subsequently, any user that tries to authenticate after that must be either defined in the user collection or in the list of approved user. One can add approved users and remove approved users from the project by pressing the plus avatar in the upper right-hand corner.

9 Project Management

During the development of MALTE we used an agile project management methodology inspired by both SCRUM and Kanban. At the beginning of the project we created a backlog of all the cards needed to fully implement the system (including nice-to-haves items). At the beginning of the project we divided work into discrete sprints, just as in SCRUM, but visualised our work with a board similar to a Kanban board, seen in Figure 6. The tool we used for our work board was GitHub's projects functionality. This turned out to work well due to the integration with GitHub issues and pull requests, which made work easy to visualise and follow; if a change was made we could always track why the change was made.

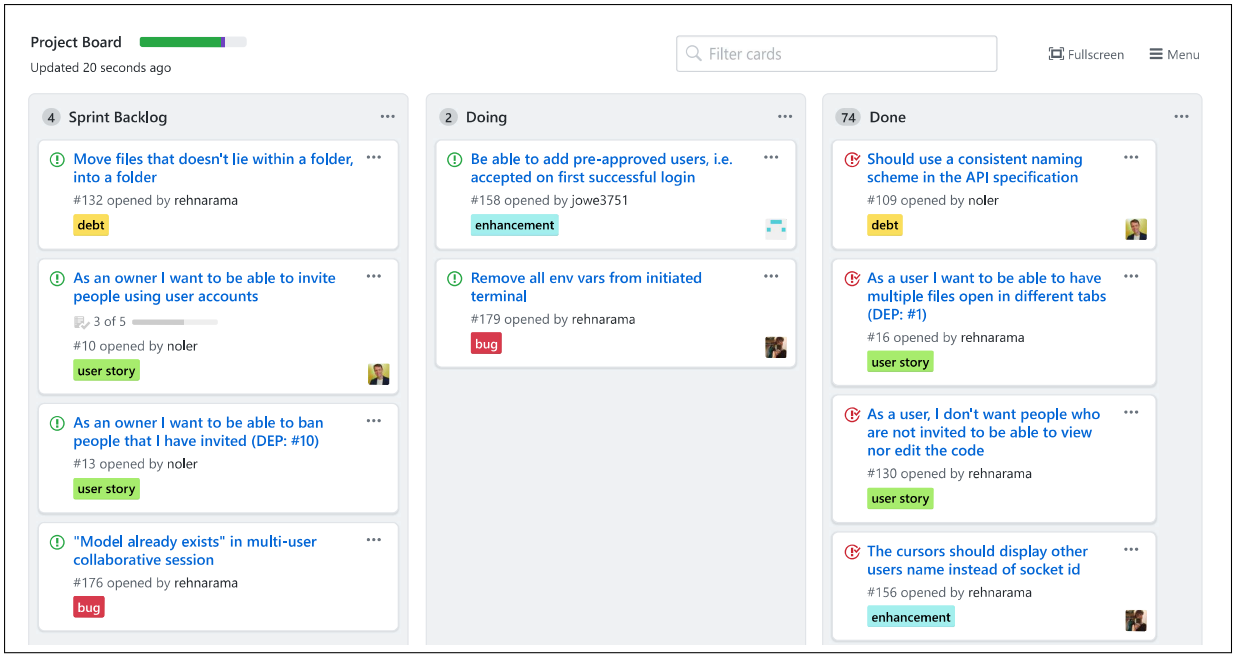

Figure 6: Example of the Kanban board used for the project for each sprint

One member of the team was working remote during all of the project which made communication key for the group work to function well. All group communication was handled via Slack, where we also had digital stand-up meetings (in true SCRUM spirit!) to share what we had worked on, what we will work on, and if we had any blockers. These "meetings" were held either via group calls in Slack, or by having each member summarize their work and plan for the day in a chat message.

In SCRUM we should deliver a functional project increment after each sprint. We do this by publishing a GitHub release after each sprint. When publishing a GitHub release, the continuous deployment process will start to build the app for release and a docker image will be automatically published at Docker Hub.

After each sprint we held a sprint review where we discussed what had gone well during the sprint and what could be improved. During one of these review meetings we discussed that the burden of reviewing pull-requests on GitHub was uneven, leading to a more equal workload regarding pull-requests the following sprint.

