Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

ANDREAS MISCHKE
MASTER THESIS

REAL-TIME COLLABORATIVE QDA

Submitted on April 2, 2018

Supervisors: Prof. Dr. Dirk Riehle, M.B.A.
Andreas Kaufmann, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Bamberg, April 2, 2018

License

Bamberg, April 2, 2018

Abstract

QDAcity is a cloud-based web application that supports qualitative data analysis. Many research projects are executed by multiple researchers and therefore require some degree of collaboration for the qualitative data analysis process. QDAcity already aids this process by managing a single data source that can be accessed by multiple researchers. Still collaborating in QDAcity requires a separate communication channel outside of the application to coordinate the collaboration and avoid concurrent changes to data inside QDAcity. In this thesis an extension for QDAcity is implemented to augment the application with real-time collaboration capabilities. This enables researchers to work on a project and even single documents inside QDAcity at the same time without the need for external coordination.

Contents

List of Figures

List of Tables

1 Introduction

Successful research projects usually require collaboration between a group of researchers. While there are already different software tools that support the individual activities of qualitative research, especially the analysis of qualitative data, those tools are mostly desktop applications that do not aid collaboration and the required coordination.

With QDAcity the Professorship for Open Source Software at the Computer Science Department of the Friedrich-Alexander-Universität Erlangen-Nürnberg created a web application for qualitative data analysis that makes use of the extensive possibilities of cloud-based systems to enhance the collaboration process.

This thesis presents the design and architecture of a system that, for the first time, allows users of QDAcity to concurrently perform the task of coding on the same data, from any client running in a standard browser environment.

In Chapter 1 of this thesis an overview of fundamental topics is given to outline its motivation. In Chapter 2 the requirements for this project are defined before getting to the main Chapters 3 and 4 about architecture and implementation of the new software features. In the end the evaluation in Chapter 5 shows if, and to which extent, the requirements were met, before the thesis closes with a conclusion and a prospect on future work in Chapter 6.

1.1 Qualitative Research and Data Analysis

Qualitative specifies the type of data that is being examined in the research. The opposite is *quantitative* data and the two types can be differentiated as follows:

> "quantitative data – which are data in the form of numbers (or measurements), and qualitative data – which are data not in the form of numbers (most of the time, though not always, this means words)." (Punch, 2014, p. 3)

"Qualitative research is an umbrella term for a wide variety of approaches to and methods for the study of natural social life." Basis for those approaches are "primarily ... textual materials such as interview transcripts, field notes, and documents, and/or visual materials such as artifacts, photographs, video recordings and Internet sites" (Saldaña, 2011, pp. 3 sq.).

According to Flick (2014, p. 3), the central step in qualitative research is data analysis: "Whatever the data are, it is their analysis that, in a decisive way, forms the outcomes of the research". Within the research process *qualitative data analysis* (QDA) is often one step after the collection of data. Other approaches conduct the collection and analysis of the data in parallel or have the analysis as the central part of the research process, guiding the way of conducting the other steps (cf. Flick, 2014, pp. 9 sq.). Prominent example for the latter case is grounded theory, where the state of the analysis decides the methods for further data collection (cf. Corbin & Strauss, 2014).

A central concept in different strategies of QDA is to code the data. This means to assign labels to parts of the examined data (e.g. paragraphs, words, visual elements), so the data can be grouped under a common aspect. In the context of QDA those labels are called *codes*. A widely used strategy is qualitative content analysis (cf. Flick, 2014, p. 11). Schreier (2014, p. 174) describes the coding frame, also known as code system, as "the heart of" qualitative content analysis. It is a hierarchical set of codes consisting of main categories and subcategories. Main categories describe the aspects that shall be examined in the analysis while their subcategories group the details about the respective aspects.

By coding the data the researcher can identify similar phrases and common sequences across the analyzed document. Also relationships and differences between categories could become visible (cf. Miles, Huberman, & Saldaña, 2014, p. 10). These findings then help to form a theory for further work.

1.2 QDAcity

There are several software tools for *computer assisted qualitative data analysis* (CAQDAS) like MAXQDA[1] or atlas.ti[2]. Another instance is *QDAcity*, a cloud-based QDA software developed by the Professorship for Open Source Software at the Friedrich-Alexander-Universität Erlangen Nürnberg.

QDAcity is a web application running in the Standard Environment of Google App Engine[3]. The server-side component is written in Java and takes advantage

[1] https://www.maxqda.de/
[2] http://atlasti.com/
[3] https://cloud.google.com/appengine/

of the various APIs provided by Google App Engine. The frontend is a JavaScript application based on Facebook's React framework[4]. Users can authenticate via their Google account.

Researchers can use QDAcity to perform qualitative data analysis. It helps especially with managing and analyzing text documents like interview transcripts or field notes. Core feature is its coding editor that is used to manage a code system and a set of documents and to apply codes to the documents.

The code systems in QDAcity are rooted tree structures of codes where each code has different attributes such as name, memo or color. Additionally there is the possibility to define relations between codes in a code system language. Each code can be applied to any number of text document sections.

Every project inside QDAcity can contain multiple text documents, which can also be edited in the coding editor. This includes changes in the documents' content, i.e. adding and removing text, but usually the documents are created outside of QDAcity and imported as RTF document. The major task to be accomplished inside the application is the coding of documents.

Since multiple users can edit and code documents on the same project there is already a certain degree of collaboration provided by QDAcity. But in the current state of the application this must not happen simultaneously on the same document as concurrent changes to the same document between the editors would cause lost edits or even corrupted data. Therefore collaborators need to coordinate which documents to code to avoid any simultaneous edits. With the features introduced in this thesis, that need of external coordination becomes obsolete.

1.3 Real-Time Collaborative Editing

Collaborative editing is the act of working together on a piece of information. A simple example is a piece of paper that is handed back and forth with each party making changes on that paper. In terms of using computer systems the piece of paper would be replaced by a document file sent back and forth via email or on a physical drive. Consecutively editing the same item in an online service is also collaborative editing.

Real-time collaborative editing (RTCE) is a type of collaborative editing where data is concurrently changed by different editors. Adapting the paper example from above, this would mean that the collaborators draw concurrently on the same sheet of paper. The first example of RTCE in computer science was demonstrated by Douglas Engelbart in 1968 at the Fall Joint Computer Conference in San Francisco

[4]https://reactjs.org/

(Engelbart & English, 1968). A more prominent contemporary example is Google Docs[5], where users can simultaneously work on text documents, spreadsheets and presentation slides.

The main problem with RTCE is to detect and resolve conflicts between simultaneous edits. There have been different approaches to address it:

- With *operational transformation* peers modify received operations to accommodate previous operations.
- *Commutative replicated data types* can be applied in any order.
- *Differential synchronization* defines an algorithm specialized for web applications.

The single approaches are discussed in more detail in the following sections.

1.3.1 Operational Transformation

Ellis and Gibbs (1989) laid the foundation of *operational transformation* (OT). The aim was to add real-time collaboration capabilities to their GROVE[6] system.

To collaborate using OT, every collaborator's application instance starts with an identical local copy of the document to be edited. All changes are immediately reflected in the user's own copy while at the same time a description of the change is sent to all other connected users. The sent changes contain a position specification (e.g. a character offset) and the action to be performed at that position (e.g. an insertion or deletion of a character).

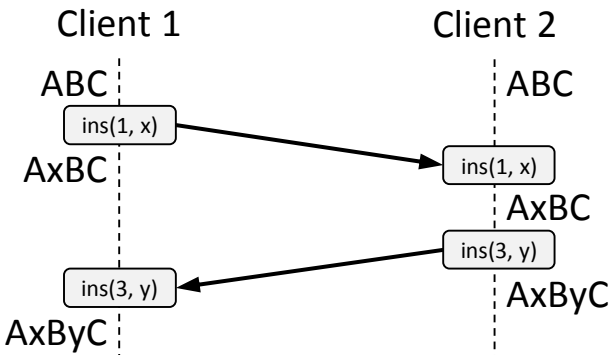

Figure 1.1: Operational transform with sequential changes (Based on Ellis and Gibbs, 1989, pp. 4 sqq.)

[5]https://www.google.de/docs/about/
[6]GRoup Outline Viewing Edit

If all changes would occur sequentially, the received changes could be executed unmodified and every collaborator would see the same document state as the others at any time. Figure 1.1 shows a simple sequential example where client 1 wants to insert an x between A and B while client 2 intends to add a y between B and C. Since the changes do not overlap, the result is as expected.

