FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real-Time user-based coverage of a sports event: A Web Application for the modern football fans

Ângelo Miguel Tenreiro Teixeira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. João Correia Lopes

Second Supervisor: Eng. Marco Sousa

June 18, 2021

Real-Time user-based coverage of a sports event: A Web Application for the modern football fans

Ângelo Miguel Tenreiro Teixeira

Mestrado Integrado em Engenharia Informática e Computação

June 18, 2021

Abstract

TODO

Today, millions of users follow their teams' games online to keep up-to-date regarding the events of a match. Some of those had a special connection to their hometown team, but since it plays in way lower leagues and without much exposure, the users end up missing information and losing the passion they once had for the hometown team. There is, however, a specific group of users that keeps following the games of the smaller teams, and most importantly: sharing updates about them.

Platforms that allow game commentary sharing enable the most passionate fans who still watch the smaller leagues to share what is going on in the game, report the events, and build the game's history, totally community-driven.

Tools for this already exist but are either somewhat outdated or do not completely let the users get involved in the commentary, restricting them to a more passive role, hence the opportunity to build something better.

This work involves creating a Real-Time web application that allows users to publish event updates for any event, increasing their connection to the lower leagues as it already happens in the upper echelons. Additionally, as stadiums internet connectivity is often poor, the application will allow users to post while offline, syncing when the internet comes back, and featuring a conflict resolution functionality to improve user experience (UX). A proposed solution is presented, as well as a plan for its development.

Keywords: Real-Time, Web, Sports, Conflict Resolution, Reputation systems

Resumo

Hoje em dia milhões de utilizadores mantêm-se a par dos jogos das suas equipas online, de forma a estar a par dos eventos ao longo destes, caso não possam assistir diretamente. Alguns tinham uma ligação especial à equipa local, mas jogando em ligas de escalão inferior sem grande cobertura televisiva ou até mesmo jornalstica, acabam por perder interesse e a paixão que sentiam outrora. Ainda assim, ainda existe um grupo de utilizadores que continua a seguir os jogos das equipas mais pequenas, e mais importante que isso: a partilhar atualizações sobre estes.

Plataformas que permitam o comentário ao longo dum jogo permitem aos fãs mais apaixonados que ainda seguem as equipas locais partilhar o que vai acontecendo ao longo do evento, reportando o que se passa e construindo a história do jogo, num esforço totalmente comunitário.

Ferramentas como estas já existem, contudo estão obsoletas, ou não permitem completamente aos utilizadores o envolvimento no comentário do jogo, restringindoos a um papel mais passivo, daí a oportunidade de criar algo único e inovador.

Este trabalho involve a criação de uma Aplicação Web que permita aos utilizadores a partilha de atualizações de um evento desportivo em tempo-real, aumentando a ligação às ligas mais inferiores, como acontece com os escalões superiores. Adicionalmente, e uma vez que a ligação à internet nos estádios é geralmente instável, a aplicação deverá permitir aos utilizadores interagir enquanto estiverem em modo offline, sincronizando assim que a ligação seja restabelecida, incluindo uma funcionalidade de resolução de conflitos de forma a melhorar a experiencia do utilizador. Será apresentada uma proposta de solução, bem como um plano para a sua implementação.

Keywords: Real-Time, Web, Sports, Conflict Resolution, Reputation systems

Acknowledgements

TODO ACKNOLEGDEMENTS

Author

"Ceci n'est pas une citation."

Ângelo Teixeira

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
OT	Operational Transformation
CRDT	Conflict-free Replicated Data Type
UX	User Experience
UI	User Interface

Chapter 1

Introduction

TODO

Today, millions of users follow their teams' games online to keep up-to-date regarding the events of a match [1]. Some of those had a special connection to their hometown team, but since they play in way lower leagues and without much exposure, the users end up missing information and losing the passion they once had for the hometown team.

There is, however, a specific group of users that keeps following the games of the smaller teams, and most importantly: sharing updates about them. One platform that allows users to do that, as of today, is zerozero.pt, from ZOS. This enables the most passionate fans who still watch the smaller leagues to share what is going on in the game, report the events, and build the game's history, totally community-driven. This tool exists and is somewhat outdated, hence the opportunity to build something better, including automatic conflict resolution, offline mode and a mobile-friendly interface out of the box.

1.1 Goals

The goal is to allow multiple users to report the incidents in a sporting event, which will show up for everyone following that match in real-time. As internet connectivity is often poor inside stadiums, the tool must allow offline work, synced whenever possible. This can generate many data inconsistencies, which must be handled by the tool.

Parallel to this, we want to provide the best possible User Experience, since inconsistencies can seem confusing for users. For this, we indend to measure and test different alternatives, in order to elicit what is the desired experience.

This project will provide an approach to this problem, and the following sections provide more details on the project's key objectives.

[1] https://www.facebook.com/business/news/insights/the-changing-profile-of-sports-fans-around-the-world

1.2 Offline Availability

As previously stated, internet connection in stadiums is poor most of the time. Thus, the users must have the option to interact with the application and synchronize once possible. This will obviously lead to data consistency issues (i.e., two users report a goal, changing the result to “1-0” for example, but one of them is offline, so when it finally synchronizes, the result is already “3-2”, and it should not be overwritten.)

More information on this topic is presented in Section 2.2 and a proposed solution will be stated in Section 3.2.3.

1.3 Conflict Resolution

Another objective of the tool is to provide users with automatic conflict resolution when possible. Some strategies are depicted in the *State of the Art* section, in Chapter 2.3. Here, it is important to preserve the truth and the most up-to-date versions of data. In this scenario, there might not be a source of truth present to verify and validate all inputs, so other strategies must be used, such as agreement-based implicit voting — if nobody questions a user’s input, it must be true until stated otherwise.

Additionally, the tool can use different strategies to solve conflicts automatically, thus improving the user experience (UX). More on this topic is available in Section 2.3 and a preliminary proposed solution can be found in Section 3.2.4.

1.4 Reputation System

The third key-objective of the application will be the reputation system. Currently, there already exists a ranking concept, and a “trusted” user, which is the equivalent to the maximum reputation and should be considered the source of truth in case of conflict.

But what about the cases where two “non-trusted” users’ inputs conflict, or even the case of two “trusted” users? Who should win? To resolve conflicts, an answer to these *conundrums* is fundamental. Ergo a new reputation system is required, and more details are available in Section 2.4. A preliminary proposed solution is presented in Section 3.2.5.

1.5 Document Structure

In Chapter 2, a comparison with a similar project is made, as well as a *State of the Art* exploration on the multiple scopes of this project. Chapter 3 defines the problem and proposes solutions for it, which are planned in Chapter 5 Conclusions are present in Chapter 6.

Chapter 2

Background and Literature Review

This chapter will dive deep into previously done work related to this project. First, a Background is provided for the reader to have context on some relevant work and information that precedes the findings present in the following sections. Second, since the goal is to develop a complete application, there will be an analysis of the specific problems and how they have been solved in the literature. Then, there will be a comparison between a similar work and similar existing applications.

2.1 Background

Since this project is intended for general use, the easiest and most common client available to users is the web client, also known as a web browser. In the early stages of the web, the applications followed a server-client architecture where the client had little to no work: it just rendered some previously compiled HTML and CSS on the page, and the user made the interaction with the server through HTML forms. Later on, JavaScript usage increased, and the pages started to be a bit more dynamic [53]. Still, it wasn't until more performant and portable devices such as the iPhone were available to the public that web applications started to tilt their focus to the client-side.

More recently, we start to see "Single Page Applications", which harness the client-side JavaScript capabilities to simulate legacy web interactions such as changing to another page or view without actually reloading the page, trading client-side work for network load [39] [20] [46] [45]. In this architecture, there still exists a server and clients (the web browsers). However, the server usually serves a REST API (Representational State Transfer Application Programming Interface) and a bare-bones HTML document, which serves as a base for the clients to render the rest of the document with JS, based on API calls results. There is also a mixed option: The server handles some logic, usually related to session management or localization, and responds with an HTML document that already has some information to prevent the client-side from taking too

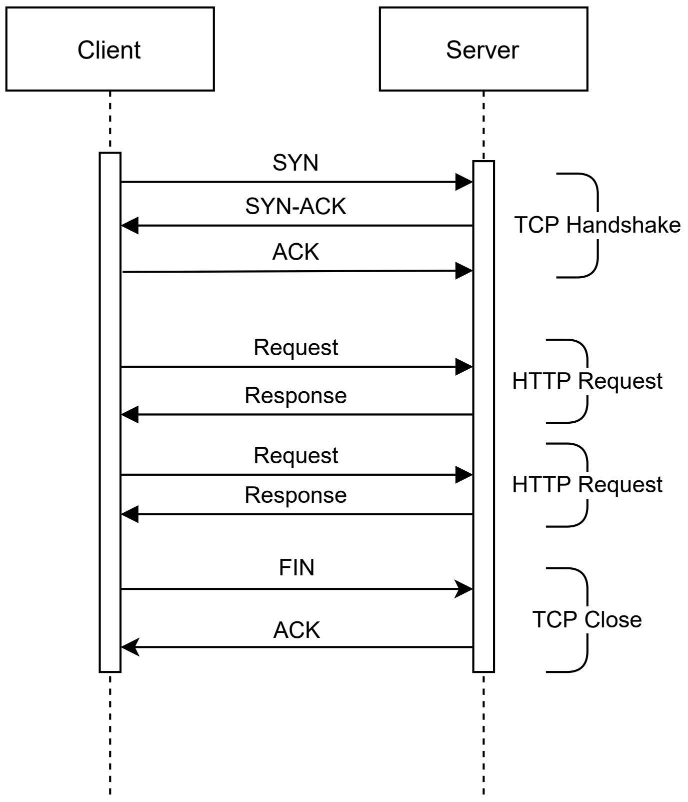

Figure 2.1: HTTP protocol example interactions

much time on work done for every request. On the web, some frameworks allow this more recent architecture, such as React[1], Vue[2] and Angular[3].

Since the web uses the HTTP Protocol [24] [14], it is quite hard to achieve real-time behavior with good performance, due to the protocol's design [54]. As shown in Figure 2.1, every data transfer starts with a request and ends with a response. It is therefore tied to this two-step process, which limits its potential.

To allow a more performant way of data transmission between client and server, similar to the well-known data sockets available in Operating Systems used for bi-directional data transfer in real-time, there exist Web Sockets [22] which serve the same purpose and allow Web Applications to send real-time updates to the clients without them needing to request them, which would only be possible otherwise using techniques such as polling (i.e., the client keeps asking the server for updates periodically), and even here, the client is actually asking for updates, only in fast succession.

When developing a Web Application, one of the most important aspects is User Experience (UX). It is even more relevant than User Interface (UI) because, even though they impact one

[1]https://reactjs.org/
[2]https://vuejs.org/
[3]https://angular.io/

another, an application with a good UI but poor UX is just a nice painting the users can look at and be amazed, but with no interaction whatsoever.

According to Tullis and Albert [59], UX involves a user, that user's interaction with the interface, and the interest in observing or measuring their experience when using the interface. A more common concept is usability, which is usually considered to be the ability of the user to use the interface to execute a task successfully.

In order to have a good UX, an application cannot require too much thinking from the user, it should be intuitive [31]. The application should be self-explanatory and "nothing important should be more than two clicks away". This involves link naming, placement, colors, and more, hence its connection to the UI.

Even though some things have specific rules (e.g., links should be sayclickable and look like it), others require further examination to understand how users are interacting with them. This is done manually in usability tests, where actual human users are recruited and product owners can see directly how they execute some requested action, and feedback is provided directly; or automatically, through UX metrics.

UX metrics can be of different types [59]: performance, usability, self-reported (i.e., satisfaction / ease-of-use rating given by the user) and behavioral / phsycological.

Performance metrics include measuring task success ratios, timing tasks (i.e., how much time does it take for a user to do something) and error collection. Usability metrics rely on the users reporting of issues they encounter when using the application. Self-reported metrics are small questionnaires filled in by users, usually at the end of an interaction/session, which give a more complete information about what the experience felt like for the user. Behavioral/Psychological metrics take advantage of sensors in order to measure reactions via eye-tracking and heart rate monitoring, for example.

As with any software application, there must some kind of tests in place, in order to guarantee a minimal quality standard. In the interface domain, tests rely more on simulation user actions, such as clicking on components, inputting text, submitting forms, etc., and asserting the results of those actions on the webpage. To develop unit-tests following this model, there are libraries such as Testing Library[4] which allows for asserting page changes in multiple frameworks — such as React, Vue or Angular, mentioned above, and more. For more complex tests, relying on the full application — including the server — which simulate the complete interaction of a user with the application, also called end-to-end tests, there are alternatives like Selenium[5] and Puppeteer[6], which simulate actions on an actual browser, not only on a simulated webpage, thus allowing that browser to dynamically change based on the server behavior and on interaction with other clients, rendering it very useful when testing multi-user applications.