10 Future Work

10.1 Software as a Service with Meta Server

At the moment, each project is self-contained and must be deployed manually by anyone who wants to our system. If a user is a part of several different teams with different groups, as often is the case in a university environment, the projects would have to be set up individually and there would be no connection between them. This would require the user to remember or bookmark each project instance to be able to return to these projects. Users would also have to authenticate themselves each time they join a new project, even though they have already authorized the application to use their GitHub credential before.

A solution to these problems could be the introduction of a Meta server. This Meta server could be seen as a hub where a user would access all of their projects from. A user would authenticate themselves once to this Meta server, and the server would keep track of which projects the user has created, joined or been invited to. This kind of server would separate out all logic regarding user management from the project servers into one central location where an owner of a project could invite and ban users.

Taking this idea one step further, the Meta server could itself be responsible for deploying new project instances, instead of only keeping track of existing ones. This would simplify the creation of a project server by removing the need for users to themselves deploy the projects. To have a single Meta server deploy the project servers would mean that the resources to run the projects would also be centralized, leading to costs for the Meta server hosts. This could be mitigated by allowing organizations to host their own Meta servers that deploy project servers locally.

10.2 Collaborative Editing Improvements

The structure used for collaborative editing, RGA, explained in subsection 6.2 has a constant time complexity for downstream operations while upstream operations have a linear time complexity, in relation to document size. This can lead to unresponsive editing which affects the user experience negatively.

Improvements to the RGA structure are possible. For example, some operations – e.g. pasting – inserts a large chunk of text directly. Instead of inserting these as separate nodes, we can insert a large chunk of text in the same node; this is called a block-wise RGA [3]. While this would not improve the algorithmic time complexity of the structure, real life performance would improve due to less overhead.

To improve the upstream time complexity of the RGA, an auxiliary tree structure can be used [3]. The tree would improve look-ups in the tree from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$, assuming it is balanced, thus improving the upstream complexity.

10.3 Research Scalability

During development we have only tested the system locally with a limited amount of users, often no more than four concurrent users. It would be interesting to benchmark the system to see how performance

would scale to a large amount of users.

10.4 Chat

To further enhance the user experience a chat functionality could be introduced, accessible in the same browser window to enhance collaboration without the need for users to be situated at the same physical location. As the project server could be used to enable users with limited hardware resources on the client side it is preferable to provide this collaboration tool running on the server in order to mitigate the need for additional software installed on the user device.

10.5 Comments

If the server is used working remotely the ability to give feedback on others code as well as properly explain your own work without bloating the code with comments would be a nice feature to incorporate. In order to provide this to the users the project server could offer the possibility to highlight code and add inline comments, only visible on the project server when clicked upon.

10.6 Debugging

A core feature of many IDEs is debugging capabilities. This could also be a feature developed for MALTE. As of now any debugging done on the project server would be made through the terminal with, for example, gdb. Developing an interactive debugger capable of debugging multiple languages was deemed to be out of the scope of this project considering the time allocated.

10.7 Shell Access

Each connected user should have its own user on the host and that user should keep a synchronized copy of the project folder, such that each user can compile the project concurrently. One naive solution to ensure that changes in code from other users during compile won't interfere would be to temporarily pause the synchronization of the project.

10.8 Compile and Run button

To mitigate the need for using terminal commands in order to compile and run the code it would be possible to extend the functionality of MALTE to incorporate buttons for doing so. The functionality of the buttons could be extended to actively consult the current open file to make smart decisions about what compiler to use and format the generated commands accordingly, before executing them in the terminal.

10.9 Backup of the project files

As the file system is located solely on the server this introduces the risk of losing all source code in the case of a server crash with no backup performed. As MALTE offers the use of a terminal with git installed on the backend it is possible to clone a git repository to the project server and hence push and pull changes to another server. But as all users share the same files on MALTE this reduces the possibility to allow for example branching in git.

To provide the possibility to extract code without the use of external software solutions a future improvement of MALTE would be to offer the possibility to download the complete file system in the form of a zip file, this would serve as a possibility to backup the project without the risks of replicating your product on an external server.

References

[1] J. Bauwens and E. Gonzalez Boix, "Memory efficient crdts in dynamic environments," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM, 2019, pp. 48–57.