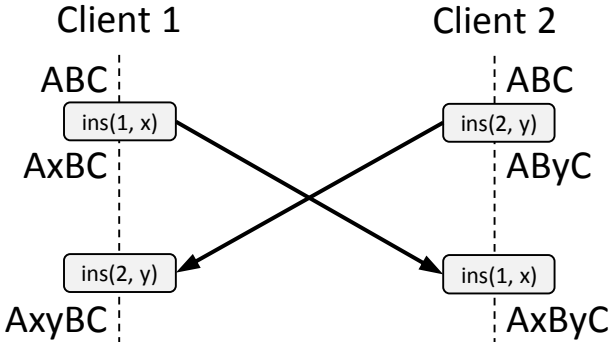

Figure 1.2: Operational transform with concurrent changes but no transformation (Based on Ellis and Gibbs, 1989, pp. 4 sqq.)

In practice there is always the possibility for changes happening at the same time. Figure 1.2 illustrates the result when the same activities as in the previous example are executed concurrently: While client 2 sees the correct result, client 1 obtains a wrong version of the document. The insertion of the x leads to new offsets for B and C. Therefore, the offset of the change received by client 2 now points to the location between x and B, instead of between B and C

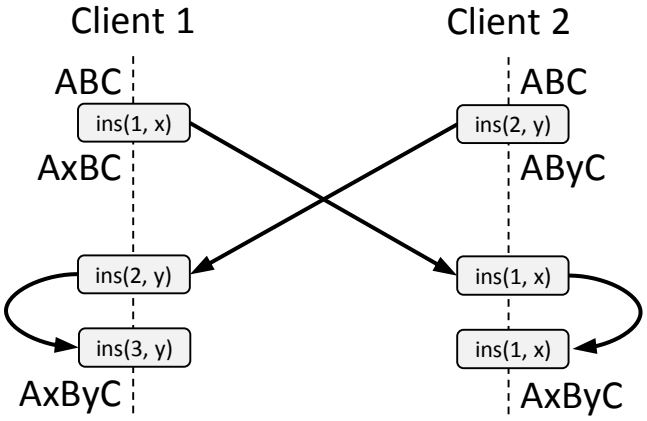

Figure 1.3: Operational transform with concurrent changes and transformation (Based on Ellis and Gibbs, 1989, pp. 4 sqq.)

The idea of OT is to transform the received changes to correct the position and other potentially affected parameters based on previously executed changes. Figure 1.3 extends the above example by a transformation step that corrects the offset for received changes. At client 1 the insertion of x requires the offsets of all later changes that point to a position right of the x (including the insertion of y)

to be incremented by 1. At client 2 no correction is necessary, since the insertion of x has a smaller character offset than the already executed insertion of y.

This solution to transform the changes according to previous changes is also a downside when using OT for more complex systems. For all possible combinations of changes a transformation has to be defined, leading to a $m \times m$ matrix with m being the number of possible changes.

1.3.2 Commutative Replicated Data Types

An approach similar to operational transformation are *commutative replicated data types* (CRDT). As with OT, all collaborators' systems start off with an identical local copy of the data and then exchange operation messages. While with OT the messages are transformed to take previous changes into account, Roh, Kim, and Lee (2006) came up with CRDT to make the application of changes independent from their order.

An exemplary data type is the increment-only counter described at Shapiro, Preguiça, Baquero, and Zawirski (2011, p. 394): Given a system with n clients, a vector of length n contains one counter per client. Only the respective client may increment the value. After incrementing, clients do not broadcast an *increment* message, but the new counter value. When receiving a change, it can always be executed in any order. In case a change message overtakes an other, the first one will be ignored since it is less than the second and the counter is monotonous increasing. Even in case of lost messages, each client will eventually receive the latest values. To get the total value of the counter a client can anytime sum up the single counters of the vector.

Downside of the CRDT approach is, that an application can only use commutative data types for synchronization. This is no problem for greenfield applications but very hard for adaption in existing applications.

1.3.3 Differential Synchronization

Differential synchronization (DS) is a method developed by Fraser (2009). The aim was to introduce a system that allows to add real-time collaboration capabilities with minimal impact on application design and thereby also suitable for being added to existing applications. "DS is a state-based optimistic synchronization algorithm" (Fraser, 2009, p. 1) that avoids the drawbacks of other approaches:

- Pessimistic approaches lock sections or the whole document, so there is no real-time collaboration possible. Further the pessimistic model is unsuitable for unreliable network connections.

- Edit-based algorithms like operational transformation (cf. section 1.3.1) mirror all single user actions over the network. Any failure that leads to an action not being mirrored causes a fork between the clients' document history from which the system has to recover.
- Systems relying on three-way merges send the whole document to the server which then merges the changes of all users into a single version and distributes that version back to all clients. The drawback of this approach is, that there must not be any changes between sending the client's version to the server and receiving the merged version. This slows users down a lot as they repeatedly have to stop their work and wait for the merge.

DS relies on a client-server-architecture, where the server keeps track of all connected clients and their document states. Fraser (2009) includes a diff-patch algorithm for plain text, but DS can be applied to all kind of data for which a suitable diff-patch algorithm exists. Figure 1.4 depicts the schematic structure that is built per client-server-connection.

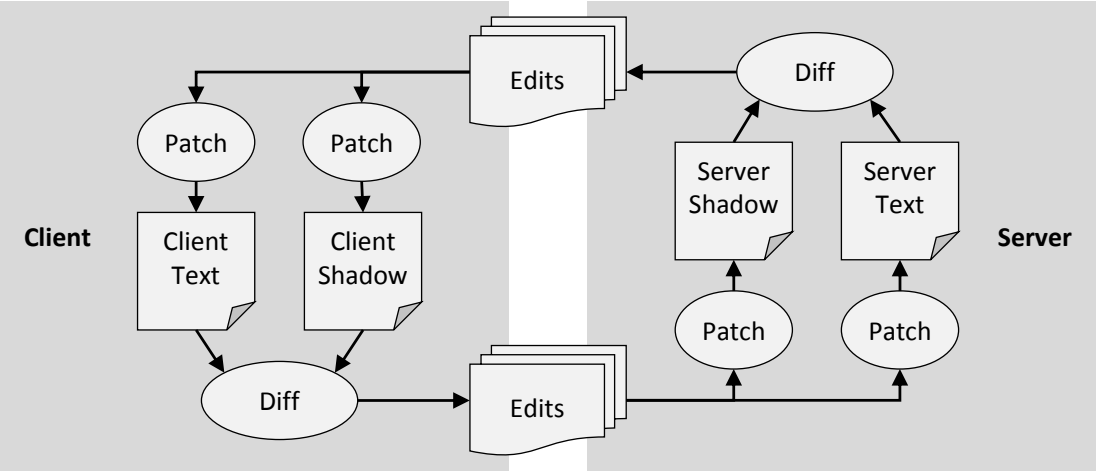

Figure 1.4: Schematic overview of differential synchronization
(Based on Fraser, 2009, p. 3)

Initially all the text and shadow instances are identical. The server has a single *Server Text* instance but manages multiple *Server Shadows*, one for each connected client. The user's actions are immediately performed on the *Client Text* and the *Server Text* may be changed by other clients in the meantime. The synchronization between one client and the server runs in cycles that do not overlap:

1. A diff is calculated between the *Client Text* and the *Client Shadow* resulting in a list of edits.
2. The *Client Text* is copied over to the *Client Shadow*.
3. The edits from step 1 are sent to the server.

4. The server calculates patches based on the client edits and applies them to the *Server Shadow* and the *Server Text*.

These steps are then repeated in the opposite direction: A diff between the *Server Text* and the *Server Shadow* reveals the edits that were made by other clients, which in return get incorporated into the *Client Text* and *Client Shadow*. After that a new synchronization cycle starts with step 1 of the above list.

1.3.4 Discussion of the Presented Approaches

In the last three sections different approaches have been presented, that address the problem of resolving or avoiding conflicts occuring with concurrent changes in distributed systems.

Operational transformation and commutative replicated data types are both very interesting solutions to the problem, but both have the disadvantage of being hard to add to existing applications. When choosing Operational Transformation the big number of different actions that can be performed in QDAcity would result in a vast amount of transformations to be implemented. Adding commutative replicated data types to QDAcity would require the reimplementation of many existing parts that work on data to be synchronized between users.

One of the design goals of differential synchronization was to have minimal impact on the application design and to be easily adoptable in existing applications. This is a great advantage over the other two approaches for the implementation in QDAcity. To apply differential synchronization a diff-patch algorithm has to be provided for data which shall be synchronized. This is expected to be a feasible task that can be managed within this thesis.

2 Requirements

This work's objective is to design and implement a *real-time collaboration service* (RTCS) for the existing QDAcity application. That service shall enable users to collaborate on documents concurrently while seeing each other's changes in real-time. It covers only changes made inside the coding editor of QDAcity, the synchronization of actions performed outside of the coding editor is not in the scope of this thesis.

2.1 Functional Requirements

The structure of the following functional requirements is based on the FunctionalMASTeR template by Rupp (2014, pp. 215 sqq.) as shown in Figure 2.1. Italic words are placeholders that have to be filled in, while dashed boxes are optional. The semantic definition of the keywords *shall*, *should* and *will* as used in this thesis can be found in Table 2.1. The used process words and terms are semantically defined in Tables 2.2 and 2.3.

