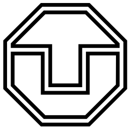

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Rechnernetze

AUTOMATISIERTE STEUERUNG EINES WEB-SERVICES WÄHREND DER PRÄSENTATION EINES PDF-DOKUMENTS

Sven Fischer

Born on: 26th January 1985 in Dresden

DIPLOMA THESIS

to achieve the academic degree

DIPLOM MEDIENINFORMATIK

First referee

Tenshi Hara

Second referee

Iris Braun

Submitted on: 27th October 2017

STATEMENT OF AUTHORSHIP

I hereby certify that I have authored this Diploma Thesis entitled *Automatisierte Steuerung eines Web-Services während der Präsentation eines PDF-Dokuments* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 27th October 2017

Sven Fischer

CONTENTS

Statement of authorship 3

1 Introduction 7
- 1.1 Motivation . . . 7
- 1.2 Contributions . . . 8

2 Related Work 11
- 2.1 The Portable Document Format (PDF) . . . 11
- 2.2 State of the ARS . . . 14
- 2.3 Auditorium Mobile Classroom Service (AMCS) . . . 15
- 2.4 Related Tools in the AMCS Ecosystem . . . 15

3 Requirement Analysis 17
- 3.1 User Types and User Environment . . . 17
- 3.2 Interfaces (User, Hardware, Software) . . . 17
- 3.3 Current AMCS interface and structure . . . 18
- 3.4 Use Cases . . . 18
- 3.5 System Features/Functional Requirements . . . 20
- 3.6 Nonfunctional Requirements (Security, Performance, User Documentation) . . . 20

4 Concept 23
- 4.1 Rendering the PDF . . . 23
- 4.2 Communication With the Webservice . . . 23
- 4.3 Proposed Workflow . . . 24
  - 4.3.1 Setup . . . 24
  - 4.3.2 Presentation . . . 25
- 4.4 Components of the Application . . . 25
- 4.5 Interface Mockups . . . 26

5 Implementation 29
- 5.1 Cross-platform PDF viewer software . . . 29
- 5.2 Integration with AMCS Angular2 Frontend . . . 29
  - 5.2.1 Displaying PDF Files . . . 30
  - 5.2.2 File Selection and Upload . . . 30
  - 5.2.3 Going Fullscreen . . . 32
  - 5.2.4 Capturing Input Events . . . 33

5.2.5 Communication between browser windows (?presentation and application?) . . . 33
5.2.6 Angular2 Change Detection Mechanisms . . . 34
5.3 Communication with the AMCS Backend . . . 35

6 Evaluation **39**
6.1 Qualitative Evaluation . . . 39
6.1.1 Methology Discussion . . . 39
6.1.2 Setup . . . 40
6.1.3 Results Task Load Index . . . 41
6.1.4 Results of Thinking Aloud . . . 42
6.1.5 Interpretation of the Results . . . 44

7 Conclusion and Future Work **47**

List of Figures **49**

TLX Evaluation Sheet **51**

Bibliography **53**

1 INTRODUCTION

1.1 MOTIVATION

During the course of my study, I witnessed many changes in the world, in the university, in the lecture hall. The changes were technical of nature, as well as social. In the first couple of semesters, professors used a lot of different media. The math professor would go blackboard-only, he got a nice wipe with his name stitched in after carrying us through calculus and probability theory. Some professors would use overhead projectors to show analog slides printed beforehand, or would make them up on the fly by writing on the slides right as they were projected onto the wall. One professor would spot a mistake in his digital presentation, switch to his favorite commandline editor, fix the error and recompile the corrected presentation from scratch. Another one would show her prepared digital presentation, but always ask for a small movable cabinet containing a desktop computer to be rolled in prior to the lecture to be able to show her presentation.

Over the years, digital presentations became more and more prevalent, to the point, that even teaching staff holding the weekly group exercises prepared digital slides to show hints or solutions. Professors got more and more comfortable and excited using digital media and started to incorporate moving pictures and various other kinds of digital media into their presentations.

Thankfully, after a few semesters, many of the digital presentations became available to help with exam preparation, either by being published by the professors themselves, or by the heroic efforts of some students writing them down and recreating a digital version. This had the upside of giving everyone the chance to use them when learning for exams, even in case they were not able to write down notes as meticulously as required, or in the rare case of not paying the lecture a visit. Many students found it easier to follow the lectures, since they did not have to jot down every word in case it might turn out to be important for the exam. On the other hand, not having to write down anything easily leads to students writing down nothing, or not visiting the lecture at all; which, for some students, removes the important channels of retaining information via (hand-) writing notes or listening to the lecture in the first place.

On the other side of the proscenium, much the same has been happening with respect to digitalization. In the beginning, there were one or two students with their laptops plugged into docking-stations, using them to write notes or gather more information about the lecture topic. Shortly after, the laptops had been replaced by notebooks, partially with touch input and a digital pen. But before notebooks became widespread, the smartphone entered the stage and soon everyone, almost, had a notebook, a smartphone, or both, available during the lecture. While notebooks were mostly brought intentionally to increase the effectiveness of the lecture, by providing easy information lookup, the ability to skip back and forth through the provided slides and making it easier to take notes, the smartphones were used during

the lecture because everyone had brought one with them anyway. But smartphones would not lend themselves as well as notebooks to increase the effectiveness of visiting the lecture: while looking up information is feasible, notetaking and poring over slides is cumbersome. Notebooks and smartphones were, of course, also used by many to distract themselves from the lecture by playing games or visiting social media.

Over the years, the situation in lectures/readings with big audiences evolved from the professor using analog slides (or no slides at all), the students having to try to discern the important parts and to write them down, neither the students nor the professor using digital devices, resulting in the risk of the student being overwhelmed and not understanding important parts of the lecture – to professors using digital devices to present slides they would share with the students, students not having to write anything down and using the ubiquitous smartphones, resulting in the students possibly not following the lecture as attentive as they could, easily zoning out or being distracted by their smartphone.

The initial increase in attention by reducing the burden on the students to follow the lecture by introducing digital devices, has turned into the opposite: students find it easy to divert their attention from the lecture.

One way to increase attendance and attention is to provoke interaction among students or between students and the lecturer. The easiest and easily most futile way was for the lecturer to ask if there were any questions left. Surprisingly, in most cases there were none. Traditionally, a better way for lecturers to check if students are still following, was to get something wrong, intentionally or unintentionally, provoking a correction. But that would only work for the few students who were sure they know the answer and were following the lecture to the point, the others would not even twitch. But, leveraging the mentioned changes in behavior and technology, other forms of increasing interactions are possible, for example: using the student's and the lecturer's affinity to technology, to create a digital channel between the student and the lecturer [16]. Different approaches are possible, and can be subsumed under the concept of "Audience Response Systems" (ARS). These systems are used to request and receive feedback from the audience/the students, react upon this information, possibly in real-time, save and analyze the gathered data. This provides the lecturer with a flexible way to collect exactly the feedback he wants, while increasing student participation and attentiveness. ARS' do not have to use the student's or the lecturer's devices, "Clicker" systems with devices only handed out for this single purpose can be used too, but employing the ubiquitous smartphones decreases the cost and time of deployment and capitalizes on the familiarity of the students with their devices. Not only can these systems be used to gather feedback about the student's understanding of the lectuere, but also to provide students with questions they may repeat at home or additional lecture material to help repeat the lecture content. Further extensions include the provisioning of different feedback material to be sent to each student, depending on their individual answers/knowledge/learning goals.

While this transforms the student's devices from a possible distraction to a valuable learning tool, the same can be done for the lecturer's digital presentation. Usually, the ARS and the lecturer's presentation are separate; additional effort has to be put in, to use both, ARS and a presentation. Going one step further, seamlessly integrating the presentation slides into the ARS would allow the lecturer to present his slides as usual, while being able to reap the benefits of using an ARS without the hassle of having to control both. To achieve that, slides created using the Portable Document Format, often used for presentations, will be integrated into the ARS "Auditorium Mobile Classroom Service" (AMCS) developed at TU Dresden, using nothing more than a current web browser.

1.2 CONTRIBUTIONS

The contributions of this thesis are as follows:

- displaying PDFs in a platform-independend way, using only a current web browser
- integration of PDF display and navigation into the ARS AMCS
- steering AMCS using the current state of the presentation
- reacting to AMCS state received through websocket connection

2 RELATED WORK

The topic of this thesis connects a number of different fields. At first, a short introduction of the Portabe Document Format is given, including history and different options for creation and display. Then two different Audience Response Systems are showcased. The Auditorium Mobile Classroom Service is introduced and projects closely related to this thesis are presented.

2.1 THE PORTABLE DOCUMENT FORMAT (PDF)

The Portable Document Format is widely popular today, its use extends from authors easily publishing books online, to designers sending the small brochures or large bills they created to the printing companies, through banks collecting data via PDF's form elements and storing it afterwards, to people using it to present information on a projector. Although the use cases probably extend beyond what inventor John Warnock imagined, they reflect the core goal PDF was designed to achieve: reliable transfer and reproduction of documents independently of software or hardware [2, 20].

PDF was designed to be an improved, electronic, lossless replacement for the prevalent FAX machines of the early 90s [20]. At this time, a proven solution for this problem was PostScript, an interpreted, Turing complete, stackbased language developed at Adobe Systems [1]. The PostScript ecosystem already had all the tools necessary to achieve the goal of the Camelot project, but the implementations of this complex language were too slow for many of the machines of this day, and improving the efficiency proved difficult. Hence, the rendering of a PostScript document was split into multiple parts that could run independently on smaller machines with acceptable performance. This has been achieved by using a feature of the PostScript language called "rebinding" which, from the original PostScript document, can produce a derived, simpler document, which only uses a reduced set of less complex commands which require less computing power and can be split more easily. An example includes the "unrolling" of certain loops. The author defines the Interchange PostScript (IPS) format, which is a valid PostScript and EPS file, so that the existing infrastructure can be leveraged. To generate IPS, a reduced PostScript interpreted was implemented, its only function is to rebind PS files to IPS files. Since this IPS binder also includes reconstituted fonts for only those characters that are used in the file, the resulting IPS file is fully self contained. The resulting system consists of using the IPS binder to convert any PostScript document to IPS and the simple and fast IPS interpreters to browse and print IPS files, which can be used even on low-end machines. This would enable new ways of storing, handling and retrieving documents which can be easily searched, printed at and transmitted to any location.

The Camelot project was later renamed to "Carousel" and the PDF standard version 1.0 was published [6]. The whitepaper layed out some fundamental principles of the format: distinct

pages, PDF describes how to paint on pages, painting is opaque and may be clipped to another shape. In contrast to PostScript, "PDF is not a programming language, it does not contain procedures, variables and control constructs," its fixed set of high level operators can be directly implemented in machine code. Each page of the PDF file may be randomly accessed, without needing to interpret the whole document up to this page, like it is the case with PostSctipt. The pages may contain images and paths to describe arbitrary shapes, font descriptors are embedded, that help the receiver of the document to get an output reasonably close to the original in case he is missing the font, without adding the whole font to the document. Initially, the software distributed by Adobe was only available commercially. Together with the limited features of PDF 1.0 (like missing hyperlinks) and the large size of the format (in comparison with plain text), this lead to a slow adoption of the format. This began to pick up after Adobe released the free Reader 2.0 together with PDF 2.0 in 1994, which included hyperlinks, introduced passwords and encryption together with a binary (instead of ASCII only) file representation.

There have been many revisons of the PDF file format, which extended existing and introduced many more useful features, e.g. interactive page elements, form data, reproduction and embedding of many kinds of external content, Javascript.

The specifications were made available free of charge by Adobe, but PDF remained a proprietary format until the release of PDF 1.7 as an open standard published by the International Organization for Standardization as ISO 32000-1 in 2008 [8]. Some extensions required by this standard, like XML Forms Architecture (XFA) are still proprietary.

After releasing this open standard, further extensions have been developed, e.g. embedding of Flash/SWF content and extensions to XFA, some of which were again released as the open ISO standard 32000-2, PDF 2.0 respectively [9].

The ISO standardized a number of subsets of the PDF standard containing features suited for certain use cases while leaving out others, e.g. PDF/X, PDF/UA, PDF/A.Of special interest is PDF/A which was at first based on PDF 1.4, later on ISO 32000-1/PDF 1.7. PDF/A is concerned with archiving and long-term preservation of documents, the standard focuses even more on the exactly reproducibility (even years in the future) than the regular PDF standard. Therefore, a PDF/A document has to be 100% self-contained, all information (e.g. fonts, images, color information) has to be contained in the file, the standard forbids features that depend on external resources, but allows external annotations like hyperlinks. More restrictions include: no audio or video content, no JavaScript, no encryption. Often, legal issues come into play as well: LZW is forbidden due to intellectual property constraints, a font can only be embedded, if it is legally embeddable, from version PDF/A-2, digital signatures are allowed, but only if they conform to the PAdES (PDF Advanced Electronic Signatures) standard.