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[3] L. Briot, P. Urso, and M. Shapiro, "High responsiveness for group editing crdts," in *Proceedings of the 19th International Conference on Supporting Group Work*. ACM, 2016, pp. 51–60.

[4] "An open source collaborative code and text editor." [Online]. Available: https://firepad.io/

[5] "Developing with Visual Studio Online," Apr 2016. [Online]. Available: https://code.visualstudio.com/docs/remote/vsonline

[6] Repl.it, "The World's Leading Online Coding Platform." [Online]. Available: https://repl.it/

[7] "Operational Transformation," Oct 2019. [Online]. Available: https://en.wikipedia.org/wiki/Operational_transformation

[8] "Conflict-Free Replicated Data Type," Nov 2019. [Online]. Available: https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

[9] "How Figma's Multiplayer Technology Works." [Online]. Available: https://www.figma.com/blog/how-figmas-multiplayer-technology-works/

Appendix

Contents

A API Specification

We utilise socket.io as a framework for WebSocket communication. Socket.io is event-driven. All custom defined events is below

You can listen on events with `socket.on('event/name', (data) => {...})` and you can emit events with `socket.emit('event/name', data)`. More information about socket.io can be found in their documentation.

The API is organised into use-case on the form `namespace[/sub-namespace]/action`, where `sub-namespace` is optional. For example, all buffer related APIs are under the namespace `buffer` and to open a buffer you would emit `buffer/open`.

A.1 buffer (WebSocket API)

A.1.1 buffer/open

Direction: Server `->` Client

The server emits this event when a new buffer has been opened, most likely as a request from a previous `join-buffer` event.

This event contains the initial buffer data which will be a serialized version of the RGA structure as seen in packages/rga/src/RGA.ts.

A.1.2 buffer/join

Direction: Server `<-` Client

The client can emit this event to request to open a buffer. The data sent should be the full path of the file to load into the buffer, e.g.:

```
{
  "path": "/tmp/hello-world.c"
}
```

If the server currently does not hold the buffer in memory, it will be loaded into memory before being transmitted to the client via the `open-buffer` event.

A.1.3 buffer/leave

Direction: Server `<-` Client

The client can emit this event to request to leave the buffer, i.e. to not receive any further update about this buffer. The data should contain the absolute path of the file open inside the buffer that the client want to close, e.g.:

```
{
  "path": "/tmp/hello-world.c"
}
```

A.1.4 buffer/operation

Direction: Server `<->` Client

Both the server and the client can transmit this event to notify an update to a buffer. The data should contain both the absolut path of the file open inside the buffer as well as the operation to apply, i.e.:

```
{
  "path": string;
  "operation": RGAInsert | RGARemove
}
```

The type definitions for `RGAInsert` and `RGARemove` is found in packages/rga/src/RGAInsert.ts and packages/rga/src/RGARemove.ts.

A.2 connection (HTTP API)

A.2.1 GET /oauth/github/auth (internal)

Causes a redirect to the GitHub website for authentication. Should not be used as an API call but rather visited in the browser.

A.2.2 GET /oauth/github/callback (internal)

The callback url for GitHub OAuth authentication. Should not be used as an API call nor visited in the browser directly, only as a result of a redirect from the GitHub website.

A.2.3 GET /oauth/github/user

An authenticated user, i.e., a user who has completed the log in with GitHub process by visiting `/oauth/auth` can fetch their current GitHub user data by making a `GET` request to this endpoint.

A.3 connection (WebSocket API)

A.3.1 connection/auth

Direction: Server `<-` Client

Client requests to authenticate this WebSocket connection. Data should be of type `string` and should contain the user id that resides in the `userId` cookie after authentication. See HTTP API.

A.3.2 connection/auth-fail

Direction: Server `->` Client

The server sends this acknowledgement to the client once the WebSocket connection has failed to be authenticated after a `connection/auth` event.

No data is sent with this event.

A.3.3 connection/auth-confirm

Direction: Server -> Client

The server sends this acknowledgement to the client once the WebSocket connection has successfully been authenticated after a `connection/auth` event.

No data is sent with this event.

A.3.4 connection/signout

Direction: Client -> Server then Server → Client

The server sends request to be signed out with the userId. Server responds with informing client that it has now been signed out on the same event.

