Mobile Modeling with Real-Time Collaboration Support

Max Härtwig

10.01.2022

Supervisor
Dr.-Ing. Sebastian Götz

Supervising professor
Prof. Dr. rer. nat. habil. Uwe Aßmann
Abstract

Modeling is an essential discipline that is especially important in the field of software engineering. Students and developers alike employ models to describe systems on an abstract level, capture requirements, and communicate with other teams. For that purpose, UML diagrams are usually the instrument of choice. Over the course of the last decade, mobile devices increased in prevalence and popularity and flexible work arrangements were introduced in a larger number of workplaces. Effective collaboration is more important than ever. However, the tools have not kept up with these developments. There exists no semantics-aware mobile modeling application that supports collaboration in real time, a gap in the market.

This thesis investigates existing applications in the mobile modeling space and their shortcomings, technologies for developing cross-platform apps, and methodologies for facilitating conflict-free collaboration. Based on the findings, it conceptualizes and implements CoMod, a proof of concept allowing users to collaboratively edit UML class diagrams in real time. The system consists of a Flutter-based client application for Android and iOS and a Node.js-based server executable. These components utilize conflict-free replicated data types (CRDTs) to merge participants’ changes and communicate via WebSocket connections. Moreover, CoMod’s feasibility is evaluated by means of a case study investigating the system’s scalability and performance characteristics.

It has been shown that CoMod is able to handle common use cases arising in software engineering teams or group projects at university. It is further kept sufficiently general to allow other types of models to be supported without having to alter the entire system.
Contents

1 Introduction 5
  1.1 Requirements .................................................. 6
  1.2 Problem Description ........................................... 6
  1.3 Objectives ..................................................... 7
  1.4 Structure ..................................................... 7

2 Background 8
  2.1 Software Modeling .............................................. 8
    2.1.1 Unified Modeling Language .................................. 8
  2.2 Cross-Platform Application Development ......................... 9
    2.2.1 Web apps .................................................. 9
    2.2.2 Hybrid apps .............................................. 10
    2.2.3 Native cross-platform apps ................................ 10
    2.2.4 Summary ................................................ 11
  2.3 Real-Time Collaboration ....................................... 11
    2.3.1 Conflict-Free Replicated Data Types (CRDTs) ............... 12
    2.3.2 Operational Transformation ............................... 13

3 Related Work 15
  3.1 Astah UML ................................................ 15
  3.2 Lucidchart .................................................. 16
  3.3 System Designer ............................................. 18
  3.4 Summary .................................................. 19

4 Concept 20
  4.1 Objectives ................................................ 20
  4.2 User Interface ................................................ 21
  4.3 Data Model ................................................ 22
  4.4 Collaboration ................................................ 23
    4.4.1 Conflict handling ...................................... 24
    4.4.2 System architecture ..................................... 25
    4.4.3 Client-server communication ............................ 26
  4.5 Summary ................................................ 27
5 Solution

5.1 Client
  5.1.1 Walkthrough
  5.1.2 Data model
  5.1.3 JavaScript subsystem
  5.1.4 Dependencies

5.2 Server
  5.2.1 Dependencies

5.3 Collaboration
  5.3.1 Client-server communication
  5.3.2 Client data flow

5.4 Testing
  5.4.1 Unit tests
  5.4.2 Integration tests
  5.4.3 End-to-end tests

5.5 Extensibility

6 Evaluation

6.1 Case Study

6.2 Technical Analysis
  6.2.1 Test data generator
  6.2.2 Client analysis
  6.2.3 Server analysis

6.3 Threats to Validity

7 Conclusion

7.1 Fulfillment of Objectives

7.2 Future Work

Acronyms

Bibliography
1 Introduction

Modeling is an essential discipline in many fields, such as engineering, natural sciences, and psychology [15]. Models describe or represent systems on an abstract level and can be used for discussions, analyses, and other purposes. They are particularly useful in software engineering and are employed throughout the industry. Common use cases include the description, documentation, and support of software systems and their development. Models capture requirements and aid in the communication between teams, thus increasing quality and productivity by reducing ambiguity and misunderstandings. With sufficient semantic information, some kinds of models can even be used to generate and verify code.

Traditionally, the process of modeling has taken place in an office setting on whiteboards, or digitally on laptops and desktop computers. However, mobile devices such as smartphones and tablet computers can also be used for modeling purposes, making it accessible to a larger group of users as more people have access to smartphones than laptops and desktop computers¹. Further, they are usually cheaper to buy and replace. Smartphones and tablets also benefit from the fact that they can be taken anywhere and used near the described problem domain or right where the model is needed. Mobility and context can be especially beneficial for modeling in certain problem domains [17]. Furthermore, mobile devices commonly have access to a wide range of sensors, such as camera and GPS, and may feature specialized input methods like styluses and multi-touch gestures than can result in an improved usability.

The aspect of collaboration has usually involved multiple people discussing changes while one person was carrying them out on a whiteboard or a shared screen. While being effective, this approach exhibits a couple of shortcomings. Collaborators have to be present in the same room or online meeting and cannot all edit the model at the same time. A better solution would be the adoption of tools that allow modeling to take place in an asynchronous and collaborative manner. This is especially important for companies with a rising number of distributed teams and colleagues working from home. This process is further accelerated by the COVID-19 pandemic and shifting work policies².

With the increasing prevalence of mobile devices and flexible work arrangements, mobile modeling applications could be valuable instruments for teams to collaborate and might even become essential tools in a world where remote work is the norm rather than the exception. Several mobile modeling applications are available today, but they usually lack vital features

that a viable collaborative mobile modeling solution should have (cf. section 1.1, chapter 3). Some general-purpose applications support real-time collaboration, but do not provide the means needed for modeling software systems.

1.1 Requirements

This section enumerates a set of requirements that viable mobile modeling applications should satisfy and explains why each one is essential. The preceding literature analysis revealed different criteria that were used by different authors. The following list was influenced by my experience as a mobile app developer and condenses these requirements into a concise set.

**Availability:** Mobile modeling apps are mainly used on mobile devices which usually run either Android or iOS. A good app should be available on both platforms to enable all members of a project to use it without requiring some of them to switch away from their preferred operating system or purchase a new device. Another advantage for an app is to be available on desktop platforms such as the web, macOS, or Windows as well. This enables team members who are at their desks to also take part in a modeling session while allowing them to make use of a larger monitor and additional peripherals such as mouse and keyboard.

**Semantics:** A model should not just be a drawing or a diagram. A good modeling app understands the model's contents and can discern its individual components. Operations performed on the model should always yield a valid result, e.g. a deleted class in a UML class diagram should not leave relationships without a target.

**Performance:** Modeling apps should be fast and responsive despite being limited by a mobile CPU. Users should not have to wait for basic actions and expensive work needs to be performed in the background to ensure that the user interface (UI) never feels sluggish.

**Usability:** Mobile devices often have relatively small screens and require apps to adapt their UI accordingly. Basic interactions should be discoverable, easy to perform, and require as few steps as possible. The aspect of accessibility also belongs to this category.

**Collaboration:** Modeling is usually performed by multiple members of a team collaborating with each other. Mobile modeling apps should enable them to do so by either allowing documents to be shared or, even better, worked on in real time.

**Offline support:** Enabling real-time collaboration is not a trivial task and usually requires an active internet connection. However, mobile devices often have to deal with unreliable cellular connections or no connection at all when wifi is not available (e.g. in the case of most tablets). Modeling apps should account for that and allow models to be edited offline as well.

1.2 Problem Description

No application available today satisfies all of the requirements for collaborative mobile modeling applications outlined in the previous section (cf. chapter 3). This represents a gap in the market that could be closed. Consequently, the main research question of this thesis is:

*Is it possible to build a mobile modeling application with real-time collaboration support?*
1.3 Objectives

The following objectives shall be accomplished:

O1 Create a concept for building an application that fulfills the requirements outlined in section 1.1.
O2 Implement this concept as part of a working solution.
O3 Evaluate the solution by revisiting the requirements and by means of a case study.

1.4 Structure

The remainder of this paper is organized as follows. First, chapter 2 explains some terms and concepts used throughout this paper. Then, chapter 3 examines existing mobile modeling applications and their properties with respect to the requirements outlined in section 1.1. Chapter 4 describes the concept of a new application satisfying these criteria without depending on specific platforms or frameworks. Subsequently, chapter 5 portrays the implementation of CoMod, a proof of concept based on the aforementioned concept. This is followed by chapter 6, an evaluation of the solution, its viability and scalability by means of a case study. Finally, chapter 7 summarizes the main findings and concludes this thesis.
2 Background

This section introduces and describes several concepts and terms that play an important role throughout this thesis.

2.1 Software Modeling

Before a software system can be implemented, it needs to be designed. This process includes defining the system’s overall structure and the behavior of the types, objects, and functions involved. Having such a specification enables teams to work in a coordinated fashion. Different members may implement separate parts of the model and the likelihood of mistakes due to ambiguity are reduced. Software models can be expressed in different ways, ranging from prose or pictures to formal modeling languages. The next section introduces one such language.

2.1.1 Unified Modeling Language

The Unified Modeling Language (UML) [12] is a standardized modeling language that is mostly used to visualize the design of software systems. After its original development at Rational Software in 1994, it was adopted by the Object Management Group (OMG) in 1997 and made a standard. The specification allows for many types of diagrams than can be used to represent different aspects of a system. Structural diagrams describe a system’s static architecture using objects, attributes, operations, and relationships while behavioral diagrams describe a system’s dynamic functionality with the help of sequences and state machines. Some types of diagrams may even be used to generate code.

UML is the de-facto standard for modeling systems in the field of software engineering and it is employed in both academia and industry [14]. Diagrams can be drawn by hand or utilizing specialized applications. Students of computer science usually learn it early in their studies and employ it as part of assignments. Professional software developers are also familiar with UML, though they often tend to use informal sketches that contain UML elements [1]. Reasons for that include the vast complexity of the UML specification and the quickly-changing nature of software systems.
UML class diagrams

The solution presented in chapter 5 mainly focuses on UML class diagrams, a type of structural diagram that describes an object-oriented system's architecture by outlining its classes, their attributes, operations, and relationships to each other. An example can be seen in fig. 2.1. Classes are represented by boxes with three distinct sections. The first shows its name and an optional label denoting it as an abstract class or interface. The second section contains the class's attributes, including their visibility and data type. The last section shows the class's operations, including their visibility, parameters, and return type. Class-level relationships such as generalization and realization are represented by arrows with solid or dashed lines. Instance-level relationships such as association, aggregation, and composition are represented by different types of lines and may have a label and multiplicity annotations.

2.2 Cross-Platform Application Development

Android and iOS are the two dominating mobile platforms\(^1\). Usually, applications for these two operating systems are developed natively using the Android or iOS Software Development Kit (SDK). However, creating and maintaining apps with separate codebases for multiple different platforms can be fairly costly and time-consuming. That is why the area of cross-platform development has emerged, intending to build applications for various platforms from a single codebase. This section presents a selection of popular cross-platform development approaches and touches on their advantages and shortcomings.

2.2.1 Web apps

When the original iPhone was launched in 2007\(^2\), it was announced that it would run Web 2.0 apps. The iOS SDK and the App Store were not launched until one year later in 2008. The first Android SDK was published around the same time. Since then, the number of apps in the

---


Apple App Store and the Google Play Store has been increasing dramatically. Despite the rise of native apps, web applications are still widely used today.

