UiO : Department of Informatics
University of Oslo

Bagadus App: Notational data capture and instant video analysis using mobile devices

Anders Emil Rønning, Jon Hammeren Nilsson

Master's Thesis Spring 2015

Bagadus App: Notational data capture and instant video analysis using mobile devices

Anders Emil Rønning, Jon Hammeren Nilsson

Abstract

Enormous amounts of money and other resources are poured into professional soccer today. Teams will do anything to get a competitive advantage, including investing heavily in new technology for player development and analysis. In this thesis, we investigate and implement an instant analytical system that captures sports notational data and combines it with high-quality virtual view video from the Bagadus system, removing the manual labor of traditional video analysis. We present a multi-platform mobile application and a playback system, which together act as a state-of-the-art analytical tool providing soccer experts with the means of capturing annotations and immediately play back zoomable and pannable video on stadium big screens, computers and mobile devices. By controlling remote playback and drawing on video through the app, sports professionals can provide instant, video-backed analysis of interesting situations on the pitch to players, analysts or even spectators. We investigate how to best design, implement and combine these components into a Instant Replay Analytical Subsystem for the Bagadus project to create an automated way of viewing and controlling video based on annotations. We describe how the system is optimized in terms of performance, to achieve real-time video control and drawing; scalability, by minimizing network data and memory usage; and usability, through a user-tested interface optimized for accuracy and speed for notational data capture, as well as user customization based on roles and easy filtering of annotations. The system has been tested and adapted through real life scenarios at Alfheim Stadium for Tromsø Idrettslag (TIL) and at Ullevaal Stadion for the Norway national football team.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I want to thank my supervisor Pål Halvorsen for great discussions, feedback, guidance and enthusiasm. I particularly want to show my gratitude to Asgeir Mortensen for his cooperation and help with on-site configurations of the prototype. Additionally I want to acknowledge the great work that has been done in the Bagadus project by other students. I also want to thank my co-author, Anders Emil Rønning for great teamwork, arguments that led to great results, and for constantly drumming on his desk, while sitting right next to me.

Finally, I wish to thank my friends and family, especially my beautiful girlfriend Ida, for words of encouragement and wisdom, as well as delicious meals.

Oslo, May 16, 2015
Jon Hammeren Nilsson

I would like to thank my supervisor Pål Halvorsen for his great guidance, feedback, advice and enthusiasm. I wish to thank my co-author, Jon Hammeren Nilsson, for great teamwork, arguments that led to great results, and for tolerating my constant finger drumming on all desks. In addition, I want to show my gratitude to Asgeir Mortensen for his cooperation and help with on-site configurations of the prototype.

Finally, I would like to thank my friends, family and my lovely girlfriend, Helene, for the constant support.

Oslo, May 16, 2015
Anders Emil Rønning

Chapter 1

Introduction

1.1 Background

In sports like soccer, the difference between winning and loosing is often dependent on small margins. Clubs and teams will do anything to gain the extra edge that can put them in an favorable position. This includes the increasing use of technology, for instance, the head coach for the Norway national soccer team Per-Mathias Høgmo, said “It’s all about winning 1-2 percent and gain a competitive advantage” [1]. Høgmo calls this the "2%-factor", stating that technology is not everything, but it can constitute the marginal difference between two equally strong opponents [2]. When watching a soccer match one can often observe the coaches standing on the sideline taking notes with pen and paper and then looking at their watch to note the time. This is to prepare for the half-time speech. Chelsea’s manager Jose Mourinho explains in an interview with official Chelsea magazine why: “I read the first half, I take my notes, I prepare my interventions at half-time based on the notes and where I feel I can help my players. So, I make the notes in order to be ready to make my changes.” [3]

Professional sports clubs worldwide are expanding their tool set with analytic and video capturing systems - systems used for both match analysis as well as player development [4]. Commercial systems include Interplay-sports, ProZone, STATS SportVY Tracking Technology and Kiswe, which all provide various analytic platforms typically by either analyzing player patterns in video-streams or by using global positioning and radio based systems. However, none of them provide a fully integrated system with support for video, tracking and notational analysis without significant manual work. Therefore, the Bagadus project was conceived [5].

Bagadus is a system being developed in collaboration between University of Oslo (UiO), University of Tromsø (UiT) and Simula Research Laboratory, which aims to fully integrate existing systems and enable real-time presentation of sport events. This is done by using a camera array capture system together with the ZXY Sports Tracking system [6] and and a

system for expert-captured annotations[1]. Earlier work [5] presents a prototype installed at Alfheim Stadium (TIL). Another prototype is currently being set up at Ullevaal Stadium - the home ground of the Norway national football team and Vålerenga Fotball.

Bagadus enables analysts to locate situations in a game that would otherwise require a lot of manual work by video seeking and editing. Manual annotations allow personnel to register interesting events during training sessions as well as matches. The annotations are registered in *hindsight* - the personnel "create" an event right after the situation they want to persist happens, and the system immediately generates video footage starting typically 15 seconds before and up to the timestamp of the registration. This allows the saved annotations to be available for replay almost instantly after registration. Possible use cases for this would be video analysis in the locker room during the half-time break, summary of the match at full-time and instant playback at stadium big screens during training sessions. Post-game analysis and construction of playlists for players', personnel's or even fans' consumption are other possible applications.

The first subsystem for expert annotations is called Muithu, a lightweight mobile system for notational analysis [7]. It was originally implemented as a stand-alone platform before the initiation of the Bagadus system, and therefore lacks integration with it. The Muithu system is based around a set of portable GoPro cameras for capturing video, an on-site computer for transferring video from the cameras, a Windows mobile application for annotating events, and a cloud service for permanent storage of video snippets tied to these events. The process of using the Muithu system requires a lot of tedious, manual work with setting up the cameras, syncing the clock between the cameras and the app, and individually uploading video from each camera after a session. A full integration is required to reduce the manual work.

1.2 Problem Definition

A quick and non-disturbing way of capturing notational data is vital during high-intensity sports sessions like soccer matches. Spending too much time in the process of registering such data might distract the analyst from the ongoing game, leading to situations where notational data should have been captured. Muithu provides functionality for rapid data capture in an intuitive manner. However, the process of combining these annotations with video requires significant manual labor. Video is not available until recording has stopped and the cameras have been manually synced to persistent storage, restricting analytical use to post-session.

Furthermore, the Bagadus system provides high-quality panoramic video in real-time,

[1] *annotations*, *notational data* and *registered events* are interchangeable terms throughout this thesis.

which is great for combining with notational data. Coupling registered annotations from an mobile application with video from the Bagadus system for near-instant playback will allow analysts to evaluate elapsed situations on the pitch immediately or shortly after they have happened.

We will examine the possibilities of integrating and expanding the annotation principles of Muithu with live panoramic video and virtual views from Bagadus. In this thesis, we will design, implement, test and evaluate a prototype of an instant analytical subsystem for Bagadus, centered around the Bagadus App and based on the principles of Muithu. User testing and deployment will be performed in a real life scenario for TIL at Alfheim Stadion and for the Norway national team at Ullevaal Stadion, where the intended end-users are able to interact with it. We wish to remove the manual labor associated with the Muithu system and add functionality for camera selection, zoom and on-frame drawing when using the Bagadus App for playback. In addition we want to extend the information captured when an annotation is made, by associating it with a user role. To achieve this, we must investigate if extending the Muithu App with additional functionality and integrate it with the Bagadus system is plausible, or if a completely new application should be developed. Requirements to platform, user interaction, responsiveness, requested functionality from potential end-users, as well as any necessary supporting components to the application must be considered.

It is essential that the system must be intuitive and easy to use, as intended end-users are sports personnel and not IT experts. To make the system responsive, light-weight and scalable, we aim to minimize the data and memory use while supporting multiple mobile platforms. We investigate how we can optimize the speed and accuracy in use of the application, while still offer a intuitive interface that can enable real-time interaction with the rest of the Bagadus system.

1.3 Limitations

The thesis will not provide a detailed background description of the complete Bagadus system, as this has thoroughly been done in [5].

In the research done around building the first version of Muithu, it was discovered that using the app was 15% faster than the traditional pen and paper method. Therefore, we do not investigate the problem on pen and paper versus a mobile application, but use this as the standard and try to improve on it.

Because of the long distance to where the prototype is installed at Alfheim Stadion in Tromsø, as well as a bureaucratic delays in regards to setting up the system at Ullevaal, we have been forced to implement parts of the system locally and developed mocks-ups of

inaccessible parts of the system during development and for certain demos.

1.4 Research Method

The research method used in this thesis follow the *design paradigm* described in *Computing As a Discipline* [8]. We design and implement a prototype of the system which is tested and evaluated in real life scenarios. The implementation is done in a iterative and agile manner to allow for new features and requests to be implemented after demos and discussions with the end-users of the system. Results from experiments and user surveys will be the basis for our design choices.

The prototype is tested by the end-users. This provided us with the ability to confirm the choices we made, and to make adjustments to better fit the need of the users.

1.5 Main Contributions

In this thesis, we design and implement a new system for instant review and analysis of soccer sessions and integrated it with the existing Bagadus project.

First, we present the design of an instant analytical subsystem and its functional requirements. The requirements are inspired by Muithu, requests from the staffs at the national Norwegian soccer team and TIL as end-users of the system, as well as the available functionality of the existing Bagadus system.

Next, we construct a new mobile application, the Bagadus App, which addresses the limitations of Muithu, and fulfills the functional requirements of the instant analytical subsystem. Through user testing and surveys, we show that a drag-and-drop based interface for the application is slower and more prove to errors than a click-based interface, which also is the user-preferred interface of the two.

Finally, we implement a playback system supporting the functionality of the Bagadus App to achieve a complete implementation of the instant analytical subsystem. The system will allow users of the app to control playback of the captured video as well as provide user drawings superimposed on the video, near-instant after registering the notational data. The video playback is available through a simple and universal web interface.

To evaluate the system, we have performed various experiments. We compare the WebSocket and HTTP communication protocols, and conduct experiments to find the most fitting protocol dependent on latency and data exchange frequency. In addition, we present and evaluate various approaches for mobile application development.

1.6 Outline

The rest of this thesis is structured in the following way:

Chapter 2 - The Bagadus System We introduce and briefly describe the existing components of the Bagadus system, before we address the limitations of the system and explain the motivation behind our work.

Chapter 3 Design of Instant Replay Analytical Subsystem We propose a new analytical real-time subsystem for Bagadus. We present a design based on the limitations introduced in the previous chapter, as well as features suggested by potential end-users.

Chapter 4 - The Bagadus App We introduce and present the design and implementation of the Bagadus App as part of the Instant Replay Analytical Subsystem, and display how this solves the limitations of the Muithu. We discuss and evaluate different graphical user interfaces and communications types for the app through tests and user feedback.

Chapter 5 - The Playback System This chapter details the design and implementation of Playback System which connects the Bagadus App with the Bagadus video subsystem. We discuss the requirements of the system, existing components to use and new components to design and implement.

Chapter 6 - Conclusion Finally, we conclude the thesis and summarize on our findings as well as discuss future work.

Chapter 2

The Bagadus System

The current systems for soccer analysis require some manual work for integrating the different analysis techniques and systems into a complete solution. For instance, in the Interplay-sports system [9], video footage must be analyzed and annotated manually through a desktop application. ProZone [10] is another application, which automates some of the manual video-analysis, specifically in the regards of evaluating movement patterns, giving statistics about speed, velocity and player positions. Kiswe [11][1] provides a great video system, but lacks the analytical tools. The common denominator of these systems is that they, to our knowledge, lack an integration of video, positional and manual notational data into a complete analytical system. None of these systems have dedicated mobile applications for registering notational data, making it harder for use by coaches on the pitch.

Bagadus [5, 12–15] was therefore built to be a sports analysis system that integrates existing systems and provides real-time representations of sports events, automating the process of combining the different subsystems. The purpose is to aid players and coaches in analyzing situations on the pitch during the match, during half-time, post-game or at training sessions, and minimize the amount of work required to do so.

The Bagadus system has gone through several iterations. In this chapter, we will describe the latest version of the Bagadus system in its entirety and discuss motivation and improvements for the analytics subsystem. We will not go into great detail about the tracking or the video subsystems, as this has been thoroughly done in [5, 12–15] and is generally out of scope of this thesis.

The Bagadus system consists of three subsystems - *video*, *tracking (sensor)* and *analytics*, where the latter will be the main focus of this thesis. Figure 2.1 shows an brief overview and interaction between the different components in the system.

[1] Previously Camargus.

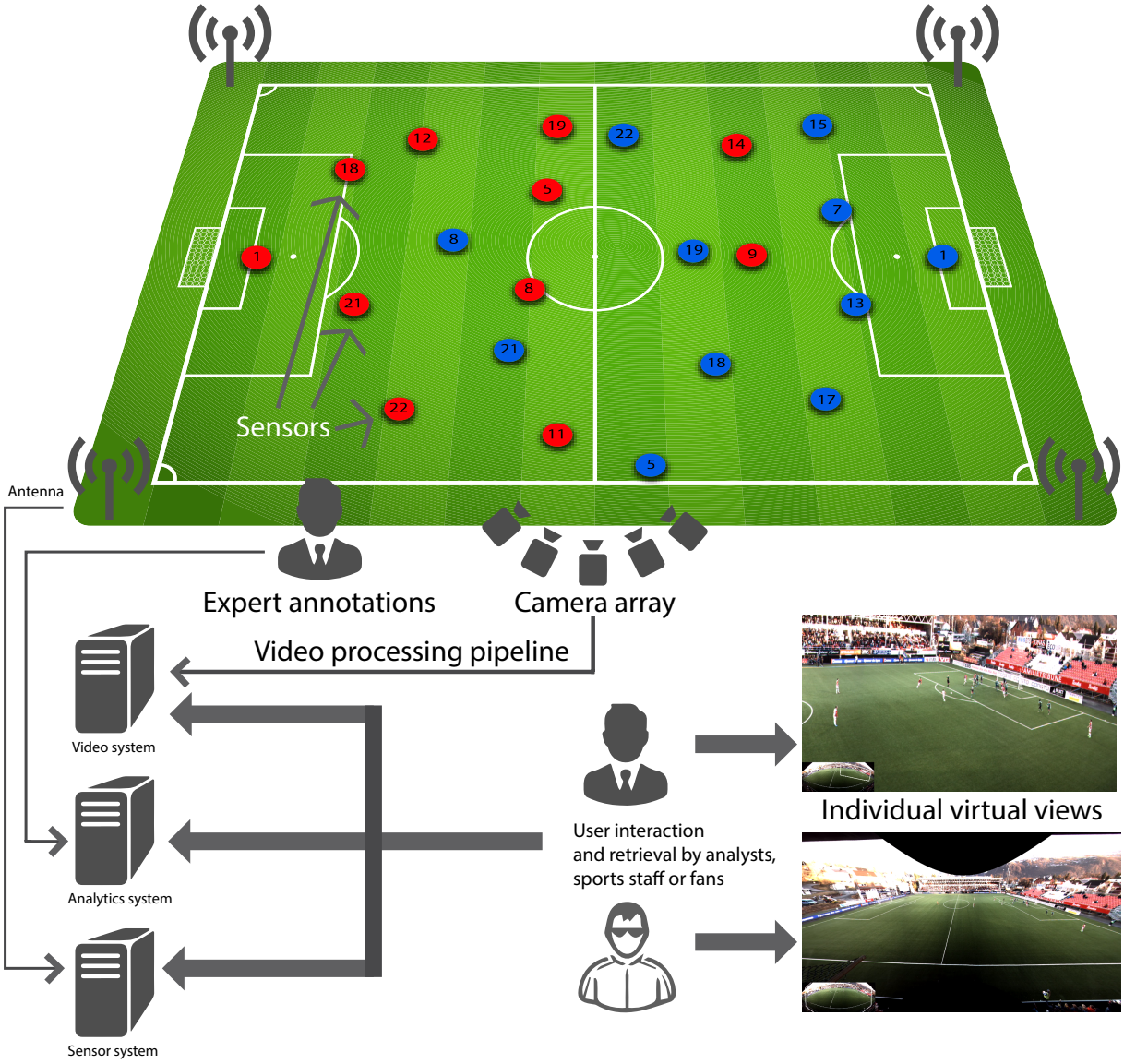


Figure 2.1: Overall Bagadus architecture

2.1 Video subsystem

The video subsystem is centered around a pipeline that handles all the steps between capturing video and producing output. The original pipeline [5] supported two separate methods of producing video: from each camera individually and stitching together the camera streams into panoramic video covering the entire pitch. The latest version has removed the individual camera video outputs in favor of a more sophisticated virtual camera viewer which allows for creating *virtual views* - a type of view created by zooming and panning into the panoramic video, acting like a cameraman with a traditional camera would pan and zoom during a match [16]. The system supports both playing back video stored on disk as well as real-time video.

The whole pipeline is subdivided into modules for better separation, so that they can be executed on different machines. This distributed computing approach allows the pipeline to provide video in real-time, as well as enabling easy scaling of the system in the future, e.g., by adding additional cameras [17].