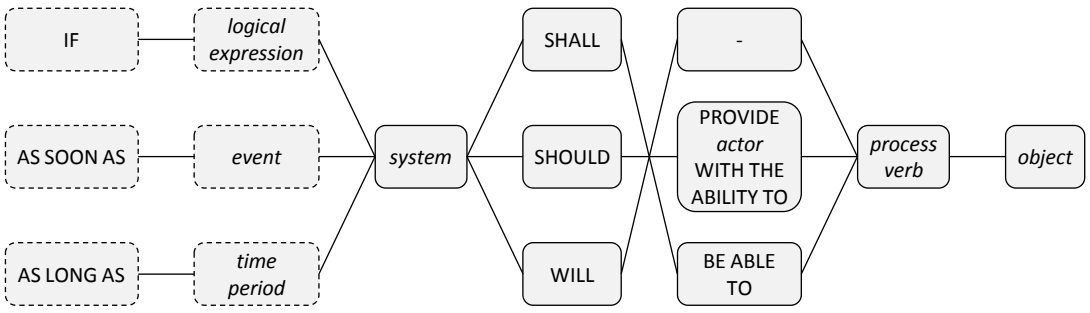

Figure 2.1: FunctionalMASTeR template based on Rupp (2014, pp. 230, 242)

Keyword	Semantic Definition of the Keyword
shall	A requirement that has to be fulfilled for the project to succeed.
should	A requirement that is important but not necessary for the software to work correctly.
will	A requirement that is not necessary but desired.

Table 2.1: Definition of the requirement keywords

Process	Semantic Definition of the Process Word
to persist	In the RTCS, *to persist* shall be defined as storing a data item in the data store of the backend by writing directly to the data store or using the backend API.
to synchronize	In the RTCS, *to synchronize* shall be defined as sending a change to all collaborators, so all collaborators' document copies are eventually consistent.

Table 2.2: Definition of the requirement process words

Term	Semantic Definition of the Term
backend	In the RTCS, a *backend* shall be defined as the existing backend of the QDAcity application.
change	In the RTCS, a *change* shall be defined as a user action and the corresponding meta information to describe the action.
collaborator	In the RTCS, a *collaborator* shall be defined as a QDAcity user who is working concurrently in the coding editor on the same QDAcity project.
document	In the RTCS, a *document* shall be defined as a document that is loaded into and edited inside the QDAcity application.
RTCS	In the RTCS, a *RTCS* shall be defined as the real-time collaboration service that is implemented in the scope of this thesis.
user	In the RTCS, a *user* shall be defined as a physical person that is using the QDAcity application.

Table 2.3: Definition of the requirement terms

2.1.1 Synchronization of Codings

Central feature of the QDAcity coding editor is the possibility to add and remove codings to or from a document. These are also the main actions that have to be synchronized by the RTCS. Therefore the following requirements can be defined:

FR–1:	As soon as a user adds a coding to a document the RTCS shall persist the change in the backend.

FR–2:	As soon as a user adds a coding to a document the RTCS shall synchronize the change to all collaborators.

FR–3:	As soon as a user removes a coding from a document the RTCS shall persist the change in the backend.

FR–4:	As soon as a user removes a coding from a document the RTCS shall synchronize the change to all collaborators.

2.1.2 Synchronization of Codes

Another set of essential actions to be synchronized are changes that apply to single codes inside the code system or the relation between two or more codes:

- adding a code,
- relocating a code, i.e. assigning a new parent code in the hierarchy,
- changing the code book entry of a code,
- changing a code's properties: name, color or author,
- adding a relation between two codes,
- changing a relation between two codes,
- removing one or all relations of a code,
- removing a code.

Besides adding and removing a code, all actions can be seen as changing one or more properties of a code. On this basis, these requirements can be defined:

FR–5:	As soon as a user adds a code to the code system the RTCS shall persist the change in the backend.

FR–6:	As soon as a user adds a code to the code system the RTCS shall synchronize the change to all collaborators.

FR–7:	As soon a user changes one or more properties of a code the RTCS shall persist the change in the backend.

FR–8:	As soon a user changes one or more properties of a code the RTCS shall synchronize the change to all collaborators.

FR–9:	As soon as a user removes a code from the code system the RTCS shall persist the change in the backend.

FR–10:	As soon as a user removes a code from the code system the RTCS shall synchronize the change to all collaborators.

2.1.3 Synchronization of Collaborators

Users need to recognize that they are not the only user in a document but have collaborators changing the same document concurrently. To achieve that, the following requirements are defined:

FR–11:	The RTCS shall display a list of all collaborators in the QDAcity coding editor interface.

FR–12:	As long as there are no collaborators the RTCS shall display a corresponding message instead of the collaborator list.

FR–13:	As soon as a user enters the QDAcity coding editor the RTCS shall add this user to all collaborators' collaborator lists.

FR–14:	As soon as a user intentionally leaves the QDAcity coding editor the RTCS shall remove this user from all collaborators' collaborator lists.

FR–15:	As soon as a user looses connection to the RTCS server the RTCS shall remove this user from all collaborators' collaborator lists.

2.1.4 Synchronization of Text Changes

Though editing text is possible within the QDAcity coding editor, the average user is expected to spend significantly more time with coding activities than with text editing activities. Usually documents are written outside QDAcity and imported once. After initial corrections, changes to the documents' content are not a common occurrence during coding. Therefore synchronizing text changes is assigned a low priority.

FR–16:	As soon as a user adds a character to a document the RTCS should synchronize the change to all collaborators.

FR–17:	If a user has added a character to a document the RTCS should eventually persist the change in the backend.

FR–18:	As soon as a user changes the styling of a character of a document the RTCS should synchronize the change to all collaborators.

FR–19:	If a user has changed the styling of a character of a document the RTCS should eventually persist the change in the backend.

FR–20:	As soon as a user removes a character from a document the RTCS should synchronize the change to all collaborators.

FR–21:	If a user has removed a character from a document the RTCS should eventually persist the change in the backend.

2.1.5 Synchronization of Text Selections

To make real-time collaboration in QDAcity more convenient, each user's text selection inside the document should be synchronized to the other editors. This helps users to identify sections where other users are actively working on, so they can avoid conflicts or duplicate work by not editing the same section.

FR–22:	As soon as a user selects text in a document the RTCS should display the selection in all collaborators' text editors.

FR–23:	As soon as a user deselects text in a document the RTCS should display the selection in all collaborators' text editors.

2.1.6 Authentication and Authorization

Authentication and authorization is already implemented in the frontend and the backend of QDAcity. Since the RTCS will accept changes and modify the backend copy of a document, it has to ensure that only changes by authorized users are accepted.

FR–24:	The authentication mechanism of the RTCS shall only allow authenticated users to make changes to a document.

2.2 Non-Functional Requirements

The formulation of the non-functional requirements is based on the PropertyMASTeR and the EnvironmentMASTeR templates by Rupp (2014, pp. 234 sqq.) as shown in Figures 2.2 and 2.3. As in Figure 2.1, the parts in dashed boxes are optional and the italic words have to be filled in.

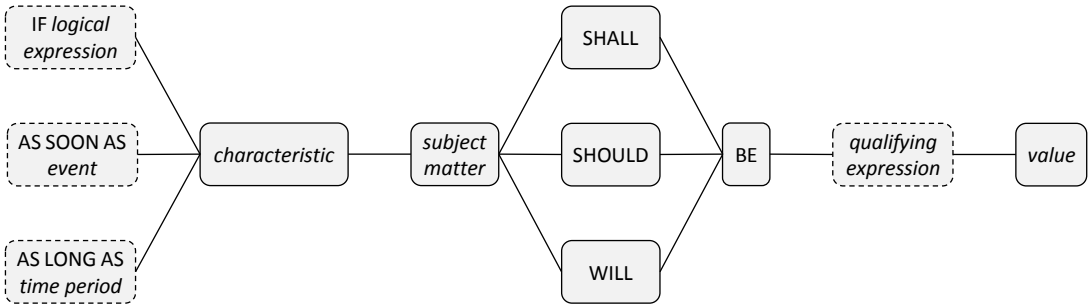

Figure 2.2: PropertyMASTeR template based on Rupp (2014, p. 239)

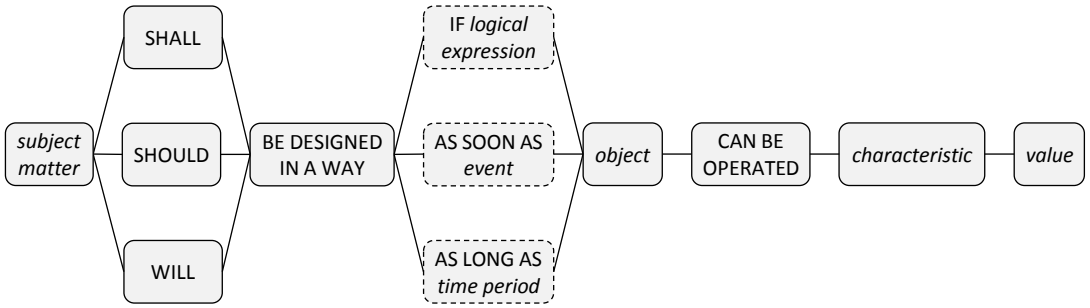

Figure 2.3: EnvironmentMASTeR template based on Rupp (2014, p. 239)

2.2.1 Technological Requirements

The nature of the existing application induces some technological requirements for the RTCS. Since QDAcity is a web-based client-server application hosted on Google App Engine, the RTCS should run in the same ecosystem.