Web pages may either be static websites containing text, images, audio, video, and links, or web apps that feature interactive elements, are customizablae, and may perform a wide range of tasks. Web apps usually employ responsive web design, i.e. they adapt their content to the screen on which they are displayed. This is especially important for small screens such as those found in smartphones. Some web apps define a web app manifest and may be classified as Progressive Web Apps (PWAs). They can be added to the home screen and behave similarly to native apps by running in their own window, showing up in the device’s search function, and providing offline support. Using advanced browser APIs, they may even receive push notifications, store data locally, and run background tasks.

Web apps are usually written using the web standards HTML [6], CSS [2], and JavaScript (JS) [3]. This allows them to run on virtually any device with a standards-compliant browser without requiring a separate download. They are also built using the same tools and languages as standard web pages and do not require developers to get to know a new ecosystem. However, web apps usually run slower than their native peers, cannot access all of the device’s capabilities, and might feel out-of-place when not following the platform’s design language.

2.2.2 Hybrid apps

Hybrid applications are web apps wrapped in native containers using technologies such as Apache Cordova3 or Capacitor4. This allows them to be written using standard web technologies and access native device features via plugins. An embedded browser view renders the app’s content and handles user interaction. Specialized SDKs such as Ionic5 provide adaptive UI components that resemble the platform’s native UI elements.

Hybrid apps are cost-effective to develop as they are built using web technologies. Plugins allow them to take advantage of native device features that web apps do not have access to. Moreover, their ability to be distributed via app stores makes them more discoverable and provides companies with an additional marketing opportunity. Nevertheless, hybrid applications face the same performance and usability challenges as web apps.

2.2.3 Native cross-platform apps

The recent decade has brought forward a number of cross-platform app development frameworks that allow native applications to be built from a single codebase. Notable examples include Xamarin6, React Native7, and Flutter8. Developers can use a language they are familiar with and are not limited to the languages the native SDKs support. Apps developed with this approach usually ship with their own runtime to execute their code while rendering a native user interface on a separate thread. This also allows them to display native UI components that feel at home on their respective platform.

Native cross-platform applications do not rely on web technologies and can make use of the full capabilities the device has to offer. This generally leads to a better performance at

---

5https://ionicframework.com, accessed 20.10.2021
runtime and users appreciate the appearance of native UI components. Flutter even uses a custom rendering engine, allowing apps to deliver a consistent user experience across different platforms. However, cross-platform frameworks often receive new features with a delay of several weeks or months and it can be quite costly and time-consuming to make an app look and feel great on every platform.

2.2.4 Summary

With several different approaches to choose from, it is impossible to identify the best one for all use cases as the choice depends heavily on the individual project. Web apps are the most cost-efficient but usually fall short in terms of features, performance, and usability. Hybrid applications can make use of more device features, adapt better to smaller screens, and can be distributed via app stores. However, they may also face performance and usability issues. Native cross-platform apps can provide truly native experiences, but may be more costly and time-consuming to develop. Every project should evaluate these approaches according to its goals and choose the one which best suits them.

2.3 Real-Time Collaboration

Collaboration is an important aspect of many projects. It fosters creative ideas, more efficient problem solving, and brings people closer together. Teams collaborate on text documents, presentations, drawings, and source code, to name a few examples. With more work becoming digital, the selection of suitable tools is more important than ever. Popular collaboration applications such as Google Docs\(^9\), Figma\(^{10}\), and Visual Studio Live Share\(^{11}\) are used by millions of people for both professional and personal projects. However, building such collaborative applications is not an easy feat. This section discusses two technological approaches that have been the subjects of research for decades and which are used to enable real-time collaboration in applications today.

A real-time collaboration system has to be able to accommodate multiple participants, each with their own device. The devices should all be connected with each other (either directly or via a server) and exchange messages. Participants should be able to start and join collaboration sessions and see each other's edits in real time. For the sake of simplicity, this section assumes a plain text document is being edited. Nevertheless, most of the following concepts should apply to non-textual and nested documents as well. All participants should be able to edit the document at the same time without anyone's document becoming locked while it is being edited by someone else. The client applications exchange messages and their documents should converge toward the same state (eventual consistency). In other words, participants' documents should have the same contents when the system is at rest, i.e. when there are no more messages to be exchanged. Any arising conflicts should be handled by the system without requiring manual intervention by any participant. Ideally, the client applications should also be able to tolerate temporary disconnects from the network and provide the option to edit a document without being connected at all (offline). An optional feature is to see other participants' cursors and selections and not just their edits/keystrokes.

\(^9\)https://docs.google.com, accessed 15.10.2021
\(^{10}\)https://www.figma.com, accessed 15.10.2021
\(^{11}\)https://visualstudio.microsoft.com/services/live-share/, accessed: 15.10.2021
A naive algorithm would be to send document versions back and forth, compare them, and try to merge their differences. It is quite easy to envision scenarios in which this approach proves to be inadequate. For example, a document consisting of the string ABC. One participant makes BC bold (ABC), while another replaces the C with a D (ABD). The correctly merged result would be ABD, but the naive algorithm does not have sufficient information to produce that outcome. It would produce ABCD, ABD, or ABDC instead, all of which are results that participants did not expect. Evidently, a better approach to facilitate real-time collaboration between multiple parties needs to be employed.

Over the course of the last decade, different approaches have emerged to solve the problem of eventual consistency. The following sections introduce the approach of Conflict-Free Replicated Data Types that is used as part of the concept outlined in the next chapter, and briefly mentions Operational Transformation as a commonly employed alternative.

2.3.1 Conflict-Free Replicated Data Types

Conflict-Free Replicated Data Types (CRDTs) are data structures that can be distributed to multiple clients in a network, modified independently and concurrently, and that can always be merged without conflicts. They started appearing around 2006 but were not formally specified until 2011 [16]. Their main task is the facilitation of optimistic replication\(^\text{12}\), i.e. all changes are applied concurrently without locking or waiting for other clients; the different replicas are eventually merged with any inconsistencies being resolved mathematically. A number of different CRDTs have been developed, including positive-negative counters, last-write-wins sets, and observed-remove sets. For the purpose of real-time collaboration, sequence-based CRDTs bear the most potential. Developing a CRDT is complex, but fortunately, a number of open-source CRDT frameworks, such as Automerge\(^\text{13}\) [7] and Yjs\(^\text{14}\) [10, 11] are readily available. The following sections describe the way CRDTs are structured and how they resolve inconsistencies.

**Structure**

Text-based CRDTs operate on the per-character level; every character is treated as a separate entity with a unique identifier and timestamp. These entities form a structure resembling a linked list; characters can be inserted or deleted at any point. If two characters are inserted concurrently at the same position, the structure devolves into a tree-like composition as both new characters point to the same predecessor. A deterministic algorithm decides which one comes first. The text to display is reconstructed by performing a depth-first traversal of this tree-like structure and applying a conflict resolution approach such as last writer wins (LWW) wherever necessary. This also helps to enforce causal consistency: if a client receives characters that reference other characters that it does not yet know of, they can be queued until the missing data is received. Deleted characters are not removed from the structure immediately. Instead, they become so-called tombstones that are hidden from the text. This allows their position to be referenced by clients that have not yet been informed of their deletion.

One concern is that CRDTs grow larger in size every time they are modified, even when characters are deleted. This concern is valid, however, tombstones may be garbage-collected when the system can ensure that all clients have received a certain update and can be considered stable.


\(^{13}\)https://github.com/automerge/automerge, accessed 15.10.2021

\(^{14}\)https://github.com/yjs/yjs, accessed 15.11.2021
Vector clocks are used to determine the concurrency of updates. Another shortcoming is the preservation of user intent. Every user action is broken down into a number of modifications to the CRDT structure and it may be hard to reconstruct which action/intent a modification was caused by. Clients' replicas are guaranteed to converge, but different independent low-level changes might cause an underlying schema (such as JSON or XML) to become invalid.

Types of CRDTs

CRDTs can be classified as either state-based convergent replicated data types (CvRDTs) or operation-based commutative replicated data types (CmRDTs). Both types are equally expressive and one may be represented by means of the other. The choice of which type to use depends on several factors.

CvRDTs propagate their entire state to other clients. They receive it and merge it with their local state using a commutative, associative, and idempotent function. Shapiro et al. showed that if all possible state values form a semilattice\(^{15}\) and updates are monotonically increasing, replicas are guaranteed to converge. Overall, CvRDTs are simpler to design than CmRDTs and do not demand special requirements from the network layer. One downside is that the CRDT's entire state has to be transmitted, hence more data has to be sent back and forth. However, there are optimized delta state CRDTs which only transmit recent changes.

CmRDTs, on the other hand, only transmit and apply update operations. Therefore, they usually require less bandwidth than CvRDTs. The operations are commutative but not guaranteed to be idempotent. Because of this limitation, the network layer needs to ensure that sent operations are not duplicated or dropped.

CRDT system architecture

There are different ways a CRDT system can be designed. The fact that they do not rely on an active connection to a central server allows for peer-to-peer networks. Whenever a client has new changes to report, they have to be sent to all other clients in the network. This can be done via a broadcast or gossip protocol\(^{16}\). Without a server that needs to process updates, all data may be end-to-end encrypted and sent over an insecure network in which only eligible clients have access to the decryption key. Nevertheless, a server may be employed to distribute messages between clients.

2.3.2 Operational Transformation

The Operational Transformation (OT) technology was first described by C. Ellis and S. Gibbs [4]. Later, some correctness issues were discovered, leading more researchers to focus on this area and propose improved concepts. However, only two forms could be proven to be correct: server-based OT [9, 18] and OT with TP2 [13]. Server-based OT requires each client to be connected to a server that coordinates changes from all parties and can act as a source of truth. Each client's changes only have to be compared to the server's version, hence avoiding the complexity of three-way merges. OT with TP2 on the other hand refers to the so-called transform property 2. Data structures having that property can achieve convergence when merging changes from three different clients. Because of this, clients can be directly connected


in a peer-to-peer network without requiring a central server. However, correct multi-way transformation algorithms are very complex and only a few data structures have reliable TP2 implementations.

A document making use of OT can be described as a sequence of operations. Possible operation types for a rich text document include insert text, delete text, and apply style. Each application needs to define its own set of operation types that can be used to describe the documents it processes. The current state of a document can be restored by replaying its describing operations. Each user edit is expressed as one or more operations; e.g. replacing a string with another string might be expressed by a Delete followed by an Insert. These operations are subsequently sent to all other participants to be incorporated into their documents. OT’s main task is ensuring that each pair of concurrent operations produces the same outcome, regardless of the order of application. For that purpose, a transformation function $T$ is used to modify an operation $op_i$ in such a way that it includes the changes of another operation $op_j$. Above requirement can be described using the equivalency $op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$ where $op_1 \circ op_j$ denotes the application of $op_i$ before the application of $op_j$. $T$ has to produce a correct result for each pair of operations (and operation types).
3 Related Work

This chapter examines a number of existing mobile modeling applications that were available to download as of October 2021. The selection includes solutions allowing the design of models that can be represented as UML class diagrams. These solutions are evaluated with respect to the requirements outlined in section 1.1.