To create a PDF document, there are a varietey of possibilities. Many operating systems have built-in capabilities to save a printed document as PDF (MacOS, some Linux distributions), while third-party software can be used to add this functionality to other operating systems (Windows). Popular word-processing softwares can directly export documents as PDF (like LibreOffice[1], MS Office[2], WordPerfect[3]). Documents produced using the LaTeX/TeX-ecosystem[4] can be output directly as PDF file using pdfTeX/pdfLaTeX[5]. Additionally there are a number of open- and closed-source products that can convert files to and from PDF. The most obvious choice to create a PDF file are Adobe's own products: Acrobat, InDesign, FrameMaker, Illustrator, Photoshop[6].

For displaying PDF documents, there are even more ways. Besides well-known stand-alone

[1] https://www.libreoffice.org – Last accessed on 29 October 2017
[2] https://products.office.com/en-us/products – Last accessed on 29 October 2017
[3] https://www.wordperfect.com – Last accessed on 29 October 2017
[4] https://latex-project.org – Last accessed on 29 October 2017
[5] https://pdftex.org – Last accessed on 29 October 2017
[6] https://www.adobe.com/products/catalog.html – Last accessed on 29 October 2017

solutions like Adobe Reader[7], Evince[8], Foxit[9], Xpdf[10] or the viewers integrated into the operating-system, like MacOS X's Preview, PDF documents can be viewed in the browser in a number of different ways. Adobe distributes their Acrobat Reader as a plugin for most common browsers, which can be used to directly view PDF documents from the internet (or from the hard drive) in the browser without opening an external program. Google's Chrome[11] which is a fork of Google's open source Chromim Engine[12] comes with an extension to view PDF files preinstalled, it is based on the PDFium[13] rendering engine which is implemented in C++. For most PDFs this works as fast and as seamlessly as the Adobe reader plugin, but it is only available in the Chromium-based browsers.

Another way to view PDF documents directly in the browser is to render them on the server and serve them as images or as HTML documents, which does not require any plugins or special browser capabilites. This can be achieved by using a software that converts PDF to images (like ImageMagick[14]) and serving them via HTTP or by using an existing service. Google Docs[15] can be used to view PDF files in the browser, it renders them to PNG images on the server and creates a HTML page containing those images and possibly the text of the document. An example url looks like this: `http://docs.google.com/viewer?url=https://www.cs.drexel.edu/~david/Classes/Papers/p91-winkenbach.pdf`

The third way to view PDF documents in the browser is using client-side rendering: the PDF file is not sent to the server and rendered there, but rendered directly in the browser, for example via JavaScipt. This approach requires the browser to provide some JavaScipt features to enable the processing of the PDF file, but it does not depend on any browser-sepecific features or external plugins. One solution is PDF.js[16], which is also used inside the browser Mozilla Firefox to render PDF documents, but can be used as a stand-alone library to render PDF documents on any website. PDF.js is an open source project that uses features of modern browsers like the canvas element, typed arrays, Web Workers (a JavaScript script running in the background to do work asynchronously possibly using another thread) and XMLHttpRequest (to fetch resources without a page reload) respectively FileReader API[17] to request and render PDF files. It is split into the components used for loading and rendering the PDF and the components used to present an user interface for navigation.

PDFObject[18] provides a simple wrapper for the embedding of PDF documents. It can detect if the browser provides support for embedding PDF files in a webpage, either through an installed plugin or built-in of the browser, and has a fall-back to use PDF.js in case the browser has no support. The developer can provide the size the PDF shopuld be rendered in, the page to be redered initially and some other PDF open parameters which the browser may or may not support. It does not enable the developer to hook into the rendering events to get or set the displayed page.

Flexpaper[19] is a hybrid, commercial solution which claims to use HTML4, HTML5 or Flash depending on the availability on the client and the publishing author's preferences. It requires special server-side support (preprocessing, rendering on server via PHP or ASP.NET using pdftk and mupdf-tools) and may require client side support (Flash), but can serve PDF content in

[7] `https://acrobat.adobe.com/us/en/acrobat/pdf-reader.html` – Last accessed on 29 October 2017
[8] `http://wiki.gnome.org/Apps/Evince` – Last accessed on 29 October 2017
[9] `https://www.foxitsoftware.com/` – Last accessed on 29 October 2017
[10] `http://www.xpdfreader.com/` – Last accessed on 29 October 2017
[11] `https://www.google.com/chrome/index.html` – Last accessed on 29 October 2017
[12] `https://chromium.org/` – Last accessed on 29 October 2017
[13] `https://bugs.chromium.org/p/pdfium/` – Last accessed on 29 October 2017
[14] `http://www.imagemagick.org/` – Last accessed on 29 October 2017
[15] `https://docs.google.com/` – Last accessed on 29 October 2017
[16] `https://mozilla.github.io/pdf.js/` – Last accessed on 29 October 2017
[17] `https://developer.mozilla.org/en-US/docs/Web/API/FileReader` – Last accessed on 29 October 2017
[18] `http://pdfobject.com/` – Last accessed on 29 October 2017
[19] `https://flowpaper.com/` – Last accessed on 29 October 2017

a browser to a very large variety of end-users. It provides an extensive API in the browser allowing to create user interactions that go beyond what is possible with PDF.

A similar project is smartlook[20]. It preprocesses PDF files on the server and "replaces all the documents on your website with beautiful online publications." The resulting presentations can be searched, zoomed and viewed in fullscreen mode, the original files can be downloaded. Smartlook seems to host all published contet themselves and uses HTML canvas to display the preprocessed pages transferred to the browser in some binary format. The service exposes a REST interface to manage the published documents and a JavaScript API to control the embedded viewer.

2.2 STATE OF THE ARS

A very lightweigth and browser-only ARS is Tweedback [5, 19]. It supports three main features: posing multiple-choice questions to students with a detailed report for the lecturer, providing the students with six different "problem buttons" and a chatwall. The problem buttons, in other applications they are called "panic buttons", can be used by the students to provide the lecturer with feedback he can immediately act on. There are six predefined categories: "a) too fast, b) too slow, c) too quiet, d) an example is needed, e) the last slide has to repeat in greater detail or f) that the auditor has a diffuse feeling" [5]. The chatwall can be used by students to ask questions that are visible to other students and the lecturer and can be upvoted, discussed and answered by students as well as the lecturer. The authors conducted an evaluation of the system on a revison course over six days for medical students' exam preparation [18]. It shows the feasibilty of modern ARS for large audiences of more than 100 students, the reduced cost and time for doing multiple-choice questionaires, the ease of use for the lecturer when polling and collecting the results and an easy way for students to review the posed questions to check their progress. The initial setup takes some amount of time, the students have to be introduced to the system and have to have their devices ready and charged to participate, but this cost was dwarved by the speed of polling when using the system for multiple questions in multiple sessions.

The ARS eduVote uses a different technical approach: students and can use the browser or an app, the lecturer uses and controls the ARS through software or plugins to presentation tools [17, 7]. To allow a better integration of the polling of students and the presentation of the results, edoVote equips the lecturer with a Microsoft PowerPoint plugin for the Windows platform. The commercial product allows the lecturer to create and edit multiple-choice questions directly in his PowerPoint slides. During the presentation, the lecturer shows the created questions and, after gathering the results, may show the results directly on another slide without leaving the presentation. Results of the polls are directly saved into the presentation file. During the polling, the lecturer may use a browser to view the intermediate results before the polling is concluded. The advantage of this approach are the better workflow for the lecturer. With the current system, the question and answer texts only exist in the lecturers presentation. They do not get sent to the server and it is not possible for the students to independently view the questions themselves, they only get to view the choices for an answer in an anonymous fashion ("A/B/C/D"). It follows, that the questions have to be shown by the lecturer on some display or projection, otherwise the students do not know what each possible answer is supposed to mean. Also, the questions cannot be reviewed later by the students. The evaluation was performed by asking lecturers that employed the system. It showed that the most important advantages of the system are "increased participation of all students"(65% of lecturers), "better activation of introverted students"(60%) and the "possibility [for students] to test themselves"(63%).

[20] `http://www.ismartlook.com/` – Last accessed on 29 October 2017

2.3 AUDITORIUM MOBILE CLASSROOM SERVICE (AMCS)

In 2012, students at TU Dresden developed the online community "auditorium"[21] [3]. During the semester, students at the university will have lots of questions about the lectures and the taught material. Not all questions can be answered by rereading the provided material or referenced literature, so the students will seek help either from lecturers or from their peers. This either forces lecturers to answer similar questions regularly, or leaves the students without an answer in case their peers cannot help or do not want to. The platform can be used by students and faculty to ask, share and answer various questions with the goal to help the students and lecturers alike by incrementally building a knowledge base. Akin to Question&Answer systems like the Stackoverflow community[22], the system provides the possibility to pose questions and collect answers about lectures, organization of the course of studies or other related topics. To organize themselves, the participants can create three different kinds of groups: related to a specific course, own study groups, groups on a certain topic which does not have to be related to a course. The interaction is faciliated through the use of tags, fulltext search and filtering, subscription to groups, presentation of recent posts, up- and downvoting, moderation through users themselves, gamification concepts. Furthermore, users can share different kinds of content like announcements, presentation slides, tutorials or videos. At the time over 1500 students and faculty have used the tool to collaborate, create and answer topics or advertize own projects and courses. Activity has been increased through the gamification elements and the presentation of recent discussions to all users, including users which have not yet shown interest in the topic of this discussion.

While increasing the qualitiy of questions and answers, requiring students to register with their full names using an official university email address proved to be discouraging to students with low self-esteem, who might deem their questions to be dumb or too easy. Additionally, the knowledge-base approach is useful in itself, but increasing student's attention and undestanding during a lecture can provide even more value for students. To achieve these goals, the ARS AMCS is an extensible, cross-platform extension of the existing "auditorium" tool at TU Dresden[12, 13].

It was developed to better reach students depending on their prior knowledge, individual learning goals and learning styles. This is achieved by providing six individual components: asking each student a select few questions about their individual interests and personal goals, providing learning questions to each student to enable them to evaluate and improve their knowledge and increase attention via cognitive promts, individual feedback depending on the student's answers to the questions, giving metacognitive prompts to better help the students achieve their goals, providing supplementary material like slide sets, enabling a "roleplay" where students get assigned a certain role which helps them view the material from a different angle or ask questions they would normally not have thought of. All of these possible interactions can be individually tuned using the student's answers to the leading questions about his knowledge/goals/motivation. The tool increased the interactivity between students and lecturer and enabled a better understanding and retention of knowledge.

The promising results have led to the further extension of AMCS, e.g. the possibility to give graphical answers or the guided suggestion of features for the lecturer to use [14, 4].

2.4 RELATED TOOLS IN THE AMCS ECOSYSTEM

AMCS is designed to be used in parallel to a presentation, for example, questions can be added that are shown when the presentation reaches a specific slide. There is no inherent connection between the lecturer's presentation slides which are shown via some presentation software,

[21] `http://auditorium.inf.tu-dresden.de` – Last accessed on 29 October 2017
[22] `https://stackoverflow.com/` – Last accessed on 29 October 2017

e.g. Microsoft PowerPoint, and the questions specified for some slide in the AMCS system. Thus, the AMCS system has to be informed which slide the lecturer's presentation is showing currently. In the original design, this can be achieved by hand using the AMCS website, but multiple projects have been developed to remedy this redundancy.

Antje Schubotz designed a system to enable the lecturer to use Microsoft PowerPoint on Microsoft Windows in conjunction with AMCS[15]. The system enables the lecturer to control the AMCS server state (the currently shown slide) using only his presentation software PowerPoint without the need to visit the AMCS website. The software is implemented as an Add-In for PowerPoint programmed using VSTO (Visual Studio Tools for Office) and can be used with Office 2007 and newer. In addition, it enables creating/updating/deleting the different question types the AMCS system offers. The presented Add-In worked as expected, events were propagated efficiently from AMCS to the presentation software and the other way around, but some tests showed a high latency of the AMCS server sparking doubt in the user about his correct usage of the prototype. While the implementation uses a different underlying technology (native code) and targets a different user group (PowerPoint users), the goal of seamlessly changing AMCS server state direclty from the user's presentation is the same as the goal of this work, so some considerations and design ideas can be built upon in this work.