[4]https://testing-library.com/
[5]https://www.selenium.dev/
[6]https://pptr.dev/

2.2 Offline Availability

As stated in [43], even though people complete interrupted tasks in less time with no quality drop, interruptions make users work faster to compensate for it, increasing stress and frustration. The lack of offline interaction would force users to interrupt their task of inputting the desired information. That would increase stress when adding it later once they connect again, as they would try to do it fast while that information still has relevance. Indeed, a real-world application was tested in both scenarios — with offline mode and without it — for which the basic version obtained a user satisfaction score of 66.5. In contrast, the offline-tolerant version had a score of 95.5 out of 100 points, proving that having an offline mode improves the user experience in a web application substantially [42].

Marco [40] exposes the interrupted internet connectivity problem applied to e-learning applications. It proposes creating a modeling tool to create interruption-resilient web applications by defining the possible operations and how they should behave in case of interruption. This model is presented later [5], proposing the Offline Model, which defines a taxonomy for interruptions on web applications. It further analyzes some properties of web applications in the presence of interruptions, such as scheme and offline support, which depend on the application and the task being performed within it.

Marco further specifies rules regarding adapting basic web applications into offline-tolerant ones [42]. On the *Application Domain Level*, the data model of the application should not be modified to support as many applications as possible. Regarding the *Hypertext Level*, the navigation on the application could be slightly modified or adapted, regarding pages within the web application, depending on the connectivity status. On the *Presentation Level*, the user interface (UI) should be adapted according to policies specific to each page element as the interaction with those elements would change depending on the connectivity status.

Therefore, it is necessary to establish which components of the web application will be available in offline mode and how. To specify the navigation, they propose the notation presented by Albertos et al. [41], representing the web pages as nodes, with edges meaning navigation within the web application. Each node has attributes referring to policies about the behavior of the components in offline mode and the mechanism to create the local copy. To specify the interface behavior in offline mode, the following policies are described:

- **Hide**: The element should be hidden, preventing user interaction
- **Disable**: The element should be shown, but interaction should not be possible
- **Save**: Save elements locally so that they can be accessed in offline mode (such as images or text documents)
- **Update**: For elements that can be interacted with when in offline mode, they should be updated — synchronized — when connectivity is regained

Yang [64] presents a mechanism to support offline collaborative work in a web application. Due to the publication's age, most of the used tools are now obsolete, but the general ideas still apply. It proposes the existence of local storage to every client so that operations done in offline mode are kept. Then, once the user comes back online, the work saved in the local storage is synchronized with the online storage, where conflict-resolution algorithms take place in order to minimize said conflicts.

Kleppmann, Wiggins, Hardenberg et al. [30] propose ideals for local-first software, being the idea that the data on users' local machines is the primary when compared to servers, not the other way around, as it often happens in web applications:

1. **"No spinners" / No loading**: Data is changed locally, and synchronization occurs quietly in the background;
    - A workaround pattern is mentioned — Optimistic UI — which consists of showing the changes immediately, while sending them to the server, which would need to be reverted in case of error, for example
2. **Multi-device support**: Users should be able to work everywhere;
3. **Optional network**: Since local-first applications store the primary copy of their data in each device's local file system, the user can read and write this data anytime, even while offline. It is then synchronized with other devices sometime later, when a network connection is available;
4. **Seamless Collaboration**: Real-Time collaboration should be possible with conflict resolution when necessary;
5. **Data Forever**: The data is independent of the company or service provider, since it is stored locally. It should therefore last forever;
6. **Secure and Private by default**: By having the data on each user's device locally, database tampering is not as harmful, as you'll have many backed up replicas. Moreover, by using end-to-end encryption, the server can only save encrypted data, thus making it private, as it can only be decrypted by the users who are allowed to do so;
7. **Ownership**: The user owns the data, the provider cannot block the access in any form since the data is local.

It further compares multiple applications in the way they handle the local-first ideals, such as Files and Email attachments, or Google Docs[7] as well as technologies to implement said ideals, such as Web Applications, Mobile Applications, CouchDB[8] — a multi-master database that allows each node to mutate the database and synchronize the changes with other nodes, meant to run on

[7]https://www.google.com/docs/about/
[8]https://couchdb.apache.org

servers — and PouchDB[9] — with the same goal of CouchDB, but meant to run on the end-user devices.

It also mentions CRDT (Conflict-Free Replicated Data Types) as a foundational technology to achieve the local-first ideals. As explained in the next section, CRDT are general-purpose data structures similar to the common lists and maps but built for multi-user environments from the beginning. CRDT merge changes from multiple users when possible; however, it does not handle changes to the same element in the structure. In that case, it keeps track of the conflicts for the application to deal with them, allowing for custom conflict resolution techniques. CRDT are generic enough to synchronize over any communication connection like server, peer-to-peer networks, Bluetooth, and others. The paper further presents *Automerge*[10] as a JavaScript (JS) CRDT implementation.

Furthermore, the authors built some prototypes using Electron, JavaScript, and React using CRDT to verify that its usage was viable to build local-first software for the web and desktop applications. The main conclusions were that CRDT worked reliably while integrating easily with the other tools and seamlessly merging data. Also, they verified that the user experience was splendid, as it allowed for offline work and sync when possible, giving the feeling of "data ownership" to the user. Finally, they state that this technology combines well with Functional Reactive Programming (FRP), a paradigm that renders the view based on a function that receives data. If the data changes, it "reacts" and redraws. A popular framework enabling this paradigm is React, but there are others such as Vue or Flutter. With such a tool, by syncing the data with a CRDT and reacting to UI changes, a real-time experience is achieved.

Finally, it added that CRDT might have a performance problem if used to save many changes (since they essentially save all the history). That is something that must be taken into account when using that technology and designing the system.

Zawirski, Preguiça, Duarte et al. [67] proposes a way of supporting many client-side applications by sharing a database of objects that they can read and update under a convergent causal consistency model, with support for application-specific conflict resolution. It relies on fast local writes, and possibly stale data reads to achieve better speeds. It allows updates to thousands of clients using only three data centers, leveraging client buffering and controlled staleness, absorbing the cost of scalability, availability, and consistency.

Kao, Lin, Yang et al. [29] proposes a system to allow mobile applications to run in a web container, joining the benefits of both: being able to run offline and cross-platform.

In order to make it possible for users to work while they are not connected to the internet, there must be a way of storing their actions locally so that they can be synchronized when they regain internet connectivity. Currently, HTML5 exposes multiple local storage APIs, which can be used for this effect.

Liu [38] discusses multiple web client storage technologies, comparing their advantages and disadvantages:

[9] https://pouchdb.com/
[10] https://github.com/automerge/automerge

- **Cookies**: Small piece of data that is included in every HTTP request (limit of 4 KB per cookie [12]). They can be managed by the server as well as the client and are mostly used to keep state in between HTTP requests, for use cases such as user sessions or keeping track of some user activity. They are very popular and widely used, but at the same time they have little capacity and can be insecure, due to their presence in the HTTP communication with the server [60] [32];

- **localStorage**: Can be used to implement cross-page communication (in the same domain) by storing data and having the web application listen for storage events, all pages of the same domain will have the updated data in the respective section of *localStorage*. It can be faster than cookies since no interaction with the server is needed. The data is stored in key-value pairs and is partitioned by domain, with no risk of inter-domain data corruption. It can be read and modified using JavaScript and all values are strings.

- **sessionStorage**: Works in the exact same way as *localStorage*, but the data only lasts until the browser is closed, i.e., the session;

- **UserData**: Older technology, specific to Internet Explorer, and has since been deprecated, it had only 128 KB of capacity;

- **Web SQL Databases**: Uses a table-structured format to store the local data. In Google Chrome, it uses the SQLite DB. The interactions are made as it were a normal SQL database running, by using the $executeSql()$ or $transaction()$ APIs. It is not supported by all browsers nor is it part of W3C standard[11];

- **Indexed Database**: Uses a NoSQL DB to store the data. Event-based so it is asynchronous by default. It might be helpful to use a Promise-based wrapper [3] such as idb[12].

Liu further concludes that *localStorage* is the most compatible among web browsers. In summary, for session id tracking or for small data storage, one should use *Cookies*; up to 5 MB, *localStorage* delivers a solution across all common browsers[13]; for more space, **Indexed Database** is recommended.

Jemel and Serhrouchni [28] presents some vulnerabilities that can arise from using the HTML5 Local Storage API as is. It also introduces a way of securing data stored using the local storage API, based on the user, not only the domain the data belongs to.

2.3 Conflict resolution

Conflict resolution is a problem that has been studied extensively in the literature, with publications dating as far back as 1986 [26]. Still, it was not until 1989 that the first algorithm was proposed to deal with this problem by Ellis and Gibbs [21].

[11]https://www.w3.org/TR/webdatabase/
[12]https://www.npmjs.com/package/idb
[13]https://www.w3.org/TR/webstorage/

In that paper, they define groupware systems as multi-user (i.e., composed of two or more users) computer systems that allow development on a common task, providing an interface to a shared environment. They propose an algorithm to solve the groupware real-time concurrency problem called **Operational Transformation (OT)** which allows concurrent editing without the need for locks, increasing responsiveness. *Response time* is defined as the time required for the user's action to be reflected on their screen. *Notification time* is defined as the time required for the user's action to be propagated to all other participants.

Real-time groupware systems have the following characteristics:

- **Highly interactive**: Short response times;
- **Real-time**: Notification times should be close to the response times;
- **Distributed**: They should work even if the participants are connected in different machines and networks on the internet;
- **Volatile**: Participants may enter and leave the session at any time;
- ***Ad Hoc***: Participants don't follow a script, it is therefore impossible to know what is the information they are trying to access beforehand;
- **Focused**: Generally users will be trying to access the same data, generating a high degree of access conflicts;
- **External Channel**: Participants are often connected among them via an external channel such as an audio or video communication tool;

A groupware system model is then defined as being formed by a set of sites and operators. Sites consist of a site process (i.e., a user's unique session), a site object (i.e., the data being read and modified), and a unique site identifier. Operators are the set of operations available for users to apply to the site objects. The goal is to maintain consistency among all the site objects at all times. The site process performs three kinds of activities: **operation generation**, where the user generates an operation to be applied to the site objects. The site will then encapsulate the action in an operation request to be broadcasted to all other sites; **operation reception**, where an operation is received from another site; **operation execution**, where an operation is executed on the local site object. The model further assumes that the number of sites is constant, messages are received exactly once, without error, and that it is impossible to execute an action before it is generated.

The paper further specifies the following definitions regarding the groupware system:

- Given two operations a and b, generated at sites 1 and 2, respectively, a precedes b iff:
  - $a = b$ and the generation of a happened before the generation of b, or
  - $a \neq b$ and the execution of a on site 2 happened before the generation of b;

- The **Precedence Property** states that if an operation *a* precedes another operation *b*, then at every site the execution of *a* happens before the execution of *b*;
- A groupware session is **quiescent** iff all generated operations have been executed at all sites;
- The **Convergent Property** states that site objects are identical at all sites at quiescence;
- A groupware system is *correct* iff the **Convergent Property** and the **Precedence Property** are always satisfied.

The proposed algorithm uses four auxiliary data structures: State vector, Request, Request Queue, Request Log and a predefined, semantics-based matrix of transformations.

State Vectors are based on the partial ordering definition in [33] and the concept of vector clocks in [37] and [23], stores the amount of operations done per site, i.e., the *i*'th component of the vector represents how many operations from site *i* have been executed in the current site. It is therefore possible to compare two state vectors s_i and s_j:

- $s_i = s_j$ if each component of s_i is equal to the corresponding component in s_j;
- $s_i < s_j$ if each component of s_i is less than equal to the corresponding component in s_j and at least one component of s_i is less than the corresponding component in s_j;
- $s_i > s_j$ if at least one component of s_i is greater than the corresponding component in s_j;

Requests are tuples in the form $< i, s, o, p >$ where i is the originating site's identifier, s the originating site's state vector, o is the operation and p is the priority associated with o. From the request state vector, a site can determine if the operation to execute can be executed immediately, or wait for needed updates from other sites, enforcing the precedence property. The *Request Queue* is a list of requests pending execution. Even thought the term "queue" is used, it does not imply first-in-first-out order. The *Request Log* stores at site *i* the executed requests at that site, in insertion order.

The Transformation Matrix defines, for every operation type pair, a function T that transforms operations so that given two operations o_i and o_j, with priorities p_i and p_j, instances of operators O_u and O_v, respectively and

$$o'_j = T_{uv}(o_j, o_i, p_j, p_i)$$

$$o'_i = T_{vu}(o_i, o_j, p_i, p_j)$$

then T is such that

$$o'_j \circ o_i = o'_i \circ o_j$$

where $\circ$ means composition of operations.

The algorithm has an initialization section, a generation section, a reception section, and an execution section. In the initialization section, the site's log and request queue are set to empty, and the state vector is initialized with all values being 0 since no operations have been done. The next section specifies that whenever a local operation is received, a request is formed, and it is added to the local queue and broadcasted to other sites. In the receiving section, when a request is received, it is simply added to the request queue. Finally, the execution section specifies how to apply the operations, handling conflicts. First, it checks the request queue to retrieve any request (with state s_j) that can be executed, s_i being the state in the local site i, and there are three possibilities:

1. $s_j > s_i$: The request cannot be executed since there are changes done in site j that were not executed yet at site i, therefore the request must be left in the queue for later execution;

2. $s_j = s_i$: The two states are equal, therefore the request can be executed immediately without operation transformation

3. $s_j < s_i$: The request can be executed, but the operation must be transformed, since site i has executed requests that are preceded by request j, r_j. Site i's log L_i is examined for requests that were not accounted for by site j (i.e., the requests that were executed in i but not on j prior to the generation of r_j. Each such request is then used to transform o_j in o'_j, according to the Transformation Matrix. o'_j is then executed and the state vector is incremented.