3.1 Astah UML

Astah UML¹ (formerly JUDE) is a UML modeling software that was initially released in 1999. Its iPad version supports the creation of UML class diagrams by allowing text, notes, classes, and interfaces to be placed on a canvas. These objects can be color-coded and connected using lines and arrows. Details such as the type's name, attributes, and operations can be edited via an overlay that appears when needed. The desktop versions offer more features and also allow other types of UML diagrams to be created.

**Availability**: The mobile version of Astah is only available on iPad (named *Astah UML pad*). The desktop version comes in two variants, *Astah UML* and *Astah professional*, and runs on Windows, macOS, and Linux.

**Semantics**: Astah recognizes the individual parts of class diagrams. The desktop version even shows an outline of all the components in the model. Documents can be exported as XMI files to be processed by other applications. Code can be imported from and exported to many languages.

**Performance**: Both the mobile and desktop versions are fast and responsive.

**Usability**: The limited feature set of the mobile version makes using the app rather simple. Most actions are performed using menus that appear when needed. A bar above the keyboard offers quick access to special characters and the names of data types. The desktop version goes a step further and provides affordances to quickly add a property or operation and even suggests targets when beginning to draw a connection.

**Collaboration**: Astah lacks support for collaboration. Files can be sent to team members, but they cannot be edited collaboratively.

¹https://astah.net/products/, accessed 23.10.2021
Figure 3.1: Astah UML pad running on iPad Pro.

**Offline support**: Projects are created locally by default and do not require an active internet connection.

### 3.2 Lucidchart

Lucidchart\(^2\) is a proprietary diagramming application that focuses on sharing and collaboration. It was initially released in 2008. At its core is a canvas upon which different shapes can be placed. Basic shapes include boxes, arrows, and text. An extensive library offers access to more shapes, such as AWS icons, flowcharts, tables, and UML diagrams. All of them can be mixed and matched and placed on the same canvas. Shapes can be formatted and connected with each other using lines or arrows. Many shapes have dedicated areas for text, e.g. the *class* shape accepts text entries for the class’s name, its attributes, and its operations.

**Availability**: Lucidchart offers native apps for Android and iOS and is accessible via a web browser on desktop computers.

**Semantics**: A major drawback is the absence of semantics. Documents are treated as collections of shapes and are unaware of the semantics of their contents. They can be exported as PDF files or images, but not in a format that would allow further processing (e.g. JSON or XML).

\(^2\)https://www.lucidchart.com/pages/, accessed 23.10.2021
**Performance**: The experience is quite snappy on iOS and does not require the user to wait for any action to complete. However, the version running in the browser might feel somewhat slower than comparable native applications.

**Usability**: The app is fairly usable and it is easy to get up to speed. The number of presented options is daunting, but not overwhelming. Lucidchart also does a good job adapting to the different screen sizes: in the browser, the editing tools are located in a toolbar (similar to typical word processors) and the shape library can be found in a sidebar; on tablets, all of the controls are located in the sidebar; and on phones, the tools can be pulled in from the bottom and only cover half of the screen to leave the user with their context. However, there might be usability differences between the platforms as the reviews for the iOS App Store (avg. 4.7 stars) and the Play Store (avg. 3.0 stars) differ by a sizable margin.

**Collaboration**: This is the area in which Lucidchart truly stands out. Other users can be invited to collaborate on documents and any edits appear on others’ devices in real time. Contributors can leave comments and associate them with a certain shape to start a discussion. The UI keeps track of this and hides comments as soon as they have been resolved.

**Offline support**: Both mobile apps allow documents to be created and edited while being disconnected from the internet. However, the offline editing feature is still in beta for the web version.

---

2 as of 23.10.2021, US region
3.3 System Designer

System Designer\(^4\) is a free mobile IDE for designing JavaScript systems developed by Erwan Carriou. It allows the design of entire systems, including model, behaviors, instances, and logging. This is done via the Metamodel JavaScript Object Notation (MSON)\(^5\) which can be edited directly. During the process, the individual components of the system are visualized within the app. The model tab (see fig. 3.3) resembles a UML diagram, though it cannot be manipulated directly. After a system has been designed, it can be run directly in System Designer and even shows logging output.

![Figure 3.3: System Designer running on iPad Pro.](image)

**Availability**: Native apps for Android and iOS are available for download. A PWA for desktop computers can be used as well.

**Semantics**: The specification via MSON enables the application to be aware of every system component on a structural and semantic level. A system can be exported or imported as JSON. Additionally, a system can be exported as JavaScript code to be embedded in a web application or Node.js\(^6\) code that can be run independently.

**Performance**: Both the native and web versions of the app are performant and immediately respond to user interaction.

---

\(^4\)https://designfirst.io/systemdesigner, accessed 23.10.2021
\(^6\)https://nodejs.org/en/, accessed 12/07/2021
**Usability:** The user interface has a clear structure. Editing system components using the built-in text editor might feel tedious sometimes, but it is fast and features autocompletion. The extensive documentation helps to learn the syntax, but it cannot be opened within the app. Unfortunately, the mobile version does not show button tooltips in contrast to the web version used on a desktop.

**Collaboration:** System Designer does not offer dedicated collaboration features. However, a project can be synchronized via a GitHub repository. This allows an asynchronous workflow, but the synchronization has to be triggered manually and conflicts might arise.

**Offline support:** None of the application variants requires an active internet connection.

### 3.4 Summary

The solutions presented above are merely a small subset of the modeling applications available today. However, they show the general situation software engineers and students who want to create models using their mobile devices face. Applications like Lucidchart allow the collaborative creation of nice-looking UML diagrams, but do not understand the models’ semantics. Hence, models cannot be validated or exported for code generation or further processing. Other applications such as Astah UML pad can be used to design proper UML models, but they are only available on a small number of platforms or lack certain features developers rely on. Specialized solutions such as System Designer are very capable but focus on a specific area, e.g. designing JavaScript systems, and are unsuitable for general-purpose modeling.

In summary, none of the solutions presented in this chapter nor other available applications at the time satisfy all of the requirements described in section 1.1. It is a common scenario that an app performs well in certain categories, but falls short in others or lacks support for one entirely. This insight identifies a gap in the solutions available today: there exists no mobile modeling application that is available on most mobile platforms, allows the creation of UML class diagrams, understands a model's semantics, is fast and easy to use, does not require an active internet connection at all times, and supports real-time collaboration.
4 Concept

This chapter describes a concept for an application that satisfies all of the requirements outlined in section 1.1 without concretizing specific frameworks or technologies to be used. This allows it to be implemented using different tech stacks. The choice of which one to use depends on which platforms are to be supported and which aspects of the solution are to be prioritized.

The field of mobile modeling is extensive and impossible to exhaustively cover with a single application. For the purpose of this thesis, conceptualizing and implementing a proof of concept, the domain of UML class diagrams (cf. section 2.1.1) has been chosen.

4.1 Objectives

This concept is designed to fulfill the following list of objectives. These objectives build on the requirements mentioned in the first chapter and extend them by specifying a list of features the application should support.

**UML class diagrams:** The app should allow the creation and manipulation of UML class diagrams, including types (i.e. classes, abstract classes, interfaces) and relationships (i.e. associations, aggregations, compositions, associations with associated classes, qualified associations). Types should have a name and be able to specify their supertypes, properties, and operations, including their visibility and data type/return type. Relationships should be able to specify the types involved, their multiplicity, and a label (if applicable).

**Semantics:** The app should retain the model’s semantics, i.e. it needs to be able to discern the model’s individual components and check their validity. Another responsibility is ensuring the consistency of the model while it is being modified. E.g. deleting a type should also delete its relationships and not leave them one-sided.

**Availability:** The app should run on multiple platforms. As this is mainly targeted at mobile platforms, Android and iOS are the prime candidates. Targeting the web is desirable, but remains optional for this proof of concept.

**Collaboration:** The app should allow multiple users to collaboratively work on the same model. Modifications need to be non-locking and able to be merged without conflicts that require manual resolution. Users need to be able to edit the model concurrently without having to wait for others. This may cause their local replica to diverge temporarily. To cope with this fact, the
app has to implement eventual consistency, i.e. all replicas need to be in the same state once all pending modifications have been transmitted and processed.

**Performance:** The app needs to be fast and responsive. Mobile devices often lack the most powerful processors due to their small size and thermal constraints. This needs to be accounted for. Expensive computations need to be performed in the background to ensure that the user interface remains responsive at all times. When collaborating, a low latency is desirable. Modifications from other connected participants should appear in real time.

**Usability:** The app needs to be usable on smartphone-sized screens. Text should be readable and UI elements have to be large enough to be touched without accidentally triggering an adjacent one. All features have to be accessible via touch without requiring a mouse or a keyboard to be connected.

**Offline support:** The app has to be usable without an active internet connection. This includes the modification of shared models. Changes during that time need to be cached and transmitted once an internet connection is established again. Unreliable connections have to be tolerated without changes being dropped.

### 4.2 User Interface

The centerpiece of every application is its user interface. It is responsible for visualizing the app’s data and providing the means for data manipulation. In order to supply the features outlined in the first objective of the previous section, the app needs at least four different views:

**Models:** A list of the different models the app has stored locally. It should also allow new models to be created (e.g. with a plus button) and existing models to be deleted. Selecting a model leads to the Model Details view.

**Model Details:** Shows the selected model’s contents, i.e. its types and their relationships. This could be a simple list or even a visual representation. From this view, users should have the option to add, select, or delete types. Selecting a type leads to the Type Details view; a back button leads back to the Models view.

**Type Details:** Shows the details of a type, including its name, type (class, interface, etc.), super-types, attributes, operations, and relationships. All these properties need to be editable from this view. A back button leads back to the Model Details view.

**Collaboration:** Allows users to start or join collaboration sessions. Starting a collaboration session should be possible after having selected a model to edit while joining a remote session should be possible from the Models view.

Fig. 4.1 depicts what above views might look like when implemented as dedicated screens and how a user might navigate between them. However, these views do not necessarily have to be presented on their own screens. It might also be possible to combine multiple views on a single screen, especially on larger devices such as tablets.
4.3 Data Model

An internal data model is required for the app to keep track of the model's contents at runtime. A UML class diagram can be interpreted as a hierarchy, thus a tree-like structure would be suitable. The tree's root is the UML model itself which in turn contains UML types and relationships. Types contain attributes and operations. Operations contain parameters. As a consequence of that structure, the model consists of heterogeneous nodes that, depending on their own type, can only contain certain types of child nodes (cf. fig. 4.2). Moreover, each type of node may only contain a certain set of properties, e.g. an operation has a name, a visibility annotation, and a return type. Table 4.1 lists all entities along with their properties and possible children types.

Some relations between different nodes, i.e. attributes based on types in the model and the types involved in a relationship, cannot be represented by a strictly tree-based structure. Both attributes and relationships reference types based on their unique IDs. This approach can be interpreted as an overlay graph (cf. dotted lines in fig. 4.2). This is important for the model's consistency, which must be maintained at all times. When a type is deleted, all referencing attributes' data types are changed to string and referencing relationships are deleted.

The data model has to be persisted to disk regularly to preserve its contents across launches of the app. One option is saving all entities as rows in a database with parent-child relations modeled using foreign keys. Another option is to serialize the data model to a binary or text-based format and write it to a file.
Figure 4.2: The tree-like structure of the app’s data model. The dotted lines indicate the overlay graph.