A similar concept was proposed by Martin Johst around the same time in 2016[11]. The goal is the same as Schubotzs' but the work was also focused on the integration of a smartwatch to change the slide in the lecturer's (PowerPoint) presentation. In the course of the work it was found, that the communication between smartwatch and the presentation software is easily realized using existing communication channels relaying it over the central AMCS server, so no additional work was done to integrate the smartwatch. Johst examined different methods of controlling the presentation software PowerPoint on MacOS, for example using a PowerPoint Add-In, creating an application using AppleScript or using NodeJS modules. The NodeJS modules "NW.js" and "slidehow" were found best suited for the task. While the implementation worked as expected, the evaluation of the prototype revealed some sluggishness of the propagation of the new slide number; it took up to five seconds for a slide-change event to propagate from the AMCS webclient on a smartphone to the presentation software. One user unfamiliar with the AMCS design and its internal workings (question types, active/inactive lectures, etc.) had problems with parts of the user interface of the Add-In, resulting in a proposal of some improvements and additional help texts. As with Schubotz' work, the used technology is different from the technology used in the work at hand, but the goals are similar, so ideas and general system design will be considered and improved upon.

3 REQUIREMENT ANALYSIS

To come up with realistic requirements for the system to be developed, the first thing to do is to look at the potential users of the system and their enviromnent. After this, the current AMCS system is presented briefly, to gather interface requirements. From initial state and potential use cases, the functional and nonfunctional requirements are derived.

3.1 USER TYPES AND USER ENVIRONMENT

AMCS is aimed at lecturers and faculty that want to collect feedback about their lecture, the student's/listener's knowledge or other arbitrary polling results. Most have prepared a slide set to be shown during the lecture/talk. The system to be developed should support those that are using PDF files to present their slides and improve the experience. They are familiar with PDF presentation software, but want an integrated solution without the need for an external PDF program. AMCS already requires a current webbrowser and reasonably current hardware, so this requirement does not change. The targeted user has their PDF either available locally, or on some webserver accessible via HTTP.

In short the user:

- has a current web browser
- has a reasonably current operating system and hardware
- wants to present PDF documents while simultaneously controlling a web service
- knows how to present PDF documents on a PDF reader, but cannot or does not want to use one currently
- has their PDF locally available or on some webpage

3.2 INTERFACES (USER, HARDWARE, SOFTWARE)

The AMCS application runs in any current webbrowser, so the user interface will be just the browser with additional custom controls to control the presentation. To be able to render PDF files, the browser mus support Javascript and rendering to Canvas or SVG. While the presentation of the PDF file can be realized without any internet connection, controlling the webservice would not work. Thus, having an internet connection available is required. The AMCS backend webservice is controlled and responds using the HTTP protocol or websockets.

In short:

- the user interface is the browser with custom controls to control the presentation
- internet connection has to be available to control the web service
- the web service is controlled over HTTP protocol or web sockets
- the web service responds via HTTP or web sockets
- the browser must support Javascript and rendering Canvas or SVG

3.3 CURRENT AMCS INTERFACE AND STRUCTURE

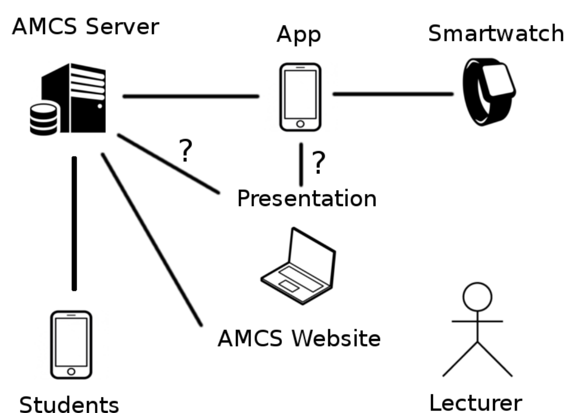

Figure 3.1: Structure of AMCS (and several other ARS)

The AMCS system has a typical client-server infrastructure: multiple clients with different capabilities (students, lecturers, admins) connect to a single server instance. For an overview, consult figure 3.1. This server keeps track of the questions, the answers, the lectures, and much more. The AMCS client can connect to the server and login, create lectures and questions, query lectures, query questions, send answers, view answers, etc. Of course, this depends on the client's credentials and associated capabilities. The initial connection happens through HTTPS, but a Websocket channel is etablished in parallel for the server to be able to push messages to the client. The server provides an extensive REST API, besides the Websocket, all communication happens using this REST API. The system to be developed has to use the same REST API and Websocket channels the current clients use. To clarify, in the figure, there are two direct connections between different clients. The App (on a smartphone) just works as a proxy for the Smartwatch. The connection between Presentation and smartphone App was one of the possible ways explored by [15, 11] to connect a presentation with AMCS, namely to have the presentation software only react to input by the App, without a connection to the AMCS server.

3.4 USE CASES

Since for all of the use cases and interactions the main actor is the lecturer, in the following the word user and lecturer are to be understood synonymously.

Before being able to do anything, the user has to connect a PDF to the lecture he wants to present. As seen in figure 3.2, there are three main ways of attaching a PDF to a lecture: using a local PDF on the users machine, attaching a remote PDF which is available through HTTP and

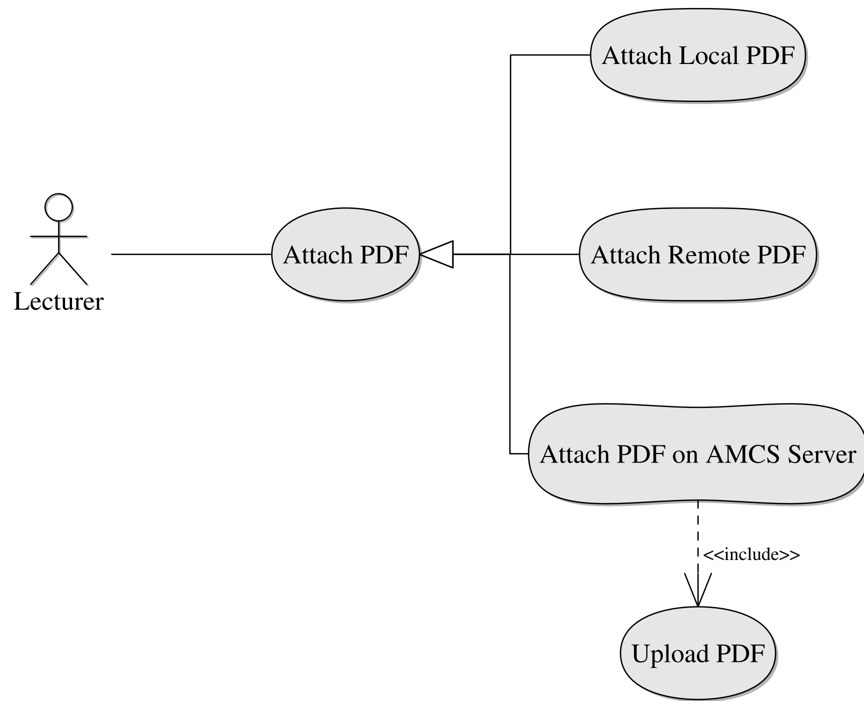

Figure 3.2: Attaching a PDF to a lecture (UML Use Case Diagram)

attaching a PDF that is uploaded to the AMCS server. Different users might prefer different options and all options will have slightly different workflows and user interface.

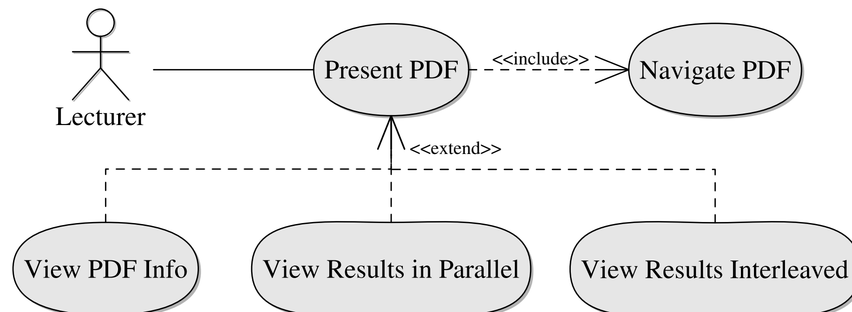

Figure 3.3: Presenting a PDF (UML Activity Diagram)

After attaching the PDF to a lecture, the user may present the PDF (see figure 3.3). Navigating the PDF is necessary to make any use of this functionality (unless there is only one slide). During the presentation of the PDF, the user may want to review information about the PDF he used, for example, the name or how many pages it has. Since the user likely has prepared polls for his audience (not shown here), he may want to check the results. To allow the lecturer to continue showing the presentation without showing the results on the same screen, the user has to be able to view the results in parallel to the presentation. Other users, may want to only use a single screen and show the results in between showing the presentation, hence there are two use cases: "View Results in Parallel" and "View Results Interleaved".

Usecases for using the PDF to support the user in editing the lecture can be seen in figure 3.4. Instead of editing a lecture through the existing AMCS application, which already supports creating, editing and deleting questions (compare figure), the user may choose to use the attached PDF for a better picture when and where he wants to add questions. Editing a lecture through a view with the attached PDF displayed is only only a special case of the current functionality.

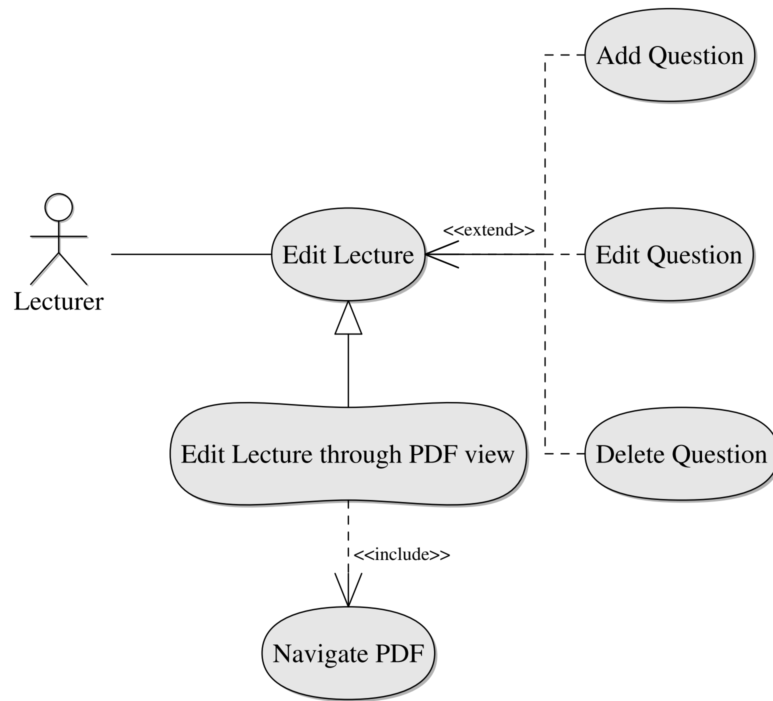

Figure 3.4: Editing a lecture through PDF view (UML Activity Diagram)

3.5 SYSTEM FEATURES/FUNCTIONAL REQUIREMENTS

From the current architecture of AMCS and the analysis of the user's knowledge and preferences, the requirements in table 3.1 can be derived.

3.6 NONFUNCTIONAL REQUIREMENTS (SECURITY, PERFORMANCE, USER DOCUMENTATION)

All communication over the wire has to be over HTTPS, this is already the case with the existing AMCS application. Reasonably large PDF files have to be displayed at interactive rates, including files up to 200 pages, multiple images per page or vector graphics. The user has to be given a short explanation about navigating the PDF and the implication the navigation has on the state of the AMCS backend.