NFR–1:	The architecture of the RTCS shall be a client-server architecture.

NFR–2:	The client part of the RTCS shall be designed in a way that it can be operated as a part of the existing QDAcity frontend.

NFR–3:	The server part of the RTCS should be designed in a way that its running instances can be managed via the Google Cloud Console.

NFR–4:	The server part of the RTCS should be designed in a way that it can be operated in a distributed way on multiple instances.

2.2.2 User Interface Requirements

The user should not experience any delay when editing the document locally. A delay when incorporating changes of other users is inevitable due to the usage of remote network connections, but should stay as low as possible.

For similar use cases, Tolia, Andersen, and Satyanarayanan (2006, p. 47) ascertained that "user productivity is not impacted by response times below 150 milliseconds. ... Above one second, users become unhappy". Therefore theses limits are targeted for the synchronization tasks in the RTCS.

NFR–5:	The RTCS should be designed in a way that it can continue operation after the synchronization of a change has failed.

NFR–6:	The RTCS should be designed in a way that the synchronization of changes is transparent to the user.

NFR–7:	The synchronization time of the RTCS should be below 150 milliseconds.

NFR–8:	The synchronization time of the RTCS shall be below 1 second.

3 Architecture

Based on the requirements in Chapter 2 this chapter presents the architecture of the RTCS. The first section defines the fundamental architecture, before the second section elaborates on some aspects of the architecture in more detail. Finally the replacement of the currently used text editor component is described and reasoned.

3.1 General Architecture

The RTCS needs to distribute changes between several users across the internet, so it is obvious to have some centralized server as communication hub. At the same time a client module is needed, that connects the QDAcity frontend to this central RTCS server. This leads to a client-server architecture.

In this section the fundamental design decisions are described, that follow on the architectural decision reasoned above. The section starts with the choice of the programming language and the RTCS-internal client-server-protocol. Based on those choices the selection of the server runtime environment and a shared in-memory database is reasoned before summarizing the section in an overview of the involved systems and the connections between them.

3.1.1 Selection of the Programming Language

In many client-server applications there is a clear separation between operations performed by the client and operations performed by the server. Often the client is focused on displaying data and accepting user input, while the server handles the business logic and persistence of the data. Or in other cases the business logic is also moved to the client while the server is confined to the data persistence. In contrast both the client module and the server of the RTCS have to execute similar operations on the data, e.g. applying changes to a document.

This leads to the consideration of using a common programming language for both the server and the client. Since the choice of client side programming languages is limited to JavaScript there is no alternative than using JavaScript for the server side as well. Using a common language allows to implement a shared library for data manipulation to be used on client and server side. This has several benefits:

- **Reduced implementation time**. Implementing and reusing a single library saves a considerable amount of time. This applies not only to the initial implementation but to every future addition of new features or bug fixing.
- **Identical handling of data**. Since the sending and receiving side are identical, the chance of having errors by differently handling the data sent back and forth is reduced.
- **Increased maintainability**. When enhancing the application with new features or when fixing errors in the application this only has to be done in one single place. That reduces work on the one hand and on the other hand prevents errors by modifying only one part of the communication structures.

Using JavaScript for the implementation of the RTCS server has also drawbacks, namely the limitation of options for choosing a server run time (cf. Section 3.1.3). Still the benefits outweigh this limitation, so JavaScript is selected as the programming language for the RTCS server.

3.1.2 Client-Server Communication

The RTCS needs to notify connected clients about changes on documents or the code system. Conventional HTTP requests are always initiated by the client, so the server cannot independently send messages to a client, but only respond to client requests. There exist solutions based on HTTP, namely *HTTP long polling* and *HTTP streaming* (cf. Loreto, Saint-Andre, Salsano, & Wilkins, 2011), both requiring "an abuse of HTTP to poll the server for updates" (Fette & Melnikov, 2011). HTTP long polling uses consecutive requests that wait for a delayed server response or a request timeout, whatever may occur first. HTTP streaming also starts with a request by the client while the server response is kept open so the server can append data to the stream at any time.

"The *WebSocket* Protocol is designed to supersede existing bidirectional communication technologies that use HTTP as a transport layer" (Fette & Melnikov, 2011). In contrast to standard HTTP requests a single TCP connection is used for full-duplex communication. This way the server saves resources by only maintaining one instead of two or more TCP connections per client. Also the protocol overhead when sending small messages is reduced as there is no HTTP header

needed on every message like it would be the case with standard HTTP requests. This helps on keeping the network latency and thereby the perceived response time low for the user.

Within this project socket.io[1] is used, an open-source JavaScript framework that provides an abstraction layer on top of WebSockets. To have a concise wording in the following sections, the data units sent from a RTCS client to a RTCS server are called *messages*, while *events* denote data units sent from RTCS servers to RTCS clients.

3.1.3 Server Runtime Environment

QDAcity is completely hosted on Google App Engine. To simplify monitoring and administration of all application parts, i.e. to have a single place for logging output and to manage all systems in a single dashboard, it's sensible to have the RTCS running in the Google Cloud Platform[2].

First choice would be to also use Google App Engine. There are two different environments, the Standard Environment and the Flexible Environment[3]. JavaScript is not supported in the Standard Environment, so this is not an option. The Flexible Environment does support JavaScript, but is also not usable for this project, since it does not support WebSockets.

The more flexible but less managed alternative inside the Google Cloud Platform is the Google Compute Engine[4]. Google's Infrastructure as a Service component provides the option to launch fully controllable virtual machines. As with the Google App Engine Flexible Environment it is possible to launch multiple instances of the same service and configure load balancing between those instances.

3.1.4 Handling of Shared Data

To support load-balancing at application level any shared data has to be handled in particular. QDAcity application data should be kept in the existing backend and only loaded for processing tasks and written back afterwards. Metadata that is needed across RTCS instances for the purpose of synchronizing collaborators and similar tasks is stored in an in-memory database.

[1] https://socket.io/
[2] https://cloud.google.com/
[3] https://cloud.google.com/appengine/docs/the-appengine-environments
[4] https://cloud.google.com/compute/

Two options for in-memory databases are considered: Memcached[5] and Redis[6]. Memcached is provided by Google as part of the App Engine Standard Environment. Since the Standard Environment is not usable for the RTCS, there is no advantage in using Memcached. The benefit of using Redis is that for socket.io there is a Redis-adapter[7] provided by the socket.io contributors. The adapter is needed for socket.io to broadcast messages to distributed socket.io server nodes. As there is no official Memcached adapter for socket.io, using Memcached would cause additional implementation effort within the scope of this thesis. The small advantage of Redis leads to the decision, to use this in-memory database not only for socket.io but for other shared application data as well.

3.1.5 System Architecture

Summarizing the previously depicted decisions, an overview of the architecture of the RTCS can be drawn. Figure 3.1 shows this overview with an emphasis on the position of the RTCS in the existing QDAcity application. Before the introduction of the RTCS the QDAcity frontend and backend communicated solely via the Google Cloud Endpoints[8]. Data that does not need to be synchronized will further on be transferred directly between the QDAcity frontend and backend. For data that has to be synchronized the RTCS server will act as a proxy between the two parties. As introduced in Section 3.1.2 the communication between QDAcity frontend and RTCS server is bidirectional and has to be implemented as WebSocket connection, while the RTCS uses the existing endpoints at the QDAcity backend.