<table>
<thead>
<tr>
<th>Component</th>
<th>Properties</th>
<th>Possible children types</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Model</td>
<td>ID</td>
<td>UML Type, UML Relationship</td>
</tr>
<tr>
<td>UML Type</td>
<td>ID, Name*, Type (class, abstract class, interface)</td>
<td>UML Attribute, UML Operation</td>
</tr>
<tr>
<td>UML Attribute</td>
<td>Name*, Visibility, Data Type</td>
<td></td>
</tr>
<tr>
<td>UML Operation</td>
<td>Name*, Visibility, Return Type</td>
<td>UML Operation Parameter</td>
</tr>
<tr>
<td>UML Operation Parameter</td>
<td>Name*, Data Type</td>
<td></td>
</tr>
<tr>
<td>UML Relationship</td>
<td>Label*, From Type ID, To Type ID, Multiplicity</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: A list of the data model’s components and their properties and possible children types. Mergeable properties are marked with a star (*).

### 4.4 Collaboration

Collaboration is an optional feature users may decide to make use of and is based on the concept of sessions. Each session is tied to a model (identified by its ID). When a model is being edited, a collaboration session may be started. This generates a link (containing the model’s ID) that others can use to join the session. Starting a session should automatically join an existing session of the same model if one is already in progress. A collaboration session may be left by any user at any point in time.

When joining a session, the session’s model has to be sent to the joining device. If a version of the model is saved locally, it needs to be synchronized; if no local copy exists, the entire model has to be downloaded. The process of synchronization (sync) includes receiving new modifications from other clients as well as sending local changes that occurred after the session was last left to other clients. Thus, the data structure also needs to support computing a difference between two versions. Being able to identify the difference also enables only sending
specific pieces of data over the network and saving bandwidth as a result. This synchronization mechanism is also used when clients reconnect to a session they were disconnected from due to unreliable network conditions.

When modifying a model, each edit has to be transmitted to the other clients of the session where they are incorporated into their respective data models. Thus, the data model needs to be able to emit and incorporate incremental changes. Processing remote changes also has to be idempotent, i.e. it must be possible to apply a change multiple times without changing the result beyond the first application. This is important, because the same modifications might originate from several different devices.

4.4.1 Conflict handling

If multiple clients concurrently modify the same model, conflicts will inevitably arise, e.g. one participant extends a type’s name while another adds a prefix to it. Possible solutions include locking entities while they are being edited or letting participants manually resolve conflicts, but it would stand in conflict with the objective of real-time collaboration. Instead, conflicts need to be automatically and deterministically resolved by the data model.

One data structure that is able to automatically resolve conflicts are Conflict-Free Replicated Data Types (CRDTs, cf. section 2.3.1). In theory, CRDTs also fulfill the requirements mentioned earlier in this chapter: they can be combined to form complex structures (such as trees), are able to compute differences, and can incorporate changes without producing conflicts that need to be resolved manually. A number of CRDTs with different implementations exist, but most of them have some properties in common. Individual elements of the CRDT have unique identifiers. This allows them to be tracked across different replicas, even if their other characteristics have been changed. Deletions are usually modeled as tombstones that continue to be part of the data structure. This allows clients to refer to deleted elements after their deletion (e.g. when they receive modifications from a client that did not yet know of the deletion).

Versions and differences are determined using version vectors\(^1\). Some of the model’s details should not merge their values. They always have to be equal to a value of a set of allowed values, e.g. an operation’s visibility needs to equal public, packagePrivate, protected, or private, but not a mix of them. This is also true when they are modeled as integers instead of strings. Instead these properties should be merged using the last writer wins principle (LWW), i.e. only the last modification is kept. When using this method of conflict resolution, the last modification is not determined by temporal timestamps, but by comparing the CRDTs’ Lamport timestamps\(^2\). If, according to these timestamps, two modifications occurred simultaneously, one of them is chosen deterministically. However, all other components should merge their values. The entire hierarchy can be safely merged; new or deleted types, attributes, operations, parameters, and relationships need to be inserted into or removed from other models. Text directly editable by users such as type names or relationship labels can be merged on the per-character level. This allows multiple users to simultaneously edit the same text without any changes being dropped.

An alternative to CRDTs is the Operational Transformation approach (cf. section 2.3.2). Instead of employing mergeable data structures, OT relies on operations that are being emitted, transformed, and then applied. Clients emit operations for every edit taking place and broadcast them to other clients. Each receiving client needs to compare the incoming operations against the ones produced locally and then transform them to ensure that each pair of concurrent

\(^1\)https://en.wikipedia.org/wiki/Version_vector, accessed 02.11.2021
\(^2\)https://en.wikipedia.org/wiki/Lamport_timestamp, accessed 02.11.2021
operations produces the same outcome regardless of the order of application. This approach would be possible to use for this project, but it would require a considerable amount of work to implement. A set of operations needs to be defined for the manipulation of a UML model and a transformation function has to be designed, implemented, and tested. CRDTs have been chosen for this project because of the availability of high-quality frameworks (thus, reducing the development and maintenance effort) and their relative simplicity. Simple CRDT types can be composed into more complex structures with predictable behavior. A functionally equivalent transformation function would need to be able to handle a large number of different operation types, making it more difficult to develop and test for correctness.

4.4.2 System architecture

A client-server model is the most suitable architecture for this project. In this model, clients communicate with a server that handles their requests (cf. fig. 4.3). This exchange is independent of the clients' platforms, they all communicate with the same server regardless of the operating system they are running on. The server also keeps track of each session's current model state. This allows it to directly send complete or partial models to connecting clients without having to request that information from an already connected client. This way, new clients can even receive updates when no other clients are connected at that particular point in time. In order to support real-time collaboration, clients need to be able to send and receive edits with as little latency as possible. Therefore, they should open a bidirectional communication channel to the server enabling them to send and receive messages without resorting to long polling.

This model is not limited to a single server. If the need arises, multiple servers may be employed to handle clients' requests. Each server should handle a distinct set of sessions (identified by their respective models' IDs). Incoming requests need to be routed to the appropriate server responsible for the model ID in question. This routing only needs to be performed when the communication channel is established. Subsequent messages are sent over the established channel.

An alternative to this client-server model would be a peer-to-peer (P2P) system in which clients directly communicate with each other. The CRDT approach would allow for this architecture, but not having a central server comes with a key disadvantage. Without a server that holds the current state of each session's model, joining clients would need to request the current state

Figure 4.3: Client-server model.

from all already connected clients, flooding the network with requests. Moreover, this requires at least one other client to be connected to that session. The presence of a server might also be useful when enforcing user permissions (though it is not in the scope of this concept).

4.4.3 Client-server communication

A client’s connection to the server can be in one of four states: connecting, syncing, connected, disconnected. Initially it is connecting and a bidirectional communication channel is opened. The channel transmits messages of different types, each one with its own semantics and properties (cf. tab. 4.2). Requests are always sent from clients to the server, responses are always sent from the server to clients. Update is the only message type that can be sent by either the server or clients.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Properties</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync Request</td>
<td>Update</td>
<td>Optional; sent when the client has information the server does not yet have.</td>
</tr>
<tr>
<td>Sync Response</td>
<td>Update</td>
<td>Sent in response to a Sync Request.</td>
</tr>
<tr>
<td>Update</td>
<td>Data</td>
<td>Sent by either client or server.</td>
</tr>
</tbody>
</table>

Table 4.2: A list of the message types used during the client-server communication. Optional properties are denoted with brackets ([ ]).

The first step is connecting to a session. For that purpose, a Connect Request that contains the model’s ID is sent. If the client joins without a local version of the model (e.g. via a link), the state vector property remains empty. The server looks up the session using the provided ID and sends back the session model’s data. If the server cannot find the session, the connection becomes disconnected. If the client has a local model version, the state vector property is populated to enable the server to compute their difference. If the server has no model, an empty Connect Response is sent back. Otherwise, the provided version vector is compared to the version vector of the server’s model. If the server’s model contains changes the client has not seen yet, they are sent as an update in a Connect Response. In any case, the server model’s state vector is also sent as part of the response. The connection transitions to syncing.

The client incorporates the server’s update (if any) into its own data model. If the server also provided a version vector, the connection becomes syncing, otherwise it becomes connected. A syncing client is responsible for providing the server with an update. The update is generated by comparing the server’s state vector to the one of the local model and sent to the server as a Sync Request. The server incorporates the update into its own data model, broadcasts the update to other connected clients, and confirms the request by responding with an (empty) Sync Response and the connection transitions to the connected state.

Whenever a connected client has modifications to report, an Update message is sent to the server which is immediately broadcast to all other connected clients. Likewise, the client may at any point in time receive Update messages that originated from other clients. Their contained data needs to be incorporated into the local data model; a response is not intended.

A connection may become disconnected if an invalid message is sent, the server is instructed to join an unknown session, or the network conditions interrupt the communication channel. When that happens, no more messages are exchanged. The client may try to reestablish the
connection by opening a new communication channel and transitioning through the **connecting** and **syncing** states again. This ensures that any edits that occurred while the connection was interrupted are not lost.

Fig. 4.4 shows the state diagram of this process. Fig. 4.5 depicts the example of a client connecting to a collaboration session with two other clients already connected. After the connection has been established, client 1 sends an **Update** to the server which is broadcast to the other clients. Afterwards, the same process is repeated with an **Update** from client 2.

![State diagram of client-server communication](image)

**Figure 4.4: Client-server communication state diagram.**

### 4.5 Summary

This chapter laid the conceptual foundations for the solution whose implementation is elaborated in chapter 5. First, the concept's objectives were listed, defining the features of UML class diagrams that are to be supported. Then, the most essential screens of the user interface were introduced. This was followed by a comprehensive description of the app's internal tree-based data model. Finally, the topic of collaboration was discussed, including explanations of how conflicts are to be resolved, which architecture to base the system on, and the details of the communication between client and server.

In theory, it should be possible to implement this concept utilizing different languages and frameworks, depending on the specific priorities of a solution that is to be employed in production. The next chapter introduces one possible implementation.
Figure 4.5: Exemplary client-server communication.
5 Solution

This chapter takes a detailed look on CoMod, the solution that was implemented based on the concept described in the previous chapter. It serves as a proof of concept demonstrating that it is indeed possible to build a mobile modeling application with real-time collaboration support. CoMod consists of a cross-platform application running on Android and iOS as well as a server executable facilitating the real-time collaboration features. Both components are examined separately, followed by a description of how they communicate with each other.

5.1 Client

The mobile app was implemented using Flutter\(^1\) (v2.5.2), a UI toolkit made by Google to build fast and responsive cross-platform applications. It is written in Dart\(^2\) (v2.14.3) and uses a custom rendering engine called Skia to draw its user interface. This allows it to provide a consistent user experience across all platforms without relying on web technologies that usually come along with a number of shortcomings (cf. sections 2.2.1 & 2.2.2). The UI of a Flutter application is structured as a tree of widgets. Widgets may be stateful (i.e. they have to be redrawn whenever their state changes) or stateless (they remain constant during their lifetime). In addition to the business logic implemented in Dart, the app also relies on a JavaScript subsystem running all the logic related to the CRDT-based data model, which is explained later on.

5.1.1 Walkthrough

This section is structured as a guided tour through all of CoMod’s screens and features.