	Description	Priority
A1	**Attach PDF to Lecture**	
A1.1	attach local PDF to lecture	high
A1.2	attach existing PDF accessible via HTTP	middle
A1.3	upload and attach local PDF	middle
A1.4	show information about attached PDF (name, pages)	high
A1.5	replace the attached PDF with another	middle
A2	**Edit Mode**	
A2.1	open and navigate attached PDF	low
A2.2	view questions on current slide	low
A2.3	add question on current slide	low
A2.4	edit questions on current slide	low
A2.5	delete questions on current slide	low
A3	**Presentation Mode**	
A3.1	open attached PDF in fullscreen mode	high
A3.2	navigate attached PDF in fullscreen mode	high
A3.3	leave fullscreen, view charts, resume	high
A3.4	open the presentation in second window	high
A3.5	state in main window in sync with second window	high
A3.6	resume a presentation after navigating AMCS	high
A3.7	send current presentation state to server	high
A3.8	receive presentation state, navigate accordingly	middle

Table 3.1: Requirements of the system to be developed

4 CONCEPT

4.1 RENDERING THE PDF

In chapter 2 (Related Work), three different ways of rendering PDF in a web browser have been introduced:

1. Rendering on the server, serving HTML and images
2. Rendering in the browser with a plugin
3. Rendering in the browser without plugin

Using server-side rendering of the PDFs the lecturer wants to show incurs an initial upload penalty and many smaller subsequent download penaties, which increase resource consumption and carry a delay for the user interaction. The initial upload can be circumvented by storing the PDF on the server. Caching might be used to prevent the delays when switching pages. There are some problems with this approach. Some users might want to keep their files locally and do not want to upload them to the server. Flipping to a specific page might be slow, even with caching, when the target page is not in the cache yet.

Rendering in the browser with a plugin prevents most of those flaws, but introduces an external dependency, which prevents the usage on all devices: the plugin might not be installed, the user might not have the rights to install it, or the user does not want to use an external plugin.

This leaves the rendering in the browser without a plugin using browser-only technology. It prevents the shortcomings of rendering the PDF on the server without introducing a plugin the user has to have installed. This still leaves open the option to render a PDF from either a remote server, the AMCS server, or some remote host.

The PDF can be rendered in windowed mode or in fullscreen mode and the user can select which screen the PDF is rendered to, if they have a setup with multiple screens. This allows the user to use different modes of presentation: everything on the same screen with switching between presentation and AMCS, or using multiple displays and having AMCS and the presentation open in parallel.

4.2 COMMUNICATION WITH THE WEBSERVICE

The PDF display component has to react to messages received via websockets and has to translate user actions into messages to the server which are sent using HTTP. The incoming messages the display component has to react to are SlideChange events, meaning the slide

to be displayed has changed to a new slide; its number is contained in the message. The messages the display component has to send to the server are also SlideChange events, but this time, they mean the user has changed to a specific slide. Messages signalling the start and the end of the presentation may be sent to the server and received from the server.

4.3 PROPOSED WORKFLOW

The current workflow for the lecturer using AMCS is as follows: after logging in and selecting the lecture, the lecturer can edit the lecture by creating/editing/deleting questions. After the user is pleased with the questions, the user can decide to start the lecture by setting the state to "active". This shows questions and messages to the students, possibly depending on the currently active slide. To change the active slide, the user may select an arbitrary slide number

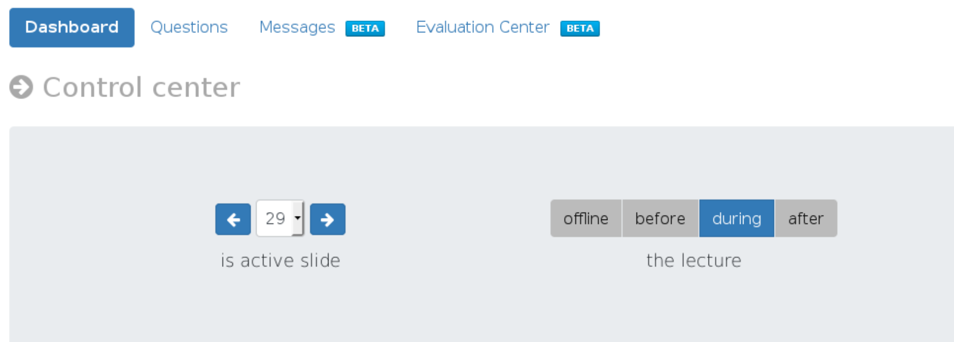

Figure 4.1: Current main lecture control in AMCS

from the dropdown or go one slide back or forward by using the arrow buttons. This has to be done in addition to changing slides in their presentation in an external presentation software.

An improved workflow is presented below.

4.3.1 SETUP

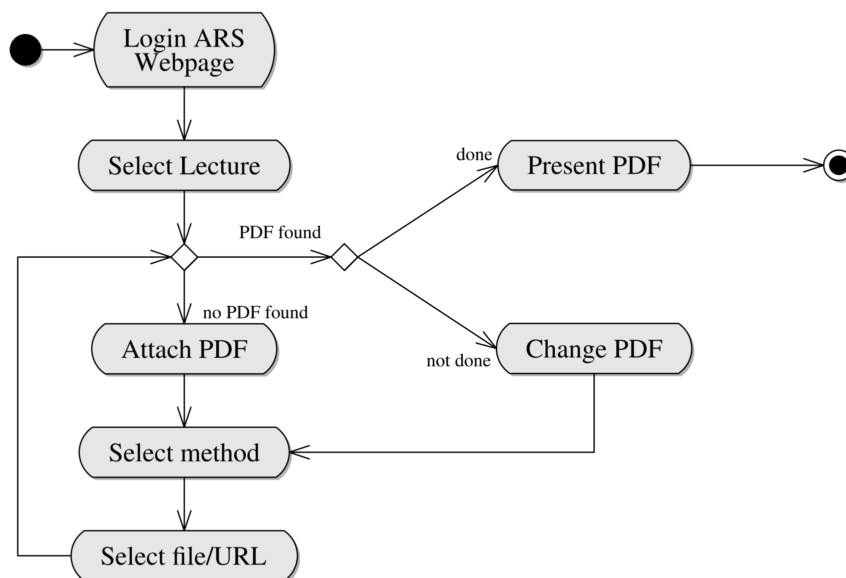

Figure 4.2: Setup of PDF presentation (UML Activity Diagram)

The current workflow gets extended by an option to attach a PDF file to the lecture, as presented in figure 4.2. After logging in and selecting a lecture, the user can decide to add a new PDF to the lecture or replace the existing PDF. This leads to dialog to choose the the method of attaching the PDF: local file, remote file or file upload to AMCS server. Depending on the preferred choice, the user gets presented a file upload dialog, or a dialog to enter the URL of the file to be attached. He may immediately choose to start the presentation of the PDF.

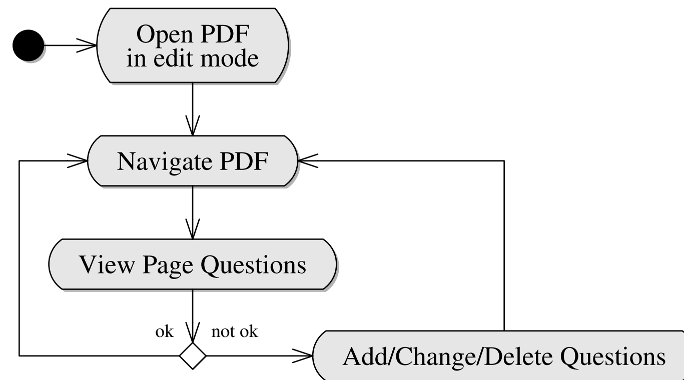

Figure 4.3: Editing questions of presentation (UML Activity Diagram)

To setup the lecture questions, instead of having to remember which slide number of their presentation he wants to add the questions to, the user can navigate the attached PDF in a “preview” mode. This allows them to see what the student will see during the presentation. The lecturer can then decide to add, edit or remove questions to this specific slide. The current workflow for other kinds of questions (questions for all slides, or questions after the lecture) is not to be altered. A schematic of this workflow can be seen in figure 4.3.

4.3.2 PRESENTATION

After attaching a PDF, the user can view some information about the attached file. Then, the user can choose to continue with one window or open a second window, as can be seen in figure 4.4. If the user wants to present their slides on the same screen as the rest of the AMCS application, he can immediately start and open the presentation, navigate through while the PDF page is automatically sent to the server which displays the prepared questions for the currently active slide. To check the polling results, the user has to quit the presentation (because it takes up the whole screen). The presentation can be resumed, if the user is not done yet. The more interesting case is, when the user has two screens available. He presses a button to open the application in another window, then, in the new window, he can start the presentation. To show the presentation on his other screen, the user has to move the presentation window to this other screen, either before or after opening the presentation. With the presentation and AMCS open in parallel, the user can navigate the PDF, the server receives the current presentation state as in the single window case, and see the results on the other screen as well. The user can also navigate through the AMCS application with the presentation still open util he decides to exit the presentation. He may then restart the presentation or close the additional window.

4.4 COMPONENTS OF THE APPLICATION

The defined features and workflow lead to the separation of the system into the components seen in figure 4.5. The PDFRenderer renders a PDF file provided by the PDFProvider into some

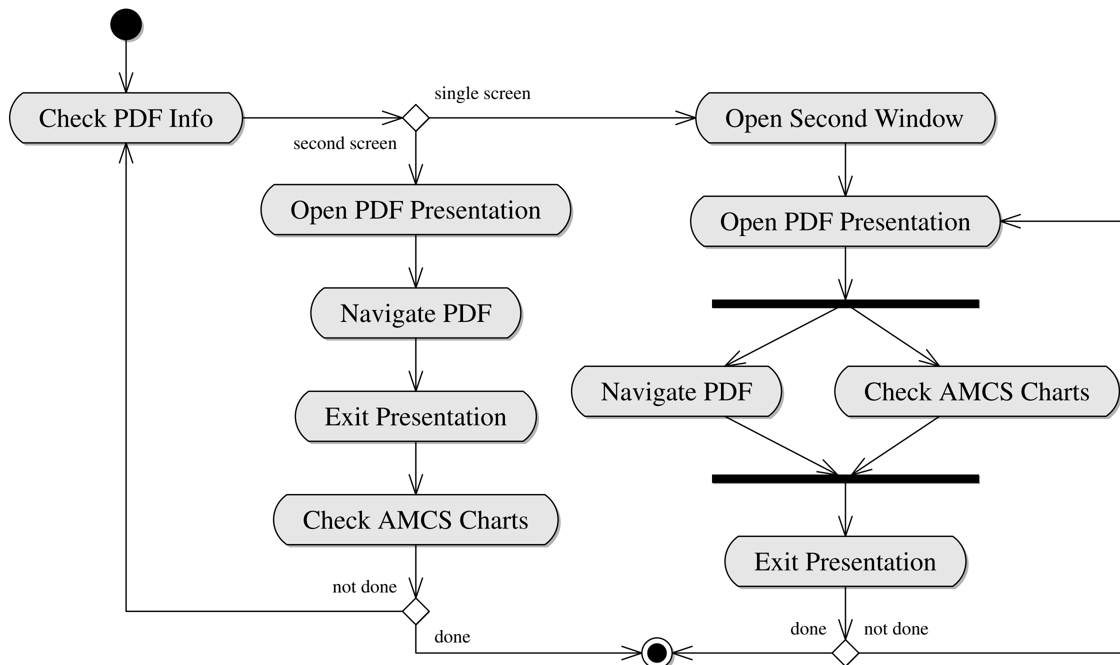

Figure 4.4: Presentation of PDF (UML Activity Diagram)

area of the screen that is watched by PDFNavigator for user input to update the UI and the server. The possibly necessary FileUpload is realized through a PDFUpload component.

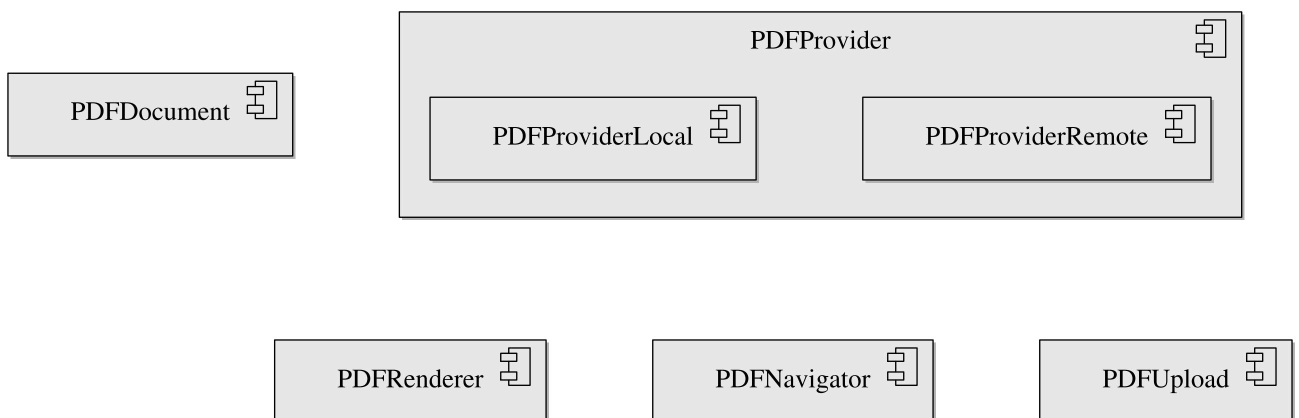

Figure 4.5: Architecture of the system

4.5 INTERFACE MOCKUPS

To enhance the existing interface, two buttons are necessary (see figure 4.6). One button attaches a new PDF or replaces an already attached PDF, another button to start the presentation. The third button may be used to remove/detach a PDF from the lecture, although this is not strictly necessary. The fourth button only plays a role when using multiple screens: it creates a new window to start the presentation in.