2.1.1 Video capturing

The camera setup consists of an array of five industrial Basler acA2000-50gc cameras capable of delivering 50 frames per second (FPS) at a maximum resolution of 2048×1088 pixels [18] arranged in a pattern so that the cameras combined cover the full pitch. Two recording machines read the camera streams frame by frame, insert a Unix timestamp in the frames' headers, and transfer the frames to the single processing machine over PCIe using dedicated Dolphin host adapters. Transfers of these raw video frames require a great amount of bandwidth. Hence, ordinary Gigabit Ethernet is insufficient. The timestamps across the machines are synchronized to the same NTP clock synchronization server, as this is crucial for frame synchronization and later video extraction from the panorama [17]. In order to ensure a smooth transition between stitched frames, each source frame must be captured at the same time. This is done by using an Arduino device, which sends a trigger signal to each camera to signal when to grab a frame. In an outdoor setting, lighting conditions can shift quickly. The cameras can adapt to this by automatically altering the exposure settings, but each camera will then adjust its own exposure leading to different lighting levels between each seam in the stitched panorama video. A static exposure setting is supported, but not acceptable due to possible change in lighting during a recording session [17]. After exploring different exposure setting possibilities in [19], the pipeline now uses the approach of having one pilot camera control the exposure setting of all cameras. In order to tackle variable lighting conditions in the same frame, e.g., when the sun is setting and the stadium itself casts a shadow over the pitch, a high-dynamic-range imaging (HDR) mode is also supported. This is achieved by capturing alternating frames with high and low exposure, and then combining them at a separate processing machine. The rest of the pipeline will then run at half the frame rate, i.e., 25 FPS, unaware of whether HDR is enabled or not [17]. Figure 2.2 shows the distributed pipeline and data flow between modules and machines.

2.1.2 Processing of captured video

The processing machine handles all the remaining modules in the pipeline. Transferred frames from the recording machines are fed into a Frame Synchronization module. As the frames may be asynchronously captured, this step is necessary for joining the video frames

into full sets of frames. Timestamps inserted in the header of each frame by the recording machine are used to determine which frames belong in a set together. If an individual frame in a set is missing, the module will drop the current frame set and duplicate the previous set to assure that the system stays real-time.

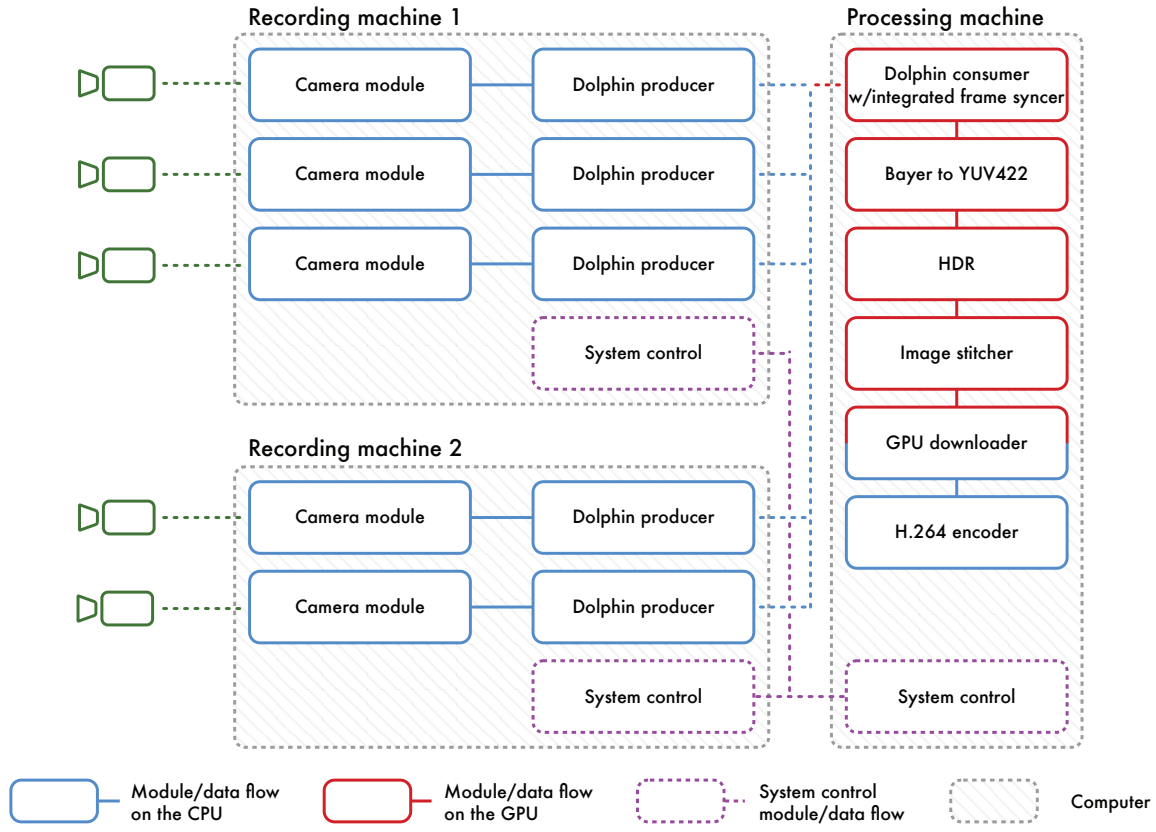


Figure 2.2: Distributed recording pipeline [20]

To maximize performance and stay within the real-time limit, the Bayer Converter, HDR and Sticher modules are all processed on the GPU. The nVidia CUDA parallell computing platform [21] is used to allow for easier programming on the GPU. Two separate modules, the CUDA Uploader and Downloader modules, are responsible for transferring video frames from CPU memory to GPU memory and vice versa after frame processing is done. Earlier publications [5, 12, 14] on the Bagadus system have shown the tremendous advantage in processing efficiency by offloading compute-intensive parts of the pipeline to a dedicated GPU.

As the cameras capture frames in the Bayer GR-8 format, the Bayer Converter module is required for converting each input image from the Bayer pixel format into YUV444 interlaced [17, 22]. Given the incompleteness of the raw Bayer pixel format, each pixel only records either red, green or blue, every image needs to be expanded into full RGB channels through the process of *demosaicing* before converting to YUV444 - the color space used in the rest of the pipeline.

The HDR module is situated between the Bayer Converter and the Stitcher modules and combines two frames with contrasting exposure levels into a single HDR image, if HDR mode is enabled. The preferred solution to creating HDR images is usually to utilize multiple images with varying exposure levels [17]. However, this is not feasible due to the maximum frame rate of 50 FPS from the cameras, as every additional frame to be used for HDR reproducing would halve the number of frames per second. In addition, using multiple frames for HDR could introduce ghosting artifacts to the image, as players may have moved noticeably between the captured frames. As such, the module uses two frames per HDR frame, producing a video with 25 FPS when it is enabled.

The main bulk of processing happens in the Stitcher module, which follows next in the pipeline. The images in a frame set are rotated, interpolated and projected onto a cylindrical surface, which is necessary for the virtual viewer system. The Stitcher has undergone significant changes since the inception of the Bagadus system. Originally, the individual images were joined with a static seam cut. However, this approach was prone to visual artifacts along the seam, especially when players cross over it. Therefore, a dynamic seam stitching process was investigated [23]. By using an implementation of Dijkstra's algorithm, the shortest seam route without cutting through any players is found. The algorithm is run for every frame, although implemented such as to avoid the seam taking a redundant route [17]. The slight color variations left are then smoothed over by a modest seam blending, improving the visual quality as shown in figure 2.3.

(a) Stitch without blending **(b)** Stitch with blending

Figure 2.3: Comparison of the static and dynamic stitchers

2.1.3 Encoding and exporting video

The final module in the pipeline is the X264 encoder. After the stitching process, the raw, planar YUV422 panorama frame is transferred back to CPU memory, as the encoder runs exclusively on the CPU. Encoding of the raw video is mandatory for persistent storing and distribution over a network. The video dimensions are downsampled according to the lookup table in the stitching module to fit 4K video, i.e., a maximum, horizontal resolution of 4096 pixels, as anything higher than this is not usually supported by the most common media players. This also has the effect of a quicker encoding speed and smaller file size [17].

Due to the nature of the Hypertext Transfer Protocol (HTTP) protocol, streaming frame by frame is generally not feasible. For that reason, common streaming protocols like HLS [24] and MPEG-DASH [25] divide video into small file segments, each segment containing a small sequence of playback time. Bagadus therefore splits video into three seconds clips with an associated live manifest to the current recording, allowing for live HTTP streaming. Each file is named accordingly to the date, the timestamp from the recording machines as well as a sequence number.

The substantial amount of static imagery, as well as typically minimal movement compared to the complete picture, makes the H264 encoder efficient at reducing the output file size by using motion estimation. The encoder finds fitting motion vectors and utilizes these to reuse pixel information from previous frames instead of saving the complete image data for each frame. Keyframes, frames that contain the complete image data, are used once at the start of each three second clip. This is to make video playback available as rapidly as possible after the client starts streaming, as the decoder needs to find a keyframe to start.

The encoder is the least predictable module in pipeline, in terms of staying within real-time constraints [17]. Scenery changes can greatly affect the processing time, as the encoder may have a difficulty finding fitting motion vectors in cases of snow, rain and other effects that may ruin the static imagery.

2.1.4 The Virtual Viewer

In section 2.1, we briefly introduced the concept *virtual view*. Virtual view describes the term of panning and zooming into cylindrical panoramic video (see figure 2.4) generated from stitching individual camera streams together. By correcting the perspective in real-time, the system provides a better and more natural zoom and panning compared to using plain crop [16].

Bagadus supports these views, allowing the user to follow a player or a group of players by interactively controlling his or her own virtual camera as presented in [16]. Unlike a conventional single camera stream, the virtual video view allows the user to focus the

video wherever he or she wishes, creating independent, dynamic video streams according to each user's preference. Through a web interface, the user can pan and zoom in the virtual view using keyboard buttons, the mouse or a gamepad. The client transmits normalized coordinates back to the virtual view system, which then updates the stream. This user-controlled stream can then be streamed to thousands of watchers through the same server. An example of such a view with the associated panorama picture can be viewed in figure 2.4.

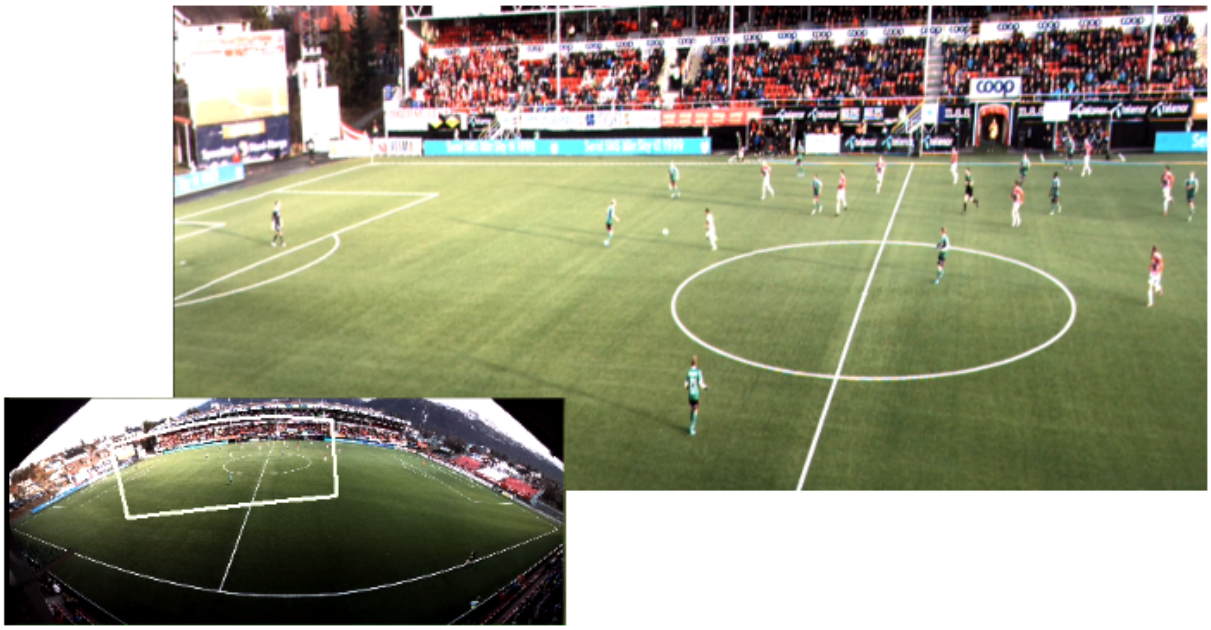


Figure 2.4: A virtual view with a thumbnail of the cylindrical panorama it has been extracted from. Note that the view is not a simple crop of the panorama.

Having continuous, complete control over the camera view may be excessive to a great deal of users. The interesting field of view is usually around the ball. Automatic virtual views where the virtual camera follows the ball, a player or a group of players have been investigated in [17], but not yet integrated.

2.2 Tracking subsystem

Bagadus uses the ZXY Sports Tracking system for tracking players' positions on the field. Players are equipped with sensor belts which transmit signals to radio antennas located around the playing field, capturing their position, foot frequency, heart rate etc. The positional sensor data is used by the video subsystem to map players unto the image pixels in the video, enabling the system to track players with an accuracy of about one meter [5]. The captured positional data can also be used to enable the video subsystem to generate the virtual views described in section 2.1.4. Further, users can utilize the positional data to query for video, as further described under section 2.2.1.

2.2.1 Automatic event extraction

The Bagadus system has support for integration with the ZXY Sports Tracking system. The tracking system is currently only installed at Alfheim Stadion (TIL) and not at Ullevaal Stadion. It enables automatic extraction of video sequences in offline mode by combining the positional data from the ZXY database with recorded video from the Bagadus video subsystem [13]. An example of a query against the database is get "all events where defender X is in the opposing team's penalty area during the first half of the game". The system will then return video sequences of all matching situations, thus automating the time-consuming process of manually scanning through hours of video footage.

2.3 Analytic subsystem

The analytic subsystem is the part of Bagadus designed to let expert users analyze situation using the video and sensor data from the video and tracking subsystems. Traditionally, coaches have used pen and paper to make notes, either during the game or by manually analyzing hours of video, thus requiring a significant amount of manual work [5]. To capture video sequences with metadata, the separate system Muithu [7] - excluding its video system - was decided to be integrated and extended with Bagadus.

2.3.1 Muithu

Muithu is a sports notational analysis system [7], which integrates real-time coach notations with related video footage. It was developed by UiT in collaboration with the coach-team at TIL. By using a cellular phone, the coaches can capture notational data during a game or training session in a lightweight, non-intrusive way. These annotations are then used to extract video captured with a set of GoPro cameras located around the pitch. The video is used by the support team and players for post-game or post-session analysis.

Architecture and design

Muithu uses a six-tier layering scheme, as illustrated in figure 2.5, where a deployment is divided into three specific groups: the components on the sports pitch, the back end modules and the web-based front end interface [7].

Video capture

A set of GoPro HD HERO2 cameras are positioned around the stadium, covering the entire pitch from different angles. Each camera captures 170° wide angle footage at 1080p. The recording equipment must be placed and synced manually before each session, and

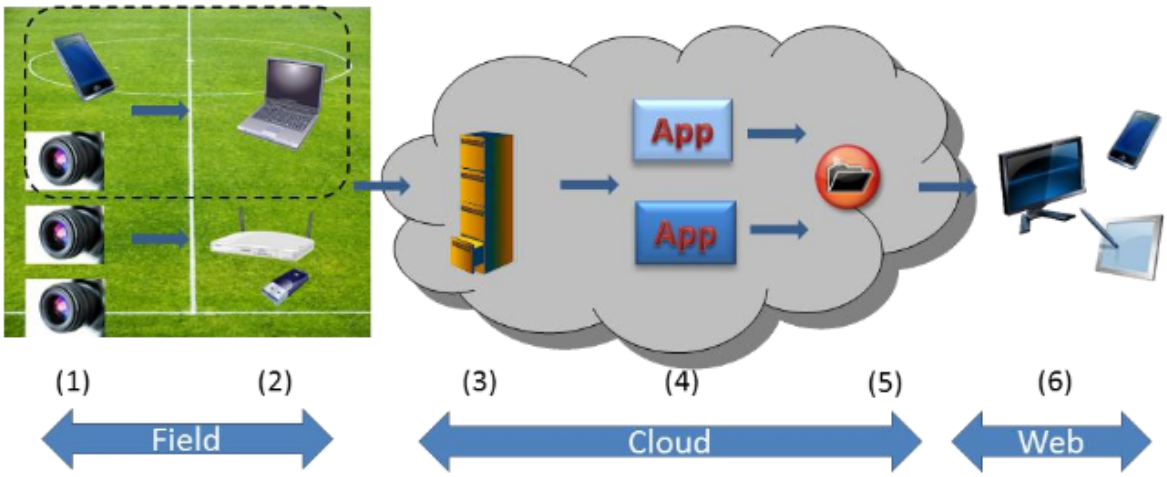


Figure 2.5: Muithu system architecture [7]

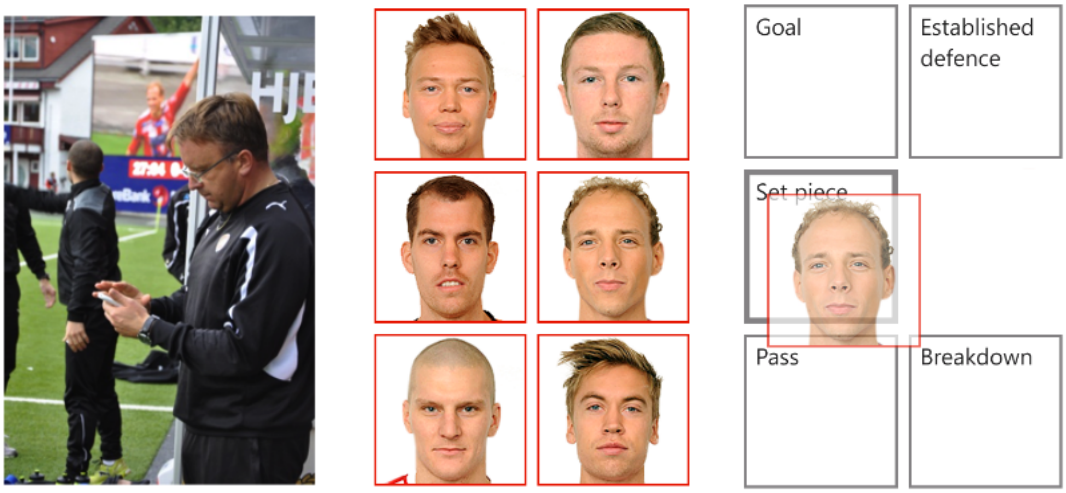
recording must be started individually on all cameras before annotations are ready to be captured, and therefore requires some manual work beforehand to get the system operational. As the GoPro HD HERO2 camera does not support streaming - although an accessory enabling it was released after Muithu was developed [26] - it is not possible to use video before post-session when the cameras have been rigged down and video data persisted to a permanent storage.