The application uses a stack-based navigation approach, i.e. screens are pushed onto and popped off the stack when navigating forwards or backwards, respectively. One of its benefits is that it is easy to follow and a gesture may be used to go back to the previous screen. The app’s color scheme is a blend of Flutter’s default light blue color and the dark blue theme of TU Dresden’s corporate design.

\(^1\)https://flutter.dev, accessed 11.11.2021
\(^2\)https://dart.dev, accessed 11.11.2021
Models screen

When CoMod is launched for the first time, an empty Models screen (cf. fig. 5.1) is shown. It allows the user to create a model from scratch or load an example to help them explore the application. After at least one model has been created (either manually or by loading the example), the screen shows the list of models (cf. fig. 5.2). Each model has a name (entered by the user) and a randomly assigned universally unique identifier (UUID). The latter is shown to help users determine which models share a collaboration session, but may be hidden in the release version. Existing models may be renamed or deleted using the ellipsis button (...) on the right-hand side of the screen. Tapping on a model performs a navigation to the Model details screen. New models may be created using the plus button (+) in the bottom-right corner. A button in the navigation bar opens a dialog that allows users to join a collaboration session by entering its ID or pasting the link to the session.

Model details screen

The Model details screen (cf. fig. 5.3) shows an overview of the model’s contents as a vertically scrollable list. Each type’s UML class diagram card is shown, containing its attributes and
operations. Tapping on a card performs a navigation to the respective type’s *Type details screen*. Relationships to other types are shown as horizontally scrollable views. New types may be created using the plus button in the bottom-right corner. The chevron button (<<) on the left-hand side of the navigation bar navigates back to the *Models screen*.

A noteworthy detail is that relationships are only shown as indicators, i.e. the line or arrow does not lead to the relationship’s target in the list, but to a smaller placeholder only showing its name. Generalizations and realizations are shown above, associations, aggregations, and compositions below the type’s card. The latter kind may also display a label and/or multiplicity annotations. This approach allows types to be viewed in their respective contexts while keeping the screen clear and concise, even if the model contains dozens of different types.

The navigation bar’s right-hand button can be used to access the app’s collaborative features. A collaboration session may be started from there. During this process, a message will show with an option to copy a link to the session or an error message, should the connection fail. The link is meant to be shared with other participants. They may either open it on their mobile devices, paste it in the appropriate dialog on the *Models screen*, or join via the collaboration button if they have a local version of the model. If a session is in progress, the button offers the options of copying a link to the session or leaving it altogether.

Figure 5.3: Model details screen.  
Figure 5.4: Type details screen.
Type details screen

The Type details screen (cf. fig. 5.4) shows a type's name, type, supertypes, attributes, operations, and relationships. All of these properties may be edited and the type may be deleted from this screen as well.

The type's name is entered via a text field that only allows characters valid for an identifier in most programming languages. The button for the type's type (class, abstract class, interface) opens a drop-down menu which allows exactly one option to be chosen (like a radio button). The supertypes button also presents a drop-down menu, but allows multiple options to be selected (like checkboxes).

The type's attributes are displayed as a list with a button to add a new one on the bottom. Each row in this list represents a single attribute with controls to modify its visibility, name, and data type. In order to conserve space, only the visibility's corresponding symbol is displayed on the button (e.g. + for private). However, when it is used to show a drop-down menu, the symbol along with its corresponding label is shown. This allows the row to look just like an entry in a UML class diagram while still giving users sufficient information during the editing process. The data type button allows choosing from a list of primitive data types, such as boolean or string, and other types (classes and interfaces) that are part of the model. The ellipsis button on the right-hand side can be used to move the attribute to a different position within the list or to delete it altogether.

The type's operations are displayed in a similar fashion. The main difference is that operations may have a variable number of parameters. These are displayed below the main row while still visually belonging to their operation's row. The ellipsis button has the additional option of adding a new parameter. When an operation is deleted, all of its parameters are deleted as well.

The type's relationships are displayed similarly as well. Their rows' centerpiece is a drop-down menu that allows the selection of the relationship type (association, aggregation, composition, association with class, qualified association). A matching icon is displayed next to the options and after a selection has been made. A text field above may be used to assign a label to the relationship. If the relationship is an association with a class, the label is replaced by a button allowing the selection of the association class. The relationship's multiplicity annotation may be edited using the text fields to the sides. The button to the right can be used to select the relationship's target. The ellipsis button offers the same options as for attributes and operations.

Summary

Overall, the application has been designed with small screens and mobile usability in mind. The three-screen structure allows users to quickly navigate models and their components without having to deal with a myriad of different screens. The UI is rendered using the Skia engine, making it fast and responsive. All the features outlined in the concept's objectives (cf. section 4.1) have been implemented.

5.1.2 Data model

CoMod's data model is based on a number of classes that form an in-memory object graph. Fig. 5.5 shows a simplified class diagram of the classes involved. The model is kept separate
from the user interface and has no dependencies on the visual layer. This allows it to be tested separately and reused should the UI be rewritten in the future. It could even be put into a dedicated package. In fact, the user interface is a function of the model. Whenever part of the model changes, the associated parts of the UI are redrawn. This functional approach reduces the amount of mutable state the visual layer needs to hold and thus, the potential for discrepancies.

![Simplified UML class diagram of CoMod's data model.](image)

As elaborated in the corresponding concept chapter, the model contains types and relationships, types contain attributes and operations, and operations contain parameters. All classes implement the `UMLElement` interface, i.e. they can be identified by a unique identifier. All model components, except the model itself, also implement the `NamedUMLElement` interface which requires them to have a name property. All parent types (containing other types) have methods for adding or removing contained types. The reason is that the contained types are stored in hash maps as opposed to simple lists. This allows them to be efficiently accessed using their ID. Removing a contained type often requires additional logic to ensure the model remains consistent.
Model consistency

The model is kept consistent at all times. Whenever a relationship is added, it points to a specific type instead of having no target. Whenever a type is deleted, it is removed as a supertype from all other types and all relationships it is part of are removed as well. Moreover, all attributes, operations, and parameters that use it as their data type are converted to string. If, for some reason, a referenced type cannot be found, it is hidden from the UI, as if it did not exist. This behavior is consistent with the approach of removing relationships involving deleted types. Furthermore, all changes related to the deletion of a type are performed as part of a single transaction, i.e. the changes are being applied to the data model atomically to ensure that they cannot be applied partially on other clients.

Nevertheless, it is still possible for users to enter inconsistent information such as inheritance cycles or invalid multiplicities (e.g. 3..1). In such cases, the data model accepts the changes and shows them in the UI, but marks them as inconsistent, usually with a bright red font. This ensures that all user modifications are being saved while warning users that they have entered inconsistent data.

5.1.3 JavaScript subsystem

CoMod makes use of the Yjs framework\(^3\) [10, 11] providing CRDTs to facilitate its real-time collaboration features. It is a JavaScript-based CRDT implementation, thus, it needs a JavaScript environment to run. Despite this limitation, the framework has been chosen because of its extensive feature set (supporting nested XML structures), solid performance\(^4\), and quality of documentation. It fulfills the requirements for the data model laid out in section 4.4: it can compute the difference between two model versions (using their respective version vectors) and is able to generate and incorporate incremental updates.

The JS environment holds a separate instance of the data model in memory. It is a tree-based model resembling an XML structure. For that purpose, Yjs offers two dedicated node types: YXMLElement and YXMLText. The former is a generic node in the XML tree that may contain attributes and child nodes while the latter is a type of mergeable string that may be put as a leaf node in XML trees. This structure was chosen because the app's data structure is expressible as an XML document and a purely list-based structure would have been much harder to work with. The native code's data model is essentially mirrored within the JS environment. Each instance of a data model class (cf. section 5.1.2) is represented as a YXMLElement with its properties being modeled as XML attributes. Furthermore, each element has an ID attribute that is the same as the corresponding ID property in the native data model. This approach allows changes in either model to be communicated and replicated on the other side. All attributes are string-based, but they are not mergeable (cf. section 4.4.1). Yjs treats them as LWW registers. Concurrent modification are being resolved deterministically, but unpredictably. Mergeable strings such as type names or relationship labels are modeled as instances of YXMLText positioned at index 0 within their parent's children.

The JS subsystem also has to be involved in persisting the data model to disk. CRDTs assign identifiers and timestamps to their internal components (e.g. each character in a string). Thus, merely saving the data model as an XML file would not work as this information would be missing. Instead, the entire model can be serialized to a binary format and efficiently persisted as a binary blob. The JS environment is sandboxed and does not have access to the file system.

\(^3\)https://github.com/yjs/yjs, accessed 15.11.2021
\(^4\)https://github.com/dmonad/crdt-benchmarks, accessed 15.11.2021
so it needs to send the blob to the native code. Likewise, whenever the model is to be loaded from disk, the binary blob has to be sent to the JS subsystem for Yjs to deserialize the data. Afterwards, the loaded model is converted to its XML representation and sent back to the native code to be replicated and displayed in the UI.

The native code and the JS subsystem cannot access each other’s memory directly. JavaScript statements may be executed as strings and results can be received using channels that convey strings. Due to the lack of type checking and proper error handling at that boundary, hereinafter referred to as bridge, the communication between the two environments is kept as lightweight as possible. For that reason, a slim API (cf. lst. 5.1) has been defined for the native code to give instructions to the subsystem. The API includes functions for creating a model, loading a model, requesting a state vector, syncing server changes, processing updates, and handling transactions. The methods `insertElement`, `moveElement`, `deleteElements`, `updateAttribute`, and `updateText` can be used to manipulate the model. Each of the API’s methods has a specific signature that determines the arguments and data types to be used. Binary data is first converted
to its Base64\(^5\) representation before being sent over the bridge. Whenever the model has an 
update to emit or it was serialized after processing local or remote changes, a corresponding 
message is sent over the bridge to be handled by the native code. Fig. 5.6 shows the data flow 
when loading or creating a model and manipulating it.

```javascript
newModel (uuid: string): void;
loadModel (uuid: string, base64Data: string): string;
stateVector (): string;
sync (serverStateVector?: string, serverUpdate?: string): string | undefined;
processUpdate (data: string): void;
beginTransaction (): void;
endTransaction (): void;
insertElement (parentID: string, id: string, nodeName: string, name: string, 
attributes?: [string, string][][], tags?: string[]): void;
deleteElements (ids: string[]): void;
updateText (id: string, position: number, deleteLength: number, 
insertString: string[]): void;
updateAttribute (id: string, attribute: string, value: string): void;
enum MoveType {ToTop = 0, Up, Down, ToBottom}
moveElement (id: string, moveType: MoveType);
beginTransaction (): void;
endTransaction (): void;
```

Listing 5.1: JavaScript subsystem API signature (TypeScript syntax).

The client’s JS code is a package that is compiled and tested separately from the native code. 
The native code calls it whenever necessary. It is written in the TypeScript\(^6\) language allowing 
it to benefit from strong static typing and more extensive compile-time checks to reduce the 
likelihood of bugs. During the compilation process (cf. fig. 5.7), the TypeScript compiler produces 
JS files that are subsequently bundled with their dependencies and minified using Webpack\(^7\). 
The result is a single JS file (~120 KB) that is included as an asset in the Flutter application. This 
text file’s contents are loaded and evaluated when the app is launched.