Some changes were later proposed to the state vector technique [34] [6] to allow dynamic entries instead of a constant number of concurrent participants. Ellis and Gibbs [21] mentioned this in the OT definition and addressed it, noting that participants can enter and leave every time the system is quiescent since the Request Logs can be reset, and it should function as a checkpoint on each site.

Nichols, Curtis et al. [47] build another algorithm on top of the existing OT, presented in [21] that uses a server managing the collaboration, instead of being peer-to-peer like the former. This reduces the need for the request priority fields in the requests for tie-breaking since the server can use a different strategy such as a reputation system, which the next section will develop upon. By removing the need for multicasting, since the server orchestrates the process and the communication is done in server-client pairs only, there is no need for message reordering logic, since a message transport protocol such as TCP [50] can be used instead, ensuring message delivery in the correct order before reaching the application layer, thus reducing the clients' workload.

Wang, Bu, and Chen [61] propose a multi-version approach to resolving conflicts in a real-time multi-user graphic designing system. It is a **Compatible-Precedence** approach since, unlike a **Conflict-Precedence** approach where the objects are locked for resolution by users in case of conflict, in this case, the conflict resolution is made by the system. It proposes a classification of operations in one of two types: Non-Multi-version Operations (NMO) and Multi-version Operations (MO). The first are operations that, when in conflict, can be resolved by choosing one of them. The latter are better suited to be decided by the users; hence keeping both versions of the operation history is necessary. To represent this model, it uses a left-subtree-child and right-subtree-sibling binary tree. The left-subtree represents the continuous conflict-resolved operations. Each node will have a right-child representing a conflicting operation, thus creating a different branch with its own history by adding left children to it. Finally, it proposes a set of algorithms to insert operations depending on their category — NMO or MO — and undo operations.

Citro, McGovern, and Ryan [16] present an algorithm to handle both exclusive and non-exclusive conflicts. Exclusive conflicts are those in which operations leading to it neither can be realized at the same time nor can be executed in a specific order since the operations leading to the conflict overwrite each other, so it is impossible to define an order for them to execute maintaining the intention of both users. Non-Exclusive conflicts are those in which the operations can be realized simultaneously, being handled using conventional consistency management techniques such as operational transformation [21].

The paper builds on top of existing techniques such as operational transformation and multi-versioning to handle both types of conflicts more effectively. Additionally, it does not suffer from a “partial intention” problem, since it allows delayed post-locking, based on the post-locking of objects technique proposed by Xue et al. [63]. With this technique, users can fully express their intentions on the object, allowing for a more informed decision when manually resolving the conflict; it also does not require a group leader or other conflict resolution roles in order to resolve conflicts.

Over the years, more implementations were published, each with its own advantages and compromises, but for the sake of keeping the focus of this document in a more general view of existing alternatives, I redirect the reader to the OT Wikipedia article, which has a good summary table[14] of most of the alternatives and their characteristics. Additionally, Sun, Chengzheng provides a FAQ page[15], serving as a knowledge base about the topic. This being a popular algorithm, many tools implement it and can be included in an application, such as TogetherJS[16], based around what they call the hub: a server that everyone in the session connects to which echoes messages to all the participants using WebSockets, or ShareDB[17], a real-time database backend supporting OT. The default implementation of ShareDB is an in-memory, non-persistent database with no queries. It is possible, however, to create connectors for other databases and connectors are provided for MongoDB[18] and PostgreSQL[19].

ShareDB is horizontally scalable using a publish and subscribe mechanism, making it relatively future-proof. Since it uses Express.js for the server, it is possible to use middleware to hook into the server pipeline and modify objects as they pass through ShareDB, adding flexibility.

Conflict-free Replicated Data Types (CRDT) are a different approach to the real-time coordination of inputs problem in a groupware system [52]. They can be operation-based [36] [10], similar to OT, or state-based [11] [7].

Operation-based CRDT are also called commutative replicated data types, or CmRDT. CmRDT replicas propagate state by transmitting only the update operation, similarly to OT. Replicas

[14]https://en.wikipedia.org/wiki/Operational_transformation#OT_control_(integration)_algorithms
[15]https://www3.ntu.edu.sg/scse/staff/czsun/projects/otfaq/#_Toc321146200
[16]https://togetherjs.com
[17]https://github.com/share/sharedb
[18]https://www.mongodb.com/
[19]https://www.postgresql.org/

receive the updates and apply them locally. The operations are commutative. However, they are not necessarily idempotent. Therefore, the communications infrastructure must ensure that all operations on a replica are delivered to the other replicas, without duplication, but in any order.

State-based CRDT are called convergent replicated data types, or CvRDT. In contrast to CmRDT, CvRDT send their full local state to other replicas, where the states are merged by a function that must be commutative, associative, and idempotent. The merge function provides a join for any pair of replica states, so the set of all states forms a *semilattice*. The update function must monotonically increase the internal state, according to the same partial order rules as the *semilattice*. Kleppmann, Martin provides good presentations[20][21] on the CRDT topic. Similarly to OT, CRDT also has available libraries and tools to be used in applications. Riak[22] is a distributed NoSQL key-value data store based on CRDT. Bet365 is an example of a large system using Riak[23]. Another example of CRDT implementation is Facebook's Apollo database[24], showing that CRDT can be used at scale as well.

A third and more recent alternative to the conflict resolution problem is Differential Synchronization [25] [2]. It provides an alternative to OT, being a symmetrical algorithm, as it has nearly identical code in both client and server; state-based, thus not requiring a history of edits to be kept by clients; asynchronous, since it does not block user input while waiting for the response over the network; network resilient, convergent, suitable for any content for which semantic diff and patch algorithms exist; and highly scalable. A working example of this algorithm can be seen in MobWrite[25].

As it can be noted, real-time applications usually apply a replicated architecture, where each client has its own replica, a user may directly edit its own version, realizing its effect immediately. The effects are then propagated to other clients. Propagation can be operation-based [48] [56] [55] [62] or state-based [25]. Chengzheng, David, Agustina et al. [15] state that most real-time co-editors, including those based on OT and CRDT, have adopted the operation propagation approach for communication efficiency, among others. It continues by comparing both solutions. Firstly, CRDT solutions have significantly higher time and space complexities than OT solutions, as revealed in [57]. Secondly, CRDT has a higher initialization cost since it needs to represent initial characters on the document, whereas OT's data structures start empty. This can have a big impact on session management and handling users that enter mid-session. Thirdly, OT does not have any additional time or space cost for non-concurrent operations. The operations buffer can be emptied with garbage collection — as was previously mentioned, the system can reset some data structures every time it is quiescent. CRDT, on the other hand, will have similar time and space costs regardless of handling concurrent or sequential operations, as all operations will be applied in the internal object sequence, which can never be emptied unless the document itself is emptied.

[20]https://www.infoq.com/presentations/crdt-distributed-consistency/
[21]https://gotocon.com/berlin-2016/presentations/show_talk.jsp?oid=7910
[22]https://riak.com/introducing-riak-2-0/
[23]https://bet365techblog.com/riak-update
[24]https://dzone.com/articles/facebook-announces-apollo-qcon
[25]https://code.google.com/archive/p/google-mobwrite/

Finally, OT does not have any additional time or space cost for local operations, as they will never be concurrent with any operation in the Requests Log. CRDT, on the other hand, will have similar time and space costs regardless of handling local or remote operations, as all operations will be applied in the internal object sequence. The longer time the local operation processing takes, the less responsive the co-editor is to the local user. Herron [1] also compares OT to CRDT, looking at the implementation in a final application, revealing some examples and problems that can arise, such as the unwanted cursor movement.

In a more general way, Sun et al. [58] present different types of conflict resolution in a creative editing system. The **preventive resolution** prohibits concurrent work on two same objects, avoiding conflicts altogether; **eliminative resolution** eliminates both operations when in conflict, eliminating the operations' history; **arbitrative resolution** elects one of the operations to be kept; **preservative resolution** keeps both options in different versions, so that the users can choose an alternative later; **creative resolution** produces a new operation, combining the two conflicting operations.

It further elaborates on the basic structure of a creative conflict resolution. There are three main components layered on top of each other. On the base, there is a conflict detector that simply detects if there are conflicts between operations. Next, there is a conflict synthesis component that creates the new operations based on the conflicts. Finally, there is the conflict management User Interface (UI), which is responsible for notifying users about conflicts and allowing them to select the desired conflict resolution strategy.

As important as the algorithm for solving conflicts themselves, the testing structure is also relevant. And in this scenario, creating relevant tests is more complex since they do not depend solely on a user's inputs and output assertions. Yu et al. [66] propose a language to define test suites based on actors and a timeline of actions portrayed by them. This allows the developer to completely specify the actions of all the participants and their interactions in one place, asserting effects more clearly.

In summary, there are three alternatives for the conflict resolution in real-time problem: OT, CRDT, and Differential Synchronization, the latter having less development in the industry and not much production use. The two main strategies are operation synchronization (OT) and state synchronization (CvRDT). Operation synchronization broadcasts operations to be replicated among peers, whereas state synchronization broadcasts the state updates directly. These can be used for the automatic merge of conflicting operations, however, multi-versioning and creative resolution are techniques that can be applied for specific conflicts.

2.4 Reputation System

There are multiple examples of how reputation can be used in multi-user systems and how it can affect group dynamics. Many refer to it as a solution to "Group Recommendations", which are based in **trust** among participants, whereas others mention its ability to induce cooperation. Haveliwala [27] shows how the PageRank algorithm can be personalized so that each link among nodes has a different weight in order to express a dynamic preference among nodes. Andersen et al. [8] demonstrates multiple trust-based recommendation systems and how they comply with a set of relevant axioms. Most importantly, it shows how the aforementioned personalized PageRank (PPR) algorithm can be used to simulate a trust network among peers by linking users with differently weighted connections. The greater the weight, the more a user trusts another, and the most likely it is for the Random Walk algorithm to choose that "path of trust". The latter also shows that PPR satisfies three out of five relevant axioms: **Symmetry**, **Positive Response**, **Transitivity**, but not Independence of Irrelevant Stuff and **Neighborhood Consensus**.

- **Symmetry**: Isomorphic graphs result in corresponding isomorphic recommendations (anonymity), and the system is also symmetric
- **Positive response**: If a node's recommendation is 0 and an edge is added to a + voter, then the former's recommendation becomes +.
- **Transitivity**: For any graph (N, E) and disjoint sets $A,B,C \subseteq N$, for any source s, if s trusts A more than B, and s trusts B more than C, then s trusts A more than C.
- **Independence of Irrelevant Stuff (IIS)**: A node's recommendation is independent of agents not reachable from that node. Recommendations are also independent of edges leaving voters.
- **Neighborhood consensus**: If a nonvoter's neighbors unanimously vote +, then the recommendation of other nodes will remain unchanged if that node instead becomes a + voter.

Dellarocas [19] shows examples of how multiple platforms handle their user reputations mechanisms. It also states that reputation systems can prevent badly intended users and deter moral hazard by acting as sanctioning devices. If the community punishes users that behave poorly and if the punishment compensates the "cheating" profit, then the threat of public revelation of a user's cheating behavior is an incentive for users to cooperate instead. It further elaborates on the reputation dynamics of a multi-user application:

- **Initial Phase** — Reputation effects begin to work immediately and in fact are strongest during the initial phase, as users try and work hard to build a reputation on themselves. Reputation effects may fail, however, when short-run users are "too cautious" when compared to the long-run ones and therefore update their beliefs too slowly in order for the long-run user to find it profitable to try to build a reputation;
- **Steady state** (or lack thereof) — In their simplest form, reputation systems are characterized by an equilibrium in which the long-run user repeatedly executes the safe action, also known as the Stackelberg action, and the user's reputation converges to the Stackelberg type (always collaborating and no cheating).

These dynamics have important repercussions for reputation systems. Dellarocas goes on to say that if the entire feedback history of a user is made available to everyone and if a collaborator stays on the system long enough, once they establish an initial reputation for honesty, they will be tempted to cheat other users sometimes. In the long term, this behavior will lead to an eventual collapse of his reputation and, therefore, of cooperative behavior.

Bakos and Dellarocas [9] present a model for a reputation system that explores the ability of online reputation mechanisms to efficiently induce cooperation when compared to contractual arrangements relying on the threat of litigation. It concludes that the effectiveness of a reputation mechanism in inducing cooperative behavior depends on the frequency of transactions that are affected by this mechanism, reminding that a minimum degree of participation is required before reputation can induce a significant level of cooperation. After this threshold is reached, however, the power of reputation manifests itself, and high levels of cooperation can be supported.

Dellarocas [18] concludes that reputation mechanisms can induce higher cooperation and efficiency if, instead of publishing updated ratings as soon as they are available, they only update a user's public reputation every *n* transactions, meaning a summary statistic of a user's last ratings. In settings with noise, infrequent updating increases efficiency because it decreases the adverse consequence of artificial negative ratings. At the same time, however, infrequent updating increases a user's short-term gains from bad behavior and thus the minimum future punishment threat that can sustain cooperation.