Notational process / mobile application

A specially designed app solely for the Windows Phone 7.5 platform was made for creating annotations. As the app had to have an intuitive and fast interface [7] to be usable during a match, input is structured in the form of tiles in a grid. Users provide input by dragging a player and dropping him unto a specific event type as shown in figure 2.6. The event types in the second layer appear when the drag action starts. An event is then persisted to a database as notational data of the player, event type and timestamp.

The notational data capturing in *hindsight* corresponds to the principle of evaluating a situation after it has happened. The non-linear nature of association football makes it challenging to predict an event beforehand. Therefore, the expert user evaluates and captures the situation in retrospect. The timestamp obtained at the annotation moment corresponds to a video sequence starting 15 seconds prior, and ending 1 second after, for a total length of 16 seconds. This was determined by the head coaches to be an appropriate length in [7]. However, the feedback we have received from both the support team at TIL and the national team stress that dynamic video sequence lengths are crucial for constructing an effective analytic report. The nature of event types differ, where for example in a counter-attack situation it may be beneficial to see the build up, which may not be completely

covered in 16 seconds. In other situations, such as a corner, a 16 second segment may be excessive. By being able to individually tune sequence lengths, a lot of uninteresting video can be pruned. A simple and sufficient method of doing this would be by allowing trimming and extending of the video when reviewing registered annotations. Muithu supports user-defined video sequence lengths by using a start and stop button in the app, but then the user would need to know when to start and stop recording, which can lead to missed situations and/or too much video. The Muithu on-premise interface used for extracting video sequences corresponding to registered notational data, does not support video editing - sequences are extracted from the full video either according to the aforementioned 16 seconds interval or intervals captured with the start and stop button.



(a) Event tagging. **(b)** Select player. **(c)** Drag to event.

Figure 2.6: Operation of Muithu during a game (a). Select a player (b) and drag the image tile to the appropriate event type (c) to register an event. Taken from [5].

The application lacks multi-user functions. Each installation is effectively a user, but the notational data does not persist any user information. Thus, there is no way of telling which individual registered an event if such information would be pertinent when analyzing the data post-session. Event types relevant to the head coach may not be interesting for the team physiotherapist and vice versa. Event types can be customized by the user in the app, but if two or more users utilize the same device, though have different wishes in regards to event types and/or available players for annotating, the settings in the application would need to be customized every time the device switched users.

As the application is restricted to one platform, additional equipment in the form of mobile phones running the Windows Phone platform would likely need to be acquired, as it only has 2.9% of smartphone market share in Q3 2014 [27]. It is beneficial that the users can use their own device.

Back end

The mobile application connects to the Windows Azure-deployed cloud storage by a RESTful HTTP application programming interface (API), where annotations are uploaded as Extensible Markup Language (XML). Video has to be transferred from the cameras to a computer for uploading, either done via USB or by inserting the memory cards directly into the machine. The user is then presented, by using an on-premise interface, the registered events with associated video sequences from all cameras. The interface allows the user to select wanted events and discard unnecessary video, for example if a situation is sufficiently covered by one of the cameras. The selected events with related video are then persisted to the cloud database.

Social network

Muithu has a social network where coaches and players can share ideas. The video snippets chosen for persistent storage by the coaching staff post-session are available through this network. Two main user roles are defined as coaches and players, with a different set of views and tools. Both players and coaches can search for and view video sequences from all available angles. Coaches have, in addition, the possibility to tag videos with players as well as a predefined set of textual tags. A commenting system allows players and coaches to address each other to reflect over captured situations.

2.4 Summary

Muithu was developed independently of Bagadus, and as such, is not integrated with it. Bagadus has been using a rudimentary form of capturing expert annotations in real-time situations, restricted to using a simple web interface.

Muithu is an offline video capturing system, in contrast to Bagadus. Video footage has to be transferred manually from each camera to storage, where wanted video sequences are then extracted according to annotations registered with the mobile app. This workflow requires a lot of manual effort, and video data is not available for use and analysis before the camera setup has been rigged down, video transferred to a computer and then uploaded to a database. The offline nature of Muithu thus restricts use cases to analysis after a training or game session has ended.

As the Bagadus video subsystem supports live streaming and instant playback, notational data with corresponding video should be available for use as quickly as possible after capture. This allows the coaches to perform on-site video analysis and make strategic decisions based on the captured data while a session is still ongoing. The support team for TIL have voiced their requests to us about using video footage during half-time to analyze

situations from the first half of the game. The head-coach can give the players feedback on failed, successful and missed situations using the notational data with video as visual support. Another application could be to investigate the opposing team's strategy, such as pointing out a hole in their defense for the players to exploit. Real-time analysis during training sessions is another possibility. The head coach should be able to utilize the Bagadus system while drilling the players on the pitch to give immediate feedback supported by captured video, e.g., by instant playback on the stadium big screens. This can be done by allowing the coach to control a virtual view stream from his mobile phone; panning and zooming via an app, as well as controlling which events to show.

Bagadus in its current state is a fully functional video capturing system, but with no support for practical use by its intended end-users: the support staff for TIL and the Norway national football team. We have shown that the current considered analytics subsystem, Muithu, needs to be redesigned in order to be implemented into the Bagadus system. There is also a requirement for several new features to be developed to meet the needs of the sports clubs using the system.

Based on the manual labor required and the Muithu's limitations, we aim to design and implement the Bagadus App, including any additional components required, to support the functionality of the Muithu App and fully integrate the analytical subsystem with the real-time video subsystem of Bagadus enabling immediate playback of events. This is presented in the next chapter.

Chapter 3

Design of an Instant Replay Analytical Subsystem

In this chapter, we introduce and present an overview of a new analytical subsystem for Bagadus, focused on instant analysis, i.e., analysis during training or match sessions. This chapter presents the motivation, design and issues regarding communication between the components in the system.

3.1 Motivation

The Bagadus system has a state-of-the-art processing pipeline producing video in real-time. However, the intended analytical subsystem Muithu is not integrated with Bagadus, and is limited in usability, especially due to its offline nature. The manual work required to set up the cameras, synchronize camera clocks with a mobile device and manually transfer of video to persistent storage restricts usage of the system to post-session analysis. By lacking a proper, intuitive analytical subsystem, the user base of Bagadus is restricted to consumers with the strong technical proficiency needed to use the system. Without a mobile, non-intrusive way of registering notational data, expert users are limited to manual video analysis or event extraction based on positional data from the ZXY Sports Tracking system.

We want our system to provide expert users with a quick method of registering notational data, and being able to play back associated video of the event immediately to clients. We have aimed at situations where near real-time analysis is required, such as half-time breaks in matches and during training sessions.

3.2 Requirements

Through dialogue with potential end-users, such as soccer coaches and support personnel, inspiration from Muithu and other sports analysis systems, as well as implemented features in the Bagadus system ready to use, we composed the following list of requirements that the new analytical subsystem should fulfill:

3.2.1 Functional requirements

Capture notational data with video The system should provide a mobile, uncomplicated way of capturing notational data, using the Bagadus App.

Multi-user The system should be able to distinguish users and enable role-based settings and annotation.

Play back video based on notational data Notational data should provide associated video from the Bagadus pipeline. This video should be easily played back to clients through controlling playback from the Bagadus App.

Create playlists of registered notational data During the half-time break, time is critical. By filtering, in beforehand through the Bagadus App, the notational data from the first half down to a few events, the coaches can show the team the most crucial and interesting situations in the half-time analysis.

Manipulate video playback in real-time The Bagadus system provides a Virtual Viewer, as described in section 2.1.4, for generating and controlling individual, virtual views. By controlling virtual views through zooming and panning via the app, analytical use cases can be extended to, e.g., showing video on the stadium big screens during practice sessions, in addition to simplifying half-time analysis. Manipulating the source input should be possible, i.e., pausing, skipping, extending or trimming the length of the event, changing event etc.

Free-hand drawing on video Drawing is a common visual aid in analyzing video. By being able use the app to create drawings superimposed on video by free-hand, experts can provide a clearer analysis to players or other support team members during half-time breaks or practice sessions.

3.2.2 Non-functional requirements

Available The system should support multiple platforms and provide the same functionality independent of the operating system.

Low latency and memory The system should have low communication latency and minimize memory usage to provide a fast and active experience for the user.

Responsive The system should be adaptive and scalable to different screen sizes and resolutions.

Non-intrusive and intuitive The graphical user interface (GUI) should be intuitive and facilitate an efficient and accurate user experience without interrupting normal tasks, e.g., when managing the team from the touch-line during a match.

3.2.3 System design

The real-time analytical subsystem is centered around the Bagadus App. In order to meet the requirements for the system, we have also implemented a supporting system dubbed the *Playback System*. It utilizes existing components of the Bagadus system, such as the Virtual Viewer. This provides the app with necessary back-end functionality, as well as serving clients with a web application for viewing video streams controlled by the app.

The following is a short description of the design of the subsystem:

Bagadus App A mobile application for expert users and other support personnel to register annotations during sport sessions, with functionality for controlling, manipulating and drawing on the video associated with the annotations via the Playback System.

Playback System A system for providing users of the Bagadus App with a way to control and stream video connected to registered annotations to other devices, such as a TV monitor in the dressing room. The users of the Bagadus App can use the app as a remote controller for controlling video on the Playback System and draw on paused video frames via the app.

Figure 3.1 shows a simple overview of the Instant Replay Analytical Subsystem. Expert analysts use the Bagadus App to register notational data. Through the Playback System, analysts, players, other sports personnel or even fans can view video from the Bagadus system controlled through the app. Implementation and further details of the Bagadus App and the Playback System are covered in chapter 4 and chapter 5, respectively.

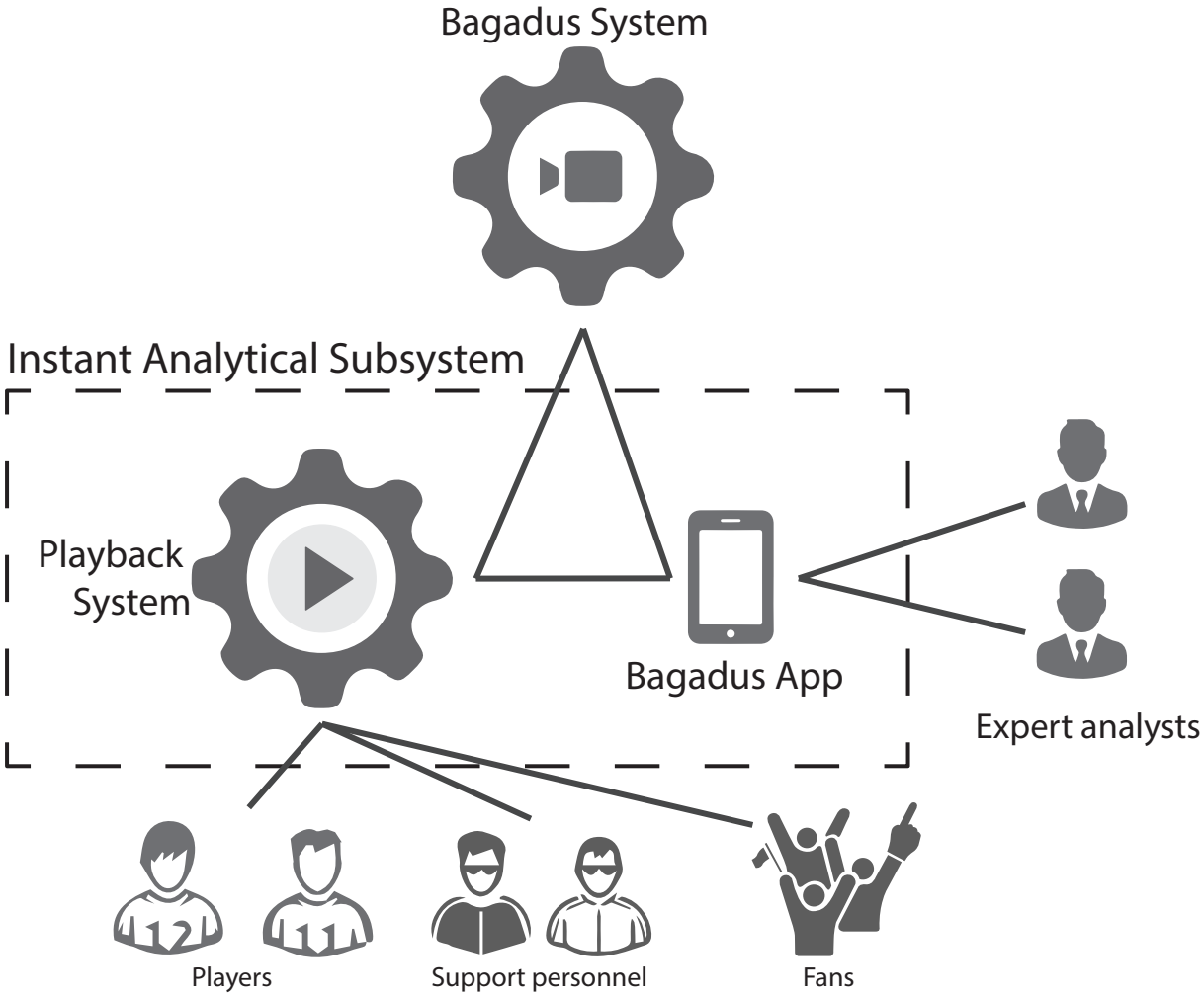


Figure 3.1: Overview of the Instant Replay Analytical Subsystem

3.3 Communication

A critical element in the system is how to handle communication between the components. A lot of the functionality will be relatively useless without a near real-time latency, e.g., manipulating video playback. In this section, we discuss different possible high-level protocols and techniques for use in the Instant Replay Analytical Subsystem.

Traditional communications over the HTTP consist of a client sending a request and a host acknowledging the request by sending a response back before closing the connection. For each request/response pair, a Transmission Control Protocol (TCP) 3-Way-Handshake must be negotiated, making a simple HTTP connection inefficient and prone to increased latency where many requests are sent in short period of time. Several mechanisms have been introduced to the HTTP protocol to reuse a connection for more than one request, such as keep-alive and long-polling.

As illustrated in 3.2b, HTTP keep-alive is a mechanism for keeping an TCP connection

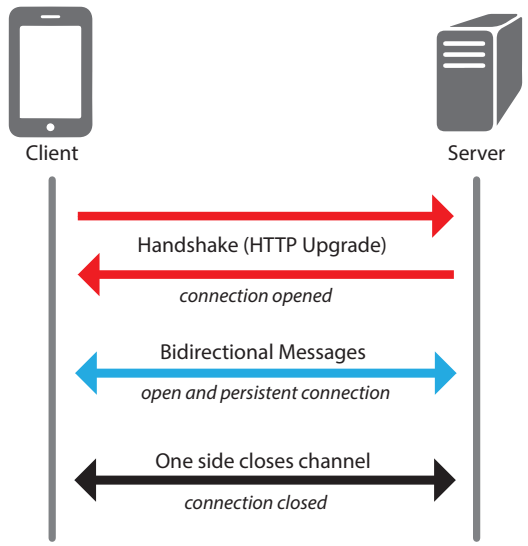
open, but idle. This can speed up the process of making new requests by reusing the same connection, but every request is still independent. It is enabled by default in HTTP 1.1 [28].

Long polling is a technique for leaving the request active for a period of time to see if any new data is available, eliminating the need to send multiple requests to get updated data. By keeping a request active, this techniques allows for emulating a push mechanism from the server to the client. If no new information is available, the host sends a response to terminate the connection. The server must repeatedly check if there is new information available, which can be expensive regarding network and CPU usage.

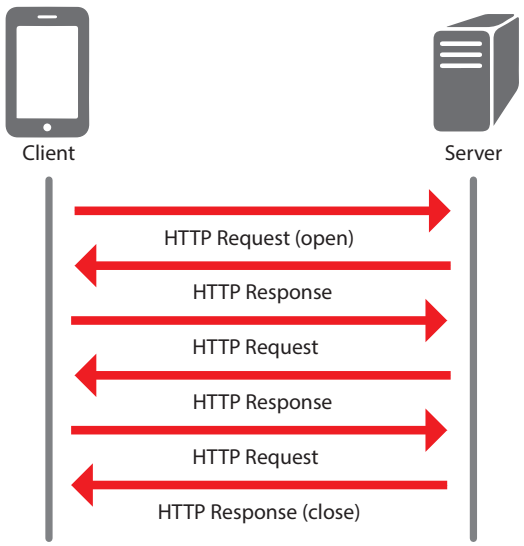
WebSocket is an application layer protocol that allows a two-way communication between a client and remote host in full duplex; communication in both directions over a single TCP connection simultaneously, as illustrated in figure 3.2a. WebSocket is a TCP-based protocol with no ties to HTTP other than the initial handshake, that is interpreted as an Upgrade request by web servers [29].