![JavaScript compilation and bundling process](image)

Figure 5.7: The JavaScript compilation and bundling process.

\(^5\)https://en.wikipedia.org/wiki/Base64, accessed 29.11.2021
\(^6\)https://www.typescriptlang.org, accessed 17.11.2021
\(^7\)https://webpack.js.org, accessed 17.11.2021
5.1.4 Dependencies

Table 5.1 lists all of the application's dependencies and their respective contribution to the codebase. For the sake of clarity, packages that only provide types for TypeScript were omitted.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Version</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>either_dart</td>
<td>0.1.2</td>
<td>Type having either a value A or a value B, but not both.</td>
</tr>
<tr>
<td>flutter_js</td>
<td>0.5.0+2</td>
<td>Enables the execution of JavaScript.</td>
</tr>
<tr>
<td>path_provider</td>
<td>2.0.2</td>
<td>Provides the path to the local document storage directory.</td>
</tr>
<tr>
<td>protobuf</td>
<td>2.0.0</td>
<td>Enables the usage of protocol buffers.</td>
</tr>
<tr>
<td>provider</td>
<td>5.0.0</td>
<td>Enables the UI to depend on the underlying data model.</td>
</tr>
<tr>
<td>quiver</td>
<td>3.0.1</td>
<td>Provides convenient hash functions.</td>
</tr>
<tr>
<td>tuple</td>
<td>2.0.0</td>
<td>Adds support for tuple types.</td>
</tr>
<tr>
<td>uuid</td>
<td>3.0.4</td>
<td>Adds support for universally unique identifiers (UUIDs).</td>
</tr>
<tr>
<td>web_socket_channel</td>
<td>2.1.0</td>
<td>Makes it possible to communicate with WebSockets.</td>
</tr>
<tr>
<td>xml</td>
<td>5.1.2</td>
<td>Enables XML parsing.</td>
</tr>
<tr>
<td>test (dev only)</td>
<td>1.16.8</td>
<td>Required for Dart unit tests.</td>
</tr>
<tr>
<td>js-base64 (JS)</td>
<td>3.6.1</td>
<td>Enables conversion to and from Base64.</td>
</tr>
<tr>
<td>yjs (JS)</td>
<td>13.5.12</td>
<td>Fast and feature-rich CRDT implementation.</td>
</tr>
<tr>
<td>mocha (JS, dev only)</td>
<td>9.0.3</td>
<td>Test framework with support for asynchronous tests.</td>
</tr>
<tr>
<td>typescript (JS, dev only)</td>
<td>4.3.5</td>
<td>Support for the TypeScript language.</td>
</tr>
<tr>
<td>webpack (JS, dev only)</td>
<td>5.49.0</td>
<td>Bundles and minifies the code into a single file.</td>
</tr>
</tbody>
</table>

Table 5.1: The client application's list of dependencies.

5.2 Server

CoMod's server facilitates real-time collaboration between clients. It is written in TypeScript and based on the Node.js8 runtime environment. It can be installed and run directly or as a Docker9 container that already includes the runtime environment and dependencies.

When the server is running, it holds a hash map of sessions in memory. Each session is identified by its session/model ID and contains the last known version of its associated data model. The server also makes use of the Yjs framework; thus, it can compare a session’s data model version to that of a client and compute the difference and generate an update if necessary. Whenever a client connects, a bidirectional WebSocket connection is established and used for the subsequent communication. An instance of the Client class is created keeping track of the client’s state and holding a reference to the connection. The first request sent contains the session ID which allows the server to add the client to the corresponding session. Whenever an update is sent by any client, the server incorporates it into the session’s data model and broadcasts it to all other clients that are part of the session. Fig. 5.8 depicts the server's components as a UML class diagram.

Sessions and their associated data models are kept in memory, even if all participants have left the session. This avoids the need of transmitting the entire data model the next time a client

---

8https://nodejs.dev, accessed 17.11.2021
9https://www.docker.com, accessed 17.11.2021
connects to that session. In case the server runs low on memory, any data model without any clients connected to its session may be purged. The next time a client connects with its ID, the server will request the data model as part of the syncing step.

![Figure 5.8: Server UML class diagram.](image)

### 5.2.1 Dependencies

Table 5.2 lists all of the server’s dependencies and their respective contribution to the codebase. For the sake of clarity, packages that only provide types for TypeScript were omitted.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Version</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>google-protobuf</td>
<td>3.15.3</td>
<td>Protocol Buffers runtime library.</td>
</tr>
<tr>
<td>uuid</td>
<td>8.3.2</td>
<td>Adds support for universally unique identifiers.</td>
</tr>
<tr>
<td>ws</td>
<td>7.5.3</td>
<td>WebSocket implementation.</td>
</tr>
<tr>
<td>yjs</td>
<td>13.5.12</td>
<td>Fast and feature-right CRDT implementation.</td>
</tr>
<tr>
<td>mocha (dev only)</td>
<td>9.0.3</td>
<td>Test framework with support for asynchronous tests.</td>
</tr>
<tr>
<td>typescript (dev only)</td>
<td>4.3.5</td>
<td>Support for the TypeScript language.</td>
</tr>
</tbody>
</table>

Table 5.2: The server’s list of dependencies.

### 5.3 Collaboration

This section elaborates on the technical details of the application’s collaboration features. A collaboration session may be started by tapping on the collaboration button in the top right corner of the Model details screen (cf. fig. 5.3). This generates a link containing the session ID that is intended to be sent to other users to allow them to join the session. They can do so by either tapping the link, pasting it into the collaboration dialog on the Models screen, or via the collaboration button if they already have a version of the model on their device. Clients that are part of the same session transmit and receive edits in real time. Whenever a client leaves the session, either voluntary or involuntary, it no longer sends or receives any updates. Any changes are recorded locally and transmitted to the server and other clients as soon as this client rejoins the session at a later point in time.
5.3.1 Client-server communication

When the collaboration feature is activated, the app opens a WebSocket connection [5] to the server. This allows the exchange of binary messages over a single full-duplex channel without the need for either party to resort to polling. The connection remains open until it is interrupted or closed from either side. If a client or the server receives invalid data, the channel is closed prematurely.

The messages to be sent are serialized using Protocol Buffers\textsuperscript{10}, a language-neutral mechanism for serializing structured data. A .proto file (cf. lst. 5.2) contains the definitions of the message types along with their fields (properties). Implementations for different programming languages (in this case Dart and TypeScript) allow messages to be constructed in code. Each message may be serialized to binary data that can be sent over the wire and deserialized on the other end, regardless of the target system's programming language. The message types and the communication protocol implement the concept described in section 4.4.3.

```plaintext
syntax = "proto3";

// Sent from clients to the server.
message CollaborationRequest {
  oneof message {
    ConnectRequest connect_request = 1;
    SyncRequest sync_request = 2;
    bytes update = 3;
  }
}

// Sent by clients when connecting to a collaboration session. Server responds // with a ConnectResponse.
message ConnectRequest {
  // The model's unique identifier.
  string uuid = 1;
  bytes state_vector = 2;
}

// Sent by clients to provide the server with a model update (or its entire // content). Server responds with a SyncResponse.
message SyncRequest {
  // [optional] A model update from the client. Contains the entire model if // the server didn't send a state vector.
  bytes update = 1;
}

// Sent from the server to clients.
message CollaborationResponse {
  oneof message {
    ConnectResponse connect_response = 1;
    SyncResponse sync_response = 2;
    bytes update = 3;
  }
}

// Sent by the server in response to a ConnectRequest.
message ConnectResponse {
  bytes state_vector = 1;
  // [optional] A model update from the server. Contains the entire model if
}
```

\textsuperscript{10}https://developers.google.com/protocol-buffers, accessed 20.11.2021
5.3.2 Client data flow

The existence of the JavaScript subsystem that handles the CRDT-related logic leads to a fairly complex data flow within the client application (cf. fig. 5.9). The native code acts as an intermediary between its JavaScript subsystem and the server. Only the subsystem can provide a state vector, process server responses, and serialize the model. However, it cannot directly communicate with the server. Therefore, any data that needs to be first sent to the server has to be sent over the bridge to the native code.

Figure 5.9: Data flow when starting a collaboration session and receiving an update.
5.4 Testing

This proof of concept relies on a combination of unit, integration, and end-to-end tests. The following sections describe each category of tests separately and showcase some examples. The share of test-related code is about 14% for the native Dart code, 50% for the JS subsystem code, and 55% for the server code. The comparatively low amount of Dart-testing code can be attributed to the majority of code being declarative code specifying the user interface, including its structure and style that does not need to be tested.

5.4.1 Unit tests

This category of tests is intended to test individual parts of source code. In the context of the Flutter application, the data model is tested this way. Examples include ensuring that the model and its individual components can be loaded from an XML-formatted input string and that attributes and operations are properly converted to strings to be displayed in the UI. All the Dart extensions (code added to standard library types) are covered by unit tests as well (e.g. lst. 5.3). The unit tests for the code running in the JS subsystem make sure that each of the API's methods (e.g. loadModel(), insertElement(), etc.) behaves as intended. This includes not only checking if the intended changes are reflected in the JS data model, but also ensuring that the appropriate messages are sent back to the native code. The server only contains a limited amount of unit tests. It implements the communication protocol and mainly benefits from integration tests.

```dart

test('Iterable compactMap', () {
    final list = [1, 2, 3, 4];
    expect(list.compactMap((x) => null), []);
    expect(list.compactMap((x) => x), list);
    expect(list.compactMap((x) => x % 2 == 0 ? x : null), [2, 4]);
});
```

Listing 5.3: Dart unit test for the compactMap method defined on Iterable.

5.4.2 Integration tests

Integration testing commonly tests multiple components of a software system acting as a group. All Flutter tests involving the user interface belong to this category. They ensure that the app can be launched and its main tasks such as creating models, performing modifications, and collaborating can be performed (e.g. lst. 5.4). However, the application does not communicate with the actual server binary, it instead uses a fake that mocks the server’s behavior. This allows both components to be tested separately. Meanwhile, the actual server’s integration tests check whether it correctly implements the communication protocol and rejects connections with invalid or incomplete data.

```dart

testWidgets('load example', (tester) async {
    await _launch(tester); // Launch and remove existing models

    await tester.tap(find.text('Load Example'));
    await tester.pumpAndSettle(); // Wait for animations to finish

    await tester.tap(find.text('Example')); // Open example model
    await tester.pumpAndSettle();
}
```

41
5.4.3 End-to-end tests

The last category of tests, also called E2E tests, intends to test the entire software system. In this case, this includes the Flutter application, its JS subsystem, and the server executable. In contrast to the other test categories, the client application communicates with the actual server instead of a fake. The main challenge of this project’s E2E tests is emulating mobile devices to run the application. Unfortunately, the current state of Flutter tooling does not allow the coordination of multiple emulated devices running integration tests. Due to that limitation, the E2E tests start the server, launch a client app, let it create a new model, start a collaboration session, and quit. Afterwards, a second client app is launched on a different device and instructed to join the session. The test succeeds if the second device receives and displays the model created on the first device. Lst. 5.5 shows the shell script executing the different steps of this process.

```
#!/bin/bash

cd ../server
npm start & # Start server in the background
NODE_PID=$!
# Remember its PID
cd

COLLABORATION_UUID="2c8932f2-7ac2-4e6d-abe9-209b407fc46". # Random, but fixed UUID

cd ../client
flutter test integration_test/e2e_test.dart
  --name client1
  --dart-define="collaboration_uuid=$COLLABORATION_UUID"
  -d iphone
flutter test integration_test/e2e_test.dart
  --name client2
  --dart-define="collaboration_uuid=$COLLABORATION_UUID"
  -d iphone
  --no-pub

kill "$NODE_PID". # Shut server down
```

Listing 5.5: Shell script driving the end-to-end tests.