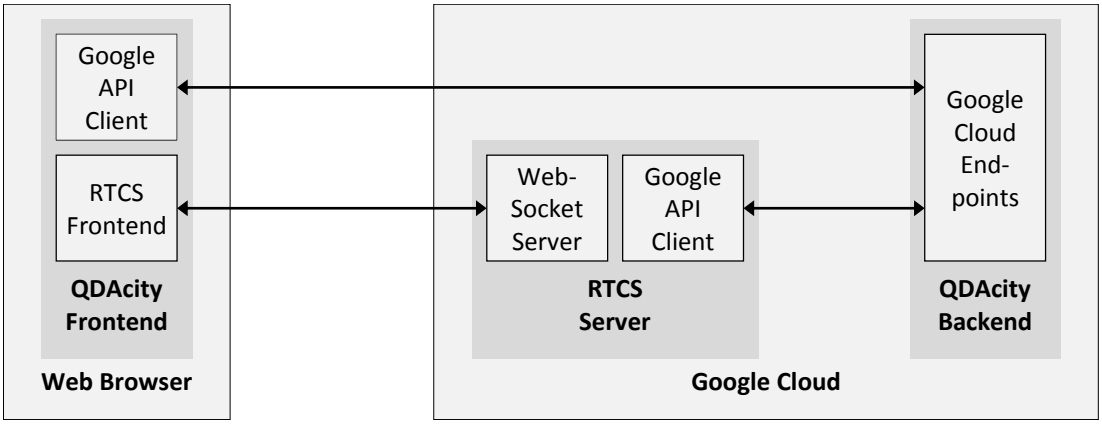

Figure 3.1: RTCS system architecture

[5]https://www.memcached.org/
[6]https://redis.io/
[7]https://github.com/socketio/socket.io-redis
[8]https://cloud.google.com/endpoints/

3.2 Architecture Details

This section elaborates on some important details of the architecture. It starts with a description of the authentication forwarding and its included backend configuration functionality. The second part refines the requirements and the design for two user interface elements that are introduced with the RTCS. In the last part a protocol enhancement is explained that optimizes the perceived system response time and therefore the usability of the application.

3.2.1 Authentication Forwarding and Backend Configuration

As defined in FR-24 (cf. Section 2.1.6) only authorized users may be able to access QDAcity backend endpoints. In the frontend, users authenticate via their Google account. Since the RTCS server acts like a usual client towards the backend, it also needs to provide authentication details. In order to ensure accountability of the single changes on application data, the RTCS has to authenticate as the respective user that initiated the change.

To achieve this the RTCS client sends the user's QDAcity backend authorization token to the RTCS server, which then instantiates a new Google API client on behalf of the user. This way the QDAcity backend can attribute every request sent via this API client instance to the corresponding user.

As further step, the RTCS client sends backend connection details to the RTCS server that have been preconfigured in the QDAcity client. This allows the use of a single RTCS server instance for multiple QDAcity application instances. This primarily benefits the QDAcity development process where every contributor runs his own free QDAcity instance (frontend and backend) on Google App Engine. By configuring a central RTCS server the contributors do not have to create and manage a separated charged Google Compute Engine instance to run a separate RTCS server.

3.2.2 User Interface Elements

The synchronization of changes happens transparent to the user, so there are no additional user interface elements needed for the synchronization of changes. But the user needs to know if and which other users are concurrently working on the same project in QDAcity. To achieve that, a user interface has to be designed.

There will be two types of lists showing other collaborators. The first will be used exactly once to list the users that are active on the same project and have the QDAcity editor opened. The other list type will be instantiated per document and show, which collaborators have that document selected in their QDAcity coding editor.

If users open multiple instances of QDAcity, e.g. in different tabs or browsers, they should appear only once in the project collaborator list, and once in the collaborator list of every document, they have opened in any of their QDAcity instance. This way other collaborators can see, which documents that user might change actively.

3.2.3 Optimistic Execution

For the synchronization of changes a communication round-trip takes place, i.e. the user input triggers a message to the RTCS server, where the change is processed, resulting in the distribution of an event to all connected clients. It became apparent that for adding and removing codings to or from documents this round-trip takes substantially longer than the targeted 150 milliseconds. Even the upper limit of 1 second could not be satisfied with round-trips taking about 1.5 seconds on average. (Cf. NFR-7, NFR-8 in Section 2.2.2).

Since this duration is not acceptable from a usability perspective, additional actions have to be taken. A more detailed examination of the time spent per server-side task showed that most of the tasks cannot be accelerated without upgrading the underlying (virtual) hardware. Most time is spent for read and write requests to the Redis server and the QDAcity backend. Upgrading these services is not an option because of unproportionally increased costs.

A simpler approach is optimistic execution. Instead of waiting for the synchronization round-trip to complete, user actions are immediately applied to the current document state. In normal operation, the states of the local document and the server side document state are consistent after the round-trip has succeeded. In case of an error during the synchronization, the change to the user's local document state is reverted to reach consistency of the document states of client and server. This way the perceived response time can be minimized to several milliseconds.

3.3 Replacement of the Editor Component

Before this project, Squire[9] was used as text editor library for QDAcity. While analyzing the existing QDAcity application it became obvious that this text editor is not suitable for the synchronization of changes via the RTCS. Main problem is the complex access to the editor's data and changes in both ways, i.e. it is hard to retrieve distinct changes that have been performed, as well as to apply changes to update the editor state. Since Squire also does not integrate with the React framework very well, it has to be replaced by another editor library. The decision process and the two considered alternatives, Draft.js[10] and Slate[11] are described in this section.

The first replacement option for Squire is Draft.js, a very popular text editor library for React, published by Facebook. After first attempts to configure the editor, it turned out to be very inflexible. Draft.js defines a fixed set of inline styles, that can be applied to the edited text, like italic, code or strike-through. For the use in QDAcity it is essential to apply metadata to a range of text that defines the existence of a coding and its details, e.g. unique id, color or author, but that can not be realized with Draft.js. Further the library does not provide a possibility to access the single changing actions executed on the document. Because of these two disadvantages Draft.js cannot be used for QDAcity.

Slate is the other option for replacing Squire. It is less popular than Draft.js and not supported by a big company like Facebook. Still the number of 145 contributors and 6,911 stars on GitHub[12] make Slate a viable option for QDAcity. The author describes it as "completely customizable framework for building rich text editors." (Taylor, 2018). Since it is also designed to work with React and is more easily adaptable to the needs of QDAcity, Slate is used as the application's new text editor component. The Slate data types and their structure are described hereafter.

Documents in QDAcity consist of paragraphs that contain only characters, but no further sub elements like lists, tables, images, etc. Characters can be formatted with the basic styles bold, italic, underline and can have a specific font size and font family. Slate supports more complex documents but since the complexity is not needed for the QDAcity editor, the following description covers only the objects and properties used in this project.

[9] http://neilj.github.io/Squire/
[10] https://draftjs.org/
[11] https://docs.slatejs.org/
[12] https://github.com/ianstormtaylor/slate (Accessed on 2018-03-24)

Most Slate object types are based on types of Facebook's library Immutable[13], i.e. every modifying method being called on an immutable object returns another immutable copy of the previous object with the applied modifications. Top-level objects that represent the Slate editor state as a whole are of type `Value`. A `Value` contains a `Document` object and the current text selection as object of type `Range`. A `Document` is hierarchically composed of sub elements to represent the document that is loaded into the editor. It contains a list of `Block` objects, which correspond to a document's paragraphs. Each `Block` consists of one or more `Text` objects. Each `Text` again has a list of `Character` objects with each `Character` optionally having one or more `Mark` objects attached.

`Mark` objects are distinguished by their `type` property and are used to represent the basic styles mentioned above. E.g. when a user selects a word and sets it to bold, each `Character` object in the selection gets a mark with `type='bold'` attached. A `Mark` can store arbitrary metadata, so this type of objects will also be used to attach font size and font family information to a text ranges. The most essential information that must be stored in QDAcity documents are codings attached to ranges of characters. With `Mark` objects any number of different codings can be attached to each character of the document.

Other relevant Slate object types are `Range`, `Change` and `Operation`. The `Range` type is used to describe a range of characters in the document, e.g. the user's current text selection as described above. A `Change` is Slate's representation for a set of modifications based on a specific `Value`. After getting a new `Change` object from an immutable `Value`, actions like `insertTextAtRange(Range, string)`, `removeMarkAtRange(Range, Mark)` or `selectAll()` can be applied to the `Change`. `Change` objects are the only Slate type that are mutable, i.e. executing any actions like the ones mentioned above alters the `Change` object directly. In the end a new immutable `Value` can be obtained from the `Change` object and be applied to the editor. `Operation` objects represent a single change in the Slate environment, like the addition of a mark to a range or the insertion of a single character. They are mainly used internally in the Slate library but may also be created and read from outside. For that purposes the `Change` type offers the method `applyOperation(Operation)` and the property `operations`.

[13] https://facebook.github.io/immutable-js/

4 Implementation

This chapter describes the implementation of the most important classes and components of the RTCS, starting with the client side parts, followed by the server side. The last section describes the shared library that is used on both the client and server side.

4.1 RTCS Client

In this section the client side components are described. First an outline of the `SyncService` class is given, followed by a distinction of the UI components that display the collaborators per project or document. In the end an overview of the React components, that had to be adapted during the implementation of the RTCS client, is given.

4.1.1 SyncService Class

The main entry point to the RTCS client module is a class called `SyncService`. As the WebSocket communication is bidirectional, also the communication between the RTCS module and the other frontend components has to be bidirectional. Messages from other components to the RTCS module are sent by calling methods of the `SyncService`. To receive data in the other direction, frontend components can listen to events being fired by the `SyncService`.