Tests were made in order to understand the reputation issues for users [4]. These were made in Waze, a navigational driving assistant with crowd collaboration for road events. Even though this and zerozero.live are somewhat different, some parallelisms can be made, and some gathered information still applies. They concluded that it was hard for users to recognize where the information came from and if it was reliable at all. Furthermore, users did not care much about their reputation when submitting information (i.e., if they heard about some road event, they would publish it without verifying it), maybe this is somewhat different from our use-case of sporting events, as users are either actually watching the game, or following it from a reliable source. Additionally, when users knew the source of data, they tended to trust people in their close circle (e.g., family and friends), and the main conclusion is that the app needed to convey the reputation of the source better to let the consumers know how much they can or should trust the source.

Resnick et al. [51] elaborate about reputation systems and their generic importance on the web. It is more geared towards e-commerce examples where people investigate the reputation before interacting with each other. It mentions three important properties reputation systems should have:

- Long-lived entities that inspire an expectation of future interaction. If the entities are short-lived, their reputation matters little;
- Capture and distribution of feedback about current interactions (such information must be visible in the future);
- Use of feedback to guide trust decisions;

In the zerozero.live case, it might be hard to get expressive feedback from users regarding other users. Therefore, it is important to have some implicit voting in place. Additionally, users are more inclined to express feedback when they disagree than when they agree, which means that the lack of negative feedback must be considered positive feedback to balance the system. Besides, users won't see the reputation of other users beforehand in order to decide to interact or not, as they simply enter the event without knowing who is also there, so it is important that they can see the reputation, or a variant of it (i.e., some relative reputation based on the current group of users) while they are at the event (e.g., showing it next to the user's name).

Melnikov, Lee, Rivera et al. [44] present a dynamic interaction based reputation model (DIB-RM), which is further evaluated in [65]. It presents a method to measure reputation as a function of user interaction frequency, also contemplating a reputation decay if the users stop contributing to the platform.

The aforementioned method is also present in [17], where the authors present a way to harness the "wisdom of the crowds", very much in line with what is required in zerozero.live, since there is no express authority during the event. It presents an example of a document sharing system and the approach to rank the documents based on the number of readers, the author's reputation, the time dynamics of reader consumption, and the time dynamics of documents contributed by the user. This last one manifests indirectly but is still relevant: it means that if a user has less-frequent readers on their documents, their reputation will decrease, so the contribution to the main document's reputation — the one they are reading now — will be smaller. Reputation values scale between 0 and 1, and they stick to the following rules:

1. Every time a user consumes a document from an author, the author gains reputation according to:
$$newRep = oldRep + (1 - oldRep) * repReward \tag{2.1}$$
repReward is a constant between 0 and 1 and should consider the number of entities in the system. As the paper states: "If the number of expected consumers is in the order of hundreds or thousands, then an overly high value of *repReward* will potentially cause popular content to quickly converge towards 1 making it difficult to differentiate between similarly popular content".

2. Every time a user consumes a document, the document gains "reputation" — meaning popularity in this case — according to the same formula of 2.1;

3. In order to take time dynamics into account, reputation should decrease over time, so that a "rich-get-richer" paradigm can be avoided. This is achieved by the following equation (both for users and for documents):
$$newRep = oldRep * decayCoeff^{k} \tag{2.2}$$
decayCoeff represents how much the reputation will change, and k is the amount of time units that have passed since the last reputation update, i.e., for a time unit of "days", k will be 0 in the first 24h, 1 in the next day, 7 in a week, and so on. This decouples the algorithm from the logistics, since the algorithm can now run in a fixed frequency, independently of

the time units, and every time it re-calculates, it will give an accurate value. However, if for example the time unit is “day”, and the algorithm updates every week only, there will be an offset of 6 days in which the value will be outdated.

4. Users with higher reputation matter more when calculating the document reputation changes:

$$newRep = oldRep * repConsumer * B \tag{2.3}$$

B is a constant within [0, 1] representing to what extent the user reputation *repConsumer* will influence the document’s reputation.

This system can be adapted and applied in zerozero.live if we map user inputs in an event as documents. However, we will be ranking users instead of inputs — “documents” in the analogy — even though they will also have reputation values. This will be explained in more detail in Section 3.2.5.

2.5 Similar platforms

On a basic level, this is a sporting-event following app. A similar platform would be 365scores.com[26], which offers the following of the same events in real-time; however, it does not offer the community-input feature of this proposed work.

Another platform that enables live viewing of sporting events is mycujoo.tv[27]. This enables the teams to live stream the game with video and mark specific events as they happen so that the viewers can revisit those moments in the video. It, too, lacks the community input feature when inserting the events; it is more geared towards the clubs sharing ability rather than the fans’.

This leaves zerozero.live as a singular app that will allow fans to contribute to the games’ events in real-time, increasing engagement, which can be complemented with the enormous football-related database which can provide real-time statistics about the game.

2.6 Similar work

Castro, João [49] has developed an application with the same goal as a Master’s Thesis as well. However, this work will not be a continuation of Castro’s work or use any of its code. It will benefit solely from the insights it can give, being a work with the same goal, with high importance in terms of literature review.

Castro’s work focused mainly on the reputation system as a conflict resolution strategy (i.e., the user with the most reputation wins an argument over the user with less reputation). While this is a valid approach to start with, it has many limitations such as highly-reputed users abusing their power in the real world. Further discussion about reputation systems in the literature is shown in Section 2.4. However, this work intends to apply a different technique that, while harnessing the

[26]https://www.365scores.com/pt/pages/about
[27]https://mycujoo.tv/en/about-us

advantages of a reputation system, aims to prevent the problems that could arise when used by real users. One of them would be using different conflict resolution strategies, depending on the conflict strategy (i.e., a conflict in the game score is way more important and thus cannot be solved by blindly applying a reputation comparison than, say, a mistake on the player substitution). The way of solving conflicts in terms of User Experience is also a matter of study. We don't want to fact-check every user input and disturb every other user experience with it, will at the same time guaranteeing the truest story possible. Finally, this work will have an "Offline Availability" goal as well, which is of great relevance in the real world, as the connectivity is not always the best, and many consistency problems result from it; thus, it's only fair that it is included in the areas of study regarding this application.

2.7 Summary

From the literature review, it is possible to realize that in order to achieve a real-time effect on the Web, one needs to use a real-time communication protocol such as WebSockets. To maintain work progress while offline, the application needs to store the interactions locally, through Web-Storage tools, such as *localStorage*. Conflict resolution is a highly researched topic with two main solutions currently used widely: OT and CRDT. These are available through multiple tools and libraries such as TogetherJS[28], ShareDB[29] CouchDB[30], PouchDB[31], Automerge[32] and GUN[33]. Finally, regarding reputation systems, it was found relevant for the reputation to be time-dependent, as well as reputation dependent, in the way that highly reputed users influence others' reputation more. It is also important for users to know who they are interacting with, in terms of reputation, as they can better filter the information given and defy it when appropriate. In the next chapter, the problem is presented in more detail. Chapter 3 uses the information in this chapter to propose some solutions to the problem exposed in this work.

[28]https://togetherjs.com
[29]https://github.com/share/sharedb
[30]https://couchdb.apache.org/
[31]https://pouchdb.com/
[32]https://github.com/automerge/automerge
[33]https://gun.eco/

Chapter 3

Problem Statement

This chapter describes the problem tackled in this dissertation, including the planned features and the expected result. First, it presents the features to be developed. Then, a set of proposed solutions are presented.

More information on how these features are planned to be executed is given on the next chapter.

3.1 Problem Definition

As mentioned in the Introduction (Chapter 1), the project's goal is to develop a web application that allows users to follow a sporting event in a real-time chat-like environment where everyone can input game events. For users that are just following, it would work as live coverage of the event; for contributors, it should be resilient to network failures due to stadiums' Wi-Fi limitations. Due to the above goal, some necessary features start to surface, such as real-time conflict resolution and the inherent reputation system for tie-breaking when necessary.

A prototype that allows event submission is already available, and since it is using React, which is an appropriate technology for this task, it will be kept. Some features still need to be polished, but most of the UI is already done, which will allow a bigger focus on the actual real-time conflict resolution problem.

The features of the prototype are described as follows, in User Story format[1]:

US1 As a user, I want to be able to join a sport event channel, where I can see details about the event in real-time (either pre-filled or contributed by other users), so that I have information about what is happening in the event.

[1]In software development and product management, a user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end-user or user of a system.

US2 As a user, I want to be able to be able to post event updates while on an event channel, in a chat-like interaction, so that I can easily contribute in an input experience I recognize.

- Event updates include: Starting players, Goals, Fouls, Set-Pieces, Cards, Substitutions, Game-Time information, and generic match information

US3 As a user, I want to be able to use the application while in offline mode, and have it synchronize once the connection is resumed, so that I don't lose information nor focus when my connection drops.

US4 As a user, I want to see a value representing the reputation of other users in a given event channel, so that I have a basis on which to decide if I trust them.

US5 As a user, I want to be able to delete inputted events, so that I can let others know that they might not be true and manifest my intention to change it.

US6 As a user, I want to be able to see if there are any pending conflicts to be resolved, so that I can clearly see if I need to solve any conflicting information.

US7 As a user, I want to be able to resolve any pending conflicts, so that I can keep the event's history clean and understandable.

US8 As a user, I want to be able to join an event channel mid-session, being able to see the previous information, so that I have more flexibility, not losing context if I arrive some time late.

US9 As a user, I want to be able to see the events generated on the ZeroZero platform (not ZeroZero Live), so that I can follow the game even if don't want to comment about it.

US10 As an official ZeroZero reporter, I want to be able to input events on the old platform, and have them synced to ZeroZero Live, so that I can use other tooling that depends on it to provide statistics, and still be able to comment manually on ZeroZero Live.

3.2 Problem Solution

In this section, a preliminary approach for the problem is presented. In some cases, it is still not possible to make a clear decision, as it requires implementation and further testing to validate that it actually works and really fits the need. These proposed approaches are based on the research done in Chapter 2, projected into the application domain and specific needs. This chapter is divided into sections, which will more precisely elaborate on their respective topic, presenting one or multiple proposed solutions for that specific area.

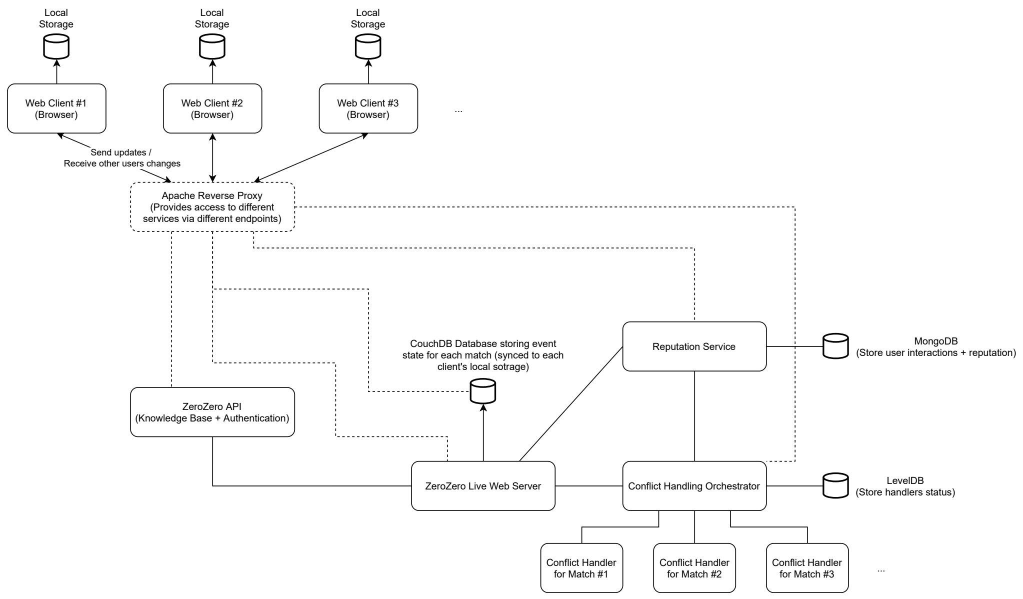
Local Storage
Local Storage
Local Storage
Web Client #1 (Browser)
Web Client #2 (Browser)
Web Client #3 (Browser)
...
Send updates / Receive other users changes
Apache Reverse Proxy (Provides access to different services via different endpoints)
CouchDB Database storing event state for each match (synced to each client's local sotrage)
Reputation Service
MongoDB (Store user interactions + reputation)
ZeroZero API (Knowledge Base + Authentication)
ZeroZero Live Web Server
Conflict Handling Orchestrator
LevelDB (Store handlers status)
Conflict Handler for Match #1
Conflict Handler for Match #2
Conflict Handler for Match #3
...

Figure 3.1: High Level Architecture design of the zerozero.live system

3.2.1 Knowledge Base

This project will benefit from the existing structure of zerozero.pt knowledge base, and will integrate with it by pulling events data from it, as well as publishing the event inputs generated by the users, storing the event history.