5.5 Extensibility

CoMod implements the concept described in chapter 4. Nevertheless, it can be extended on multiple levels. Additional platforms may be supported by adding the appropriate targets in Flutter. Because of the separation of software modules, the UI may be changed without modifying the model layer. The support for additional model types is particularly interesting as it would allow the solution’s foundation to be (re-)used for editing other model types, perhaps even in entirely different modeling domains. Because of that, the implementation has been
kept sufficiently general, such that other kinds of models (e.g. ER or CROM [8]) could be supported without requiring a large amount of changes. The server is able to handle any models that can be represented using the Yjs framework. Because of Yjs's support for XML-based data structures, this includes all models that can be represented using XML documents. Thus, the server's executable can be used as is. Furthermore, the communication protocol is independent from the model's contents as well. So, the server can even handle sessions with different kinds of models at the same time. Code-level changes are only required on the client side. The JS subsystem exposes an API that allows the manipulation of any XML-like structure using the generic methods insertElement, deleteElements, moveElement, updateAttribute, and updateText. Therefore, it does not need to be adapted to other types of models. The subsystem's native counterpart does have to be changed either. However, a different kind of model has to be expressed with a different underlying data model and the application's user interface has to be adapted in order to visualize and allow interactions with the new model.
6 Evaluation

This chapter evaluates the solution presented as part of the previous chapter by means of a case study. Two distinct common cases are presented, followed by a technical analysis of CoMod’s scalability. Subsequently, limitations and threats to validity are pointed out and the chapter is summarized in the end.

6.1 Case Study

The following case study focuses on the collaborative aspects of CoMod as this is the main contribution of this thesis. The characteristics of the application being used without the collaboration features have been covered by the unit and integration tests described in section 5.4. The correctness of merges and merge conflict resolutions is already covered by the unit tests of yjs.

This application can be employed in a multitude of different situations. One such case is one person exclusively editing a model while many people receive updates on their devices. This scenario is subsequently referred to as 1WnR (1 writer, many readers). Examples include an exercise in the context of a class lead by a tutor or a software architect sharing their work with a number of developers in real time.

The other case examined as part of this study is a group with every member contributing to the model, subsequently referred to as nWnR (many writers, as many readers). This is a common scenario within group projects in an academic setting or project groups at software companies.

6.2 Technical Analysis

This analysis evaluates the feasibility of the cases described above by examining how well the system scales to accommodate these cases. A test data generator has been implemented to provide pseudorandom input data for the following analyses. The client application (henceforth "client") and server executable (henceforth "server") are analyzed separately.

6.2.1 Test data generator

Both of the following analyses include the transmission of pseudorandom input to the subjects under test (SUT). This represents the stream of model updates that is received from other clients. The input is pseudorandom, because it should resemble a series of model changes applied by a user. An entirely random series of changes would less accurately represent how CoMod would be used in practice.

Each run of the test data generator (TDG) generates data for a single session based on two parameters: the number of iterations $i$ and the number concurrently editing clients $c$. At the start, the model consists of a single type. This model serves as the baseline for the first iteration. During each iteration, $c$ concurrent updates are generated. After each iteration, all generated updates are merged with the previous baseline to form a new baseline for subsequent iterations. After all iterations have been processed, the output of $i \times c$ updates is written to a text file. Each update is represented as a Base64 string on its own line.

Each update represents a change to the model. To produce an update, a random number is generated determining the kind of change to apply. There is a 6% chance for a component (type, attribute, operation, parameter, relationship) to be added, a 2% chance for a component to be deleted, a 12% chance for a component's property (e.g., visibility, data type) to be changed, and an 80% chance for a component's name (or label in the case of relationships) to be modified. These percentages have been chosen to let a model grow as more changes are being applied to it (it is likelier for a component to be added to the model than it is for a component to be deleted). The relatively large percentage for textual changes has been chosen, because each keystroke (character addition/deletion) is represented by a separate update.

For each kind of change (insertion, deletion, property, textual), a random component is chosen. In the case of an insertion, it serves as the parent for the new component to be created. If it is the model itself, a type or relationship will be inserted; if it is a type, an attribute or operation will be inserted; and if it is an operation, a parameter will be inserted. The inserted component's properties (e.g., visibility, data type) are determined at random. In the case of a deletion, the chosen component and its children are deleted. In the case of a property change, a random property of the chosen component is changed to a random, but different, value. In the case of a textual change, there is a $\frac{2}{3}$ chance for a character to be inserted at a random location and a $\frac{1}{3}$ chance for a random character to be deleted within the chosen component's name or label. The generator ensures that it only produces changes that are possible for the baseline model, e.g., an empty model does not allow any changes apart from insertion.

6.2.2 Client analysis

This analysis evaluates the client application's performance when confronted with an increasing number of incoming updates. For that purpose, a special server has been set up exhibiting largely the same behavior as the actual server with the exception of sending a certain number of updates each second after a client has successfully connected to a session. Its input is a file produced by the TDG. An instrumented instance of the client is launched and instructed to connect to that server. The server sends a certain number of updates equally distributed over the course of each second. The client measures the time it takes for incoming updates to be processed (in ms). All updates are processed by the JavaScript subsystem (cf. section 5.1.3). Thus, they are processed serially on a single thread. Ideally, each update's processing time should be below $1s - n$ milliseconds where $n$ is the number of updates received per second. If an update's processing time exceeds that limit, the app receives more updates than it can
Multiple experiments using two different input files produced by the TDG were carried out on an iPhone 11 Pro (2019), each with a different number of updates sent per second. The first input file represents 5000 iterations for one editing client while the other represents 2000 iterations for 5 concurrently editing clients. These two files correspond to the two cases described in section 6.1. Measurements were taken after the application of 10, 100, 500, 1000, 1500, 2000, and 2500 updates and represent the average time (in ms) it took for the application to process the 10 updates preceding the point of measurement. The results can be seen in tab. 6.1 and tab. 6.2. An experiment was stopped when it was either running too long (in the case of 1 update/s) or when the application became unresponsive (as in the case of 30 updates/s).

The results show that the more updates were applied before them, the longer it takes for subsequent updates to be processed. This difference can be quite significant, e.g. when receiving 5 updates per second, it takes 9.9ms on average to process the first 10 updates, but 60.1ms to process the updates 2491 through 2500. The reason for this finding is the increasing complexity of the model. After the application of each update, the model is serialized to be written to disk. Larger models take a longer time to be serialized. In fact, models are monotonically increasing in size as even the deletion of an element or character leaves a gravestone. The results also show that, on average, it takes slightly longer to process updates originating from 5 concurrent writers than the same number of updates from a single writer. The reason is likely the need for arising conflicts to be resolved. Surprisingly, when a higher rate of updates is received, updates are processed faster. Possible explanations include the just-in-time (JIT) compiler of JavaScript increasingly optimizing the code at runtime and the operating system giving the JavaScript thread an increasingly higher priority the more work it does.
Tab. 6.3 shows the results of multiplying the values in tab. 6.1 by the number of updates received per second. It is apparent that some update rates are unfeasible in combination with larger models. In practice however, update rates exceeding 10-15 updates/s should be a rare occurrence. In certain cases, multiple participants typing text at the same time might still produce a large number of updates. This causes a high peak rate, but not a high sustained stream of updates. In this scenario, incoming updates might not be processed in real time, but the queue of outstanding updates should clear rapidly. In conclusion it can be stated that the client application is able to handle a large rate of updates (up to 15-20 updates/s) without becoming unresponsive, though the rate of updates should be much lower in practice and momentary peaks can be handled. Consequently, both of the scenarios investigated as part of this case study are feasible.

<table>
<thead>
<tr>
<th>Updates/s</th>
<th>10</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>1500</th>
<th>2000</th>
<th>2500</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.4</td>
<td>22.2</td>
<td>41.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>49.5</td>
<td>89.5</td>
<td>215.5</td>
<td>274.0</td>
<td>250.0</td>
<td>312.5</td>
<td>300.5</td>
</tr>
<tr>
<td>10</td>
<td>64.0</td>
<td>114.0</td>
<td>240.0</td>
<td>281.0</td>
<td>317.0</td>
<td>385.0</td>
<td>416.0</td>
</tr>
<tr>
<td>15</td>
<td>112.5</td>
<td>106.5</td>
<td>211.5</td>
<td>331.5</td>
<td>438.0</td>
<td>537.0</td>
<td>594.0</td>
</tr>
<tr>
<td>20</td>
<td>120.0</td>
<td>110.0</td>
<td>256.0</td>
<td>424.0</td>
<td>550.0</td>
<td>690.0</td>
<td>768.0</td>
</tr>
<tr>
<td>25</td>
<td>130.0</td>
<td>127.5</td>
<td>282.5</td>
<td>497.5</td>
<td>680.0</td>
<td>870.0</td>
<td>965.0</td>
</tr>
<tr>
<td>30</td>
<td>156.0</td>
<td>144.0</td>
<td>315.0</td>
<td>585.0</td>
<td>816.0</td>
<td>1044.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Average processing time of all updates received over the course of a second (in ms). Obtained by multiplying the values in tab. 6.1 by the number of updates received per second.

6.2.3 Server analysis

This analysis evaluates the server’s performance when handling an increasing number of concurrent sessions and updates. The following experiments were carried out on an E2-medium general-purpose virtual machine with 2 virtual CPU cores and 4 GB of memory on Google Cloud Platform (GCP)\(^2\) located in Frankfurt. Apart from the underlying hardware resources, the server’s runtime performance depends on some other factors such as number of sessions, number of clients per session, rate of updates clients send, and, as determined as a relevant factor by the client analysis, the complexity of each session’s model.

The experiments’ setup is as follows. A specific number of sessions \(s\) with a fixed length \(l\) (number of updates) is created over the course of the first session’s length. TypeScript-based clients send a number of updates \(r\) each second. The first session starts immediately, the second session starts after \(l = s \times r\) seconds, and the last session starts just before the first session ends. After a session has ended, its clients disconnect and a new session is started. Thus, after the first session has ended, \(s\) sessions will be in progress, each at a different point in its lifetime. This is the point in time at which the experiments’ measurements begin to be taken. This approach prevents the complexity of models to affect the results the longer an experiment runs. It also prevents potentially hundreds or thousands of clients from connecting at the exact same time, a situation unlikely in practice that could easily overwhelm the server. After the measurement phase has begun, each session reports its average latency, defined as the average time it takes for each of its clients to receive an update a writer sent to the server, when it ends. The average of \(s\) sessions’ latency is recorded as the result for the experiment.

\(^2\)https://cloud.google.com/gcp, accessed 01.01.2022
The latency metric was chosen, because it has a direct impact on users as opposed to other metrics such as CPU utilization or memory pressure that are not directly perceivable to users.