When a user triggers an action that has to be synchronized, the processing React component calls a method on the `SyncService` which sends the corresponding message to the RTCS server. Optional acknowledgments to the message are then returned to the calling React component. When an event is received by the RTCS server, the `SyncService` calls all event listeners that were registered by calling `on(String, Function)` on the `SyncService`.

The `SyncService` class has two sub components, each handling the communication for a single logical group of changes. The `CodesService` handles messages and events regarding changes on codes, while the `DocumentService` handles the communication about adding and removing codings to or from documents. Messages and events regarding updates of user data is handled directly by the `SyncService`, since these data are strongly connected to the lifetime of the WebSocket connection, e.g. the data has to be sent as soon as possible but not before the WebSocket connection has been established or when the connection has been reestablished after a connection abort. Table 4.1 gives an overview of all messages and events and the handling classes on both client and server side. The corresponding server side classes are described in Section 4.2.

Message	**Event**	**Client Class**	**Server Class**
user.update	user.updated	`SyncService`	`Socket`
code.insert	code.inserted	`CodesService`	`CodesHandler`
code.relocate	code.relocated	`CodesService`	`CodesHandler`
code.remove	code.removed	`CodesService`	`CodesHandler`
code.update	code.updated	`CodesService`	`CodesHandler`
coding.add	coding.added	`DocumentService`	`DocumentHandler`
coding.remove	coding.removed	`DocumentService`	`DocumentHandler`

Table 4.1: Messages and events used in the RTCS

When the user enters the coding editor of the QDAcity application the `SyncService` is initialized, connects to the RTCS server and starts to listen for incoming events. As soon as the user leaves the coding editor or the whole QDAcity application the WebSocket connection is closed.

4.1.2 Collaborator Lists

There are two React components to display collaborator lists in the user interface of the QDAcity coding editor. The `CollaboratorList` component renders the list of collaborators on the same project, while the `CollaboratorBubbles` component is used to list the collaborators that have a specific document opened. Both components are quite similar but differ in small details. They share the general design of small circles containing the profile picture of the respective user, with an appended white area for the user's full name that is shown when the interface is hovered.

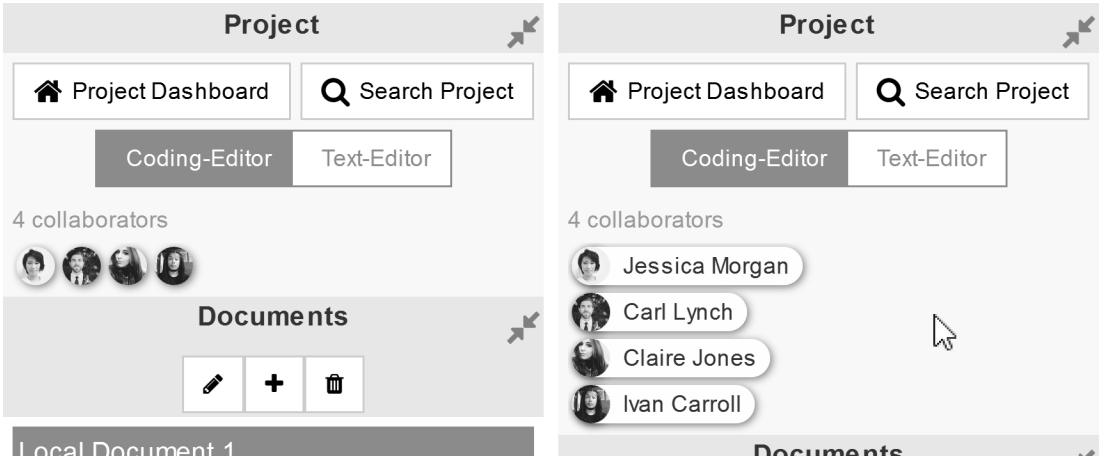

Figure 4.1: Screenshot of the CollaboratorList, on the left in normal state, on the right while hovering with the mouse.

The `CollaboratorList` component (cf. Figure 4.1) is adjusted to utilize the space of the sidebar's project panel, showing all user circles aligned in rows. When hovering the rows, the `CollaboratorList` transforms into a vertical one-column list showing all users and their full names.

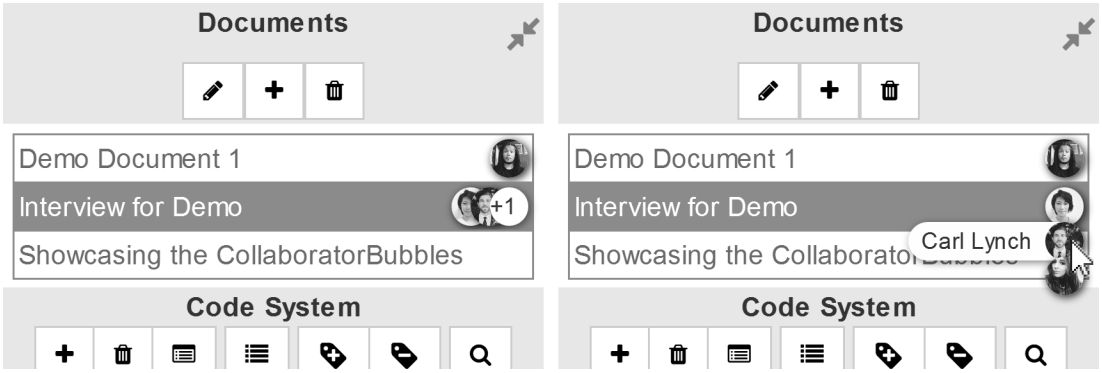

Figure 4.2: Screenshot of the CollaboratorBubbles, on the left in normal state, on the right while hovering with the mouse.

The `CollaboratorBubbles` component (cf. Figure 4.2) in contrast has a more condensed style with slightly overlapping horizontally aligned circles, to occupy as few space of the document title area as possible. It only shows the first three users on the document on first sight. If there are more than three users actively working on the document, the first two are displayed directly with the third circle displaying the remaining count of document collaborators. When hovering the circles, they transform into a vertical list, showing all users for that document and their full names.

4.1.3 Adapted React Components

Some of the already existing React components have to be adapted during the implementation of the RTCS. In particular the `CodingEditor` and `DocumentsView` experience some changes while the `TextEditor` is completely rewritten. To use the `CodingBrackets` component further on, a wrapper component is introduced to convert the data.

`CodingEditor` Class

Since the coding editor delimits the scope of components that might connect to the `SyncService` and thereby the RTCS, some managing tasks are added to the responsibility of the `CodingEditor` component. It initializes the single `SyncService` instance that is used in all child components and sends updates about the basic user data to the `SyncService`.

`DocumentsView` Class

Before the introduction of the RTCS the `DocumentsView` was responsible for sending changed documents to the backend when a coding was applied or removed. This functionality has to be adapted, so the information about the actions of applying and removing is sent to the RTCS server instead.

When applying a coding, the `DocumentsView` computes the necessary Slate `Operation`, applies it to the local document state and passes the same `Operation` to the RTCS. In the RTCS server that `Operation` is also applied to the server's document copy and on success distributed to all other clients in the same project. If the application of the `Operation` fails in the server, the server sends an error back to the initiating client. The `DocumentsView` then does a rollback, removing the optimistically applied coding and notifies the user that the coding could not be applied due to a server error.

When removing a coding from the document the tasks performed in the `DocumentsView` are similar, with a small difference. The Slate `Operation` objects are also computed for optimistic local application, but instead of those objects, information about the code to be removed and the `Range` where to remove the code is sent to the RTCS. This raw information is needed, because the RTCS server also takes care of coding splitting, i.e. the assignment of new coding IDs, if a coding is split into two parts by removing the coding form a sub-range in the middle of the coding.

`Codesystem` Class

Similar to the `DocumentsView` for coding changes, the `Codesystem` component manages the backend communication for changes regarding codes. With the introduction of the RTCS the `Codesystem` is adapted to call methods of the `SyncService` instead of the Google API client.

Possible user actions that will be synchronized are adding and removing a code, relocating a code in the code system hierarchy and updating a code's properties. The synchronization round-trip for these tasks is much faster and therefore no optimistic execution is needed. The `Codesystem` component is also responsible for accepting events coming in from the RTCS and applying them to the local code system state.

`TextEditor` Class

The `TextEditor` class has to be completely rewritten, since the existing version was designed to work with the previously used Squire editor. The new implementation acts as a wrapper around the Slate `Editor` component of the `slate-react` module[1]. It manages the Slate `Value` used by the `Editor` component, handling updates from the `Editor` and actions triggered via the editor toolbar. Furthermore it provides rendering methods for `Block` and `Mark` objects. Facing the other React components of QDAcity, the new `TextEditor` offers the same methods as its previous version. To be able to keep the existing `CodingBrackets` component, the wrapper component `SlateCodingBracketAdapter` is introduced. Its purpose is to transform the data from the Slate `Value` of the `TextEditor` into the format required by the `CodingBrackets` component.