3.2.2 High Level Architecture

Figure 3.1 presents the proposed high level design for the application. The ZeroZero API is already available. The relevant parts are the ZeroZero Live Web Server, which will serve the web pages to the clients, mediating the state synchronization among them. It will communicate with the existing ZeroZero API for authentication and access to existing event information. It will also communicate with the other two micro-services. Reputation Service will handle the reputation calculation regarding the interactions during the event. Conflict Handling Service is responsible for handling the conflicts in real-time, resolving them automatically whenever possible, and cleaning up the event state. The Web clients will contain the frontend web pages with the logic to store changes locally and allow for offline working, as well as listening for the real-time updates to keep the state synchronized.

3.2.3 Offline Availability

According to the research done in Chapter 2, there needs to be a way of storing state locally if the user is offline. Since this application is a web application, from the alternatives presented, **localStorage** seems to be the most available among browsers[2], while fitting the project's needs.

[2]https://caniuse.com/?search=localstorage

Currently, in the already existing “Proof-of-Concept”, there is an already implemented solution, which uses a local queue of requests stored in *localStorage*, so as not to overwhelm the server with changes every second. This queue is “dumped” every 10 seconds. That is, every 10 seconds, the operations batched in the queue are sent to the server. While this allows for the prevention of operations loss in case of sudden network connectivity failure, it might not be the best approach in terms of real-time and user experience. For example, let’s say that a user generates a conflicting operation, but that operation is only sent 10 seconds later. Only then will the user see that effect on their device, when it could have happened sooner.

My proposal is to reduce the waiting time to a more reasonable 2 seconds, which, while preventing the overwhelming of messages to the server, reduces notification time and still lets the user cancel the operation before it is sent.

3.2.4 Conflict Resolution

This is a topic on which there is no clear solution. As it was seen in the research, the most well-known and used approaches are OT and CRDT. The community cannot, however, elect a clear winner[3], they are just *different*. On the OT side, a common approach is to use ShareDB[4], together with the JSON OT type definition[5].

Another alternative would be to use CouchDB[6] with its web browser counterpart, PouchDB[7]. It allows for replication of state among all users, with complete control on conflict resolution: When CouchDB encounters a conflict scenario, it arbitrarily chooses a winner, deterministically; however, it keeps the conflicting version as well, which can be used to solve the conflict with custom application logic, for example, based on a reputation system, or on merging the two inputs — Creative Resolution (Section 2.3)

On the CRDT side, there are two options: Automerge[8] and GUN[9]. Since the latter’s documentation seems to be lacking, I am removing it from the options pool. Automerge is flexible enough, allowing for server-client network architecture, as well as peer-to-peer, and it works like JSON CRDT were described to work: each user has a local copy of the JSON state, which can be locally mutated, even when offline, and it will sync automatically with other nodes. It works similarly to the CouchDB/PouchDB conflict handling, in being as much automatic as possible but letting the application know about existing conflicts and handle them.

At this point, the ShareDB option seems to work on a lower level than the other two, and it might be unfeasible to have good results in the expected timeframe. Between CouchDB/PouchDB and Automerge, there is no clear winner, the only distinctive characteristic being that CouchDB

[3]https://news.ycombinator.com/item?id=22039950
[4]https://github.com/share/sharedb
[5]https://github.com/ottypes/json0
[6]https://couchdb.apache.org/
[7]https://pouchdb.com/
[8]https://github.com/automerge/automerge
[9]https://github.com/amark/gun

is more mature and has been used in production for much longer. Nevertheless, the wise decision here would be to try both in a small "Proof-of-Concept" and further verify their usability.

3.2.5 Reputation System

As was mentioned in the Literature Review (Section 2.4), an effective method to achieve a fair reputation system, which takes into account the time dynamics of user interactions as well as their current reputation, is to implement a personalized PageRank algorithm. It takes into account the reputation of users when calculating vouching or invalidation in order to achieve a weighted voting system so as to provide long-term reputable users with a prize for their good behavior. Recalling the system present in [17], there are 4 rules involved in adapting the system to our use case:

1. Every time a user consumes a document from an author, the author gains reputation;
2. Every time a user consumes a document, the document gains "reputation" (i.e., popularity);
3. In order to take time dynamics into account, reputation should decrease over time, so that a "rich-get-richer" paradigm can be avoided (both for users and for documents);
4. Users with higher reputation matter more when calculating the document reputation changes;

With this in mind, I propose the following rules to adapt this to our scenario:

- Every time a user agrees with an input, he will improve the input's reputation according to rules 2 and 4;

$$newIR = oldIR + (1 - oldIR) * maxRepReward * userRep * userRepInfluence \quad (3.1)$$

  where *IR* means *Input Reputation*

- Every time a user disagrees with an input (either by inputting a real-conflicting input or reporting as false/inaccurate) he will worsen the input's rep according to rules (2's reverse) and 4;

$$newIR = oldIR * (1 - maxRepPunishment * userRep * userRepInfluence) \quad (3.2)$$

- Every time a user submits a falsely-conflicting input, meaning that both users submitted the same information resulting in duplicated information, it should act as an explicit agreement with the other user's input, so it should count more, according to an *explicitAgreementBonus* constant, which must be greater than zero to achieve the bonus effect;

$$newIR = baseIRIncrement * (1 + explicitAgreementBonus) \quad (3.3)$$

  where *baseIRIncrement* is calculated based on 3.1

- The user gains reputation according to the average of its inputs' reputations. Only takes into account the latest inputs, referring to the last event which will trigger the reputation update;

$$newUserRep = oldUserRep + (1 - oldUserRep) * \frac{\sum IR}{numInputs} \tag{3.4}$$

- Each user has a reputation decay according to rule no. 3, the time unit should be 1 week since there's at least one relevant game per week. This prevents users that generate a lot of inputs in a single game to enjoy their reputation boost for many more games, since they need to be consistent every week: it matters more if they make an input every week than 20 inputs once every 2 or more weeks. This decay is on a higher level than the events, creating 20 inputs in an event is roughly the same as 1 input in an event (since the football events last around 90 minutes)

$$newUserRep = oldUserRep * decayCoefficient^{timeSinceLastUpdate} \tag{3.5}$$

- The reputation values are updated at the end of each event, according to the event's history.

3.3 Summary

In summary, the problem to be tackled is the development of a multi-user real-time application allowing the coverage of sporting events, with offline tolerance. That entails the existance of conflict resolution strategies, as well as a reputation system, which should take time into consideration, making reputation more dynamic.

Chapter 4

Solution

This chapter presents the solutions to the problems exposed in Chapter 3. It explains the rationale behind them, as well as why they were preferred over their alternatives.

First, the general architecture is described, presenting the multiple components of the application. Afterward, each component is described in more detail, regarding its functionality and implementation.

4.1 General Services Architecture

This section describes the general architecture of the system. It is composed of multiple microservices, each handling a specific part of the problem. The main services are the ones dealing with the frontend application — the website itself —, the conflict handling, and the user reputation. Additional services are build in order to achieve a functional whole, and they will be described below. Fig. 4.1 shows the system architecture design and how the services connect to one another.

All of these services are managed using Docker [1]. Each service is built in a Docker Container, allowing it to be easy deployed and replicted across machines for scalability and fault tolerance. It also ensures that every deployment is identical, no matter the host machine and environment it is deployed on.

Additionally, Docker Compose [2] is used to define how the different containers in a multi-container application should interact with each other, allowing us to start the application as one entity, instead of multiple isolated containers. It conveniently provides an DNS so that containers can reach each other by their service name.

Finally, a microservice architecture promotes isolation and independence of services, which in turn allows for easier development and testing. Different teams can work on each service, which has a single purpose, and they only need to agree on the interface and how they can communicate between them.

[1]https://www.docker.com/

[2]https://docs.docker.com/compose/

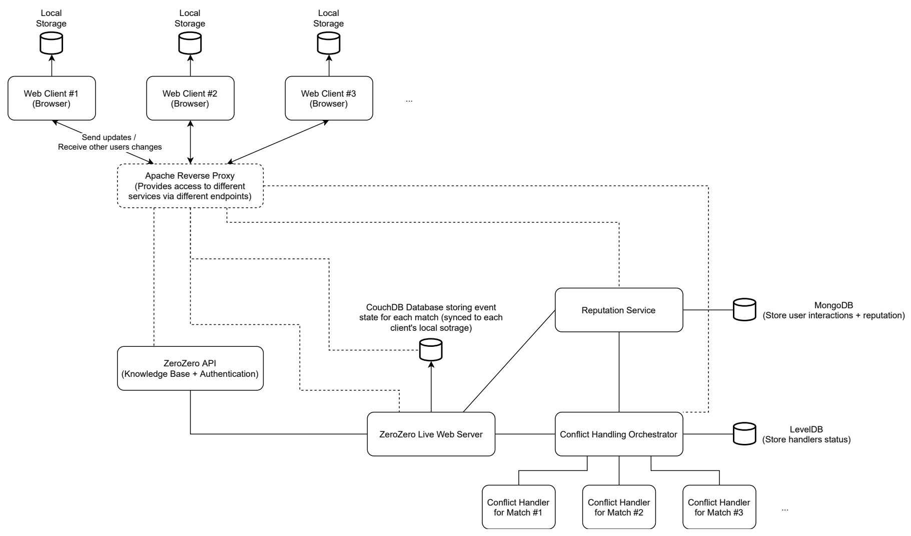
Local Storage
Local Storage
Local Storage
Web Client #1 (Browser)
Web Client #2 (Browser)
Web Client #3 (Browser)
...
Send updates / Receive other users changes
Apache Reverse Proxy (Provides access to different services via different endpoints)
CouchDB Database storing event state for each match (synced to each client's local sotrage)
Reputation Service
MongoDB (Store user interactions + reputation)
ZeroZero API (Knowledge Base + Authentication)
ZeroZero Live Web Server
Conflict Handling Orchestrator
LevelDB (Store handlers status)
Conflict Handler for Match #1
Conflict Handler for Match #2
Conflict Handler for Match #3
...

Figure 4.1: Architecture design of the zerozero.live system

4.1.1 Reverse Proxy

In the zerozero.live deployment, only the 443 — HTTPS — port is exposed to the internet, thus, to expose multiple services, it cannot be done in multiple ports, at least not from the outside. Due to this, a reverse proxy is required to map each service — and its specific port — to an endpoint on the main domain zerozero.live:

- **/** Redirects to the client application — the website
- **/api** Redirects to a small proxy that communicates with the ZeroZero API
- **/db** Redirects to the CouchDB, holding the match events in real time
- **/manager** Redirects to the conflict handling services orchestrator, which monitors the conflict handlers for each match
- **/reputation-manager** Redirects to the reputation service, which registers the interactions and calculates the reputation updates for users

4.1.2 CouchDB

CouchDB is a document-oriented, multi-master database, which allows for easy replication between nodes. This enables data to flow seamlessly from web browsers to the database and vice-versa, powering offline-first applications while being developer friendly, since the API that interacts with the local database — powered by PouchDB — is the same that interacts with the remote database — CouchDB.

When compared to the previously analyzed alternatives, namely ShareDB and CRDT solutions such as Automerge, CouchDB is older and thus more *battle tested* than the rest, while also having frequent releases — the latest stable to date is from September, 2020. ShareDB has also been around for some time, since 2013, however, it offers a lower level alternative to conflict resolution using OT. It allows integration with any database and offers similar features to CouchDB, but I have found it difficult to program custom conflict resolution strategies, very much important to this work. From the three, Automerge is the youngest (2017), and it relies on a different approach to conflict resolution, based on CRDT. As it does not require a central server, it merges documents automatically and I could not find how to add custom resolution of conflicts, which led me to choose the CouchDB alternative.

It is based on documents which are JSON objects with at least an id and a revision identifier. CouchDB then uses this to detect conflicts — if it has the same id, it may conflict — and to resolve them. It works by creating a tree of documents, where the leaf nodes are the current winners, branches represent conflicting revisions, which are kept to be resolved by the application if needed. CouchDB will choose a winner automatically and deterministically, so that every client has the same version of the data. However it is possible to resolve the conflicts afterwards, by choosing a conflicting version as the winner, which appends a new node to the tree, after the previous winner, making the previously conflicting document the new leaf node, and thus the new winner for that document id.

Finally, this has the advantage of having a browser version, powered by PouchDB, which replicates the database to the browser's IndexedDB, and its API is the same as CouchDB's. This makes the application offline first, since all the changes are made to the local database, and continuously synchronized to the remote database.

4.1.3 Conflict Handling

4.1.3.1 Conflict Detection

After generating match events, conflicts may arise. The conflicting events are the ones whose documents share the same id. If we want documents to conflict, they cannot have unique ids, so the following schema was defined for each event id:

$$eventMinute - eventMinuteExtra - eventCategory \tag{4.1}$$

By having this schema, when any event from the same category is inserted, refering to the same game time as another, they will conflict. Event categories are defined in such a way that events from different categories do not conflict. Some category examples include goal related events (*Goal*, *Own Goal*, *Big Scoring Opportunity*, etc.), card related events (*Yellow Card*, *Red Card*), or time events (*Start Whistle*, *End of 1st Half*, *Final Whistle*, etc.)

This will ensure that we have a bascic conflict detection, however it can easily be noticed that even in the same category there may be events on the same minute: There can be a *Big Scoring Opportunity* immediately followed by a *Goal*, or even two *Yellow Cards* for two different players, or even the same one.