WebSocket offers a technology that is useful for real-time applications where bidirectional communication is needed, such as games and chat rooms. It can offer a great discount in the latency and amount of data by reducing the overhead of the HTTP headers needed in every HTTP message [30].

Figure 3.2b illustrates the differences between HTTP and WebSocket. Traditional HTTP establishes and closes a new TCP connection for every message, while WebSocket only does this once, reducing the latency for new messages [31]. WebSocket allows for a more TCP socket-like type of communication. When HTTP requests are made rapidly, HTTP keep-alive reuses the same connection.



(a) Flow of a WebSocket connection. Multiple messages are possible over a persistent connection.



(b) Flow of HTTP messages. Connection kept alive for some time.

Figure 3.2: WebSocket and HTTP protocols

Based on the functional requirements of the analytical subsystem, we have identified two situations where low-latency communication is crucial: controlling video playback and drawing on video - both actions that are done from the Bagadus App and emitted through the Playback System to clients. When controlling video, it is desirable that requests reach clients as quickly as possible. If pausing playback takes more than a few milliseconds to happen, the result could be widely inaccurate, e.g., if the video pauses after a goal when the coach wanted playback to pause right before said goal. Low latency is not as crucial for drawing on video as for controlling playback, but still necessary for a responsive experience. In addition, continuous drawing requires sending tens, if not hundreds, of messages with coordinates if the drawing should update contiuously at the client. An alternative way would be to send the drawing in chunks or wait until the drawing is complete. However, using a WebSocket implementation, continuous drawing to clients should be trivial, due to the constant connection. This is further investigated in section 4.5.

3.4 Summary

In this chapter, we have presented an overview of the new, real-time analytical subsystem of Bagadus. We began the chapter by discussing the motivation behind the system, outlining the missing aspects of the current analytical subsystem, Muithu. We then presented a list of functional requirements for the system based on features of Muithu and additional requests from potential end-users. Next, we described the overall design of the system based on these requirements. Lastly, we discussed the aspect of communication between the components in the system and presented possible protocols to use. In the next chapter, we introduce the Bagadus App - the main component of the real-time analysis subsystem - and its design and implementation.

Chapter 4

The Bagadus App

4.1 Motivation

In chapter 2, we discussed the features and limitations of the Muithu system, and the need for an updated analytical subsystem for the intended end-users to be able to use the Bagadus system for notational analysis. Further development and adaptation of the Muithu app would be a possibility. However, given the restrictions to one platform, as well as the simplistic nature of the app, we concluded that designing and developing a completely new app would be the most efficient solution. In accord with the requirements of the system listed in section 3.2, through evaluation of the Muithu app, as well as discussing new requirements and functionality with potential users, we propose the following set of criteria for the new app:

Notations Users should be able to capture notations with minimal effort. The annotation should have information about the user assosiated with it, such as roles. Captured notations from the same session should be available for view in the app.

User specific The app should be able to distinguishes users, and the users should be able to customize the default settings, and user roles.

Control video Playback of video clips on other clients should be controlled via the mobile device. Users should be able to filter out irrelevant events in the app before playing them. Users should be able to manipulate event video sequences by adjusting the length of the event. The camera stream's field of view should also be possible to adjust from the app to center on areas of interest in the video.

Drawing as a visual aid When controlling video through the app, drawing on the video should be possible for analytical purposes.

Multi-platform The app should be available for a majority of mobile devices by supporting as many platforms as feasible.

Portability The app should be portable, i.e. it should adapt to network changes, caching events if instant upload is not possible.

Intuitive Soccer coaches are not IT professionals, and the app should take this into account. The GUI should be intuitive to use immediately without extended usage courses.

Precise Users must be able to annotate events as quick as possible and with great accuracy.

4.2 Mobile application development platforms

Mobile application (*app*) development has exploded over the last few years, correlating with the sharp increase in the number of smartphones sold [32]. The two biggest smartphone operating systems are Google's Android and Apple's iOS, with marked shares of 47.06% and 43.86% respectively as of September 2014 [27]. A common challenge, applicable for our use case, in mobile app development is deciding which platforms to support. The expected user group must be taken into account as well as development costs for the platform and future support. Fortunately, the major platforms provide similar functionality, e.g., interfaces towards GPU, camera, GPS etc., meeting the needs for most types of apps. The Muithu app exclusively supported Windows Phone - a platform with a marginal market share of 2.38% as shown in 4.1. We want a broad as possible support for the Bagadus App to minimize extra equipment needed to use the system. By supporting Android and iOS, our app can reach 90.92% of mobile and tablet users. Therefore we focus on these two platforms due their high market share and dominating position in the smartphone and tablet operating system market. As such, all references to *mobile platforms* in the following chapter will include only the iOS and Android platforms unless stated otherwise.

Both mobile platforms have their own development environment with restrictions to programming language, integrated development environment (IDE) and methods of app distribution. The iPhone Software development kit (SDK) is only available on the Intel-based Mac computers. A developer licence is also required. The Android SDK, on the other hand, is open source, based on Java, C and C++ and available on Linux, OS X and Windows.

There are various ways of approaching multi-platform support for mobile applications. Factors such as resources, time, features to support, app distribution and hardware required must be taken into account. Both mobile platforms evolve rapidly, leading to support for older devices being dropped and new features being supported. Ideally, there should be a

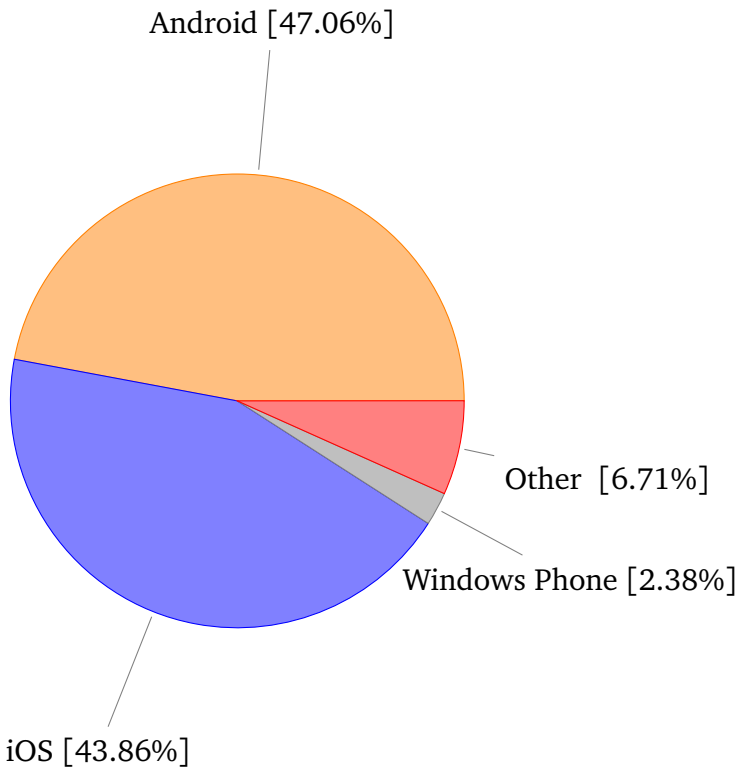


Figure 4.1: Market share of smartphone operating systems in September 2014 [27]

consistent experience for users independent of devices and platforms. However, this is not always possible, as the great deal of devices available - especially for the Android platform - differ in hardware features and software support. Compromises may have to be done in order to keep a broad device support and consistency. The mobile app environment today consist of three types of apps: *native apps*, *Web apps* and *hybrid apps*.

4.2.1 Native apps

A common practice is to develop native mobile applications - apps developed specifically to a certain hardware and operating system. The developer programs using the platform language and SDKs directly. This gives the advantage of complete access to the platform API, to common user interface elements leading to a more consistent user experience conforming to the platforms design principles[1], as well as generally better performance due to compiled languages and less abstraction, especially for resource demanding apps such as 3D games or image processors [34]. One disadvantage to the native app approach is the time and resources needed to develop the same app for different platforms. Native iOS apps can only be distributed through the App Store, which is further described in section 4.2.4.

A possibility is to port an app from one platform to another, e.g., Android to iOS. However, this essentially requires a complete re-development to a new programming

[1]Apple takes into account design principles when reviewing apps submitted to the App Store [33].

language and platform. Tools for converting from one programming language to another exist, such as Java to Objective-C [35]. These tools might be helpful in limited cases, but there are still a great deal of environment variables and platform-specific APIs to automate such a conversion process. The Java Native Interface (JNI) [36] provides a framework that allows native applications and libraries written in C and C++ to call Java code running in a Java Virtual Machine (JVM) and vice versa. In this regard, we could write a wrapper and provide an interface from iOS to the JNI functions through callbacks, but adapting the application to the platform framework would still be necessary. Another option is to write the base application in C/C++ and natively compile it for each platform to be supported, and then create interfaces between the application core and the different platform APIs. Third-party development platforms using this approach are further investigated under section 4.2.3.

4.2.2 Web apps

Web browsing is a central part in the smartphone experience. A mobile version of Google Chrome is available for both iOS and Android, running on forks of the same layout engine, the same JavaScript (V8) engine and supporting the same features as its equivalent desktop application. As such, all Web applications running on desktop computers are supported on these mobile platforms. A Web app is implemented as a Web site optimized for mobile applications, i.e., adopting a mobile first philosophy by taking into account screen size, touch gestures and network restrictions. However, the extra abstraction of running everything in the context of a Web browser, i.e., creating a user interface through a software rendering engine using interpreted languages like HTML, CSS and JavaScript instead of painting pixels directly through the GPU using the platform APIs in compiled code, makes non-trivial Web apps more resource-intensive than an equivalent native app [34]. Nonetheless, the constantly evolving browser ecosystem as well as rapidly improving hardware and software capabilities in smartphones are leading to Web rendering engines operating more efficiently on smartphones. For instance, the WebKit engine allows for hardware acceleration of CSS transforms, leading to smoother running animations and transitions between view states. The Blink rendering engine, a fork of the WebCore component of WebKit, used by the Chrome browser from version 28, and Android's built-in browser from Android 4.4 utilizes the concept of GPU accelerated composition for even better performance [37].

Apple initially embraced the concept of Web apps; Web apps were the only approach to developing apps for iPhone until the iOS SDK was released and the App Store launched in 2008 [38]. Tooling for such applications were provided through their Dashcode project. This project contained templates and libraries for making Web apps in nature with the iOS platform's native design. Between May and June 2013, as evident by the Wayback Machine

[39], Apple removed the Web apps directory - `http://apple.com/webapps` - from their website. It now redirects to `http://apple.com/iphone`, signaling their intention to halt supporting Web apps. This is also evident by the ostensibly discontinued Dashcode project, which has not been updated since mid-2012.

Web apps circumvent the distribution platforms completely, thus avoiding any approval process. On the other hand, a Web app has restricted access to the platforms APIs. Certain features such as access to the device microphone and geolocation are available through HTML5, but e.g., permanent storage is only available through HTML5 localStorage, which is deleted when clearing browser data. Thus, this type of storage is prone to accidental deletion. It is also limited to saving simple key-value pairs, so metadata without extra measures will be lost and any objects with behavior will have to be rebuilt when retrieving the data. Bagadus App should be able to save settings and cache annotations, so this is a possible issue. A constant Internet connection is also required, as the app is stored on a Web server.

4.2.3 Hybrid apps

Hybrid apps is another approach to cross-platform development. The term *hybrid app* usually includes two different approaches:

- Wrap a Web app inside a native container utilizing the device's embedded browser control for displaying content. A hybrid app framework like Apache Cordova [40] does this by compiling a native app consisting of a borderless browser view serving Web files from the device's storage. This circumvents the need for a Internet connection. If convincingly designed, i.e. not using default HTML/CSS graphics, a user should not be able to tell such an hybrid app apart from a native app. Cordova provides a plugin API to create interfaces between JavaScript and the native APIs. This allows developers access to far more mobile device features than a pure Web app, e.g., push notifications and device storage. Several frameworks, such as Ionic [41], jQuery Mobile [42] and Sensa Touch [43] help easen the process of designing Web technology based hybrid apps by providing mobile optimized visual components and touch optimization.

- Use a software development platform like Xamarin [44] to write an app in one language and compile it to native apps for each supported mobile platform. In Xamarin's case, this is achieved through several layers of abstraction, where the base app logic is shared among all platforms with individual platform-specific code on top. An advantage of this approach is native apps with less code-writing, but the developer must be vary to specific environment knowledge to avoid problems, such as data structures and generics working in different ways than with the native programming

languages and APIs [45, 46].

4.2.4 Distributing mobile apps

Android and iOS have their own platforms for distributing mobile apps to users: Google Play[2] and App Store[3] respectively. Both services have a screening process for applications: Google Play uses an automated system, Google Bouncer, which scans apps submitted to the store for malware and other issues related to malicious activity, while Apple's App Review, whose technical details have not been disclosed, employs mostly static analysis in combination with some manual analysis e.g. to insure the app meets design criteria etc.

4.2.5 Selecting platform

There are three mobile app environments today: native, web and hybrid. Table 4.1 shows some of the advantages and disadvantages of these different types of mobile applications.

Type	Advantages	Disadvantages
Web application	- Single codebase	- Limited access to device APIs - Needs a Internet connection
Hybid application	- Access to many of the device APIs - Single codebase	- Abstraction layers can prevent native-like performance
Native application	- Full access to device APIs - Optimal hardware utilization for better performance	- Platform specific codebase

Table 4.1: Advantages and disadvantages of different application types

With the Bagadus App, we want a broader support of devices than the Windows Phone-exclusive Muithu App. Developing multiple native apps would be too time-consuming to both develop, debug and update. Maintaining a consistent design between the platforms could also prove to be challenge as well, if using the native graphical interface elements of each platform. The extra processing power a native app provides is not necessary in the context of the Bagadus App because none of the planned features require really high processing power. Some features like drawing, that do require some power, is supported through the HTML5 canvas. Through a plugin interface, native API features such as hardware acceleration are accessible. If needed, this can speed up an app by running intensive tasks on the GPU or both CPU and GPU. Thus, the approach of converting a native app deemed too complex and unnecessary for the Bagadus App's criteria.

A pure Web app would need to support multiple browsers, or we would have to dictate the user's browser use, as there exists several third-party browsers for Android and iOS.

[2] https://play.google.com/store
[3] https://itunes.apple.com/us/app/apple-store/id375380948?mt=8

A hybrid app uses the platform's native browser, and thus minimizes the need for cross-browser testing. Figure 4.2 shows that the time spent surfing through a mobile browser is decreasing, giving the impression that mobile users prefer dedicated apps for Internet services. In [47], it was established that users find native or hybrid apps to be more responsive as they provide one-click access, unlike a Web app where a user at minimum must open a Web browser and click a bookmark or enter an URL.

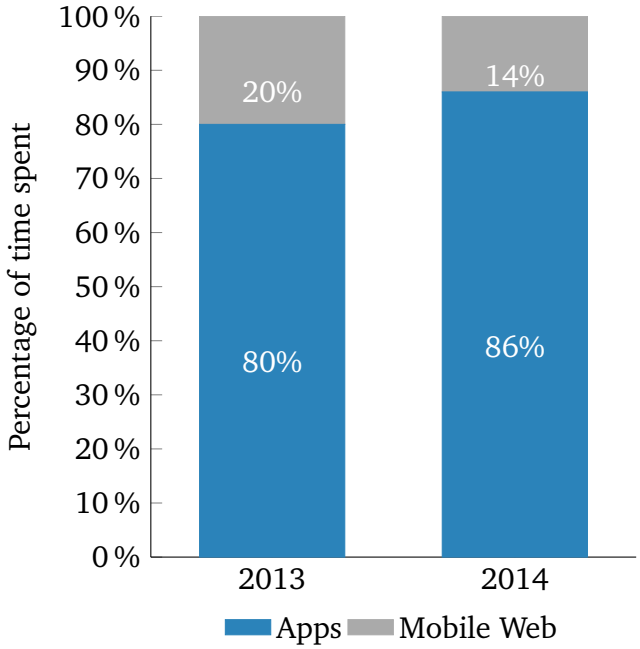


Figure 4.2: Percentage of time spent using apps and surfing in a mobile Web browser [48]

As such, we chose the hybrid approach using the Apache Cordova mobile development framework. By developing a single app that is supported across multiple platforms, we minimize the amount of redundant development and updating needed. In addition, the Bagadus App does not have resource-intensive requirements justifying the native approach. Using Xamarin could provide a more native-feeling app, but it is a commercial and non-free alternative. The development platform is also only available for Windows and OS X. By using free and/or open-sourced alternatives, we hope to encourage future development of the system.

The Bagadus App should be available through the respective app stores, so users can easily install the app. Only authenticated users can use the app, as it requires logging in. Cordova only provides means to package the Web page into an app as well as interfaces to platform APIs, giving no specific support to designing the app. Thus we decided to use the mobile Web framework Ionic. It is both a CSS framework and a JavaScript library for building an optimized mobile hybrid app, aiding in mundane tasks such as ensuring the user interface scales properly on devices with different display resolutions. The design structure of Ionic is similar to iOS, which is convenient for our app to pass Apple's App Review.

4.3 User interaction

A soccer match is usually highly intensive, giving coaches limited time to capture notational data. Therefore, we wanted an intuitive, responsive and quick user interface to minimize the time used in this process. We set a time limit for annotating an event to a very maximum of 5 seconds - the same as for the Muithu app [7]. While a notation is done either by clicking on a tile or holding the same tile down for more options in the Muithu, we saw this as a possible time-consuming element that could be improved.

4.3.1 Feel and responsiveness

A challenge we identified with hybrid apps is capturing the native look-and-feel of the different platforms, as well as performance issues compared to native apps [49].