4.2 RTCS Server

The server-side component of the RTCS is a completely new software, without existing code to build upon. The following sections describe its most important parts and their purpose. They are grouped into three parts: The first group consists of application parts involved in the initialization of the server node. Helper classes form the second group and the third group contains classes that are responsible for the processing of synchronization messages.

[1] https://www.npmjs.com/package/slate-react

4.2.1 Server Initialization

The RTCS server's entry point is a file named `server.js`. On startup it loads the configuration and initializes all other modules. First it creates a new express[2] application and a new socket.io instance. The express application is used as HTTP server and will forward requests to the `/socket.io/` route to the socket.io instance.

Next step is the Redis initialization which is bundled in the `redis.js` file. Three Redis connections are established. The socket.io instance requires two connections to communicate with socket.io instances on other RTCS server nodes via Redis Pub/Sub[3]. The third connection is used to store and retrieve RTCS application data. Further a keep-alive mechanism is initialized to clean up Redis data. Every server node periodically sends a keep-alive message by updating a unique Redis entry identifying that node. If the server node fails to update the entry because it lost the connection to Redis or completely broke down, the entry expires and is processed in the next cleanup. All server nodes periodically perform such a cleanup, i.e. they search for application data that was written by a failed server node and delete that data entry.

The last step in server startup is the initialization of the sync service module. The only task performed here is to attach a listener to the socket.io instance that binds incoming WebSocket connections to a new instance of the RTCS `Socket` class. More details about the `Socket` class and its periphery are given in Section 4.2.3.

4.2.2 Helper Classes

This section describes the most important helper classes, that support the synchronization process on the RTCS server.

`Endpoint` Class

For the communication to the QDAcity backend the `Endpoint` class provides two methods: `updateParameters(String, String, String)` is used by the `Socket` class to configure a distinct backend connection for each connected client. The three parameters (API URL, API version, authorization token) are defined by the RTCS client, so the connection can be established on behalf of the respective user (cf. Section 3.2.1). To send a request to the backend the `request(String, Object)` method can be used. It requires an endpoint name as first and the payload as second parameter. An important feature of the `Endpoint` is request queuing. Since

[2] https://expressjs.com/
[3] https://redis.io/topics/pubsub

the backend connection is not configured until the user authorization has been completed in the frontend, it might occur that the `request(String, Object)` method is called before the connection configuration is completed. In this case the request is added to a queue and processed as soon as the connection is configured.

`DocumentLock` Class

Originally the synchronization of concurrent changes was planned to be implemented with use of the differential synchronization algorithm (cf. Section 1.3.4). This could not be implemented for time reasons. Instead a simpler locking approach is implemented in the `DocumentLock` class. Locks are set as data entries in Redis, utilizing two optional parameters on the Redis `SET` command that save a lot of implementation effort for the RTCS. If the parameter flag `NX` is set, the `SET` command only writes the data entry, if no data entry with the same key exists. This way it is ensured, that an already acquired lock can not be acquired a second time. The other helpful parameter is `PX <milliseconds>` that sets a time-to-live on the data entry, so a lock is released after a defined amount of time at the latest. A `DocumentLock` is instantiated for a specific document, whereas multiple instances may work on the same document. It provides methods for acquiring, refreshing and releasing a lock.

`DocumentCache` Class

To speed up the synchronization of documents and avoid unnecessary backend requests, a `DocumentCache` class is introduced. It provides methods to store and retrieve JSON representations of documents in or from Redis.

`Logger` Class

The `Logger` class provides an abstraction for logging. Instead of only writing to standard output stream via the global JavaScript `console` object, the `Logger` uses the winston[4] library. Winston provides the possibility to define multiple log targets. In case of the RTCS, logs are written to the standard output stream, which are redirected to local files, and Google Stackdriver[5]. The latter is chosen to have a single location for all logs regarding QDAcity, since the QDAcity backend log messages are also sent to Stackdriver.

[4] https://github.com/winstonjs/winston
[5] https://cloud.google.com/stackdriver/

4.2.3 Server-Side Synchronization Classes

The RTCS server's main task of synchronizing changes is performed by the `Socket` class and its two sub components `CodesHandler` and `DocumentHandler`. These classes are described in this section.

`Socket` Class

The `Socket` class is the server-side counterpart to the RTCS client's `SyncService`. It manages a single client connection, processes incoming messages of that connected client and emits events to the connected and all other clients. Like the `SyncService`, the `Socket` has also sub classes to handle specific sets of messages. Code changes are handled by the `CodesHandler` class while the `DocumentHandler` handles the application and removal of codings to or from documents. Updates of user data are handled by the `Socket` instance directly.

`CodesHandler` Class

The processing of code changes does not require additional tasks, therefore the `CodesHandler` is very simple. Incoming messages are translated to the corresponding backend request and executed via a call to the `request(String, Object)` method of the `Endpoint` class. If the request is successful, the response is emitted to all clients on the same QDAcity project. If the request fails, the backend response is forward to the client that caused the request.

`DocumentHandler` Class

The implementation of the `DocumentHandler` is more complex because conflicting changes on documents are much more probable than on the code system. When a message from a client is received, the first step is to acquire a lock on that document. This is retried every 100 milliseconds for up to 5 seconds, in case the lock is already set. If the lock cannot be acquired in 5 seconds at maximum, a failure messages is sent back to the initiating client. After successfully acquiring the lock, an attempt is made to read the document from the document cache, falling back to a backend request, if no cache entry was found. If the document has been loaded from the backend, it has to be deserialized from an HTML string to a Slate value.

The document is then altered according to the message type and data received by the client. When adding a coding, the Slate operations are applied directly to the document. The removal of a coding requires an additional check, if the removal

leads to a coding split, i.e. removing the coding from the given range results in a total of two ranges (one before, one behind the given range) with the same coding and the same unique coding ID. In that case a new unique ID has to be assigned to the latter of the two ranges. The operations for removing the coding and, if needed, assigning a new ID to the remaining coding, are then applied to the document.

If the document change was performed without failures, the Slate value is serialized back to HTML. The document including the deserialized Slate value is sent to the document cache and a copy of the document without the Slate value is uploaded to the backend. The lock on the document is released and the applied operations are distributed to all clients on the same project. If any error occurred while processing the message, an error is sent back to the initiating client.

4.3 Shared Library

The choice of a single programming language for both client and server leads to the advantage that a shared library can be implemented with code that is required on both the client and the server. It turned out to be not as much shared code as expected, but there are still two essential parts that are implemented in the shared library. For both parts it is very important, that the functionality works identical on client and server.

One part is the `SlateUtils.js` file, a library of pure functions that operate on Slate objects. It provides convenience functions like `findCodingStart(String, Immutable.List)` to search a list of characters for the first occurrence of a specific coding identified by its ID string. The library also contains essential functions like `deserialize(String)` and `serialize(Value)` to convert HTML strings to Slate `Value` objects and vice versa.

The other part is the `constants.js` file, that contains constant definitions for the message and event types. For each action that can be performed collaboratively in QDAcity a message and an event are defined pairwise: The message is sent from the executing client to the RTCS server. After all necessary server side actions are performed, the server emits the matching event to all clients in the same QDAcity project. All clients, including the initiating client, then apply the action to their local application state. An exemplary pair would be the `code.insert` message and the `code.inserted` event, that is sent or emitted when a user adds a new code to the project's code system. A full list of messages and events can be found in Table 4.1.

5 Evaluation

This chapter reviews the requirements from Chapter 2 and compares them to the implementation described in Chapter 4. It is grouped by functional and non-functional requirements.

5.1 Functional Requirements

In this section the functional requirements are evaluated. These requirements target the synchronization of codings, codes, collaborators, text changes, text selections and the authentication and authorization of users.

5.1.1 Synchronization of Codings

Being the central functionality of QDAcity, the synchronization of codings was also the most important requirement. As defined in FR-1 to FR-4 (cf. Section 2.1.1), when adding or removing a coding, the change should be persisted in the QDAcity backend and distributed to all collaborating users.

With the RTCS, both adding and removing codings to and from QDAcity text documents are now synchronized within a narrow time frame to all collaborators that concurrently view the same project as the acting user. As it already was the case before this project, both types of changes are persisted in the QDAcity backend data store. Difference is, that now the QDAcity frontend application does not directly communicate to the QDAcity backend in order to achieve this, but the persisting backend API call is performed by the RTCS server. Even though the synchronization does not work as fast as anticipated, it is adequately usable.

5.1.2 Synchronization of Codes

The next group of requirements approaches the synchronization of codes. All actions regarding codes have been considered, adding and removing a code as well as any modifying action regarding codes. All these actions had to be persisted in the backend and synchronized to all collaborators (cf. FR-5 to FR-10 in Section 2.1.2).