These buttons can be in different states, depending on the state of the PDF (figures 4.6, 4.7, 4.8).

To edit a lecture with the support of the PDF view, a view like in figure 4.9 is proposed. The current slide number can be easily seen as well as a short indication about the current questions

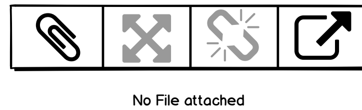

Figure 4.6: Controls to attach PDF to lecture (inital)

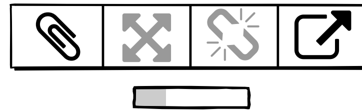

Figure 4.7: Controls to attach PDF to lecture (loading)

on this slide. This combo box can be used to create/update/delete the questions on this slide.

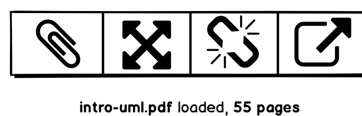

Figure 4.8: Controls to attach PDF to lecture (loaded)

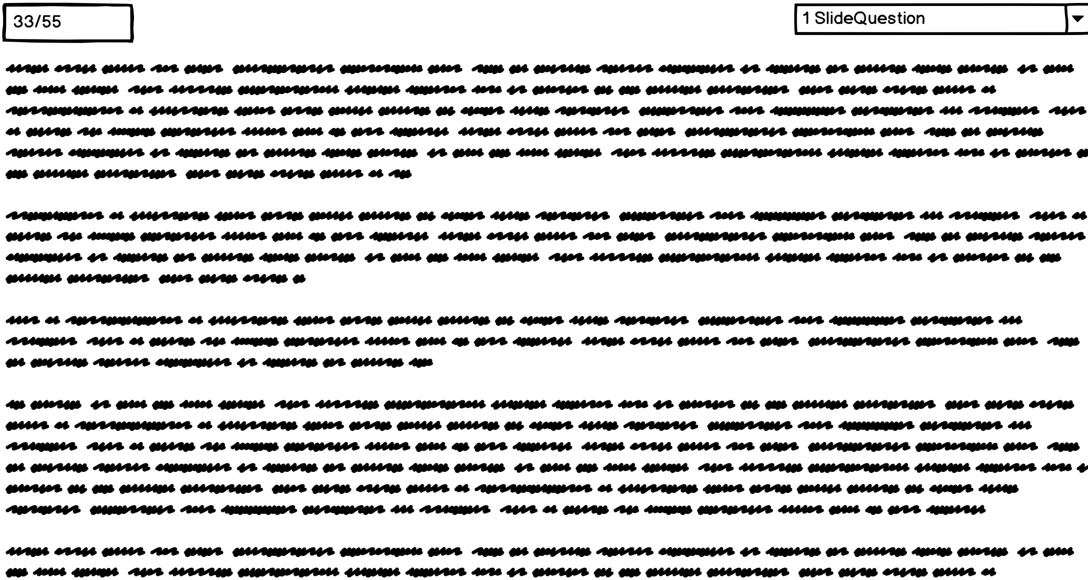

Figure 4.9: Controls to edit lecture through PDF view

5 IMPLEMENTATION

This chapter starts with a short review of the requirements and the concept to discuss possible choices of technology to implement the designed system. Then, the integration with the existing AMCS frontend and backend is presented. The chapter is concluded by showing how specific issues of the proposed concept were handled and how problems were solved.

5.1 CROSS-PLATFORM PDF VIEWER SOFTWARE

The concept proposes to use client-side rendering to be able to render the PDF without upload and without having to touch our existing backend. In section 2.1 a number of solutions for client-side rendering are presented. The rendering via plugins that have to be installed by the user defeats the purpose of a browser-only itegrated solution.

Embedding the PDF inside some webpage of the AMCS application and using the PDF renderer built into the browser is one possibility. To communicate with the AMCS backend to get and set server state, it has to be possible to access the navigation events emitted by the browser when navigating the PDF. This is possible to a certain extent: the PDF rendering of Mozilla Firefox, for example, can be accessed from the surrounding page and events can be intercepted. However, there is no official API to do this sort of thing. This solution depends on the internal DOM structure and events of the browser's PDF rendering engine, which may change between versions and obviously differs between the many different web browsers. Because of these shortcomings, this approach was not pursued further.

Mozilla's PDF.js was chosen over the other solutions to render PDFs directly in the client presented in 2.1, because it is open source, mature, and used in a major browser (Mozilla Firefox) as the default PDF rendering engine.

5.2 INTEGRATION WITH AMCS ANGULAR2 FRONTEND

The AMCS application is built like any Angular2 application: it consists of multiple modules, components and services. Modules provide a way to split the application in smaller chunks which may be lazy loaded. Compontents are responsible for the visible parts of the application and for reacting to input events. Each component consists of a Typescript file, a markup file (containing some HTML derivate) and one or multiple stylesheet files (usually SCSS). Html and SCSS may be embedded in the Typescript file, but this gets unwieldy quickly when the files grow. Services are used to get and set data in an application-wide manner, they can communicate with a web server to propagate or receive changes. Everything is wired together via dependency injection.

The lecture is presented and controlled by the files contained in the directory `src/client/app/lecture/`. The `LectureComponent` contained in the files `lecture.component.*` handles creating, editing and deleting lectures, subscribing to the websocket to receive changes, adding slides to the lecture and removing them. The actual management of the lecture during its presentation by the lecturer is handled by the `control-lecture.component.*` files. This `ControlLectureComponent`, among other things, presents buttons to change the currently active slide in the AMCS system, to change the status of the lecture from "offline" to "before" to "during" to "after". For an example, see figure 4.1. Since the PDF is to be presented during the lecture with the lecturer using this view almost exclusively, it is only natural to augment this control with the necessary user interface to control the PDF presentation.

5.2.1 DISPLAYING PDF FILES

The Angular2 ecosystem offers a rich selection of plugins (providing directives, services or components), which can be installed and added to an existing Angular2 project easily using the NPM package manager. One of the plugins suited for displaying PDF files inside an Angular2 application is "ng2-pdf-viewer"[1]. It can be integrated well into existing Angular2 applications. To use this plugin, it has to be added to the project via `npm install --save-dev ng2-pdf-viewer`. The plugin provides the following Inputs and Outputs:

- src: an URL to the file to be read.
- page: the current page to be displayed, can be Input and Output
- render-text: if the PDF should be rendered with an additional text layer
- zoom: a zoom factor to influence the rendered size
- rotation: the rotation of the displayed PDF page (multiples of 90 degree)
- after-load-complete: an event/output that triggers when the PDF has been loaded initally

During the work on this thesis, the plugin went through a number of iterations, temporarily using SVG instead of Canvas. For the rendering of the PDF files, the functions of the plugin are well-suited.

5.2.2 FILE SELECTION AND UPLOAD

The user has to be able to select the PDF file to be presented. PDF.js can be used to display any PDF file that can be accessed via URI by the browser, with some restrictions discussed later.

Three main methods of providing and opening PDF files through the user have been identified:

1. Having the file saved locally on the same machine the web browser runs on, without the need to upload the file before the lecture or during the lecture to present it.
2. Uploading a local file to the AMCS server or providing the URL of a remote file which the AMCS server downloads, stores and serves as it is viewed.
3. Providing the URL of a remote file for the PDF viewer in the browser to open directly when the PDF is to be viewed.

[1] `https://www.npmjs.com/package/ng2-pdf-viewer` – Last accessed on 29 October 2017

Method 1 provides the user with the advantage of immediately being able to open his PDF file, without the need for up" or download. The downside is, the user cannot easily open some remote file, which he published on his university web site, for example. Technically, this can be realized using a standard HTML5 file selector/file upload dialog. The resulting file can be opened directly in the browser using a blob URI generated from the file open dialog.

Method 2 makes the PDF available anywhere the lecturer may want to present the lecture. Additionally, it could enable switching to server-side rendering in the future, without changing the workflow of the user. The availability and linkage of the PDF to some lecture could be used to provide the students with additional learning material. This method requires up" and download of the file (possibly on multiple devices), changes made to the PDF would require a new upload by the user, and the user would have to make public files he wishes to keep to himself. Since the AMCS backend server would have to be extended to support uploading of files, permissions for downloading, increasing storage requirements, this method has been dismissed, although it may provide interesting possibilities in the future, with respect to uploading different kinds of content to enhance student and lecturer experience.

Method 3 enables the lecturer to change the PDF (e.g. by storing it in the cloud) at will without going through an additional upload procedure. This method would probably be preferrable since it combines ease of use of local and remote files. The challenge with this approach is that PDF.js, using the Javascript technology, is bound by the web application security model. Like any Javascript application, PDF.js can only open documents in conformance with the same-origin policy. This means, in general, no PDF from other hosts can be opened. A way around this could be cross-origin resource sharing, but this has to be implemented by both the AMCS server and the remote server, that is controlled by a third party. Possible solutions like routing the file through an external or internal proxy provided by the AMCS backend, which sets the appropriate headers were considered but not further explored, taking into account that external proxy services add an additional layer of compexity and faiure potential while adding this feature to the AMCS backend is not of high priority.

Opening local files is implemented like shown in listings 5.1 and 5.2.

Listing 5.1: HTML Code to open a file selector in Angular2 app

```
<label for="file-pdf" class="btn btn-default btn-file" style="margin-bottom: 0;">
        <i class="fa fa-folder-open-o" aria-hidden="true"></i>
        <input type="file" id="file-pdf" style="opacity:0; position:absolute; z-index: -1;"
                (change)="fileChange($event)" placeholder="Upload file" accept=".pdf" hidden>
</label>
```

Listing 5.2: Typescript Code to support file opening in Angular2 app

```
    fileChange(event: any): void {
        const fileList: FileList = event.target.files;
        if (fileList.length > 0) {
            const file = fileList[0];
            this.pdfSrc = URL.createObjectURL(file);
        }
    }
```

These two files show the general separation of markup and code in Angular2: a heavily modified version of HTML to output and input data (displaying data and updating the component's model through one" or two-way data binding or events) and a Typescript file containing the component's logic. Under most operating systems, the default display of the file picker does not integrate well into the AMCS layout. Accordingly, it was replaced by a custom button. The file input element cannot be styled at will, so the code shown hides the default file input element, wraps it into a label `for="file-pdf"` which, when clicked, automatically propagates the event to

the input with the id `file-pdf`. This would even work without Javascript enabled. The styling of the label happens via FontAwesome icons[2]. It only accepts one PDF file. The change listener `(change)="fileChange($event)"` on the file input element links the event generated by the user selecting a file and confirming, with the Typescript code in listing 5.2. This event handler is embedded into the `ControlLectureComponent` and extracts the first file of the selected files (which should only be one in total) and sets the instance variable `pdfSrc` of the object `ControlLectureComponent`. Before assignment, an ObjectURL gets created from the file, which continues to point to the file blob in the browser memory and can be opened just like a real URL. If the file object gets destroyed, the URL will produce a network error when being accessed by the browser. This URL variable is used by the ng2-pdf-viewer to load and render the PDF file.

5.2.3 GOING FULLSCREEN

The loaded PDF document can be displayed inside a part of the Angular2 application in the browser window. But to be able to give a typical presentation, the lecturer must have the possibility to present the PDF in fullscreen mode. Modern browsers can be set to a borderless mode, showing no menu but only the webpage. This mode can be activated using the F11 key on Windows and Linux for the browser Firefox. It was considered utilizing this method, but some users might be unfamiliar with it and some users might prefer to continue using their browser in normal mode, i.e. with controls, tabs and possibly not as a maximized window. One familiar and long-established example of another way to show content in fullscreen is Youtube. The videos can be played inside the normal browser window, surrounded by the other content of the website, but at the click of a button, only the video is shown, filling the screen completely. This feature requires browser support and can only be triggered by user interaction, not automatically. This is implemented like this to protect the user from malicious sites. Subsequently, exiting the fullscreen mode cannot be prevented by the application, the user is in full control of his experience and can exit the fullscreen mode by pressing the Escape key on most systems. Browser vendors implemented the Fullscreen API in major browsers by the end of 2011.