When developing hybrid apps it is necessary to test the application on all the platforms you want to support, as they may behave variously as they use different browsers. This can be avoided by bundling a browser with the app and force it to be used across platforms, although this will increase the size of the app. Android 4.4 and newer versions use a default browser built upon Chromium - the same project Google Chrome is based on - iOS Safari is powered by the WebKit engine, while Google Chrome uses the Blink engine, which is a fork of the WebKit project [50–52]. This means that Google Chrome is a well-suited and free candidate to use in the development for hybrid apps.

As such, we used Google Chrome for the development of the Bagadus App to avoid having to compile and build it for all the platforms every time a change was made to see if any unexpected results were produced. The browser also provides the ability to emulate mobile devices by simulating touch-based events using Chrome Developer Tools. Therefore, we could focus on developing the app for one browser and reduce the need to constantly build and test it for different platforms. Although choosing to use hybrid technologies gave us the opportunity to support more platforms with the same code base, relying mainly on testing done in Google Chrome proved more difficult than first anticipated. This was particularly evident in small adjustments that had to be done in order to get the desired functionality and behaviour on the iOS platform. E.g., with incoming WebSocket messages including information that may take a moment to load, such as an image, the data may still be loading when the incoming-message-callback is invoked. If injecting the image data object into an HTML `<img>` tag, the Document Object Model (DOM) must be notified when the image is completely loaded. We found that in Chrome and on Android, the browser does this automatically. For iOS, we must manually check if the image is ready before injecting. As such, we discovered that hybrid apps cannot be considered "one-size-fits-all", just as a web developer must take into account supporting different browsers.

4.3.2 Accuracy in use

We wanted to make the GUI of the Bagadus App as similar to Muithu as possible, for a smooth and convenient transition for existing users of the analytics system at TIL. Therefore, a tile setup was decided to closely resemble the original system shown in figure 2.6 under section 2.3.1. We also utilized some of the same design principles, such as large tile sizes to minimize user errors [53].

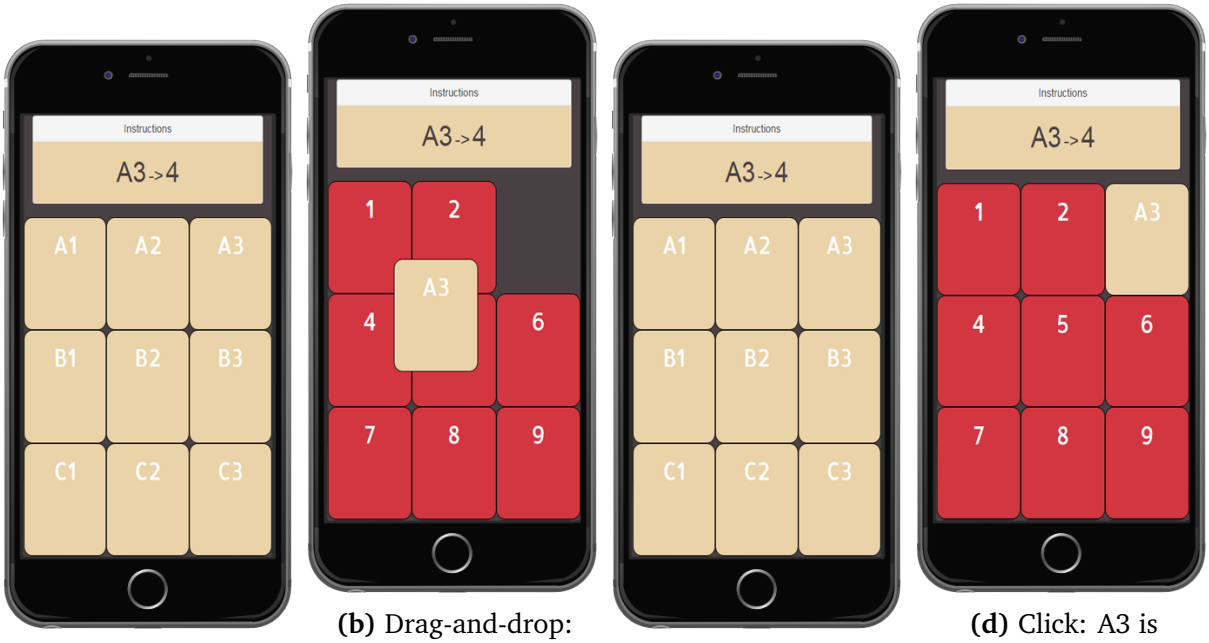
If a user initiates the action of annotating, but is distracted and looks away from the mobile device, it should be clear what the user's intention was when resuming the action. This includes being able to complete the action or easily cancel it.

The user interface should be easy to learn and intuitive to use, as well as fast and readily accessible. To provide an interface that enables the user to perform a task as fast as possible, we followed the three-click rule, stating that the user should be able to reach their intended destination of the app within three clicks [54]. We also wanted to examine if a drag-and-drop interface is the most suited gesture for an application that require fast and accurate actions.

In the early implementation stage of the app, we quickly discovered some of the look-and-feel disadvantages of an hybrid app. Cordova provides access to some hardware APIs through a native wrapper, but drag-and-drop is not among them [55]. Therefore, a drag-and-drop solution to annotating events would be processed by the device's browser engine, and on older devices, this would result in a different feel and smoothness than the user expects. However, we came up with an alternative solution, utilizing a simple two-screen interface, where the user is presented with a screen of possible registrants, and then is routed to a new screen with events to register on the selected registrant. We discovered that simple clicking is not as prone to differences in feel between platforms as drag-and-drop. We also found that a drag-and-drop interface can be slower in use and be more prone for mistakes, such as dropping in the wrong area or accidentally letting go while in the middle of the drag action [56, 57]. As one of the most important aspects of the Bagadus App is speed and accuracy, we set up a test for evaluating what interface we should base the Bagadus App on: drag-and-drop or click.

4.3.3 Drag-and-drop vs. click

To test user preference, speed and accuracy when annotating, a test app was made to simulate the two different solutions. For simplicity in distribution, we made a Web app instead of a hybrid app. The users used their own mobile device so the test was carried out on multiple different devices, but was instructed to use the device built-in browser, i.e. the same browser as a Cordova hybrid app would use. This provided the same basis as a hybrid app.



(a) Drag-and-drop: Initial view of a test

(b) Drag-and-drop: Dragging a box from A3 to 4

(c) Click: Initial view of a test

(d) Click: A3 is clicked, 4 should be clicked next

Figure 4.3: An example of a drag-and-drop test and a click test with the same parameters.

The users were given a total of 12 tasks. First, 6 using the drag and drop interface shown in figure 4.3a and figure 4.3b, and then 6 using the click interface shown in figure 4.3c and figure 4.3d. All the sub tasks were given in random order, so that no user could anticipate what the next task in the click part would be, based on the order of tasks given in the drag-and-drop part of the test.

The tasks are simple and intuitive. We managed to get 26 willing participants to do the test. The partakers were a mix of men and women with age ranging from 18 to over 40 years old. A short, textual introduction was included to make sure no test subjects had any trouble understanding the assignment.

The test was structured like this:

1. The user is given 6 drag-and-drop tests presented as a grid of tiles.

2. For each test, the user is given instructions in the form "A1 → 3", corresponding to indexes in the tile-grid, on where to drag-and-drop.

3. The user is given 6 click tests presented as a grid of tiles.

4. For each test, the user is given instructions equal to the those in (2), but with clicking instead of drag-and-drop.

5. The user is prompted to select their preference: drag-and-drop or clicking.

The app recorded the length of the time spent in each test, i.e., the time frame from when a user started dragging an object until it was dropped for the drag-and-drop tests, and correspondingly the time frame between clicks in the click tests. We also noted accuracy of each test, i.e., the number of tasks that were performed correctly. This information was not given to the user, making sure that they would not just try to race through the tasks, or focus too much on just getting everything right. At the end of the test, each user was asked which of the two interfaces they preferred. The results showed an overwhelming preference for the click solution, as shown in figure 4.5, where **all** the participants said that they preferred the click interface over drag-and-drop. In addition, the click interface scored highest in terms of accuracy, as shown in figure 4.4.

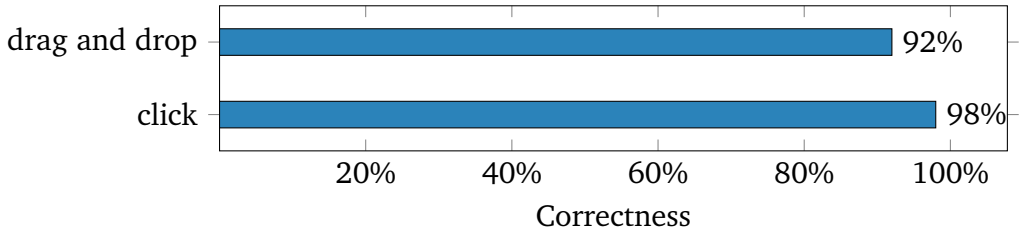


Figure 4.4: Accuracy: click versus drag-and-drop

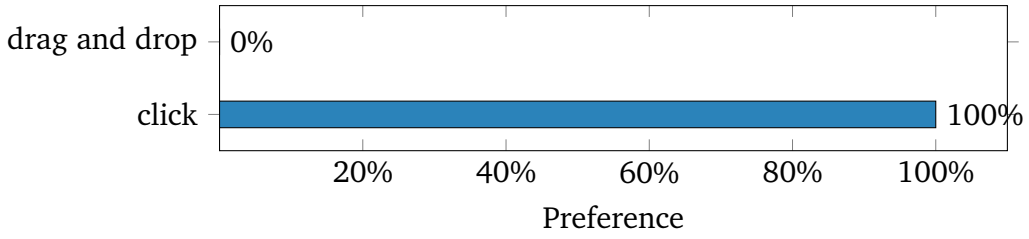
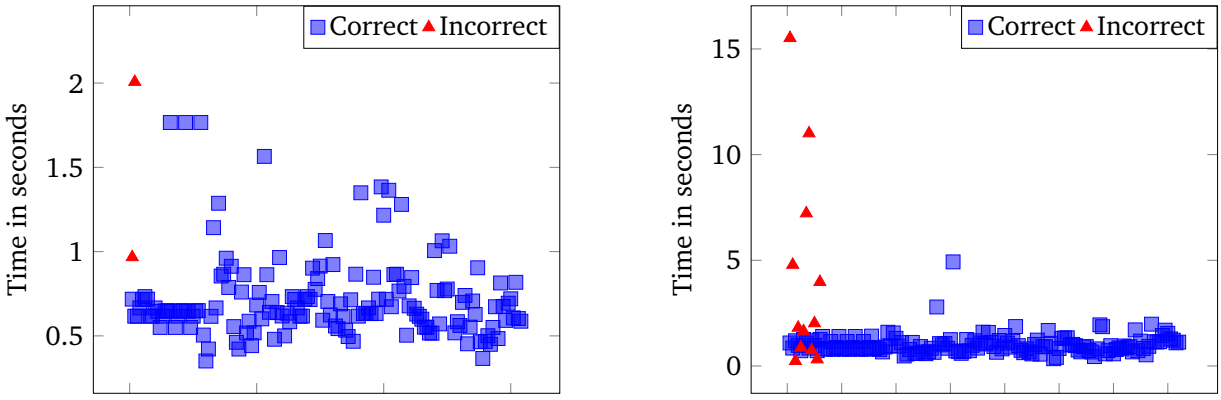


Figure 4.5: Preference: click versus drag-and-drop

Figure 4.6 shows the time spent on each task and if the task was performed correctly. The average time spent on one task was just over 0.7 seconds for the click interface. This is an improvement of about 442% faster than the average time of the Muithu interface found in a similar experiment [53]. Although the paper covering the Muithu test describes the test as very similar in nature to ours, we assume that the circumstances must have been different, as the results are very dissimilar for two essentially identical experiments.

The average for the drag-and-drop interface was just over 1.2 seconds. The results also show that the lowest time value of a single task was from a drag-and-drop test, but further investigation showed that this task was performed incorrectly and is therefore considered invalid. Likely the user accidentally dropped the box right after starting dragging. The lowest value for a task that was performed correctly belonged to the click interface with 235 milliseconds, while the lowest valid value for drag-and-drop was 312 milliseconds. The largest value for drag-and-drop was nearly twice the size of the largest for click: 4425 milliseconds and 2276 milliseconds, respectively.



(a) Click

(b) Drag-and-drop

Figure 4.6: Time: click versus drag-and-drop. Note the different scales of the y-axis between the two figures.

Some factors that can contribute to the fact that drag-and-drop is slower and less accurate is that, once you start dragging, you will have to hold your finger down and not let go until you have dragged it to the desired area. Some may start the dragging process before they have identified where it should be dropped and therefore having to hold their finger down for a long time or accidentally letting go too soon. If the dragging distance is large, e.g., from the bottom left corner to the upper right, and you start the drag with your thumb, the movement may become awkward and difficult to do effectively, as it is challenging to reach the upper right corner without repositioning your hand, while still keeping the thumb pressed down to avoid a drop in the wrong area. As such, you will probably end up using both hands to complete the task correctly.

Even though the drag-and-drop interface is a fun and popular way to interact with various applications, our tests, as well as other research papers [56, 57], show that it is prone to errors, worse annotation accuracy and slower speed. Click provides a more efficient interface that requires less effort. The click interface was preferred by all the participants in our test and produced the best results in regards to accuracy and speed. We therefore decided to base our annotation interface on clicking.

4.3.4 Architectural overview of interface

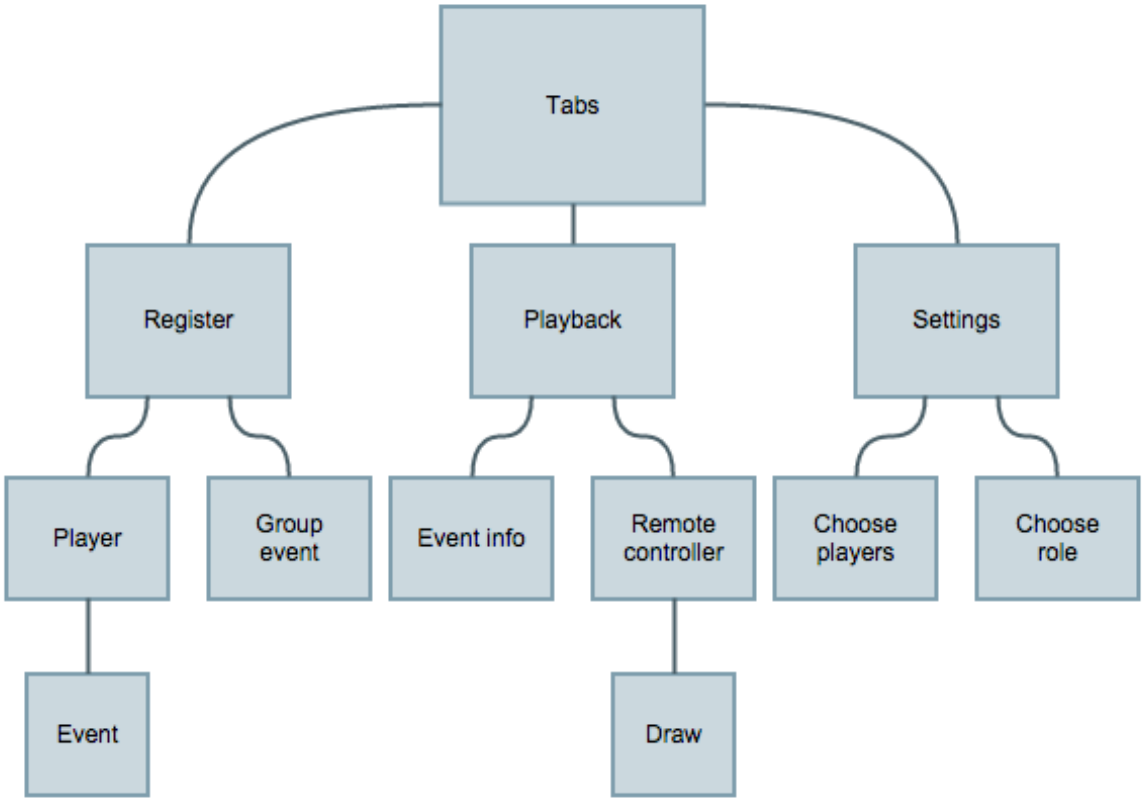


Figure 4.7: State machine representation of interface

Figure 4.7 shows a state representation of the interface with the tabs state as root. All the states that are direct children of the root state are accessible from any other state, except from the draw state, this is because we wanted to make the drawing area as large as possible by hiding the navigation bar. This makes the only possible navigating from the draw state, back to the remote controller state. States are preserved when switching, to ensure consistency for the user. The states that are direct children of the root state also provide a overview of the core functionality of the Bagadus App.

The following is a short description of the different states:

Register Register annotation on player or group

Player List of players. Select player.

Event List of events. Select event to register on player.

Group event List of group events. Select a combination of group and event to register.

Playback List of registered events. If a registered event is marked as relevant for the playback playlist, clicking it will lead to the remote controller. If not, the interface transitions to the event info state.

Event info Shows three frames grabbed from the registered event video. Allows the user to see a visual of the registered event and mark it as relevant for the playback playlist.

Remote controller Illustrates a remote controller with buttons for manipulating playback of video.

Draw Shows a picture of the current, paused playback which can be drawn on. Buttons allow for changing draw color and redo/undo draw actions.

Settings A list of settings pages.

Choose players A list of all players for selecting which players show up in the "Register" tab. Players can be selected from the list, or by the group buttons: "Defenders", "Midfielders" and "Defenders"

Choose role Select a role for annotation. The selected role is included when registering events, allowing for, e.g., better easier filtering during post-analysis.