To improve the experience, a conflict handler is needed, in order to automatically solve conflicts as best as possible.

4.1.3.2 Conflict Handler

As previously noted, the Conflict Handler will help resolve some existing conflict automatically, based on the conflicting events and the match context, in order to only rely on users to resolve pending conflicts as a last resource.

This is a Node.js process that will use PouchDB to connect to the remote CouchDB and listen for changes. It has two main responsibilities:

- If the document does not have conflicts sync with ZeroZero API, by inserting, editing or deleting the event from their database, depending on the operation
- If the document has conflicts, try to resolve them according to the specified rules, by either deleting the conflict and keeping the current winner, deleting the conflict and adding it *after* the previous winner, to choose it as winner, or keeping both documents, by deleting the conflict and adding a replicated document with a unique suffix added to the id so that it won't conflict again.

As it was mentioned earlier, all CouchDB documents have an *_id* and a *_rev* (as in *revision*). When *putting* a document — CouchDB has a *put* operation that allows insertions, edits and deletions, depending on the given argument — if there is already an object with the same *_id*, CouchDB will try to update it, however if the *_rev* does not match, a conflict is generated. For insertions, it only requires an object with at least an *_id* field. For edits, it requires the object with an *_id*, referring to the document to be edited, and a *_rev*, corresponding to the current revision of the document to be edited. For deletions, it requires the *_id*, *_rev* and a *_deleted* field with a value of *true*, which is a special case of the edit operation. The deleted documents are not really removed from the database until a compaction operation[3] is performed to clean the deleted documents and older revisions are stripped, leaving only the necessary data required for conflict resolution operations.

Document revisions have two components: a number indicating their level in the tree of updates of that document, and a unique identifier. When a document is inserted, the generated revision starts with 1, and every time it updates, the successive revisions will have consecutive values e.g., 2, 3, and so on. This means that, considering two users are synchronized with the remote database which has a document with a revision of “1-xyz”, if they both want to update that document at the same time, they will both try to *put* a document with “1-xyz” as its parent node. When CouchDB receives the requests, it will generate a revision for each, starting with 2, since their parent node revision starts with 1. One of them will be chosen as winner, and the other will conflict, since there are two documents on the same level: 2. The winner is automatically and deterministically chosen by CouchDB, so that every replica of the database is consistent, and it

[3]https://docs.couchdb.org/en/stable/maintenance/compaction.html

will store the conflicts for each document as well, then it's up to the application layer to resolve the conflict as needed. The primary focus of this work starts here, by applying automatic conflict resolution strategies between the automatic CouchDB winner selection and the manual user conflict resolution that may happen afterwards.

4.1.3.3 Base conflict resolution strategies

A base set of operations was defined in order to help solve conflicts. As per design, every conflict is resolved indivdually, i.e., if there are three documents conflicting, the conflict resolution will take palce two times, one for each conflict pair (current winner, conflict). They all receive the pre-selected winner and conflict documents as arguments:

Leave Conflict This is the most basic resolution operation. It does nothing and leaves the conflict in the database, so that it can be resolved later in the frontend application;

Ignore Conflict This is another basic resolution operation. It simply deletes the conflicting revision, basically agreeing with the automatic decision done by CouchDB;

Choose a Winner This operation deletes the document passed as conflict and inserts a new revision as the winner;

Keep Both This operation resolves conflicts by keeping both documents. Since there can't be two documents with the same id in CouchDB, otherwise they would conflict, it first deletes the conflicting revision, creates a new revision for the current winner, and inserts a new document equal to the conflicting one, but with a UUID[4] appended to its default id. This ensures that it won't conflict with the existing document, however it has the downside that it won't conflict with anything ever again. While the initial document remains in the database, there will always be a "representative" of that set of ids, that will trigger conflicts every time a new document with that id is inserted. If that document is deleted, there is a workaround in place that selects a document from those with the same base id, and removes the random suffix, so that there is always some event to detect the conflicts for future inserts;

Keep All This operation;

Additionally, there are specific resolutions that are triggered on the frontend — not automatically —- if the user resolves a conflict by choosing a winner or by keeping all conflicting documents:

Choose One - Frontend When choosing a winner, since there may be multiple conflicts at the same time, it needs to delete them all to fully resolve the conflict. However, if multiple users select different winners by each resolving the conflict differently, in the end they would end up delting all documents. In order to avoid this, after deleting

[4]A UUID (Universally Unique Identifier) is a random 128-bit identifier. [35]

the loser documents, a new revision for the winner is inserted, so that if those multiple users select different winners locally, their choices will still conflict with each other after each "round" of conflict resolution.

Keep All This operation is similar to the *Keep Both* operation in concept, but applied to *n* documents at a time, not only 2 documents. It does so by first deleting all conflicts corresponding to a document id, then inserting new documents equal to the deleted conflicts, but with a UUID appended to the id, to make them unique, and avoid the conflict;

4.1.3.4 Composite conflict resolution strategies

By leveraging the aformetioned strategies, it's possible to create complex rules that apply to different event types that conflict with each other, provided that they happen in the same match minute and belong in the same event category.

Force Winner By Event Id Chooses as winner the document that has a given event id;

Highly Conflicting; Ignore if same player If the player is the same, ignore conflict; Otherwise leaves the conflict intact to be resolved by the user;

Goal vs. Own Goal In case a user reports a goal from team A, but another reports own goal from team B, they are both referring to the same goal, so the own goal event is chosen as winner, since it is more specific. Otherwise, leave the conflict to be resolved by the user;

If same player, keep the one with given event id If the events refer to the same player, choose as winner the one that has the given event id; Otherwise leave the conflict to be resolved by the user;

If different player, use reputation; Otherwise ignore If the referred players are different, choose as winner the event whose user has the highest reputation; Otherwise, ignore conflict;

If same player, use reputation; Otherwise, keep both If the referred players are the same, choose as winner the event whose user has the highest reputation; Otherwise, keep both documents;

If same player, ignore conflict; Otherwise, keep both If the referred players are the same, ignore the conflict; Otherwise, keep both documents;

If same player, ignore conflict; Otherwise use reputation If the referred players are the same, ignore the conflict; Otherwise, choose as winner the event whose user has the highest reputation;

Substitution Handler If the involved players are the same, ignore the conflict; If there are intersecting players, i.e., an event mentions player A out for player B, and the other event mentions player A out for player C, the conflict is left intact to be resolved by the user, later. Finally, if the involved players are different, keep both documents;

Minute based resolution Receives a *minuteCondition* function that takes the documents' minute as argument, a resolution function to execute if true, and a resolution function to execute if false. If *minuteCondition* returns true, the first function is called, otherwise the second one is called;

Choose biggest reputation Choose as winner the event whose user has the highest reputation;

4.1.3.5 Synchronization with ZeroZero API

In order to fulfill User Stories US9 and US10, specific logic was implemented to fetch missing events from the API, and to add them after consensus is reached on the ZeroZero Live platform i.e., there are no conflicts.

That being said, on the same loop that listens for changes and verifies if there are conflicts to resolve, the handler also two more branches of action: when the event is being synced from the API into CouchDB, and when the event has no conflicts and was not synced from the API.

Starting with the latter, the handler needs to know which operation was done regarding each event, in order to call the correct ZeroZero API. Each document comes as if we were receiving a CouchDB's "put" call, meaning that we don't know explicitly if it was an insert, edit or delete. In order to know if it is a delete operation, we can check for the *_delete* field. As was mentioned earlier, this field is true when the document is deleted. Regarding the distinction between inserts and edits, it's not as simple. On a first approach, revisions were used: if the revision level was 1, it was obviously an insert, otherwise it would be an edit to some existing document. This seemed rational, and it works to some extent, but not always. Recalling CouchDB delete behavior explained above, documents are not really deleted until the compaction operation is run. That means that if a document is deleted, another with the same id can be inserted aftewards, but since the first was not really erased from the database, the revision level of the newly inserted event will actually be different than 1, since the history for that document id is still there. In this case, this approach would consider some inserts as edits which would cause multiple events from being synchronized to the API correctly. In order to fix this, a special flag was added to each document, meaning if the document was meant to be inserted or not. On the client side, every event creation would generate a document with the *insert* flag with a value of *true*. Whenever the handler inserted a document, it would edit it — create a new revision — setting the *insert* flag back to *false*, so that it was not inserted every time there were changes to it. The final approach was then to check for the *insert* flag in order to distinguish from insert and edit operations.

Regarding the synchronization from the ZeroZero API — their internal storage of events, managed by the older platform, and used to show the match events on their website — into ZeroZero

Live, it's a bit more complex. First it shouldn't be continuous, or it would easily cause problems, since we would be adding events there and immediatelly receiving them back would probably cause some unwanted conflicts. With this in mind, every time there is a match page load — on first visit, or on page refresh — for every event that is on the ZeroZero API that has no correspondent document on CouchDB, there will be a synchronization attempt in order to add it to the CouchDB database. The reason because this is not as simple as inserting every event that is currently not in the ZeroZero live database yet, is because of potential conflicts: Events that are already on the ZeroZero side, should not conflict with each other, and they also should not conflict with CouchDB documents, on the synchronization phase, at least.

First, let's explain how events are known to be synchronized or not. Every time there is an insert of a document on ZeroZero Live, and it is synchronized *to* the ZeroZero API — as per the logic described above — the API returns an id for that event, that is different from the document's *_id*. That id is used identify the event on the ZeroZero side, allowing us to call the edit and delete APIs on the events. When the CouchDB document is synchronized to the ZeroZero API — the insert API is called —, the document is edited to set the *insert* flag to false, as mentioned above, but it also sets the *id* field, meaning that the document has a ZeroZero's event counterpart.

Based on this, for every event coming from the ZeroZero API, we can check if there is a CouchDB document with a corresponding *id*, if there isn't, that event needs to be inserted into CouchDB.

For those that need to be synchronized, they are inserted as normal documents, as if they were inserted by the user, however, the document *_id* has an appended "synced_from_api" suffix, so that the handler will know to use the synchronization algorithm, instead of simply inserting the event.

Figure 4.2 shows the flowchart relative to the algorithm of synchronization of events into CouchDB. It first checks if there are documents starting with the same base *_id*. The base *_id* is the $eventMinute - eventMinuteExtra - eventCategory$ string used to identify documents and allow for conflict detection. Due to custom resolutions like *Keep Both* or specific document types that never conflict — such as comments — some documents will have a base *_id* followed by a unique part, in UUID format, thus it's important that it is ignored, otherwise some events would never be caught by the synchronization algorithm. It's also important to note that even if there is a document with a same base *_id*, they may not refer to the same event, and not even correspond to the same event type, it just means they happened at the same minute, and belong to the same event category. The only way there is to know if a synchronization candidate — i.e., the one with "synced_from_api" appended to the *_id* — is the same as an existing document, is by comparing their *id*s, not *_id*s, as that value comes from the ZeroZero API.

First, if there are no documents in the CouchDB database that have a same base id, we can know for sure that that event is not in the CouchDB database yet, thus, the "synced" document is deleted, and a clone is inserted with the correct *_id* — i.e., the base *_id* — so that it can conflict in the future with other events. However, imagine that while this is happening, someone inserts an event at the same minute with the same category. That will prevent us from inserting the

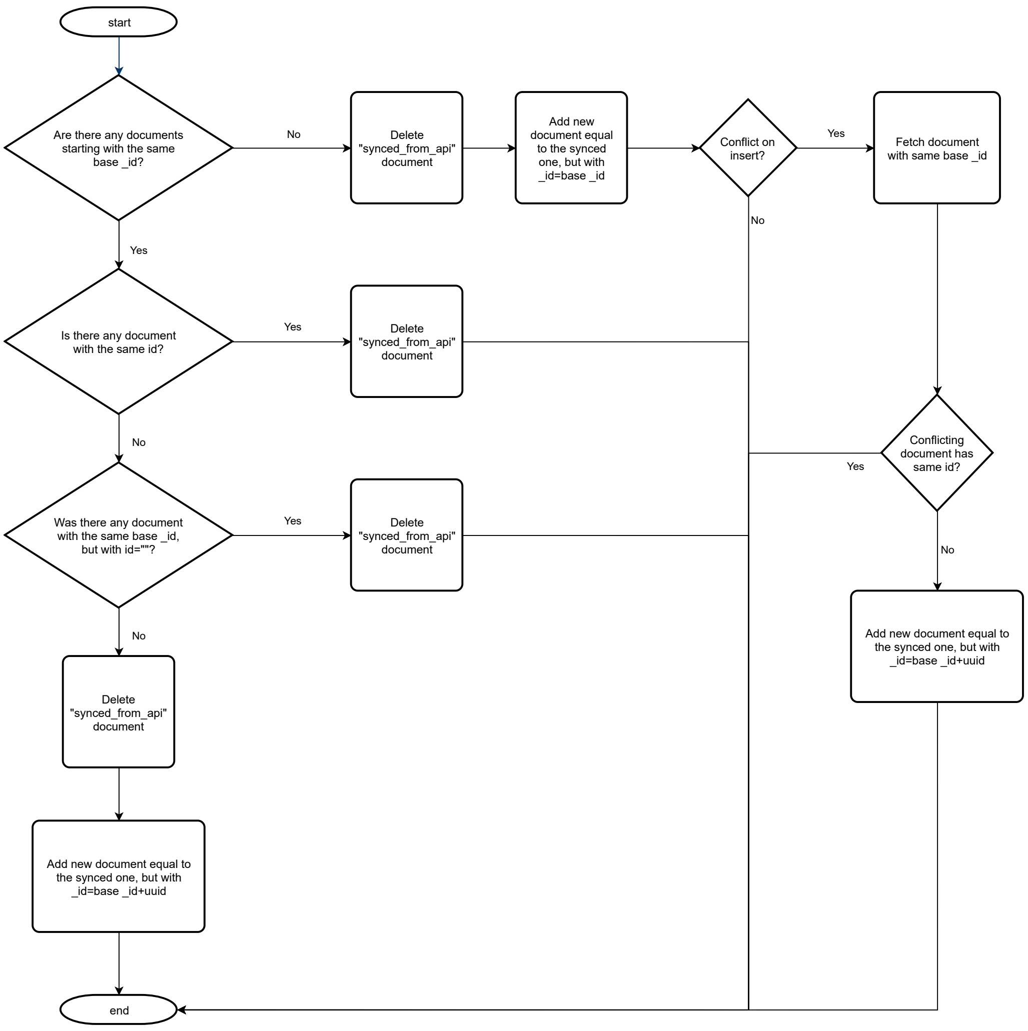
start
Are there any documents starting with the same base _id?
No
Delete "synced_from_api" document
Add new document equal to the synced one, but with _id=base _id
Conflict on insert?
Yes
Fetch document with same base _id
No
Yes
Is there any document with the same id?
Yes
Delete "synced_from_api" document
No
Conflicting document has same id?
Yes
Was there any document with the same base _id, but with id=""?
Yes
Delete "synced_from_api" document
No
No
Add new document equal to the synced one, but with _id=base _id+uuid
Delete "synced_from_api" document
Add new document equal to the synced one, but with _id=base _id+uuid
end