Two different session scenarios were analyzed, one with a single writer and 19 receivers (20 participants in total), the other with 5 concurrent writers. These scenarios correspond to the cases outlined in section 6.1. Both were run with a session length of 1000 updates (200 TDG iterations in the case of 5 writers). Tab. 6.4 and 6.5 show the respective results. Ping requests to the server's IP address had a roundtrip time of 20.0ms. This can be thought of as a baseline\(^3\). The general trend is that the latency increases as the number of sessions and updates increases. This is most likely due to the fact that the server has a higher workload and is not able to process all incoming updates right away. The values marked with a dash could not be collected, because some clients were not able to connect to their session (due to the high number of concurrent WebSocket connections).

<table>
<thead>
<tr>
<th>sessions</th>
<th>1 update/s</th>
<th>2 updates/s</th>
<th>3 updates/s</th>
<th>4 updates/s</th>
<th>5 updates/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 session</td>
<td>24.0</td>
<td>20.1</td>
<td>25.1</td>
<td>24.5</td>
<td>19.3</td>
</tr>
<tr>
<td>50 sessions</td>
<td>19.1</td>
<td>18.8</td>
<td>19.4</td>
<td>20.0</td>
<td>20.3</td>
</tr>
<tr>
<td>100 sessions</td>
<td>35.1</td>
<td>128.0</td>
<td>137.9</td>
<td>192.9</td>
<td>226.6</td>
</tr>
<tr>
<td>150 sessions</td>
<td>152.2</td>
<td>141.2</td>
<td>171.6</td>
<td>184.9</td>
<td>176.2</td>
</tr>
<tr>
<td>200 sessions</td>
<td>146.4</td>
<td>155.2</td>
<td>147.2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>250 sessions</td>
<td>152.3</td>
<td>192.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.4: \(1WnR\) scenario. Average client latency (in ms) when connected to a server with the given number of concurrent sessions (20 participants, one of which is a writer) and rate of updates (per second).

<table>
<thead>
<tr>
<th>sessions</th>
<th>1 update/s</th>
<th>2 updates/s</th>
<th>3 updates/s</th>
<th>4 updates/s</th>
<th>5 updates/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 session</td>
<td>21.4</td>
<td>23.9</td>
<td>24.5</td>
<td>24.1</td>
<td>24.0</td>
</tr>
<tr>
<td>50 sessions</td>
<td>21.3</td>
<td>45.9</td>
<td>36.4</td>
<td>41.1</td>
<td>37.0</td>
</tr>
<tr>
<td>100 sessions</td>
<td>21.8</td>
<td>36.9</td>
<td>52.7</td>
<td>198.8</td>
<td>238.9</td>
</tr>
<tr>
<td>150 sessions</td>
<td>25.9</td>
<td>133.3</td>
<td>270.9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>200 sessions</td>
<td>32.9</td>
<td>272.7</td>
<td>279.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>250 sessions</td>
<td>57.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.5: \(nWnR\) scenario. Average client latency (in ms) when connected to a server with the given number of concurrent sessions (5 writers each) and rate of updates (per second).

The results show that a commonly available medium-sized server is able to support up to approximately 200 concurrent sessions, depending on the number of participants per session, the rate of updates sent per client, and the complexity of sessions’ models. Clients usually experience latencies of less than 200ms or even less than 50ms when the server is not too busy (assuming that they are colocated in a similar geographical region). The server’s main limitation is the amount of concurrent WebSocket connections it can handle rather than the processing power required to process incoming updates. Due to the low latencies, the collaborative features of the app are perceived as happening in real time. Consequently, both of the scenarios investigated as part of this case study can be classified as feasible.

\(^3\) though it is possible to receive WebSocket messages with a lower latency, because TCP might experience more preferential networking conditions than ICMP used by ping
6.3 Threats to Validity

This section describes the threats to the validity of this thesis. The internal threats include the lack of previous studies, the use of pseudorandom test data, the need to make use of simulated clients, and the instability of the experimentation environment. Whereas the external threats include inconsistent networking conditions, the unpredictable complexity of models, and the unique composition of each session.

The first limitation of this evaluation is the lack of previous studies in that area. Similar applications exist, but none of them combine mobile modeling with real-time collaboration. Due to the unexplored nature of that niche, a standard process for evaluating such systems is lacking. As a result, the means of a case study including an analysis of the solution's scalability and performance characteristics were chosen as an appropriate evaluation technique. Another limitation is the use of pseudorandom data. The TDG aims to produce updates for a model that is being edited and that is growing as a result. However, it does not reflect the way human beings would create a model. In order to produce more accurate results, a large amount of sample data (collected from the app used in practice) would need to be used either directly or indirectly by analyzing it to improve the TDG, an approach out of scope for this evaluation. Furthermore, clients and their edits had to be simulated because of the difficulty to emulate moderate to large numbers of mobile devices. These simulated clients might produce changes that differ from the changes the actual application would produce. These discrepancies, though minor, might have impacted the results. Finally, the instability of the experimentation environment needs to be mentioned as well. The experiments' results depend not only on the experiments' parameters, but also on external factors such as the available computing resources of the machine they were carried out on and the networking conditions at the time. Factors like these tend to change on a constant basis and might also have impacted the results.

When being used in practice, the system might experience unforeseen conditions that were not covered by this evaluation. One example is inconsistent networking conditions. Mobile devices might experience excellent conditions in one area, but high latencies or poor service coverage in others. This might have a significant impact on the real-time collaboration features. Another example is the unpredictable complexity of models created by human beings. Users might produce huge models over the course of several months that the system has to handle. Models of these proportions might cause the app or the server to exhibit poorer performance characteristics. The last example mentioned as part of this section is the unique composition of each session. A session might have two participants of which only one is editing, or thousands of participants all generating changes at the same time, or anything in between. In practice, it is impossible to predict the conditions of every possible session composition that will be experienced.

In summary, it can be stated that this evaluation has been subject to a number of limitations. The feasibility and characteristics of a specific session composition, model complexity, and network conditions needs to be evaluated on an individual basis.
7 Conclusion

Modeling is an important discipline within the field of software engineering. However, the tools have not kept up with recent developments such as distributed teams and working on mobile devices. Often, vital features such as collaboration or support for semantics are missing. Chapter 3 concluded that no application available today satisfies all of the requirements laid out in section 1.1, even though many of the required technologies do exist (c.f. chapter 2).

As part of this thesis, a concept for an application that fills the identified gap was developed (cf. chapter 4). Its main focus is support for real-time collaboration, a feature that no other semantics-aware modeling application supported. Chapter 5 described CoMod, a solution comprised of a cross-platform client application based on Flutter and a server executable based on Node.js. Its real-time collaboration capabilities are facilitated by the yjs framework and WebSocket connections. The solution was tested thoroughly and its potential for extension were explored. Subsequently, a case study was conducted to evaluate CoMod's scalability and performance characteristics (cf. chapter 6).

The following sections revisit the research question and the objectives set in section 4.1 and examine if they have been accomplished. Furthermore, some thoughts and suggestions for future work are outlined.

7.1 Fulfillment of Objectives

The solution presented in chapter 5 serves as a proof of concept that allows an affirmative answer to the main research question:

Yes, it is possible to build a mobile modeling application with real-time collaboration support.

Furthermore, all objectives set in section 1.3 have been accomplished. A concept for a mobile modeling application supporting real-time collaboration has been created (cf. chapter 4), implemented as a working solution that is ready to be used in practice (cf. chapter 5), and evaluated by means of a case study (cf. chapter 6). CoMod also fulfills the objectives listed in section 4.1:

- **UML class diagrams**: CoMod allows the creation and manipulation of UML class diagrams and their components.
• **Semantics**: Individual model components are recognized and changes are carried out in a way that keeps the model consistent at all times.

• **Availability**: It is available for Android and iOS devices.

• **Collaboration**: Users are able to start or join sessions to collaboratively edit a model. Conflicts are resolved automatically and do not require manual intervention.

• **Performance**: The app makes use of Flutter’s ahead-of-time code compilation and fast rendering engine. The client analysis (cf. section 6.2.2) showed that the app can handle about a dozen or more incoming updates each second without becoming sluggish on a two-year-old phone.

• **Usability**: The client application is designed to be intuitive. It only has three different screens (apart from the collaboration dialog) and displays model components as they would look like as part of a UML diagram (cf. section 5.1.1).

• **Offline support**: The app never requires an active internet connection. Models (even shared ones) can always be edited locally.

### 7.2 Future Work

This proof of concept shows that it is indeed possible for a mobile modeling application to support real-time collaboration. The system could be used in practice “as is”, e.g. as a tool for software development teams or computer science students’ group projects. However, there is still work to be done.

The client application could be extended to support additional platforms, such as the web, allowing users working on desktop computers to take part in collaboration sessions. As mentioned in section 5.5, the app could be extended to support a wider range of models (ER, CROM, etc.). This change could go along with a feature allowing models to evolve in phases, each representing a further refinement step. Furthermore, there is a lot of potential to improve the app’s performance. The JS subsystem could be instructed to process multiple updates before performing the expensive task of serializing the model. Ideally, the entire JS subsystem could be replaced by a native implementation of yjs.

The system as a whole also bears the potential for improvement. One aspect is enabling it to handle a large amount of concurrent sessions and participants. Section 4.4.2 describes how scalability could be supported by routing requests for each subset of sessions to a specific server instance. Another useful addition would be the implementation of a system of roles and permissions. It would allow model creators to assign roles such as editor or reader to participants and set the permissions they have accordingly.
Acronyms

API  Application Programming Interface. 10, 35, 36, 41, 43
AWS  Amazon Web Services. 16
CPU  Central Processing Unit. 6, 46, 47
CRDT  Conflict-Free Replicated Data Type. 2, 3, 12, 13, 24, 25, 29, 34, 37, 38, 40
CROM  Compartment Role Object Model. 43
CSS  Cascading Style Sheets. 10
E2E  end-to-end. 42
ER  Entity-Relationship. 43
GCP  Google Cloud Platform. 47
HTML  Hypertext Markup Language. 10
ICMP  Internet Control Message Protocol. 48
IDE  Integrated Development Environment. 18
JIT  just-in-time. 46
JS  JavaScript. 4, 10, 18, 19, 29, 34–37, 40–43, 45, 46, 51
JSON  JavaScript Object Notation. 13, 16
LWW  last writer wins. 12, 24, 34
MSON  Metamodel JavaScript Object Notation. 18
OMG  Object Management Group. 8
OT  Operational Transformation. 3, 12–14, 24
PDF  Portable Document Format. 16
PWA  Progressive Web App. 10, 18
SDK  Software Development Kit. 9, 10
TCP  Transmission Control Protocol. 48
TDG  test data generator. 45, 46, 49
TP2  transform property 2 (in the context of OT). 13, 14
UI  user interface. 6, 10, 11, 17, 19, 21, 22, 27, 29, 32–35, 37, 41–43
UML  Unified Modeling Language. 2, 6, 8, 9, 15, 16, 18–20, 22, 23, 25, 27, 30, 32, 33, 37, 38, 51
UUID  universally unique identifier. 30, 37
XMI  XML Metadata Interchange. 15
XML  Extensible Markup Language. 13, 16, 34, 35, 37, 41, 43
Bibliography


Statement of authorship

I hereby certify that I have authored this document entitled *Mobile Modeling with Real-Time Collaboration Support* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 10.01.2022

Max Härtwig