The adding and removing of codes to and from the code system are synchronized and persisted by the `CodesHandler` module that has been implemented within the scope of this thesis. The same applies to the actions of relocating a code and changing its properties like name, color and author. Because of time reasons the synchronization of the remaining modifying actions could not be implemented. These include changing a code's code book entry, as well as adding, changing and removing the relation between two codes. These features can still be used and any changes are persisted via a direct call to the QDAcity backend as it was the case before this thesis.

5.1.3 Synchronization of Collaborators

Within this thesis only few user interface elements had to be implemented. As defined in FR-11 to FR-15 (cf. Section 2.1.3) users should be able to see if and which other users are actively working on the same project or document.

The project panel of the QDAcity coding editor has been augmented by a collaborator list, that shows the number of collaborators and the profile picture of each collaborator. When hovering the list, additionally the full names of the collaborators are shown. Accordingly a list of users that are active on a specific document is shown at the corresponding document's entry in the document list. Users are added to all collaborators' user lists, as soon as they enter the coding editor. When they leave the coding editor or the whole application, they are removed from all collaborating users' lists. With these two interface elements, the defined requirements have been fulfilled, as users can always see which other users are active on the same project and document.

5.1.4 Synchronization of Text Changes and Selections

The requirements for synchronizing text changes are defined in FR-16 to FR-21 of Section 2.1.4. They include changes on the content, i.e. adding and removing characters, and changing their styling, i.e. marking characters bold or italic. FR-22 and FR-23 define requirements for the synchronization of the users' text selections.

These requirements are defined as optional, since text editing is not a common task inside the QDAcity application and the absence of the selection synchronization does not limit the tasks that can be achieved with QDAcity.

The implementation of the synchronization of text changes and text selections has been left out, since the extent of these features would have gone beyond the constraints of this thesis. Still the current version of the RTCS provides a solid foundation to synchronize text changes and selections in the future.

5.1.5 Authentication and Authorization

One functional requirement regarding authentication and authorization has been defined in FR-24 (cf. Section 2.1.6). It states that only authenticated users may be able to apply changes to a document.

The existing frontend and backend of QDAcity already implement authentication and authorization. Users can only enter the QDAcity frontend after they authenticated using their Google account. The backend endpoints are secured and can only be accessed if an authentication token is provided in the request. The authorization of the authenticated user is checked within the backend endpoint methods.

The authentication forwarding of the RTCS (cf. Section 3.2.1) ensures that requests to the backend are always performed with the authorization level of the initiating user. The authentication token, that is received by the RTCS client and forwarded to the QDAcity backend, certifies the authentication and the authorization of the user at the same time.

5.2 Non-Functional Requirements

The evaluation of non-functional requirements is performed in this section. Non-functional requirements are grouped in technological and user interface requirements.

5.2.1 Technological Requirements

Section 2.2.1 defines four requirements of technological nature (NFR-1 to NFR-4).

In NFR-1 a client-server architecture is required for the RTCS. The RTCS has been designed as a client-server application as described in Chapter 3 and therefore this requirement is satisfied.

NFR-2 requires the inclusion of the RTCS client into the existing QDAcity frontend. As shown in Section 3.1.5 this is fulfilled, since the RTCS client is a module inside the QDAcity frontend.

The RTCS has to be manageable via the Google Cloud Console (cf. NFR-3). The Google Compute Engine has been selected as runtime for the RTCS server (cf. Section 3.1.3) and like all Google Compute Engine instances, the RTCS server is also manageable via the Google Cloud Console.

As last technological requirement, the RTCS should be designed for operation in a distributed way on multiple instances. The choice of the Google Compute Engine ensures that multiple RTCS instances can be grouped behind a load balancer. Further all application data of the RTCS that is relevant across the whole application and not only to a single instance is stored inside the Redis in-memory database. With these two aspects, NFR-4 has been fulfilled.

5.2.2 User Interface Requirements

For the user interface and user experience regarding RTCS features, the requirements NFR-5 to NFR-8 have been defined in Section 2.2.2.

The first of these four requirements states that the RTCS should be able to continue operation if the synchronization of a single change has failed. All failures that can occur while synchronizing a change are handled within the RTCS server or client. Still possible are errors caused by a complete node outage or software errors outside the RTCS application.

According to NFR-6 the synchronization of changes should be transparent to the user. This has been achieved partially, since no additional action is required from

the user to enable or perform synchronization. The interface elements that are used to trigger an action are placed at the same location and work the same as before. Transparency breaks if errors occur, since the RTCS has to inform the user that a synchronization task failed and therefore the intended change could not be applied to the server nor the local version.

Requirements NFR-7 and NFR-8 define two limits for the time that may pass while changes are synchronized via the RTCS. The optimum is at 150 milliseconds (cf. NFR-7) and the upper limit is at 1 second (cf. NFR-8). Both limits could not be observed and the optimization would have exceeded the scope of this thesis. At least the perceived synchronization time could be reduced to several milliseconds by performing optimistic execution (cf. Section 3.2.3), so the application is already usable without interruptions.

6 Conclusion

Motivation for this thesis was the need for a software tool to support real-time collaboration in qualitative data analysis. QDAcity provides a solid base for comprehensive software support in qualitative data analysis. By introducing the real-time collaboration service implemented in this thesis, QDAcity could be augmented with real-time collaboration capabilities.

This thesis started with the description of the problem, especially the possible conflicts when editing data concurrently. Different algorithms were then discussed that address the avoidance or resolution of conflicts. The functional and non-functional requirements for the real-time collaboration service have been defined within this thesis and were used as foundation to develop an architecture for the resulting software. In the implementation many features could be realized, first of all the base functionalities to provide support for synchronizing different kinds of data across the QDAcity application. On top of this basis the synchronization of codes and codings has been implemented, as well as the synchronization of active collaborators.

Some aspects have not been accomplished in their entirety (cf. Chapter 5) but still the RTCS lays a solid foundation to build upon and further improve the real-time collaboration in QDAcity. Researchers can use QDAcity already today to work concurrently on the coding of documents. This reduces the need of communication and coordination outside of QDAcity and is a great benefit that supports real-time collaboration in qualitative data analysis.

References

Corbin, J. & Strauss, A. (2014). *Basics of qualitative research*. Sage Publications.

Ellis, C. A. & Gibbs, S. J. (1989). Concurrency control in groupware systems. In *Sigmod '89 proceedings of the 1989 acm sigmod international conference on management of data* (Vol. 18, *2*, pp. 399–407). ACM.

Engelbart, D. C. & English, W. K. (1968). A research center for augmenting human intellect. In *Proceedings of the december 9-11, 1968, fall joint computer conference, part i* (pp. 395–410). ACM.

Fette, I. & Melnikov, A. (2011). *The websocket protocol* (RFC No. 6455). Retrieved from https://tools.ietf.org/html/rfc6455

Flick, U. (2014). Mapping the Field. In U. Flick (Ed.), *The SAGE Handbook of Qualitative Data Analysis* (Chap. 1, pp. 3–18). London: Sage Publications.

Fraser, N. (2009). Differential synchronization. In *Proceedings of the 9th acm symposium on document engineering* (pp. 13–20). ACM.

Loreto, S., Saint-Andre, P., Salsano, S., & Wilkins, G. (2011). *Known issues and best practices for the use of long polling and streaming in bidirectional http* (RFC No. 6202). Retrieved from https://tools.ietf.org/html/rfc6202

Miles, M. B., Huberman, A. M., & Saldaña, J. (2014). Qualitative data analysis. a methods sourcebook 3rd edition. London: Sage Publications.

Punch, K. F. (2014). *Introduction to social research: Quantitative and qualitative approaches*. London: Sage Publications.

Roh, H., Kim, J., & Lee, J. (2006). How to design optimistic operations for peer-to-peer replication. In *International conference on computer science and informatics (jcis/csi)*. Kaohsiung, Taiwan.

Rupp, C. (2014). *Requirements-Engineering und -Management: Aus der Praxis von klassisch bis agil*. Munich: Carl Hanser Verlag.

Saldaña, J. (2011). *Fundamentals of Qualitative Research*. Understanding Qualitative Research. Oxford University Press.

Schreier, M. (2014). Qualitative Content Analysis. In U. Flick (Ed.), *The SAGE Handbook of Qualitative Data Analysis* (Chap. 12, pp. 170–183). London: Sage Publications.

Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). Conflict-free replicated data types. In X. Défago, F. Petit, & V. Villain (Eds.), *Stabilization,*

safety, and security of distributed systems (pp. 386–400). Berlin, Heidelberg: Springer.

Taylor, I. S. (2018). Introduction - slate. Retrieved March 24, 2018, from https://docs.slatejs.org/

Tolia, N., Andersen, D. G., & Satyanarayanan, M. (2006). Quantifying interactive user experience on thin clients. *Computer*, *39*(3), 46–52.