The API allows to call `element.requestFullscreen()` on any DOM node which presents this DOM node in fullscreen. Some care has to be taken to fix CSS inconsistencies between browsers. On entering and exiting fullscreen mode, the `fullscreenchange` event is emitted. The fullscreen mode can be exited programmatically using the `document.exitFullscreen()` call. To get information about the current state of the fullscreen status, one can query whether the browser is in fullscreen mode through `document.fullscreenEnabled`, the element displayed is `document.fullscreenElement`. Fullscreen mode is exited automatically, when the user navigates away or changes tabs, but is is continued, if the user merely changes to another window, which might be another browser window as well.

The Angular2 plugin ng2-bigscreen encapsulates this functionality and provides it conveniently inside Angular2 without directly accessing the DOM which, as a general rule, is to be avoided using Angular2. The integration with the existing AMCS application proved difficult. Recently, the build process was switched to Ahead-of-time compilation with Tree-shaking, which aims to reduce the load-time of the application in the browser by “shaking” unused dependencies out of the build and compiling the templates in the build step already, not in the client browser. Unfortunately, this AoT functionality requires the plugins to follow certain guidelines, which the plugin does not. Since the Fullscreen API is not that extensive, the API is accessed without a plugin.

The interaction is realized using another button, when it is clicked, `requestFullscreen` is called on the DOM node where the PDF is displayed. On the `fullscreenchange` event, the PDF size gets recalculated to fit the screen. On several occasions, `document.fullscreenEnabled` is queried to be

[2] http://fontawesome.io/icons/ – Last accessed on 29 October 2017

able to handle input events differently in fullscreen or hide the PDF if not in fullscreen.

5.2.4 CAPTURING INPUT EVENTS

The change of the PDF file was aready covered in 5.2.2. Since Angular2 aims to provide an abstraction of the DOM, that can also be compiled to be used without the browser runtime (NativeScript), it is recommended to not register event listeners directly, like in normal Javascript, but use the `@HostListener` abstraction like in listing 5.3

Listing 5.3: Typescript Code to listen for keydown events in Angular2

```
@HostListener('window:keydown', ['$event'])
        public keyboardInput(event: KeyboardEvent): void {
                if (!this.bigScreenService.isFullscreen())
                        return;

                //use deprecated event.keyCode instead of event.key
                //because IE uses different codes in the recommended event.key
                switch(event.keyCode) {
                        case 40: //ArrowDown
                        case 39: //ArrowRight
                                event.preventDefault();
                                this.changePDFPage(1);
                                break;
                        case 38: //ArrowUp
                        case 37: //ArrowLeft
                                event.preventDefault();
                                this.changePDFPage(-1);
                                break;
                }

}
```

This code annotates the method `keyboardInput` (of the `ControlLectureComponent`) with the annotation `@HostListener` with the arguments event type and arguments to pass from the event to the method. After skipping the event when the page is not shown in fullscreen mode, the event is translated to the pressed key and the page number of the PDF is updated depending on the key. After this, the server has to be updated, so the existing code to move forwards/backwards one slide is called. Equivalently, pressing the left mouse button moves to the next PDF page and updates the server with the new slide. There is no need to capture the event of leaving fullscreen, since this gets handled by the bigscreen service, which gets queried by the HTML markup which hides the PDF

5.2.5 COMMUNICATION BETWEEN BROWSER WINDOWS (?PRESENTATION AND APPLICATION?)

To enable the user to open the presentation on a different screen (e.g. a projector) while still being able to view the questions simultaneously, several possible solutions exist. It is not possible to open the mentioned fullscreen mode on a different screen than the browser is displayed on. This implies, there have to be two different browser windows: one for the presentation (which might be hidden under the presentation while the PDF is in fullscreen) and one “main” window for the AMCS statistics and results.

While the main window has to load the AMCS application to provide all the existing functionality, the extra "presentation" window could be running an instance of the AMCS application as well, or just run the PDF viewer (outside Angular2) and talk with the AMCS server directly through HTTP or indirectly through messages to the main window. A plain window running only the PDF presentation with direct communication to the server would require some of the existing functionality in the AMCS application to be reimplemented. Even if a bit duplicated, this is certainly feasible and would make for a leaner experience without Angular2 running in the back of the presentation (see Evaluation 6). The user would open this extra window through the AMCS application and could load the PDF in the extra window. Talking indirectly with the server through messages between windows would work similarly, carry the implementation overhead of implementing the bidirectional message passing, but would not duplicate the communication code with the server. Since it was a requirement to be able to use only a single window to show both, the presentation and the AMCS application interleaved in the same window, the solution implemented works both with one or multiple windows. To achieve this, the AMCS Angular2 application is loaded in each window: the initial window and any windows spawned by this window (e.g. to show the presentation). Then the full functionality can be used by the presentation window. With other words, each window is a client on its own.

With the decision to use local PDF files, the AMCS application suddenly has important state that cannot be shared with the server, but is local to the client. If the PDF is opened in only one of the windows, the other window would not show this. This would prevent the user from seeing any info about the PDF in the main window and also prevent the user from opening the PDF in a window that is not the presentation window. To share this state, messaging between windows comes to mind.Another option is to use the localstorage API[3], which was implemented (see listing 5.4). As a sideeffect, this solution also enables the application to remember the PDF blob URL if the user navigates inside the application, which normally unloads the PDF component and any variables stored inside the lecturer component.

Listing 5.4: Typescript Code use the local storage API

```
1                 const pdfSrc = localStorage.getItem('pdfSrc');
2                 const pdfName = localStorage.getItem('pdfName');
3                 const pdfDate = localStorage.getItem('pdfDate');
4 /////////
5                 localStorage.setItem('pdfSrc', this.pdfSrc);
6                 localStorage.setItem('pdfName', this.pdfName);
```

5.2.6 ANGULAR2 CHANGE DETECTION MECHANISMS

After the first preliminary tests of the implementation, it was found, that the browser slowed down heavily when changing the page of the PDF, to the point, where rapidly switching between slides became impossible. Performance analysis showed, that not the PDF rendering was at fault, but that some code in the original AMCS application was taking up all the CPU ressources for multiple seconds after a change of the PDF page.After the rendering of the new PDF page was complete, this part of the code was called hundreds of times, coming from Zone.js.Further investigation revealed, this is caused by Angular2 change detection: Angular2 hooks, using Zone.js, into any events that might change something in the application ((input) events, timeouts and data arriving over the wire). Since Javascript does not give guarantees about the immutability of objects, Angular2 has to run change detection over the whole tree of components everytime it detects a possible change. A nice explanation of the whole change

[3]https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API – Last accessed on 29 October 2017

detection mechanisms in Angular2 can be found at [4] and [5].

The problem is twofold. Some root component of the AMCS application is doing expensive work (decoding a JSON Web Token for permissions to determine if some part of the application has to be displayed) everytime the change detection gets triggered. Together with the PDF renderer triggering a myriad of changes while it renders a PDF page, this leads to a massive performance degradation. To obviate this, Angular2 offers different mechanisms to either make change detection of some subtrees completely unnecessary (by using Observables or immutable objects) or to make a component not trigger change detection on every event, but only in an interval or completely manually. Introducing Observables or Immutables could make the application more performant as a whole, but would require a revamp of the whole infrastructure. Instead, triggering change detection in the PDF rendering component only when a page is complete gets to the root of the problem. This can be done by rewriting parts of the PDF rendering component, injecting NGZone and ChangeDetectorRef and using runOutsideAngular as in 5.5. The PDF is rendered outside of Angular, Angular does not get any events from the rendering, after the rendering is complete, we changeDetectorRef.markForCheck() is called, to mark this component as possibly dirty.

Listing 5.5: Typescript Code to prevent Angular2 from triggering change detecion on every event

```
ngOnChanges(changes: SimpleChanges) {
        this.zone.runOutsideAngular(() => {
                if ('src' in changes) => {
                        this.loadPDF();
                } else if (this._pdf) {
                        if ('renderText' in changes) {
                                this.setupViewer();
                        }
                        this.update();
                }
        });
}

private render() {
        this.zone.runOutsideAngular(() => {
                //rendering
        });
        this.changeDetectirRef.markForCheck();
```

5.3 COMMUNICATION WITH THE AMCS BACKEND

The existing AMCS application already has the functionality to communicate with the AMCS backend built in. The required functionality for realizing the PDF display and controlling the webservice is:

1. Sending a request from the client to the server that sets the active slide to a new value.
2. Receiving a push message from the server that changes the active slide.

[4] https://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html – Last accessed on 29 October 2017

[5] https://vsavkin.com/change-detection-in-angular-2-4f216b855d4c – Last accessed on 29 October 2017

In the current AMCS implementation, two ways exist to change the currently active slide: clicking the forward/backward button, which switches to the next/previous slide and directly selecting a slide from the dropdown. Both methods call the method `setActiveSlide` which calls `slideService.setActiveSlide` and subscribes to the resulting Observable to handle success and error cases. The existing code is displayed in listing 5.6.

Listing 5.6: Typescript Code of existing method to handle slide change requests

```
setActiveSlide(slide : Slide) {
        this.slideRequestCompleted = false;

        this.slideService.setActiveSlide(this.lectureId, slide.id).subscribe(
                data => {
                        this.slideRequestCompleted = true;
                        console.log("set active slide to slide "+slide.position)
                },
                error => {
                        this.slideRequestCompleted = true;
                        // set activeSlide back to old slide object (position is still not changed)
                        for(let i=0; i<this.slides.length; i++) {
                        if(this.slides[i].position === this.lecture.activeSlide.position) {
                                this.lecture.activeSlide = this.slides[i];
                                break;
                        }
                }
        });
}
```

The slide service internally calls the backend's REST API. The actual update of the client's active slide does not happen through the responses to these requests, but through another channel: the websocket subscription. This can lead to multiple requests being inflight at the same time, if they get sent in rapid succession. This can cause updates coming in through the websocket to overwrite the current slide with stale data. In the GUI, which normally shows the last slide set by the user, this can be observed by selecting the dropdown, and depressing the down arrow key for a couple of seconds. At first the displayed slide increases just by the user's change to the GUI, then the websocket receives the individual events, the GUI updates itself and the slide number is set to the different numbers the user went through until it reaches the slide the user stopped at.

This does not happen often during normal use, since the slide number is normally just incremented or decremented by one, or the slide is set to one specific slide and left there for a couple of seconds. But it can happen often, if the user flicks through the PDF pages in rapid succession, for example when navigating to their literature page or when answering questions. A naive propagation of the incoming websocket events to the displayed PDF page would cause flickering of the pages and result in bad user experience. To remedy this, the updates coming in from the websocket can either be ignored completely, or the change events created by navigating the PDF are debounced for a couple of seconds. Ignoring the websocket updates when a PDF is presented prevents the PDF page to be updated from external devices the lecturer might use, like a smart phone. When working with two windows, it also prevents the user from selecting a slide in one window via the dropdown and seeing the updated PDF page in the other window. Debouncing, on the other hand, keeps all the websocket functionality in place and only delays sending slide change requests for a set number of seconds. When the slide gets changed again in the meantime, the original timer gets canceled and the timer is started anew. Schubotz uses a 5 second debounce [15]. This is implemented in listing 5.7

Listing 5.7: Typescript Code to debounce slide changes

```
1  changePDFPage(delta: number): void {
2          this.page += delta;
3
4          //only set slide to active, if we stay 5000 ms on the same PDF page
5          if (this.timeoutChangeActiveSlide) {
6                  clearTimeout(this.timeoutChangeActiveSlide);
7          }
8          this.timeoutChangeActiveSlide = setTimeout(() => {
9                  this.setSlideIdToActive(this.slidePositionToId(this.page));
10         }, 5000);
11 }
```

6 EVALUATION

6.1 QUALITATIVE EVALUATION

6.1.1 METHOLOGY DISCUSSION

The International Organization for Standardization gives a recommendation for key metrics to evaluate the usability of a system in ISO/IEC 9126-4. [10] The standard recommends the usability metrics to include:

- Effectiveness: How accurate and complete the user can solve specific problems
- Efficiency: How much resources the user had to invest in relation to the level of completeness of solving a specific problem
- Satisfaction: How the user accepts the system and the comfort he uses the system with

Effectiveness is typically measured using completion rate (number of tasks completed successfully versus number of total tasks) or number of errors per task. Since the number of different tasks in this evaluation is low, completion rate will only provide high confidence, if the number of participants is high. Therefore, the simpler metric of errors per task is chosen. "Errors can be unintended actions, slips, mistakes or omissions that a user makes while attempting a task. You should ideally assign a short description, a severity rating and classify each error under the respective category."[1] Together with the categorization of the mistakes, this may provide qualitative feedback of what to improve, in contrast to the merely quantitative feedback of completion rate.