Figure 4.2: Synchronization of events from ZeroZero API into ZeroZero Live database

new version of the "synced" document, so there is a check in place that verifies the conflicting document and if it has the same id, the algorithm stops, since the event is already there, otherwise, it will add it, but with a unique uuid suffix, so as to avoid the conflict on insertion.

If there are any documents with the same base *_id*, however, it will try to find out if among those there is some that has the same id. If so, it will simply delete the "synced" document, it is already synchronized. If not, and if among the searched documents there's some that have an empty id — meaning they are in the middle of the process of getting one, after the insert call — the algorithm will delete the "synced" document as a preemptive measure, hoping it will be there in the future. The next time this algorithm runs, that document will already have an id, and the check above can be done.

Finally, if there were documents with the same base *_id*, but none had the same id nor had empty ids, the synchronization candidate is not present, but must be added with an unique suffix, to prevent insertion conflict.

TODO SHOW CONFLCIT RESOLUTION RULES (?) in annex

falar das chamadas ao rep service disagree vs explicit agree

4.1.3.6 Conflict Handling Orchestrator

In order to have multiple handlers at once, which is needed as there may be multiple matches happening at the same time, one needs to have a controller that handles their initialization, manages their status and stops them once they are not needed anymore.

That is the responsibility of the Conflict Handling Orchestrator. It is a simple Node.js application that exposes an HTTP API, that lets us initialize a conflict handler for a given match, list active handlers, stop a conflict handler for a given match, and *kill* any *zombie* handler references that might be left over after a crash. The state of the handlers is kept in a LevelDB[5] database, which is a simple key-value database. Each record will have the match id as the key and the value is an object containing the PID of the handler process for that match, as well as the timestamp of when the handler was started.

Even though the orchestrator is the parent process of the conflict handlers, in case the conflict handler process fails, the orchestrator is independent enough that it won't terminate as well. If the orchestrator itself fails for some reason, it is built to auto-restart instead of making the docker container restart, which would make it lose context and terminate all other processes, including child conflict handlers. This is why information about the handlers is kept in a separate database, instead of in-process memory, so that the orchestrator can recover all of it in case of failure.

POST /initConflictHandler/:matchId This endpoint launches a child process running a conflict handler for the specified match, as described in the previous subsection. It is made in such a way that it won't allow the launch of multiple handlers for the same match simultaneously. After a specified timeout, the child process is killed, to avoid wasting resources on finished matches.

POST /stopConflictHandler/:matchId This endpoint terminates a process associated with the given match.

GET /active-handlers This endpoint returns a list of active match conflict handlers, together with their respective PIDs.

POST /cleanup-zombies This endpoint will verify that every stored match handler is actually running. If not, it will update the database. This is important because after some failure, conflict handlers might not be running as the database shows, and this endpoint will restore the truth.

[5] https://github.com/google/leveldb

4.1.4 Reputation Service

In order to aid the automatic conflict resolution, in many cases, the reputation of the users will be used to determine which event to choose as the conflict winner. This is only done for conflicts that are not critical. For any important conflict — such as *Goal from team A* vs. *Goal from team B* — it is left for users to resolve manually on the UI.

The Reputation System is based on the fact that people that input conflicting events disagree with each other, as well as who edits or deletes an event. Every time a user receives an event from someone, they implicitly agree with it, until they act to mark disagreement, executing any of the actions described before. Finally, users can explicitly agree with each other, by inputting conflicting events, that are exactly the same i.e., if two users report a goal from team A at the same time, they are not disagreeing with each other, but rather agreeing. This is considered a more *powerful* agreement, since they both explicitly inputted the event, instead of just waiting for someone to do it and passively reading it. These are the three interaction types that are regsitered by the reputation service.

The Reputation Service is a Node.js microservice that exposes an HTTP API, and uses a MongoDB database to persistently store data. It has endpoints to register interactions between users and inputs and another endpoint to calculate reputation updates after a specific match interactions.

The database will store the interactions, which are documents containing the *matchId*, *eventId*, *userId*, *eventOwnerId*, *isProcessed* and *interactionType* fields. *matchId*, *eventId*, and *userId* uniquely identify an interaction and are used to prevent the same user from registering multiple interactions for the same event. Additionally, there are checks in place so that a user cannot register interactions for an event they have inserted. *isProcessed* is a boolean flag that marks the interaction as processed by the reputation calculation, so that it is not used multiple times when calculating reputation updates. Finally, the *interactionType* defines the type of interaction, which affects the reputation update calculation, e.g., *Agree*, *Disagree*, *ExplicitAgree*

Additionally, the database has another collection to store the user and their respective reputations, to be used in the conflict resolution. Each document has *userId*, *reputation* and *updatedAt* fields. The *reputation* represents the numerical value of the reputation of the user with id *userId*, and the *updatedAt* field is used to calculate the reputation decay — the more time passes after the previous reputation calculation, the more its reputation will decay due to inactivity.

WHY MONGO REP calc algorithm, maybe no endpoint respetivo? falta endpoint de get rep from user - para ser usado no conf.res

POST /:matchId/:eventId/:eventOwnerId/:userId/agree

POST /:matchId/:eventId/:eventOwnerId/:userId/disagree

POST /:matchId/:eventId/:eventOwnerId/:userId/explicit-agree

POST /:matchId/updateRep Talk about constant values choice e graficos com predicts e exemplos de user behavior

4.1.5 Web Application (Frontend)

Chapter 5

Planning and Methodology

TODO this should only talk about methodology (the user validations questionary, etc, maybe after solution in order to be able to discusss results)

5.1 Methodology

This work is intended to be tested on real users during its development. Thus, an agile methodology seems to be more fitting than a waterfall approach [13]. The most common are Scrum, XP, and Kanban. However, in a small team, Scrum and XP might be exaggerated, and Kanban should be more easily employed. Regardless of the methodology used, the really important aspect is to develop iteratively, employing some kind of "sprints" where work is focused on a set of immutable tasks, allowing user testing at the end of the Sprint for validation.

To get validation from users, I expect to collect usage information and simple forms regarding their opinion on the platform, mostly regarding usability and ease-of-use.

5.2 Planning

5.2.1 UX Monitoring

In order to better understand how the user is interacting with the application and pinpoint some aspects that might be worth improving, we intend to measure some aspects of the interaction. Next are some proposed metrics, which are relevant to this study. They are divided on their type for clearer understanding.

1. Performance metrics:

   (a) Number of automatically-unsolved conflicts during an event per user

   (b) Number of total generated conflicts

2. Self-Reported metrics, asked in the form of a Likert scale[1], when appropriate:

   (a) The tool allowed the user to narrate the game without issues
   (b) The user considered the number of conflicts... (1 - low; 5 - high)
   (c) The user believed the events to correspond to the match's truth
   (d) The conflicts were easily to locate
   (e) The conflicts were easy to solve

   (f) The user has used another mean of communication with friends while following the match in order to discuss it (*)
   (g) The user would use the tool again in the future (*)

   (h) Open answer to allow users to give whatever feedback they might have

(*) Yes or No questions

[1]A typical item in a Likert scale is a statement to which respondents rate their level of agreement. The statement may be positive (e.g., "The terminology used in this interface is clear") or negative (e.g., "I found the navigation options confusing"). Usually a five-point scale of agreement like the following is used: 1. Strongly disagree 2. Disagree 3. Neither agree nor disagree 4. Agree 5. Strongly agree

Chapter 6

Conclusions

TODO

This document had two goals. First, it allowed for better research on the existing work about the topics that will be covered by the application, providing a better context and improving future design decisions. Second, it served as an initial look at how the problem could be tackled, proposing some initial solutions and planning them.

This dissertation proposes to create a real-time web application to allow users to input sports events' updates while featuring an offline mode and automatic conflict resolution whenever possible. Throughout the document, three main areas were outlined: *Offline Availability*, *Conflict Resolution*, *Reputation System*.

Regarding *Offline Availability*, the application will require some local storage to keep users' work when they don't have internet access. Multiple solutions were analyzed, and *localStorage* seemed the best fit as it is widely available on browsers and provides plenty of storage space.

Concerning *Conflict Resolution*, multiple strategies can be used, most notably OT and CRDT. One of the first things to do in the project will be to try both of them and analyze which works best, based on the experience.

As for the *Reputation System*, some rules were analyzed, such as "Should the reputation value be available for everyone to see?", or "When to update the reputation values?". Every system is different, and different strategies must be tried and evaluated on the user base. However, a system based on time dynamics and "higher-reputation-counts-more" is proposed, due to the research shown in Chapter 2.

All of these decisions might impact UX, and we intend to study the effects of our design decisions on application usage. This is accounted for in the project planning.

6.1 Expected Contributions

At the end of this work, this project can have two main contributions. First is the actual application, which will allow many sports fans to connect to their local teams and follow their games with the community. Second, work regarding the testing of a multi-user real-time application can be useful for anyone developing such an application. I intend to create a tool on top of existing testing frameworks, allowing easier testing of these applications.

Appendix A

Services Architecture

This annex contains the architecture diagram of the implemented solution, representing all of its components, as explained in Chapter 4.

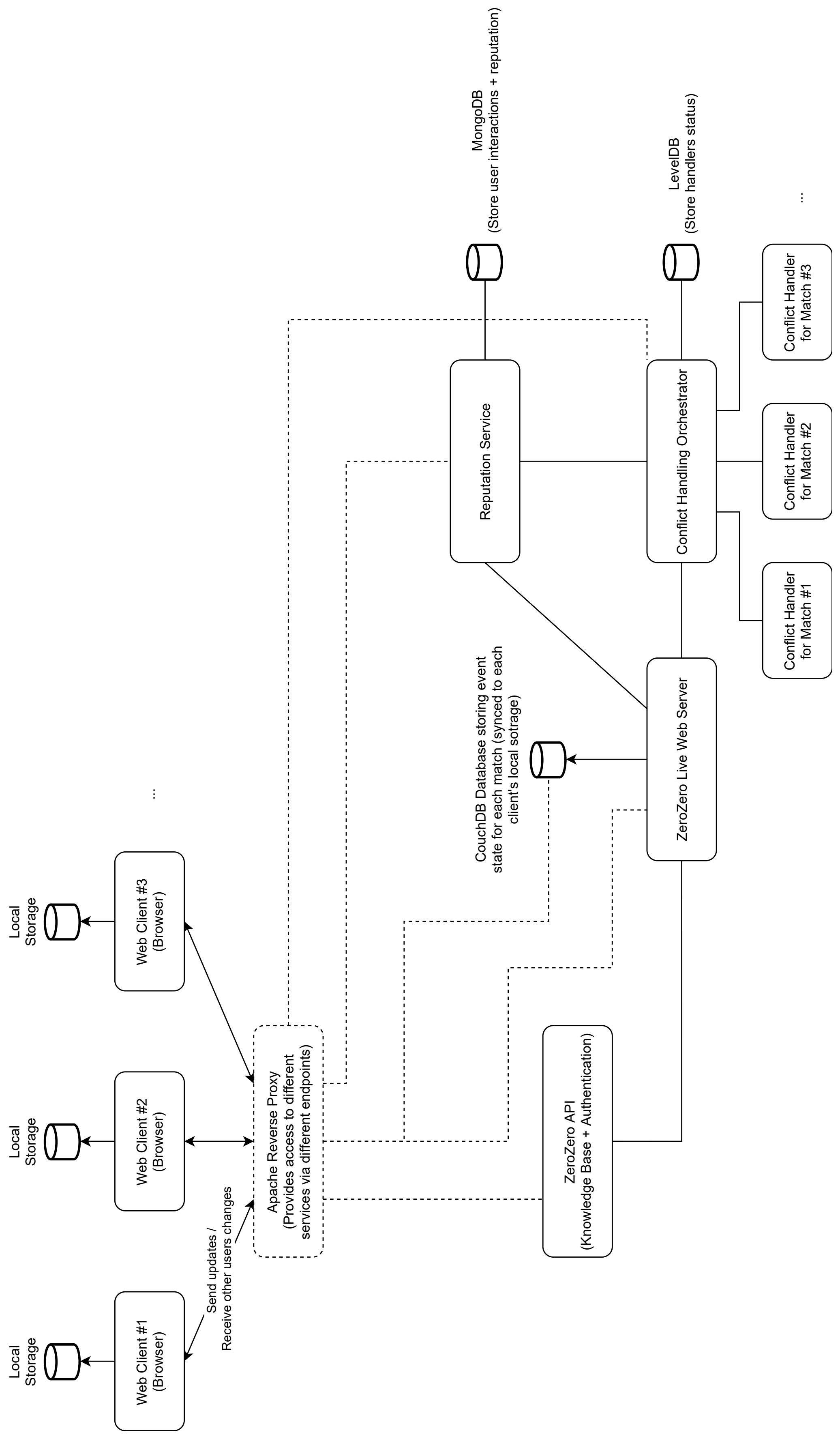
MongoDB
(Store user interactions + reputation)
LevelDB
(Store handlers status)
...
Reputation Service
Conflict Handling Orchestrator
Conflict Handler for Match #3
Conflict Handler for Match #2
Conflict Handler for Match #1
CouchDB Database storing event state for each match (synced to each client's local sotrage)
ZeroZero Live Web Server
...
Local Storage
Web Client #3 (Browser)
Local Storage
Web Client #2 (Browser)
Local Storage
Web Client #1 (Browser)
Send updates / Receive other users changes
Apache Reverse Proxy (Provides access to different services via different endpoints)
ZeroZero API (Knowledge Base + Authentication)