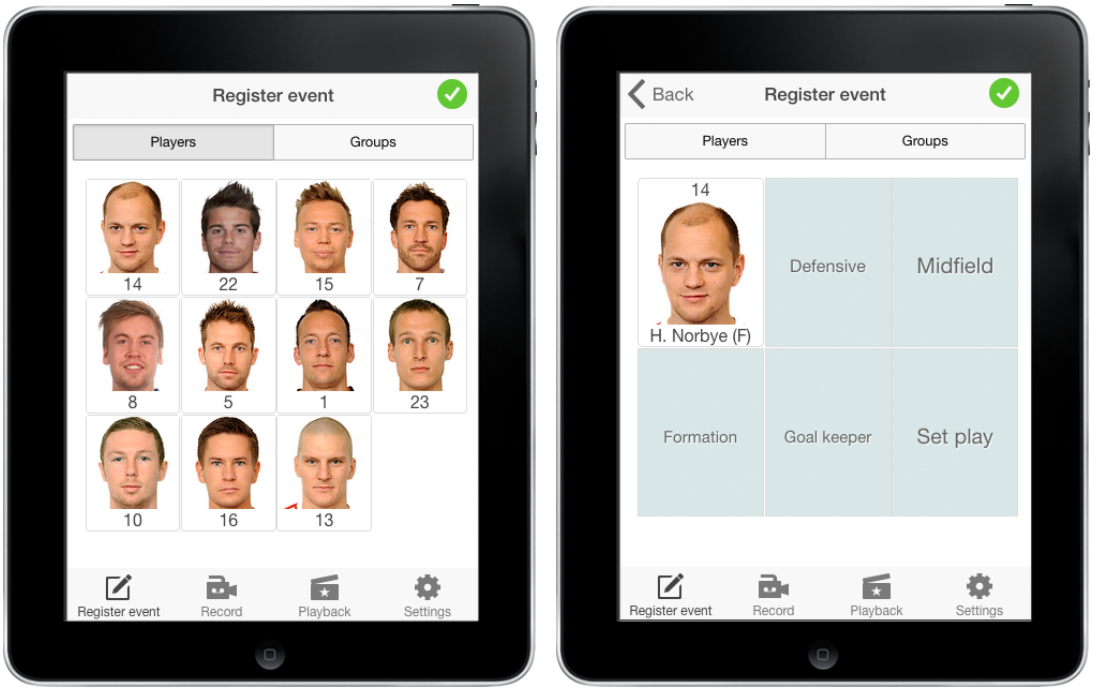
Details regarding the different states and their functionality is elaborated in the rest of this chapter.

4.4 Annotating events

When annotating an event, the user have initially two options i.e., to register an event on a specific player, or register a general event on the whole squad or group.

When registering on a specific player, the user is presented with a grid of players, as shown in figure 4.8a. When pressing on a player-tile, the user is then presented with the different types of events that can be registered on the selected registrant - i.e., the player - as shown in figure 4.8b. A tile of the same size as the events-tiles are reserved for a picture of the player. Clicking on the player tile, will take the user back to overview of players, aborting any registration. Clicking on an event tile will register an event on said player. This is designed to be used when the user wants a more specific event. It can be used as a tool in player development during training sessions, or otherwise where the user wants to capture a player-specific situation.

The user can also register an event on the whole squad, by only specifying the type of event or category, such as defending or attack. This is illustrated in figure 4.9. This is done by selecting the group tab in the upper right corner. Here the user is presented with tiles



(a) Annotation: Choose player **(b)** Annotation: Player event

Figure 4.8: An example of a annotation on player

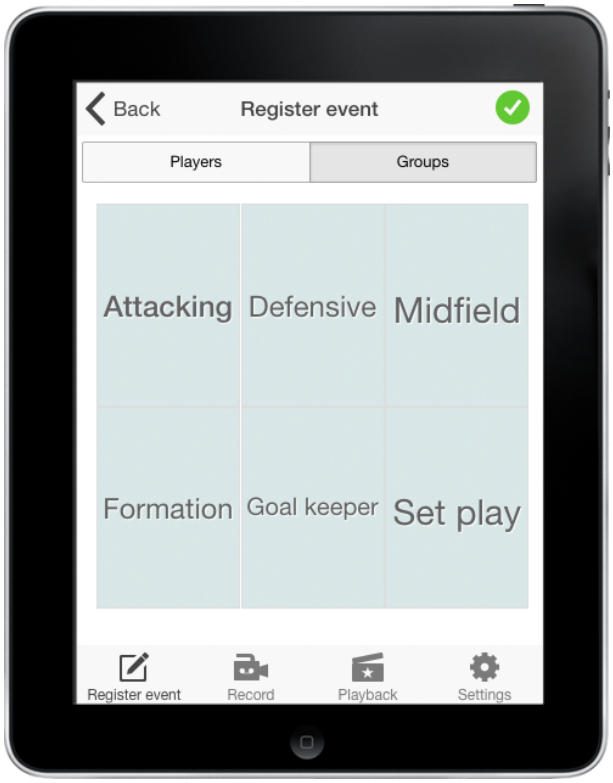


Figure 4.9: Annotation: Group event

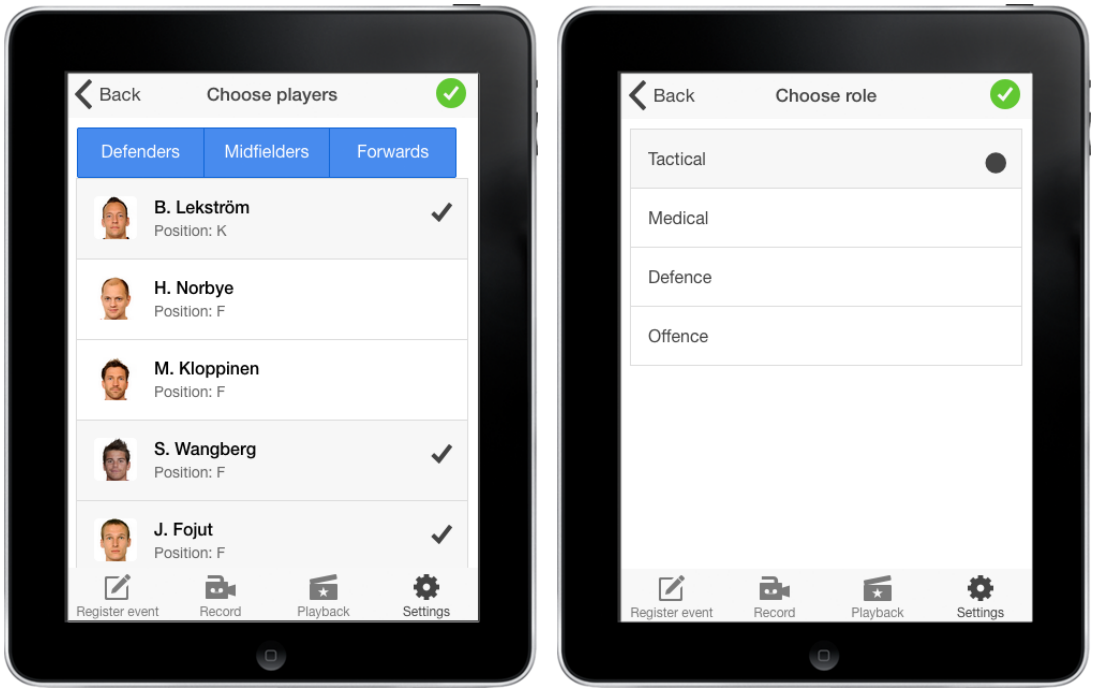
containing a descriptive text about the group or event type. Annotation is done simply by clicking on the desired event tile. The interface state remains the same after registration. This is to provide a efficient and rapid way of annotating general events, designed to be

used in a live game situation.

In early development of the Bagadus App, we added the functionality of scheduling and starting the recording via the application. This feature was later removed, and the cameras are now set to constantly run in periods when the stadium is used for the sake of simplicity. Having to make sure the recording was running before annotating is something one easily can forget, creating frustrating incidents with possibly missed annotations. Therefore the video pipeline is always running, saving video up to one year - a convenient approach for the end-user.

It is also possible to choose which players you wish to be available to register annotations on by going into the setting tab shown in figure 4.10a. This can be beneficial in live game situations when you have a starting squad of 11 players. The players on the bench or those not in the squad for that particular game will take up unnecessary space in the interface, and make it more challenging to do accurate, fast registrations, e.g., if the user has to scroll to find the correct player.

Certain event types may be more relevant to certain users than others. As such, it is possible to select a role through the settings tab shown in figure 4.10b. The role filters the type of events to match that selected role. The role is also included in the notational data when registering events, allowing for additional filtering in post-analysis.



(a) Setting: Player select **(b)** Settings: Roles

Figure 4.10: Setting: lineup and roles

The Bagadus App posts information about events to the Web API where the notational data is registered and corresponding video connected to the event. Processing of the data and video is detailed in chapter 5.

In section 3.3 we discussed different web communication protocol. As we identified the WebSocket protocol to, in theory, be a more suitable choice for high-frequency traffic, we wanted to investigate if we should use WebSocket for posting annotations as well. The WebSocket protocol is intended to be used when you need low latency, bi-directional communication between client and host. Additionally, why not use it for posting events, if it is, as suggested, faster and produce less data traffic?

When developing applications for hand held devices, there are some considerations that one needs to address, such as screen size and limited battery capacity. It is also worth noting the amount of data traffic, as the device might have limited mobile network coverage and bandwidth. In the following subsection, we conduct two experiments to see if there is a significant difference in data usage and battery life when using the WebSocket protocol versus the more traditional HTTP protocol.

4.4.1 Socket.IO and HTTP data usage

To test the data usage, we created two web applications: one using WebSocket and one using HTTP POST with two servers that the web applications could send data to, respectively. To implement the WebSocket protocol we used the Socket.IO library for JavaScript, which uses WebSocket when it is supported by the device and browser. Socket.IO never assumes that WebSocket will work, it establish a HTTP connection right away, and then tries to upgrade it to a WebSocket connection [58]. We chose Socket.IO because it provides easy implementation, reconnection logic and the support of namespaces. This choice and why this functionality is important to the Instant Replay Analytical Subsystem is detailed in chapter 5.

Other than the different protocols for sending data, the two web applications and servers were identical. We used Wireshark [59], a tool for capturing and analyzing network traffic to inspect the data. Wireshark can be used to inspect various protocols and can be set to only capture data from or to a specific host, or from a specific protocol such as HTTP. As such, we configured Wireshark to only capture data from and to the server. For the first experiment, two single, identical messages were sent - one over WebSocket and one over HTTP. The results are shown in table 4.2.

	HTTP POST	**Socket.io**
Number of packets	10	36
Total size	2233 bytes	5557 bytes

Table 4.2: HTTP POST vs Socket.IO data usage 1 post

The statement that using WebSocket produces less data traffic than HTTP seems to be inaccurate in this case. This is because our Socket.IO application uses several packets

to establish the WebSocket connection by first establishing a HTTP connection, and then gradually upgrading by checking **and** testing what the host is supporting, before it switces to the WebSocket protocol [60]. As shown in 4.11 the actual data is not ready to be sent via WebSocket before line 7.

Line 1 Check support for Polling.

Line 2 Test polling.

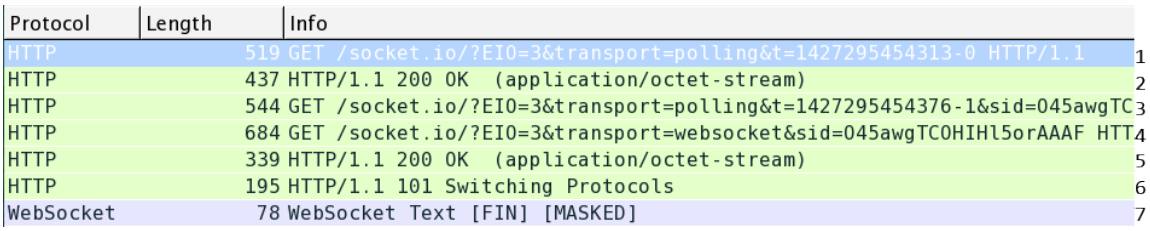
Line 3 Use polling.

Line 4 Check support for WebSocket.

Line 5 Test WebSocket

Line 6 Use WebSocket

Line 7 WebSocket up and running



Protocol	Length	Info	
HTTP	519	GET /socket.io/?EIO=3&transport=polling&t=1427295454313-0 HTTP/1.1	1
HTTP	437	HTTP/1.1 200 OK (application/octet-stream)	2
HTTP	544	GET /socket.io/?EIO=3&transport=polling&t=1427295454376-1&sid=045awgTC	3
HTTP	684	GET /socket.io/?EIO=3&transport=websocket&sid=045awgTCOHIHl5orAAAF HTT	4
HTTP	339	HTTP/1.1 200 OK (application/octet-stream)	5
HTTP	195	HTTP/1.1 101 Switching Protocols	6
WebSocket	78	WebSocket Text [FIN] [MASKED]	7

Figure 4.11: Socket.IO: Upgrade

Socket.IO is designed to immediately connect with HTTP, check if an upgrade is supported, test the upgrade, and then switch. This is repeated until it has established a WebSocket connection, if WebSockets is supported. With a raw WebSocket implementation[4], this would be done in one step, i.e., it would have sent a request to upgrade to WebSocket immediately.

The HTTP application requires less additional requests to initiate a connection, and performs the post immediately after the initial handshake, reducing the total size of data sent and received.

We investigated how much data was produced by sending the actual application payload, ignoring all other data sent and received. Listing 4.1 shows the test payload, which consist of just 31 bytes.

[4]One that is purely RFC-6455 compliant

```
{
    "id": 1,
    "type":2,
    "time": 123123
}
```

Listing 4.1: Data sent in JSON format

This resulted in a total data size of 186 bytes when using the WebSocket protocol, as illustrated in figure 4.12. The same data sent with HTTP produced a total of 945 bytes, as shown in figure 4.13. Again, just by looking at the actual message for the test payload. In this example, the JavaScript Object Notation (JSON) data in the request consist of 31 bytes, the HTTP request in total is 554 bytes and response is 391 bytes, making the total size of the message 945 bytes. As such, the actual JSON data is about 3.3% of the whole message. Doing the same calculations on the WebSocket message show that JSON data is about 16% of the whole message.

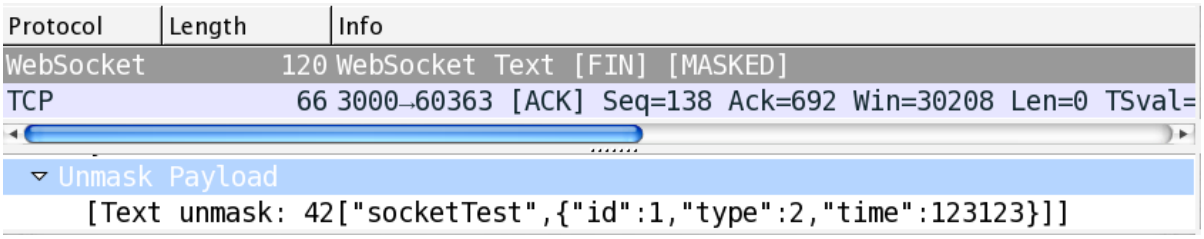


Figure 4.12: One post Socket.IO

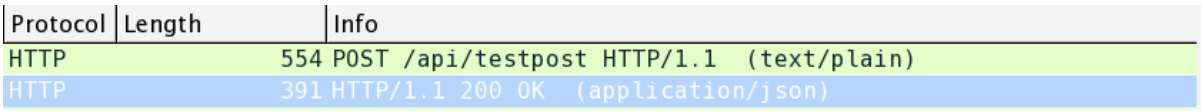


Figure 4.13: One post HTTP: filter HTTP

HTTP produced an increase of over 400% in total data sent and received compared to WebSocket, and show the extra overhead associated with the HTTP protocol e.g., metadata about the request and responce: accept types, information about the user agent, encoding, instructions about caching, etc. Note that the JSON data is compressed when sent i.e., all whitespace is removed. We suspected that if we increased the size of application data sent, the ratio between the protocols would even out, as the overhead will be relatively constant. To see if this was true, we used a JSON data file of 6.5 kilobytes and one of 15 kilobytes. The results confirmed our predictions - the amount of overhead evened out between the two protocols.

- Application data size : 6500 bytes
  - WebSocket: 7337 bytes
  - HTTP: 8272 bytes
  - HTTP: 12% larger than WebSocket
- Application data size : 15000 bytes
  - WebSocket: 16787 bytes
  - HTTP: 17743 bytes
  - HTTP: 5.7% larger than WebSocket

When sending the initial JSON data of 31 bytes, HTTP produced results that was 400% bigger than the results with WebSocket. When the data sent was increased, this ratio was lowered to just over 5%. This shows that when sending large data, HTTP and WebSocket produce similar results in regard to data size, while WebSocket will produce better result if the data sent is small.

So far, we had only looked at the result for performing one post. We wanted to investigate further to efficiency of WebSocket when sending multiple messages. Next, we conducted the same experiment, but this time with ten posts. To ensure an equal basis between the tests, we wrote a Protractor [61] script that did this automatically. Protractor is a JavaScript framework for testing. The script sent one message every minute for ten minutes on both applications. As visualized in table 4.3, the application using HTTP POST generates more data traffic.