Efficiency can be measured in terms of time the user needs to successfully complete a task. This is easily measured by timing each participant for each task and taking special care of users no being able to complete a task. Several systems exist to condense those individual measurements into a single number over multiple tasks and multiple participants, but for detailed analysis the individual measures are better kept.

To measure user *satisfaction* there are many different ways, varying in granularity, degree of complexity and effort of the survey:

- Task Level satisfaction (e.g. NASA Task Load Index, TLX): done after each task, measures satisfaction for individual tasks/workflows.
- Test Level satisfaction (e.g. System Usability Score, SUS): done after using the whole system, measures satisfaction with whole system.

[1] `https://usabilitygeek.com/usability-metrics-a-guide-to-quantify-system-usability/` – Last accessed on 29 October 2017

The developed system does not have many different possible user interactions, there are only a handful of different workflows, so it is feasible to get an individual usability score for each of the workflows. This enables the individual assessment of each workflow and the identification of weak areas and possible improvements. Using a Test Level Satisfaction measure to get the "big picture" of the system would be more useful for bigger systems, where it becomes increasingly more time-consuming to do individual tests on a per-task level.

6.1.2 SETUP

Most test subject's native tongue is German, so the initial explanation of the test, the individual tasks and the scoring sheet, as well as any other verbal communication were given/done in German. The used documents can be reviewed in the appendix.To save time, all participants were briefed about the general flow of the experiment together in the same room. Then, each participant was called into a separate room to be given some information necessary to complete the tasks. An example of the scoring sheet without a task, but explanations for each individual scale was presented and read. The participant was asked to assume the role of a lecturer about to present his (fictional) lecture "Introduction to document management" having prepared a set of slides in the file *presi.pdf* in the directory */Downloads*. General questions about the experiment and the scoring were answered.

The following equipment was used: a HP Elite Book 8440p (laptop) with on-board GPU (Intel HD), Intel i7 M620 Dual Core CPU (4 virtual cores via Hyperthreading) with 2.67 GHz and 4 GB RAM running Windows 10 Enterprise. One participant asked to use their own laptop. While this would make for a more natural and realistic experience for the participant (he is used to the input devices, operating system, screen, software), which might make the experiment capture more real-world data, for the sake of consistency and reproducibility, the request was denied. The laptop was connected to a projector using the extended desktop mode to simulate the setup of most lectures/presentations: a beamer and a second computer screen that can display other data. After 3 of 5 participants, the experiment had to be moved to another room, which did not have a projector. To compensate, an external monitor was used in place of the projector. To solve the tasks, the participants were offered the choice between a mouse, a hand-held presentation tool or the laptop's touchpad with keyboard. All participants chose to use no external device. Most remarked, they would naturally prefer this method of input for giving presentations. The experiment was conducted using our test server backend, the frontend ran on the same machine the participants were using. The frontend was operated through the web browser Mozilla Firefox 56 which had the AMCS start page already open.

During the experiment, each participant had to do all of the following tasks in succession, each task followed by a TLX sheet:

1. After logging into the AMCS platform and choosing the lecture, link the document you prepared to the lecture.
2. It strikes you that you made a last-minute change to the presentation. Check, if you indeed linked the corrected document "presi_dokumentenverwaltung.pdf".
3. Link the correct document "presi_dokumentenverwaltung.pdf" in the same direcory.
4. The lecture starts. After setting the lecture state to "active", start to present your slides. Navigate to the slide "Herausforderungen".
5. On this slide, you previously prepared some questions, that are presented to the students now. Peer over the charts generated by the system.
6. Up to now, you presented the slides and the analysis of the questions on the same output device. However, you have two screens at your disposal, and want to follow the coming

polls without interrupting your presentation. Therefore, open the presentation on the other screen.

7. In the middle of the presentation, you want to show your literature references, but they are one slide before the end of your presentation. Navigate there and, after a short moment, back to the current slide.
8. Navigate to the slide 17 ("einzelner Fokuspunkt in 3D"). You provided questions for this slide, make sure, the system shows the questions to the students and you can see the analysis of the questions.
9. You want to show your other lectures to the students briefly. End the presentation and navigate back to the overview of your lectures. Shortly after, navigate to the current lecture and restart the presentation.
10. While the presentation is active on one screen, open the "old" document "presi.pdf" anew and make sure, it is really different from the other document (there are slides missing at the end).

The tasks are formulated like this, to only tell the participant, *what* they have to achieve, but leave open *how* to achieve the tasks. While this may improve the measurement on how intuitive the system is to use, this open-endedness may result in worse assessment of the individual features the system provides (if the participant uses another feature of the system, not supposed to be evaluated, but also satisfies the task).

To gather more feedback about *what could possibly improve the the system*, not only about *which part needs improvement*, all tasks were completed as "Thinking Aloud Test". This prompts each participant to utter any thoughts they have while completing each task. Participants are encouraged to verbalize the steps they are doing/have to do in order to complete the tasks. Thoughts and problems encountered (and verbalized) by the participants are recorded to provide possible improvements. Each participant is, after the completion of all tasks, asked explicitly what tasks/features they had problems with and what could be improved in the evaluated system.

While this thinking aloud protocol may provide many cues about possible improvements, it slows down the completion of each task, sometimes severely. The participants take more time to think about their tasks and possible improvements, even referring to the task they just completed. It became apparent, that timing each participant for each task was not yielding reliable times to measure *Efficiency*. Therefore, only time taken for all tasks in total has been measured.

Effectiveness has been measured by recording errors and hiccups in parallel to recording the thinking aloud protocol.

6.1.3 RESULTS TASK LOAD INDEX

All 5 participants were already familiar with the system before the evaluation. Some were involved in the programming or have already used it in production.

The TLX Sheet was slightly altered by inverting the direction of the scale for "Performance". This has the advantage that the direction of the scales is always the same (from low to high), not confusingly inverted in the middle. On the other hand, for all scales but "Performance" a "desirable", "easy task" value would be on the left, so in this might be confusing as well. This was displayed wrong on the explanation sheet given, but only one participant seemed to get the "Performance" scale wrong, which was corrected by questioning him after the test. For the final scores, lower is better for each scale.

Plot 6.1 shows the first five tasks, while 6.2 shows tasks 6 to 10. For each task, each scale is shown as box plot. Each plot consists of the median (the line inside the box), the box that

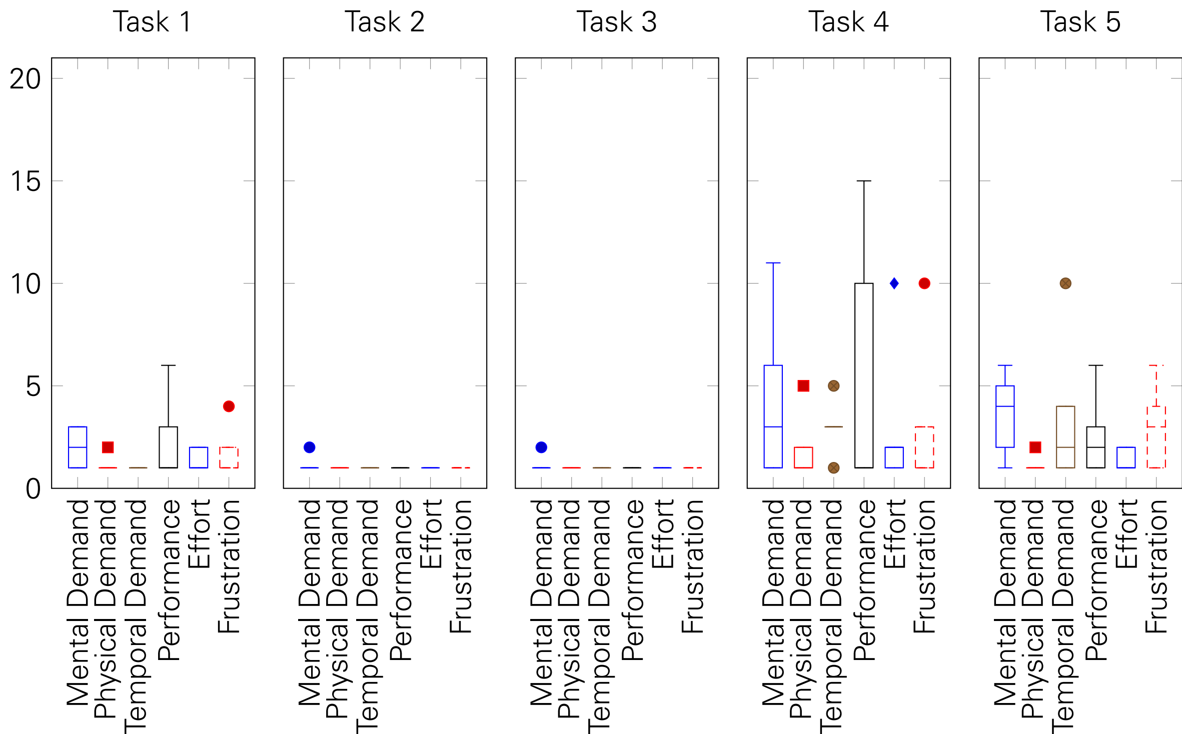

Figure 6.1: TLX results of tasks 1 to 5

ranges from the (lower) 0.25-quantile of the data to the (upper) 0.75-quantile, the lower whisker (the smallest data value still larger than the lower quantile - 1.5 * the inter quartile range), the upper whisker (likewise) and outliers (data points outside the whisker range). Since there are so few datapoints, some scales are a bit "boring".

Presentation tool does Navigating in BildHoch BildRunter

6.1.4 RESULTS OF THINKING ALOUD

Task 1: Every participant understood the task, some mentioned, that it was good, that only one button can be activated and the others are grayed out.

Task 2: Some participants clicked on unlink first, instead of directly replacing the attached PDF.

Task 3: The bad layout when selecting a PDF with long filename was mentioned by two participants. One participant tried to open multiple files at once and was relieved that that did not work out.

Task 4: One participant did not find the choice of the current symbol for the fullscreen action intuitive. Another complained, that the presentation did not start automatically on the projector.

Task 5: The majority of participants tried to press the *Alt+Tab* key combination to leave fullscreen mode. After they recognized this did not work, all participants tried the *ESC* key and completed the task.

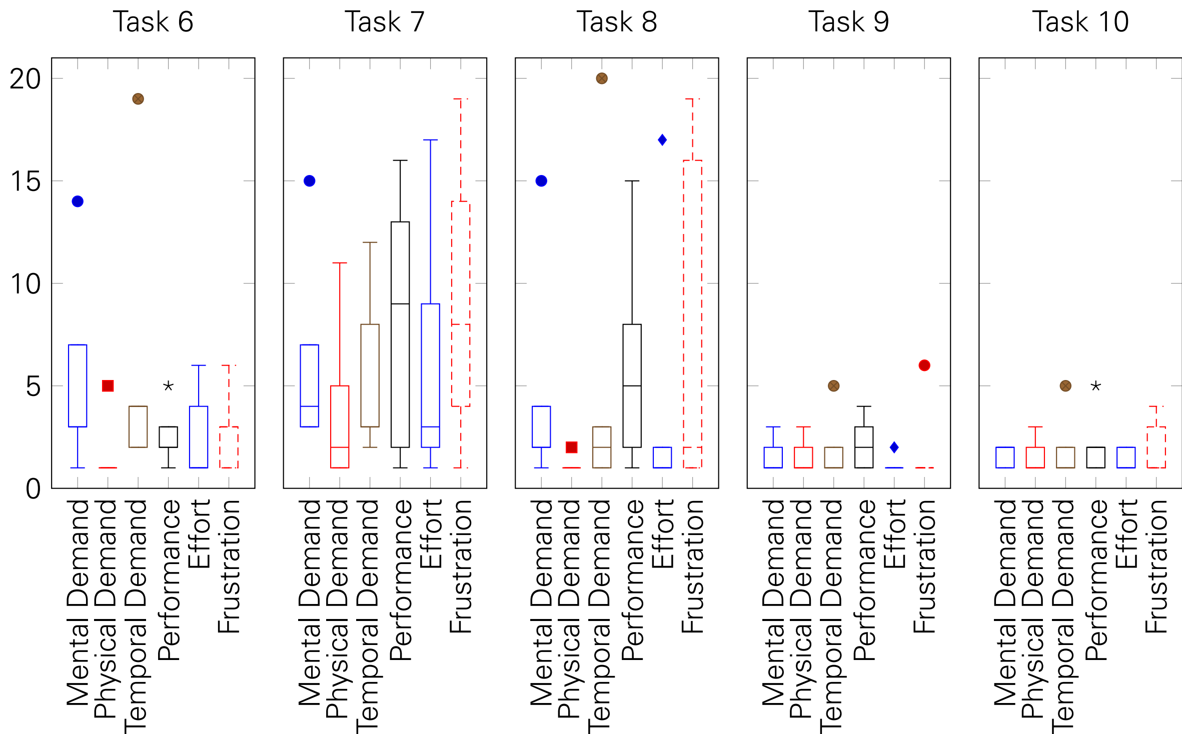

Figure 6.2: TLX results of tasks 6 to 10

Task 6: All participants recognized they had to open a second window, but most failed to do this by opening another tab or cloning the curren tab. Instead, participants tried to move the current window to the other screen, switched multiple times between fullscreen and normal AMCS window, tried to click the fullscreen button while holding down the *Ctrl* key, wanted to open in external viewer or tried all other buttons.