A.4 cursor (WebSocket API)

A.4.1 cursor/move

Direction: Server <- Client

Sent from client to the server to notify about a new cursor update. Data should be on the following form:

```
{
  path: string;
  id: RGAIdentifier;
}
```

A.4.2 cursor/list

Direction: Server -> Client

Sent from server to the client to notify about new cursor movements

```
Array<{
  path: string;
  id: RGAIdentifier;
  socketId: string;
  login: string;
}>
```

A.5 file (WebSocket API)

A.5.1 file/tree

Direction: Server -> Client

Transmits the file tree of the current workspace root as a JSON object.

Up-to-date type definitions of the structure is found here: packages/malte-common/src/TreeNode.ts Example structure:

```
{
  "name": "9w0kM5",
  "path": "/tmp/9w0kM5",
  "type": "directory",
  "children": [
    { "name": "a", "path": "/tmp/9w0kM5/a", "type": "file" },
    { "name": "b", "path": "/tmp/9w0kM5/b", "type": "file" },
    {
      "name": "c",
      "path": "/tmp/9w0kM5/c",
```

```
      "type": "directory",
      "children": [
        { "name": "d", "path": "/tmp/9w0kM5/c/d", "type": "file" },
        { "name": "e", "path": "/tmp/9w0kM5/c/e", "type": "file" },
      ]
    }
  ]
}
```

A.5.2 file/tree-refresh

Direction: Server `<-` Client

The client can send this request to request a refresh of the file-tree. The response, if any, will be send over the `file-tree` event.

No data can be sent with this event.

A.5.3 file/operation

Direction: Server `<-` Client

The client can send this request to do an operation in the file system. Data must be sent with this event specifying the operation.

```
{
    "operation": OP,
    "dir": PATH,
    "name": FILENAME
}
```

where `OP` is one of the following: `mkdir`, `rm`, `touch`, `mv`. all keys are always required, except when specifying the root directory one can ommit the `dir` key.

Examples:

```
{
    "operation": Operation.touch,
    "name": "hello_world.c"
}
```

Creates `hello_world.c` in the root folder of the workspace. If root directory, one may ommit the `dir` key.

```
{
    "operation": Operation.mkdir,
    "dir": "/existing_cool_folder/cool_sub_folder",
    "name": "my_new_folder"
}
```

Creates `my_new_folder` in e.g. `/home/ubuntu/workspace/existing_cool_folder/cool_sub_folder`

Note: `Operation` is implemented as an enumerate type, therefore its acutal value will be an integer, but that's not something the developer have to think about as long they use the shared type defined in `malte-common`.

A.6 pty (WebSocket API)

A.6.1 pty/data

Direction: Server `<->` Client

The server emits the `pty-data` event when the output of the terminal changes. The data consist of a `string` which is the direct output of the current user shell. Terminal escape codes may occur.

A client can emit the `pty-data` event to write to the input of the current user shell. The data type should be `string`. Terminal escape codes may occur.

A.6.2 pty/resize

Direction: Server <- Client

The client can emit this event to request the underlying pty to resize. The data should container both the number of columns and rows of the pty, e.g.:

```
{
  "columns": 80,
  "rows": 60
}
```

A.7 user (WebSocket API)

A.7.1 user/list

Direction: Server -> Client

The server an emit this event to notify a client whenever the list of connected users has changed, e.g. if a user joins or disconnects. The data should look like the following

```
{
  "users": [
    "name": string;
    "avatarUrl": string;
    "id": number; // Usually GitHub user id
  ]
}
```

A.8 authorized (WebSocket API)

A.8.1 authorized/add

Direction: Server <- Client

The Client emits to the server that a user is to be added to the pre-approved list.

```
{
  "login": string;
}
```

A.8.2 authorized/remove

Direction: Server <- Client

The Client emits to the server that a user is to be removed from the pre-approved list.

```
{
  "login": string;
}
```

A.8.3 authorized/list

Direction: Server -> Client

The Server emits a list of pre-approved users, this is an event that should occur every time the collection of pre-approved users is updated or upon request from the client. The data emitted has the form of:

```
{
  ["login": string;]
}
```

A.8.4 authorized/refresh

Direction: Server <- Client

The Client requests the pre-approved user list after rendering the UI. The server then answers with an "authorized/list" reply to the requesting socket.