	HTTP POST	**Socket.io**
Number of packets	67	144
Total size	20432 bytes	13548 bytes

Table 4.3: HTTP POST vs Socket.IO data usage 10 posts

Thereafter, three more experiments with the same setup were conducted: one performed 1 post every 5 minutes for 50 minutes, one with a single post every 5 seconds for 1 minute, and finally, 1 post with an interval of 100 milliseconds until 500 posts were concluded. The results are shown in table 4.4

As the results illustrate, when the frequency of sent messages is low, i.e., 1 every 5 minutes, our HTTP implementation yields better results in regard to data usage, about 77% less than Socket.io, but when the posting frequency increases, the Socket.IO implementation gives the best results, about 416% less than the HTTP application. One reason is that the WebSocket connection uses a heartbeat message to check if the client still is connected. The heartbeat interval is set to 25 seconds by default [62]. We observed through Wireshark that

	HTTP POST	Socket.io	
Number of packets	113	380	1 post every 5 minutes for 50 minutes
Total size	16822 bytes	29873 bytes	

Number of packets	43	66	1 post every 5 seconds for 1 minute
Total size	12602 bytes	8272 bytes	

Number of packets	1559	1052	1 post every 100 milliseconds, for 50 seconds
Total size	516519 bytes	100080 bytes	

Table 4.4: HTTP POST vs Socket.IO data usage

each heartbeat produced 211 bytes. The heartbeat is not issued if the server has received a message from the client within the interval. Posting every 5 minutes gives rise to 11 such heartbeats, as the one post replace one heartbeat. Doing this for 50 minutes totals 110 heartbeats, with a total size of 23210 bytes. If we subtract the combined heartbeat data from the total test data size of 29873 bytes, the result is 6663 bytes - less than $1/4$ of the total test data for the WebSocket connection.

The heartbeat is used to keep the WebSocket connection open, but this highlights how much extra data the heartbeats contribute when the message frequency is low. Thus, WebSocket communications may not be the most efficient choice in such a case. The two other tests show that the Socket.IO implementation produces less data usage, as the posting frequency is lower than the heartbeat interval. Therefore, the connection does not need to issue heartbeats, reducing a significant amount of unnecessary data sent. Despite of the HTTP test using keep-alive, so that all requests are sent over the same underlying TCP connection, the WebSocket test is more efficient in terms of data usage.

4.4.2 Socket.IO and HTTP battery life

A potential problem with using WebSocket is the reportedly excessive battery consumption on mobile devices [63]. For handheld devices, this is something one would wish to avoid as they have limited battery life before needing to be charged. As the main field of application for the Bagadus App on the sports pitch, where charging the device may be impossible or at the very least inconvenient, the battery usage of the app is an important element for its usability. As suggested in [64], a WebSocket connection can produce unacceptable battery life for mobile devices. We wanted to test this statement with our implementation.

For the experiment, we built two separate apps which connected to two separate servers. The sole functionality of the two apps was to post the same amount of data to their respective servers. Each pair of app and host would solely use HTTP or WebSocket to communicate. Apart from the communication protocols, the two apps were identical, as to eliminate any other potential disturbances that could affect the battery life.

The test were carried out on two devices: Ipad 4, Samsung Galaxy S4. All devices were charged to 100%, and then instructed to send two HTTP POST messages to the server every 5 minutes for 50 minutes, which is about the amount playtime in one half of a soccer game with additional overtime. The amount of power left was then measured, and the device recharged to 100%. The test was then repeated with the app using the Socket.IO implementation of WebSocket.

We also monitored the connection with Wireshark by setting up a computer as a WiFi hotspot to ensure the packets were delivered and the WebSocket connection held open. The screen on the devices were kept alive during the tests, to stop the devices from sleeping or otherwise affect the tests. Results in table 4.5 shows the percentage left after 50 minutes of posting two times with an interval of 5 minutes.

	HTTP POST	**Socket.io**
Samsung Galaxy S4	81%	83%
IPad 4	99%	100%

Table 4.5: Battery charge remaining after 50 minutes

There is a marginal difference between the two applications, and based on these results we can conclude that using WebSocket has no negative effect on the battery usage in our implementation. If anything, it extended the battery life of both systems contra HTTP. Furthermore, as suggested in [64], the heartbeat interval of the WebSocket connection, i.e. the amount of time between a heartbeat message is sent to see if the client still is connected, can have an impact on the power consumption. This is set to be 25 seconds by default[5] [62] and increasing this value can reduce the power consumption. We did not test this as we found that having a WebSocket connection did not produce any higher power consumption. The simple nature of this experiment makes it highly inaccurate., as we cannot accurately control the system processes on the device or precisely measure the battery. However, it produces an indication of the battery consumption between the two implementations.

4.4.3 Choosing communication protocol

Based on the different experiments and tests, we decided to use Socket.IO only where real time and rapid data sending is needed, such as for drawing and controlling playback of video from the app. In [7], the annotation frequency during training sessions was around once every 4 minutes. The average was 16 for a full 90 minutes match. The average frequency was not presented, but if equally distributed throughout the match, this would approximate to once every 5 and a half minutes. As such, for posting notational data we use the HTTP

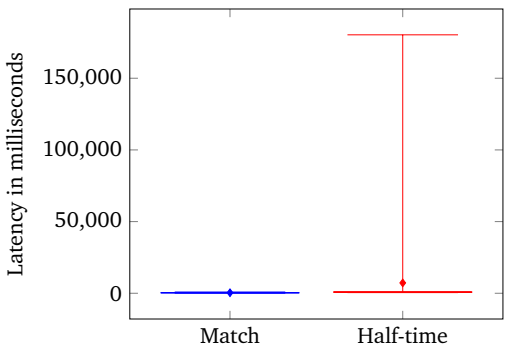
[5]We found this to be true in practice through Wireshark

protocol, as a WebSocket connection through Socket.IO did not prove to be a more efficient solution when the posting frequency was that low.

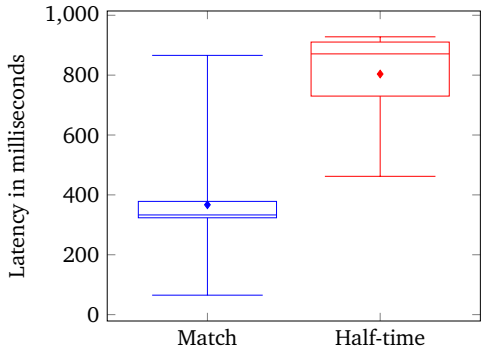
The research done in [63, 64] indicated that using the WebSocket protocol in mobile applications can cause higher battery consumption. Our simple battery tests showed the contrary. We suspect this is due to newer and better WebSocket implementations in the latest versions of Android and iOS.

4.4.4 Caching data

The biggest soccer stadiums in the world have capacities of close to 100 000 spectators. Although the largest in Norway, Ullevaal Stadium, in comparison, can accommodate *only* 27 200 people, there is still a significant possible amount of active cell phones in the arena during a match. 9 580 soccer fans attended the match between Norway and Estonia on 12th of November 2014. We performed tests by registering notational data during the match to test for contingency on the 4G mobile network of Telenor. The measured times are the network latency from the request is sent from the mobile device until it reaches the server, i.e., the one-way trip time.



(a) Shows the data including an extreme outlying value



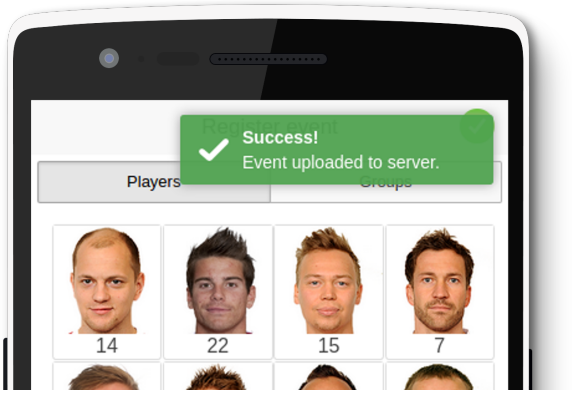
(b) Shows the data excluding the outlying value.

Figure 4.14: Two box plots illustrating the latency of registering events during the Norway vs Estonia soccer match on 12th of November 2014.

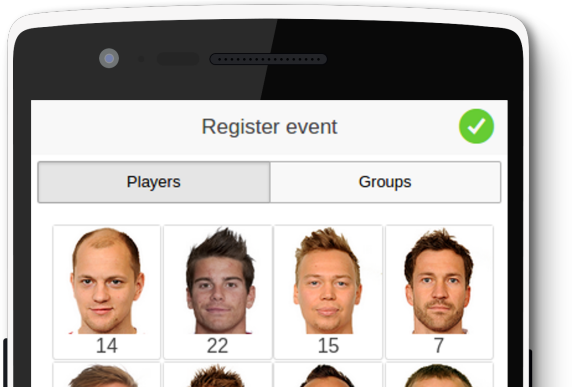
The results were stable during game time. However, during the half-time break, one request timed out. If a request timed out, the application would repeat the request, with the initial timestamp from the first request try, until a successful response was received. This outlying value had an extreme response time of 180 seconds, and is included in figure 4.14a.

Figure 4.14b shows the data without the outlying value. Here the values from the match and the break are much more contiguous, but the extra latency on the network is clearly visible by the over double mean time, 369ms vs 804ms, for registering events during half-time.

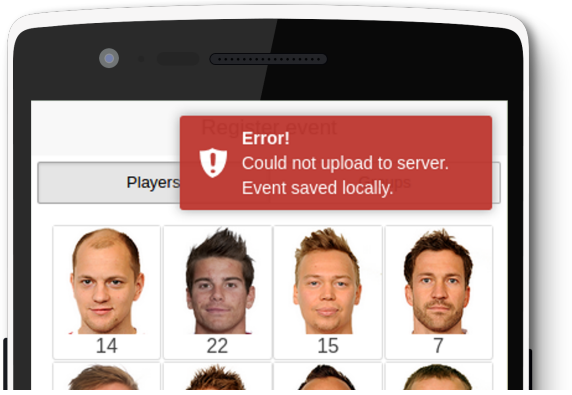
To combat situations with unstable network and unnecessary data usage, we opted for caching as much data as possible. This includes players, events as well as registered notational data. When a request to register an event fails due to a network error, the data is cached on the device. This cache is persistent, and will not be cleared unless the app successfully syncs with the Web API, the app is uninstalled, or the user manually clears the data.



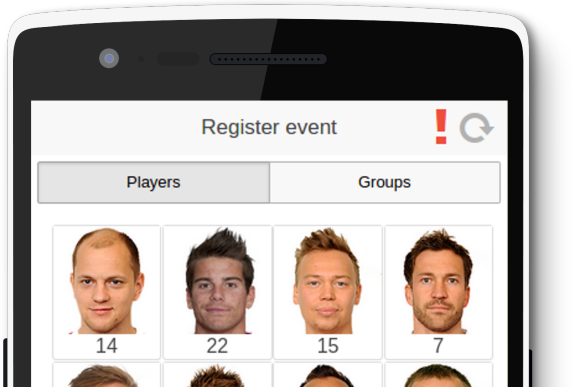
(a) Pop-up box when registered events are successfully uploaded.



(b) All registered events are synchronized.



(c) Pop-up box when an error has occurred during synchronization.



(d) There are registered events cached that have not been uploaded. The spinner icon is clickable, and will force a synchronization attempt.

Figure 4.15: The figures depict the various synchronization statuses between the app and the Web API for registering events.

If the device is offline, events are saved to cache immediately, making the device possible to use if network connection should drop. However, for the initial launch of the app, a mobile or WiFi connection must be available in order to synchronize time and fetch player, event and registered events data from the Web API. The players, events and registered events are cached for as long as the app is running. The user is able to manually upload the collection of cached registered events by pressing the spinner icon shown in figure 4.15c Figure 4.15 shows all the different synchronization statuses and messages.

4.4.5 Speech annotation

An interesting alternative to the click approach of annotating is to use speech recognition. By using speech, the process of registering notational data could be even more effective, as the coaches could keep their attention focused on the pitch even while registering.

Speech recognition systems are computationally intensive, likely too intensive for mobile devices with limited computational power [65], but this restriction can be circumvented through distributed computing by splitting the processing into client-based feature extraction and server-side recognition. As such, we investigated if this was possible to implement as an alternative method of annotating in the Bagadus App.

We identified two possible ways to achieve speech recognition in hybrid apps: by using the Web Speech API [66] or through plug-ins to native platform speech recognition SDKs. Android provides an official platform API for recognizing speech, while only third-party frameworks are available on iOS. The Web Speech API is not supported by Android browsers. One approach to provide consistent results between platforms, would be to transmit recorded voice messages to the Playback Server, which could perform the speech recognition or contact a third-party service like Google Speech. Extra machine learning could be done on the Playback Server based on the current user of the app for more precise recognition.

To test the plausibility of using voice recognition for annotating, we opted for using the Android SpeechRecognizer through the Cordova Plugin API for a simple proof-of-concept, as this was straightforward to setup without being too time-consuming. This service uses Google's vast cloud processing power and dictionary for processing speech and recognizing words. As the detailed operation behind the speech recognition service is only known to Google, we treat it as a black box by simply feed it with voice recordings and looking at the results. For the test, we used real event types specified by TIL: *angrep mellomrom (AMe), angrep bakrom (AB), angrep korridor (AK), angrep bak spisser (ABS), angrep mål (AMå), forsvar mellomrom (FM), forsvar bakrom (FB), forsvar korridor (FK), forsvar mål (FMå), offensiv dødball (OD), defensiv dødball (DD)*, as well as three generic categories in English for added variety: *attack (A), defense (D)* and *midfield (M)*.

The tests were performed with a Samsung Galaxy S4 mobile phone under three different settings: in a completely quiet room using the phone's microphone, in a conference room at Simula Research Laboratory with several speakers blaring out a sound clip of stadium noise[6] using the phone's microphone, and another with stadium noise, but this time using a hands-free set. The mobile device was held approximately 15 cm from the speaker's mouth when using the phone's microphone. The hands-free set microphone was roughly 7 cm from the speaker's mouth. Each event type was tested 10 times per event type per setting

[6] http://www.mediafire.com/listen/ni75bfkb9uy5hx0/small+football+crowd+by+FNC.mp3

by 2 different users. Both speaker's have a typical *bokmål* dialect and American English pronunciation.

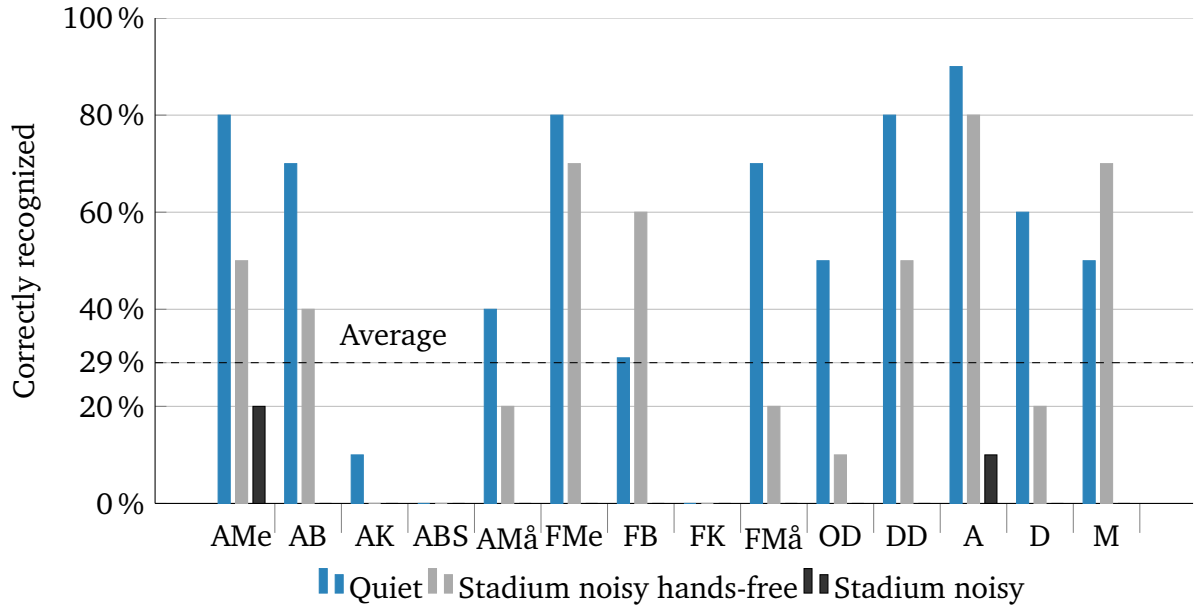


Figure 4.16: Test showing the percentage of correctly recognized event types with Google's speech recognition engine.

Figure 4.16 shows the results of the test. The test yielded a 51% correctness for the *quiet* setting, 2% correctness for the *stadium noisy* setting and 35% correctness for the *stadium noise hands-free* setting. This amounts to 29% of all tests correctly recognized. It is evident of the extremely bad score for the stadium noisy setting that the speech recognition service has trouble parsing voice commands with a lot of background noise. Using a hands-free set improved the results somewhat, but the background noise was still interfering.

Certain words were consistently misinterpreted as e.g., *dødball* and *korridor*. This may be due to the unusual word combination[7], which the search engine may interpret as a spelling mistake. Having a soccer specific dictionary or having the recognition service correct to the nearest valid event type through the use of edit distance or other metrics would likely improve the results. Another factor to consider is the considerable amount of background noise at a soccer stadium. The sound level in the two noisy tests were set at what we perceived from experience to be the approximate amount of background noise at a soccer game. As such, these settings are not completely accurate when compared to a live setting in a soccer stadium, but gives an indication of how well voice can be recognized in an loud environment. A study showed that the vuvuzela, famous from the 2010 FIFA World Cup, produces 113 dB(A) at 2 metres [67] - a considerable sound pressure level likely to interfere with speech recognition. We also observed that a clear diction with a focus on separating words when speaking improved the results, although this was mainly evident in the quiet

[7] "*angrep korridor*" yields three results on Google Search.

setting, as the mobile phone struggled to even separate the speech from the noise in the other settings.