Task 7: One participant tried to enter the page number directly, another tried to use the drop-down, which was not working. Two participants directly used the *End* key. One participant managed to break the application, the window was restarted.

Task 8: Most participants used the *Home* key, and then the arrow keys to get through the other slides. Most participants remarked, that 5 seconds is way too long for the change of the slide to become active and sent to the server, 1 second was proposed more than once.

Task 9: One participant was a bit puzzled by the task description, but solved it alright after a short while.

Task 10: Two participants found the example PDFs to be chosen badly, because it was not easy to see if the two were different at all. One participant found it unusual/unexpected to be able to change the PDF while it was open on the other screen, the normal workflow would be to close it first.

All Tasks: Sometimes, participants did not know what each button meant, they did request the addition of tooltips on mouse over. It was mentioned, that the shown date the PDF was opened on is not very helpful or confusing. There was a lot of confusion caused by the numbering of pages: the PDF used had slide numbers from 1 to 46, but consisted of 94 single

pages. This is caused by hiding some elements first and fading them in later on the same slide. This produces a PDF with a larger page count than the "real" number of slides. Instead of a whole new AMCS application, one participant whished for a lean window which acts more like a popup or overlay window that can easily be positioned on the other screen. When testing the hand-held presentation tool, it was found, that it does not emit *Arrow-Key* events like expected, but *PageUp, PageDown* events, that are not yet supported.

The measured number of failures, since every test subject completed every task.

The measured times are very similar, they range from 13 to 20 minutes.

6.1.5 INTERPRETATION OF THE RESULTS

Since the tasks were open-ended, it is not trivial to match each task to a requirement.

Task 1: Users were satisfied, the TLX scores show requirement A1.1 is satisfied.

Task 2: Users were satisfied, the unlink button may be superfluous, the results show requirement A1.4 is satisfied.

Task 3: Changing the attached PDF worked as expected, TLX results are very good as well, this satisfies requirement A1.5.

Task 4: This task tests presenting the PDF in fullscreen mode and simple navigation as well. The TLX results can be explained either by a bad experience when changing to fullscreen mode, or by bad experience navigating. With the information collected from the Thinking-Aloud-Test, it is likely, that the problem is bad design of the interface (icons, where the presentation starts), and not the navigation itself. The task shows that requirement A3.1 and A3.2 works, but can be implemented using better iconography and maybe a short introduction.

Task 5: While every web browser shows a short message that the user can leave fullscreen mode using the *ESC* key, most users did not use this hint. This suggests that a small introduction or explanation may help to better fullfill requirement A3.3.

Task 6: The TLX and the complaints of the participants suggest the feature A3.4 is not that easy to use. This can probably be alleviated by introducing a keyboard shortcut to open a second window, or displaying yet another button to make this function more prominent.

Task 7: This task is the most frustrating and overall most dissatisfying task. While navigating is a very important task, navigating many pages happens seldomly. This task does not even have a matching requirement. Nevertheless, this task has to be improved by correctly implementing the propagation of the slide number from the dropdown and possibly some shortcuts remarked by the participants, e.g. directly entering a slide number.

Task 8: This task also got very mixed results; lowering the delay of the SlideChange event to one or two seconds may improve this significantly, together with the suggested improvements for task 7. The requirement A3.5 is not satisfied sufficiently.

Task 9: Navigating the AMCS system during a presentation works good, TLX and user comments affirm that. This satisfies A3.6.

Task 10: TLX scores suggest users having an easy time with this task, but the comments suggest, that it was not clear which feature this test was aimed at.

This leaves open requirements A1.2, A1.3 which require changes to the backend. All requirements in the group A2 are not satisfied, but they were low priority nice-to-have features, that already exist in AMCS in another form. Requirements A3.7, A3.8 were not evaluated stringently, but at least A3.7 was fullfilled during the test.

7 CONCLUSION AND FUTURE WORK

After presenting the current AMCS, and some related work to control the AMCS system, use cases have been created to let a use control the AMCS server while giving a presentation using the PDF file format. From these use cases, requirements have been derived. A system to satisfy these requirements has been proposed and implemented. The evaluation of the implemented system with users alredy familiar with AMCS showed that the core requirements have been satisfied, but give many hints for improvement.

Possible improvemts include:

- enhancing the amcs backend to store files, this can be usedfor PDF and other presentation formats like PPT.
- more navigation options (directly skipping to a page by entering the page number, supporting *PageUp, PageDown* keys
- adding tooltips and/or a short explanation of the fullscreen functions
- implementing a lean presentation window, not a second window running a full AMCS application
- enhancing AMCS by implementing cheaper change detection and/or using Immutables or Observables to improve performance
- to prevent overwriting client state with the client's own requests, vector clocks or unique IDs could be implemented
- to improve performance, PDF.js could switch to Web Assembly/asm.js

LIST OF FIGURES

3.1 Structure of AMCS (and several other ARS) 18
3.2 Attaching a PDF to a lecture (UML Use Case Diagram) 19
3.3 Presenting a PDF (UML Activity Diagram) 19
3.4 Editing a lecture through PDF view (UML Activity Diagram) 20

4.1 Current main lecture control in AMCS 24
4.2 Setup of PDF presentation (UML Activity Diagram) 24
4.3 Editing questions of presentation (UML Activity Diagram) 25
4.4 Presentation of PDF (UML Activity Diagram) 26
4.5 Architecture of the system 26
4.6 Controls to attach PDF to lecture (inital) 27
4.7 Controls to attach PDF to lecture (loading) 27
4.8 Controls to attach PDF to lecture (loaded) 27
4.9 Controls to edit lecture through PDF view 28

6.1 TLX results of tasks 1 to 5 42
6.2 TLX results of tasks 6 to 10 43

BIBLIOGRAPHY

[1] Adobe Press. *PostScript Language Reference Manual*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1985. ISBN: 978-0-201-10174-4.

[2] Adobe Systems Incorporated. *What is PDF? Adobe Portable Document Format | Adobe Acrobat DC*. 2017. URL: https://acrobat.adobe.com/us/en/why-adobe/about-adobe-pdf.html (visited on 09/26/2017).

[3] Lars Beier, Iris Braun, and Tenshi Hara. "auditorium-Frage, Diskutiere und Teile Dein Wissen!" In: *GeNeMe*. 2014, pp. 117–124.

[4] Iris Braun et al. "Onlinegestützte Audience Response Systeme: Förderung der kognitiven Aktivierung in Vorlesungen und Eröffnung neuer Evaluationsperspektiven". In: (2015).

[5] Clemens H. Cap. *Tweedback*. 2016. URL: http://www.tweedback.de/ (visited on 09/28/2017).

[6] Richard Cohn and Tim Bienz. *Portable Document Format Reference Manual*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN: 978-0-201-62628-5.

[7] Michael Eichhorn. "Elektronische Abstimmungssysteme in der Hochschullehre–Empirische Untersuchung zu Erfahrungen mit dem Audience Response System eduVote". In: *Gesellschaft für Informatik eV (GI) publishes this series in order to make available to a broad public recent findings in informatics (ie computer science and informa-tion systems), to document conferences that are organized in co-operation with GI and to publish the annual GI Award dissertation.* 2016, p. 191. URL: https://www.researchgate.net/profile/Michael_Eichhorn3/publication/308118998_Elektronische_Abstimmungssysteme_in_der_Hochschullehre_-_Empirische_Untersuchung_zu_Erfahrungen_mit_dem_Audience_Response_System_eduVote/links/57daa2ab08aeea1959329096/Elektronische-Abstimmungssyst
in-der-Hochschullehre-Empirische-Untersuchung-zu-Erfahrungen-mit-dem-Audience-Response-System-eduVote.pdf (visited on 09/28/2017).

[8] International Organization for Standardization. *ISO 32000-1:2008 - Document management – Portable document format – Part 1: PDF 1.7*. Tech. rep. 2008, p. 747. URL: https://www.iso.org/standard/51502.html (visited on 09/26/2017).

[9] International Organization for Standardization. *ISO 32000-2:2017 - Document management – Portable document format – Part 2: PDF 2.0*. Tech. rep. 2017, p. 971. URL: https://www.iso.org/standard/63534.html (visited on 09/26/2017).

[10] International Organization for Standardization. *ISO/IEC TR 9126-4:2004 - Software engineering – Product quality – Part 4: Quality in use metrics*. URL: https://www.iso.org/standard/39752.html (visited on 11/01/2017).

[11] M. Johst. "Entwicklung einer Applikation zur Steuerung von Präsentationen in einer Vorlesung mit Smartwatches und der Kopplung mit der ARS-Plattform AMCS". 2016.

[12] F. Kapp, I. Braun, and H. Körndle. "Aktive Beteiligung Studierender in der Vorlesung durch den Einsatz mobiler Endgeräte mit Hilfe des Auditorium Mobile Classroom Services (AMCS)". In: *Symposium auf dem 49. Kongress der Deutschen Gesellschaft für Psychologie; Verbesserung von Hochschullehre: Beiträge der pädagogischpsychologischen Forschung*. 2014.

[13] F. Kapp et al. "AMCS: a tool to support SRL in university lectures based on information from learning tasks". In: *Summerschool Dresden 2014, Dresden, Germany* (2014).

[14] T. Kubica. "ENTWICKLUNG EINES PROTOTYPS ZUR AUSWAHL UND ZUM EINSATZ TECHNISCHER WERKZEUGE/WERKZEUGKOMBINATIONEN IN UNTERSCHIEDLICHEN LEHRFORMEN". 2016.

[15] A. Schubotz. "Entwicklung einer Lecturer App zur Kopplung einer Präsentationssoftware mit der ARS-Plattform AMCS". 2016.

[16] Alia Sheety. "If you can't beat them, join them". In: *E-Learn: World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education*. Association for the Advancement of Computing in Education (AACE), 2015, pp. 921–925.

[17] SimpleSoft - Buchholz Wengst GbR. *Audience Response System | eduVote*. 2017. URL: http://www.eduvote.de/ (visited on 09/28/2017).

[18] Jonas Vetterick, Attila Altiner, and Clemens H. Cap. "Enhancing Medical Students Exam Revision Course with Modern Classroom Response Systems". In: (2014). URL: https://www.researchgate.net/profile/Jonas_Flint/publication/268870261_Enhancing_Medical_Students_Exam_Revision_Course_with_Modern_Classroom_Response_Systems/links/5479a1530cf205d1687fa779.pdf (visited on 09/28/2017).

[19] Jonas Vetterick, Martin Garbe, and Clemens H. Cap. "Tweedback: A Live Feedback System for Large Audiences." In: *CSEDU*. 2013, pp. 194–198. URL: http://ai2-s2-pdfs.s3.amazonaws.com/ef5d/b9aec93f62e9cbdf4cb0c14ac22e7e0d5803.pdf (visited on 09/28/2017).

[20] John Warnock. *The camelot project*. Tech. rep. 1991. URL: http://www.eprg.org/G53DOC/pdfs/warnock_camelot.pdf (visited on 09/26/2017).