Figure A.1: Services Architecture of the Application

References

[1] Building real-time collaboration applications: OT vs CRDT. https://www.tiny.cloud/blog/real-time-collaboration-ot-vs-crdt/.

[2] Neil Fraser: Writing: Differential Synchronization. https://neil.fraser.name/writing/sync/.

[3] Promise API JS MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[4] Orlando P. Afonso, Luciana C.de C. Salgado, and José Viterbo. User's understanding of reputation issues in a community based mobile app. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9742, pages 93–103. Springer Verlag, 2016.

[5] Felix Albertos-Marco, Victor M.R. Penichet, Jose A. Gallud, and Marco Winckler. A model-based approach for describing offline navigation of web applications. *Journal of Web Engineering*, 16(1-2):1–38, mar 2017.

[6] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval Tree Clocks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5401 LNCS, pages 259–274. Springer, 2008.

[7] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9466:62–76, oct 2014.

[8] Reid Andersen, Christian Borgs, Jennifer Chayes, Uriel Feige, Abraham Flaxman, Adam Kalai, Vahab Mirrokni, and Moshe Tennenholtz. Trust-based recommendation systems: An axiomatic approach. In *Proceeding of the 17th International Conference on World Wide Web 2008, WWW'08*, pages 199–208, 2008.

[9] Yannis Bakos and Chrysanthos Dellarocas. Cooperation Without Enforcement? A comparative analysis of litigation and online reputation as quality assurance mechanisms. Technical report, 2003.

[10] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making Operation-Based CRDTs Operation-Based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Carlos Baquero and Francisco Moura. Using structural characteristics for autonomous operation. *Operating Systems Review (ACM)*, 33(4):90–96, 1999.

[12] A Barth. HTTP State Management Mechanism. RFC 6265, RFC Editor, 2011.

[13] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001.

[14] M Belshe, R Peon, and M Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). Technical Report 7540, RFC Editor, may 2015.

[15] S. U.N. Chengzheng, S. U.N. David, N. G. Agustina, C. A.I. Weiwei, and C. H.O. Bryden. Real differences between OT and CRDT under a general transformation framework for consistency maintenance in co-editors. *Proceedings of the ACM on Human-Computer Interaction*, 4(GROUP):1–26, jan 2020.

[16] Sandy Citro, Jim McGovern, and Caspar Ryan. Conflict management for real-time collaborative editing in mobile replicated architectures. In *Conferences in Research and Practice in Information Technology Series*, volume 62, pages 115–124, 2007.

[17] Elizabeth M. Daly. Harnessing wisdom of the crowds dynamics for time-dependent reputation and ranking. In *Proceedings of the 2009 International Conference on Advances in Social Network Analysis and Mining, ASONAM 2009*, pages 267–272, 2009.

[18] Chrysanthos Dellarocas. How often should reputation mechanisms update a trader's reputation profile? *Information Systems Research*, 17(3):271–285, 2006.

[19] Chrysanthos Dellarocas and R. Smith. Reputation Mechanisms. 2005.

[20] Anna Derezińska and Krzysztof Kwaśnik. Evaluation and Improvement of Web Application Quality – A Case Study. In *Advances in Intelligent Systems and Computing*, volume 1173 AISC, pages 187–196. Springer, 2020.

[21] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume Part F1301, pages 399–407, New York, New York, USA, jun 1989. Association for Computing Machinery.

[22] I Fette and A Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, 2011.

[23] C Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. 1988.

[24] Roy T Fielding, James Gettys, Jeffrey C Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, 1999.

[25] Neil Fraser. Differential Synchronization. In *DocEng'09, Proceedings of the 2009 ACM Symposium on Document Engineering*, pages 13–20, 2 Penn Plaza, Suite 701, New York, New York 10121-0701, 2009.

[26] Irene Greif, Robert Seliger, and William E Weihl. Atomic Data Abstractions in a Distributed Collaborative Editing System. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 160–172, New York, NY, USA, 1986. Association for Computing Machinery.

[27] Taher H. Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):784–796, jul 2003.

[28] Mayssa Jemel and Ahmed Serhrouchni. Content protection and secure synchronization of HTML5 local storage data. In *2014 IEEE 11th Consumer Communications and Networking Conference, CCNC 2014*, pages 539–540. IEEE Computer Society, 2014.

[29] Yung Wei Kao, Chiafeng Lin, Kuei An Yang, and Shyan Ming Yuan. A web-based, offline-able, and personalized runtime environment for executing applications on mobile devices. *Computer Standards and Interfaces*, 34(1):212–224, jan 2012.

[30] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. Local-first software: You Own Your Data, in spite of the Cloud. In *Onward! 2019 - Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2019*, pages 154–178. Association for Computing Machinery, Inc, oct 2019.

[31] Steve Krug. *Don't make me think, revisited: A common sense approach to web usability*. New Riders Publishing, Upper Saddle River, NJ, 3 edition, 2013.

[32] Hyunsoo Kwon, Hyunjae Nam, Sangtae Lee, Changhee Hahn, and Junbeom Hur. (In-)Security of Cookies in HTTPS: Cookie Theft by Removing Cookie Flags. *IEEE Transactions on Information Forensics and Security*, 15:1204–1215, 2020.

[33] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[34] Tobias Landes. Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications. pages 31–37, 2006.

[35] P Leach. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, RFC Editor, 2005.

[36] Mihai Leţia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. In *Operating Systems Review (ACM)*, volume 44, pages 29–34, apr 2010.

[37] Barbara Liskov and Rivka Ladin. Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 29–39, New York, NY, USA, 1986. Association for Computing Machinery.

[38] Wen Tao Liu. Research on offline storage of web page. In *Applied Mechanics and Materials*, volume 518, pages 305–309, 2014.

[39] Hector M. Lugo-Cordero and Ratan K. Guha. A secured distribution of server loads to clients. In *Proceedings - IEEE Military Communications Conference MILCOM*, volume 2015-Decem, pages 372–377. Institute of Electrical and Electronics Engineers Inc., dec 2015.

[40] Félix Albertos Marco. Supporting offline interaction with web sites resilient to interruptions applied to E-learning environments. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8295 LNCS, pages 310–314, 2013.

[41] Félix Albertos Marco, José Gallud, Victor M.R. Penichet, and Marco Winckler. A model-based approach for supporting offline interaction with web sites resilient to interruptions. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8295 LNCS, pages 156–171, 2013.

[42] Félix Albertos Marco, Víctor M.R. Penichet, and José A. Gallud. User interaction with offline web applications: A case study. In *ACM International Conference Proceeding Series*, volume 07-09-Sept. Association for Computing Machinery, sep 2015.

[43] Gloria Mark, Daniela Gudith, and Ulrich Klocke. The cost of interrupted work: More speed and stress. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 107–110, 2008.

[44] Almaz Melnikov, Jooyoung Lee, Victor Rivera, Manuel Mazzara, and Luca Longo. Towards Dynamic Interaction-Based Reputation Models. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, volume 2018-May, pages 422–428. IEEE, may 2018.

[45] Ali Mesbah. Ajaxifying classic web applications. In *Proceedings - International Conference on Software Engineering*, pages 81–82, 2007.

[46] Ali Mesbah and Arie Van Deursen. Migrating multi-page web applications to single-page AJAX interfaces. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 181–190, 2007.

[47] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *UIST (User Interface Software and Technology): Proceedings of the ACM Symposium*, pages 111–120, New York, New York, USA, 1995. ACM.

[48] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, pages 259–268, New York, New York, USA, 2006. ACM Press.

[49] João Pedro Sousa Castro. *Mobile application for online monitoring and collaboration of a sporting event.* PhD thesis, Faculdade de Engenharia da Universidade do Porto, sep 2020.

[50] Jon Postel. Transmission Control Protocol. STD 7, RFC Editor, 1981.

[51] Paul Resnick, Richard Zeckhauser, Eric Friedman, and Ko Kuwabara. Reputation systems. *Communications of the ACM*, 43(12):45–48, dec 2000.

[52] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6976 LNCS, pages 386–400. Springer, Berlin, Heidelberg, oct 2011.

[53] Leon Shklar and Rich Rosen. *Web Application Architecture: Principles, Protocols and Practices.* Wiley, Chichester, UK, 2 edition, 2009.

[54] Spero and S. E. Analysis of HTTP Performance problems. *http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html*, 1994.

[55] Chengzheng Sun, David Chen, Xiaohua Jia, Yanchun Zhang, and Yun Yang. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, mar 1998.

[56] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work - CSCW '98*, pages 59–68, New York, New York, USA, 1998. Association for Computing Machinery (ACM).

[57] David Sun, Chengzheng Sun, Agustina Ng, and Weiwei Cai. Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW1):1–30, may 2020.

[58] David Sun, Chengzheng Sun, Steven Xia, and Haifeng Shen. Creative conflict resolution in realtime collaborative editing systems. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, pages 1411–1420, New York, New York, USA, 2012. ACM Press.

[59] Thomas Tullis and William Albert. *Measuring the User Experience, Second Edition: Collecting, Analyzing, and Presenting Usability Metrics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2013.

[60] Sai Lahari Velagapudi and Himanshu Gupta. Privacy, Security of Cookies in HTTP Transmission. In *2019 4th International Conference on Information Systems and Computer Networks, ISCON 2019*, pages 22–25. Institute of Electrical and Electronics Engineers Inc., nov 2019.

[61] Xueyi Wang, Jiajun Bu, and Chun Chen. Research on Conflict Resolution and Operation Consistency in Real-Time Collaborative Graphic Designing System. In *Proceedings of the International Conference on Computer Supported Cooperative Work in Design*, volume 7, pages 145–150, 2002.

[62] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Proceedings - International Conference on Distributed Computing Systems*, pages 404–412, 2009.

[63] Liyin Xue, Kang Zhang, and Chengzheng Sun. An integrated post-locking, multi-versioning, and transformation scheme for consistency maintenance in real-time group editors. In *Proceedings - 5th International Symposium on Autonomous Decentralized Systems, ISADS 2001*, pages 57–64. Institute of Electrical and Electronics Engineers Inc., 2001.

[64] Y. Yang. Supporting online Web-based teamwork in offline mobile mode too. In *Proceedings of the 1st International Conference on Web Information Systems Engineering, WISE 2000*, volume 1, pages 486–490. Institute of Electrical and Electronics Engineers Inc., 2000.

[65] Ekaterina Yashkina, Arseny Pinigin, Joo Young Lee, Manuel Mazzara, Akinlolu Solomon Adekotujo, Adam Zubair, and Luca Longo. Expressing Trust with Temporal Frequency of User Interaction in Online Communities. In *Advances in Intelligent Systems and Computing*, volume 926, pages 1133–1146. Springer Verlag, 2020.

[66] Lian Yu, Wenping Xiao, Changyan Chi, Lin Ma, and Hui Su. Test case generation for collaborative real-time editing tools. In *Proceedings - International Computer Software and Applications Conference*, volume 1, pages 509–516, 2007.

[67] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Middleware 2015 - Proceedings of the 16th Annual Middleware Conference*, pages 75–87. Association for Computing Machinery, Inc, nov 2015.