Using speech recognition for annotating events was thus deemed as unfeasible for our implementation without further research. However, we assume that with some tweaking and more testing, speech recognition may be usable at training sessions, where the noise level is considerably lower than during matches, as the quiet setting produced 51% correct results and was often very close to the right words. Perhaps using special, easy recognizable trigger words instead of the event type name may make this more feasible. Investigating the performance of other speech recognition services may also yield better results. This simple test shows that annotation through speech is a possibility, but further research must be conducted.

4.5 Remote controller

The Bagadus App app can be used as a remote controller for the Playback System, which clients can connect to in order to view video. This makes viewing video connected to notational data easy and accessible without any manual work. The user can annotate and view the video of the annotation, all from the same app, and near instantly after annotating. All commands from the remote controller is sent to the Playback Server, which is a part of the Playback System. How these are handled and connected to video is detailed in chapter 5. All communication with the Playback Server is done over a Socket.IO WebSocket connection.

When navigating to the playback tab in the app, the user is presented with a list of registered events, as shown in figure 4.17a. By clicking on an event, the interface is routed to a view containing pictures of the registered event, as shown in figure 4.17b. These are extracted from the annotation video, and provide a glimpse of the start, middle and end of the situation. This can quickly illustrate the event for the user without having to watch video.

4.5.1 Filter events

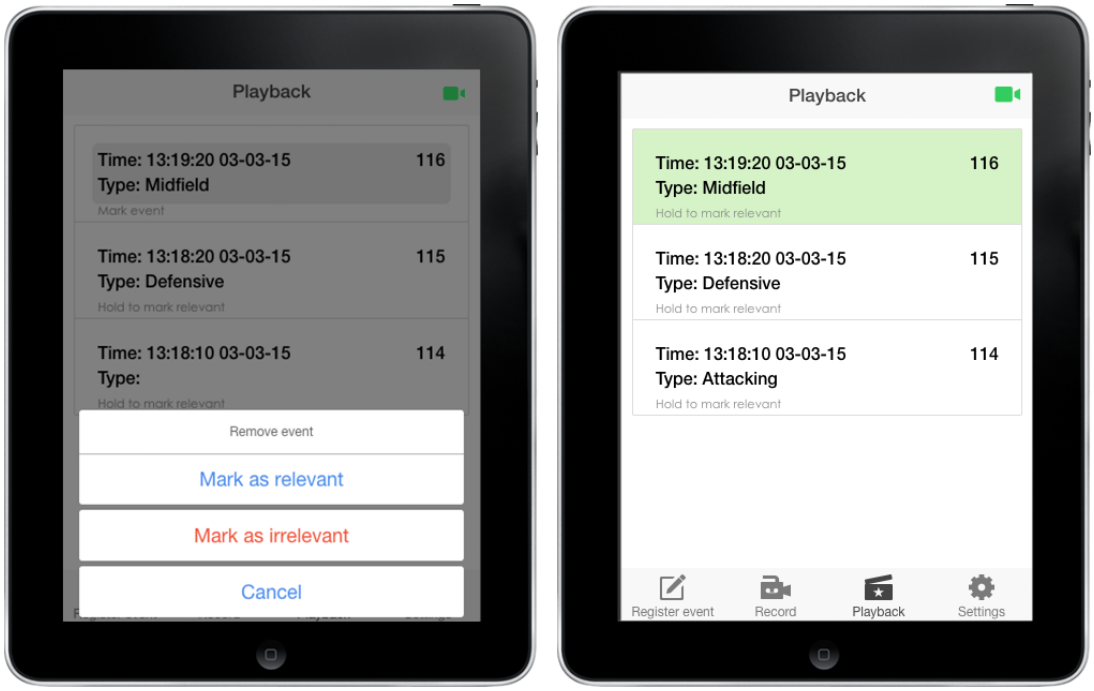
After reviewing the annotation, the user can choose to mark the event as relevant through a button at bottom of the screen. When pressed, the interface is routed back to the list of registered events. An annotation can also be marked or unmarked as relevant by long-pressing on it in list until a context menu appears, and then selecting it as relevant/irrelevant. This action is illustrated in figure 4.18a. Marking an event will make the event item green, as shown in figure 4.18b, to indicate it as relevant. The user can mark multiple events as relevant and thus creating a playlist of events. All events are irrelevant by default.

(a) List of registered events **(b)** Pictures of the event

Figure 4.17: Playback: list of events and pictures of event

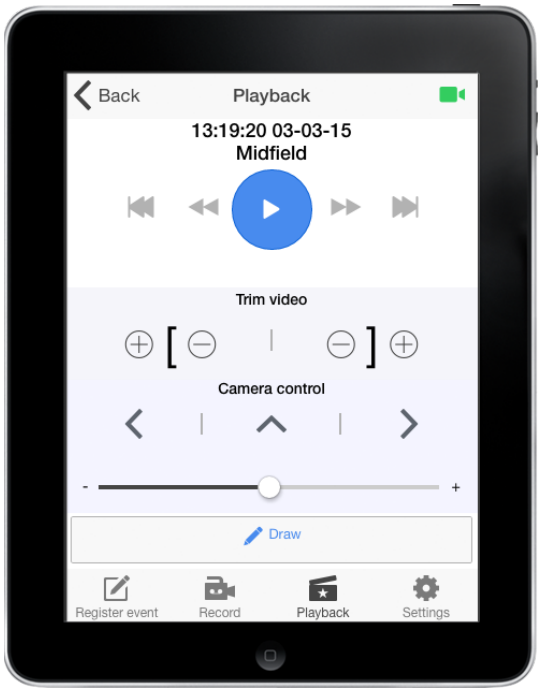
Filtering annotations into a playlist is useful in a live match setting, where the user may have tagged many events during the first half. Coaches have limited time to prepare for, as well as conduct, the half-time analysis. As such, providing a means of quickly narrowing down the list of annotations to the most important ones is crucial. By either filtering events during play-time, or in the first few minutes of the half-time break, the coaches can have a compact playlist of registered events to cover in the locker room with the players. TIL requested additional filtering, where registered events are grouped into defensive or offensive. This feature is highly personalized to TILs group events, as the filtering is based on regular expressions of their event type names. As such, this additional filtering is only present for TIL users of the app.

When a registered event has been marked as relevant, clicking it will route the app to the remote controller interface, as shown in figure 4.18c, while sending a command to Playback Server to start video playback. In this screen, the user can control playback of video of the registered event by sending commands to the Playback Server. The buttons represent the available actions: pan and zoom the camera, pause, extend or trim the video. The next and previous buttons will play the next or previous event in the playlist that the user has made by marking events as relevant. The fast-forward and fast-backward buttons will make the video skip 3 seconds forward or backwards.



(a) Dialog for marking event as relevant

(b) Event marked as relevant



(c) Remote controller for marked event

Figure 4.18: Long-pressing a registered event brings up a dialog, which allows the user to mark the event as relevant. When selecting a marked event in the list, the user is routed to the remote controller view for the selected event.

4.5.2 Camera angle

The camera control arrows allow for adjust the angle of the camera, i.e., panning in virtual views, to three predefined views. The middle one is centered on the pitch, the left is centered on the left goal area, and the right on the right goal real. The slider beneath

allows for zooming. Initially when using the zoom slider, commands were sent for every value change reported. This lead to hundreds of commands sent when using the slider, which felt excessive and unnecessary. Thus, we implemented a throttling function for the zoom slider listener, so that it would, at a maximum, only emit zoom commands every 0.2 seconds. The three predefined views combined with the ability to zoom, covers the whole playing field.

4.5.3 Trimming and extending video

The trim video buttons allow the user to trim and extend each video clip - both at the start and at the end. Often, the default video segment length, 15 seconds, may be excessive or insufficient. By individually tuning each clip length, users can minimize the amount of unnecessary video.

4.5.4 Drawing

During a demo of the Instant Replay Analytical Subsystem at the Norwegian Centre of Football Excellence, the Head of Performance Analysis voiced a request for the ability to draw on paused video via the app. When pressing the draw button, a command to pause video and a request for a still image of the current playback frame is sent to the Playback Server. The app changes to the draw interface, where the user is presented with a loading screen before the picture appears, as in figure 4.19. This state is further detailed under section 4.6.

Figure 4.19: Drawing state

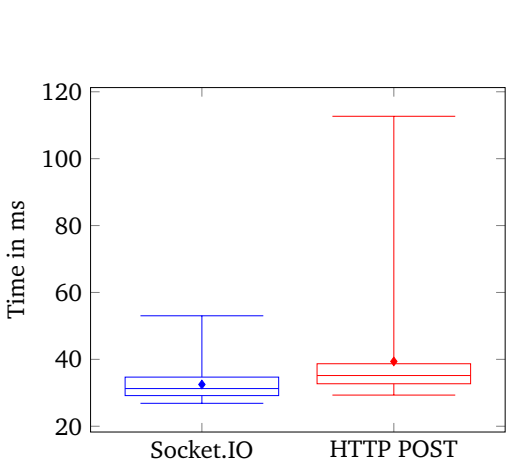
4.5.5 Latency

Using the app as a remote controller, with the ability to draw on remote video, requires the response time to be as quick as possible. A high latency can make for a frustrating and inaccurate experience for the user. If a command to, e.g., pause video, takes too long to execute, video will pause at a later point than the user intended. Furthermore, the time available for half-time analysis is limited, so every second counts. As such, we investigated both the approach of using HTTP and WebSocket through Socket.IO for this purpose as well, to find which protocol was most efficient. We configured the app to post notational data to the Playback Server using HTTP POST or WebSocket. The elapsed time between the button press event in the app, until the message was received at the server, was then captured.

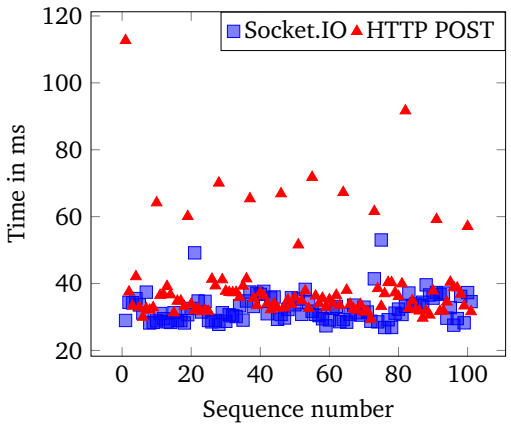
4.5.6 Socket.IO and HTTP POST response time

Using HTTP POST and Socket.IO, we simulated the situation of playing back a video clip 100 times by sending the same amount of data corresponding to the ID of the clip we wanted to show. Considering that we are interested in predictable and stable communications to the Playback Server, we calculated the variance and standard deviation of the 100 trails. The results are given in table 4.6, and visualized in figure 4.20.

The results present a small difference in average latency, but by observing the variance and standard deviation, it is reasonable to say that the Socket.IO implementation is more



(a) Socket.IO vs HTTP POST box plot



(b) Socket.IO vs HTTP POST distribution plot

Figure 4.20: HTTP POST vs Socket.IO for 100 posts

	HTTP POST	**Socket.io**
Lowest value	29.315 ms	26.856 ms
Largest value	112.672 ms	53.026 ms
Mean	39.373 ms	32.485 ms
Variance	170.420 ms	17.625 ms
Standard deviation	13.054 ms	4.198 ms

Table 4.6: Results Socket.IO vs. HTTP POST 100 times

reliant in terms of consistent latency. The largest value using HTTP POST is over twice the size of the largest using Socket.IO.

To achieve an even statistical analysis, we performed the same experiment with 1000 messages instead of 100. As this was a tedious task to do manually, we wrote a script using Protractor. The example code in listing 4.2 shows a script generating 1000 requests with 1 request every 100 milliseconds to the Playback Server installed on a server located in Amsterdam, while serving the application from localhost.

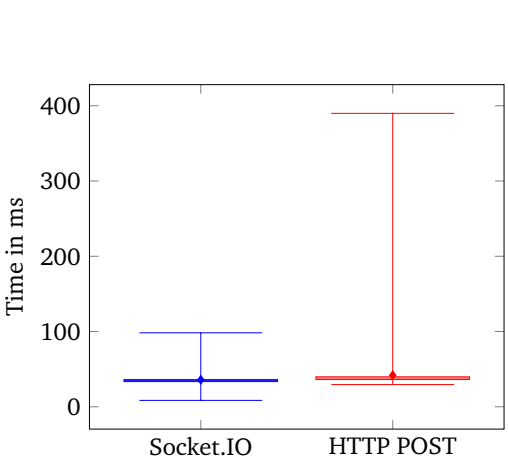
```
describe('Post event', function(){
        ptor = protractor.getInstance();
        browser.get('http://localhost:8100/#/tab/post');
        it('Should post 1000 posts', function(){
                var x = 0;
                var btn = element(by.id('post'));
                while(x <= 1000){
                        btn.click();
                        x = x+1;
                        ptor.sleep(100);
                }
                expect(true).toBe(true);
        },50000000);

},50000000);
```

Listing 4.2: Code for performing 1000 posts

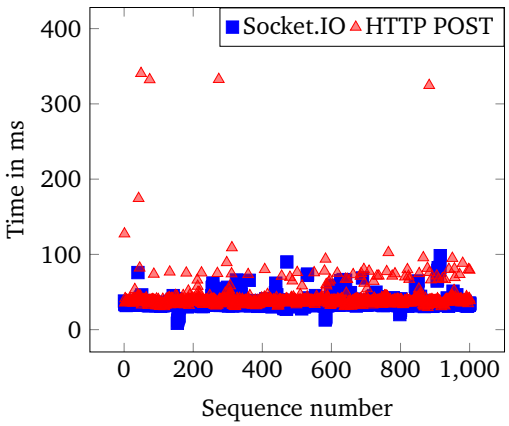
The script posted a play command to the playback server 1000 times. Results are shown in table 4.7 and visualized in figure 4.21

	HTTP POST	**Socket.io**
Lowest value	29.495 ms	8.504 ms
Largest value	389.909 ms	98.298 ms
Mean	41.962 ms	35.925 ms
Variance	579.989 ms	49.068 ms
Standard deviation	24.083 ms	7.005 ms

Table 4.7: Results Socket.IO vs. HTTP POST 1000 times



(a) Socket.IO vs HTTP POST box plot



(b) Socket.IO vs HTTP POST distribution plot

Figure 4.21: HTTP POST vs Socket.IO for 1000 posts

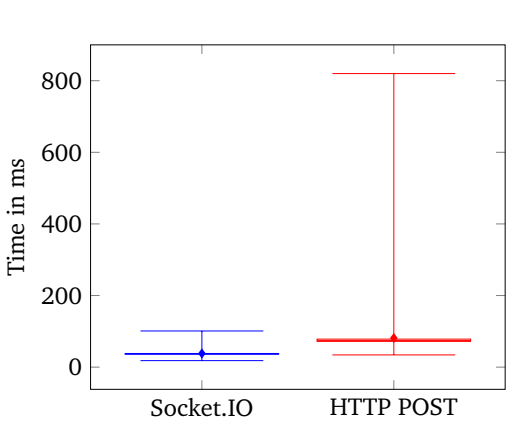
The results tell the same story. However, we noticed that the response time for an HTTP POST request was slower when given a certain timeout between tests. This was due to

HTTP keep-alive making the connection stay open for a certain amount of time, and the script continuously making requests at 100 ms apart, and therefore taking advantage of HTTP keep-alive. Although this is default behaviour, the timeout is usually set to a low value, this is useful when multiple request are made rapidly, for example when loading a new web page with many resources. Making requests at this rate from the remote controller in a live setting is unlikely. Thus, to provide a more realistic test, we disabled the keep-alive setting by forcing the server to close the connection between each request.

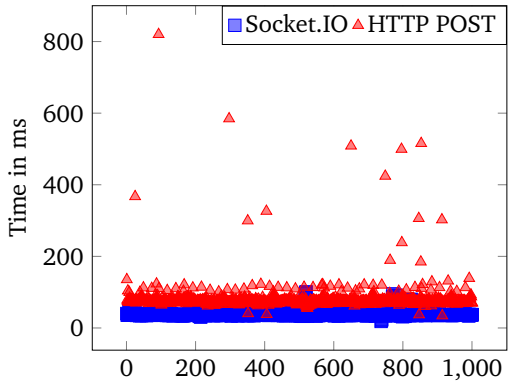
As shown in table 4.8 and in figure 4.22, it is clear that the HTTP test is outperformed by the Socket.IO test. The largest latency value of an HTTP message is over eight times larger than the slowest Socket.IO message, which is just over 100 milliseconds, and the Socket.IO results are more consistent. With no keep-alive header, the HTTP POST approach got very unstable results, as accentuated by the nearly doubled mean and standard deviation values compared to with keep-alive enabled. The standard deviation is over 560% larger than standard deviation of the Socket.io implementation. This is the consequence of HTTP having to establish a new connection for each request, while Socket.IO only does this once, keeping a single WebSocket connection open for the duration of the test.

	HTTP POST	**Socket.io**
Lowest value	34.132 ms	18.182 ms
Largest value	819.943 ms	100.995 ms
Mean	81.251 ms	38.094 ms
Variance	1925.459 ms	43.045 ms
Standard deviation	43.880 ms	6.560 ms

Table 4.8: Results Socket.IO vs. HTTP POST 1000 times without keep-alive



(a) Socket.IO vs HTTP POST box plot

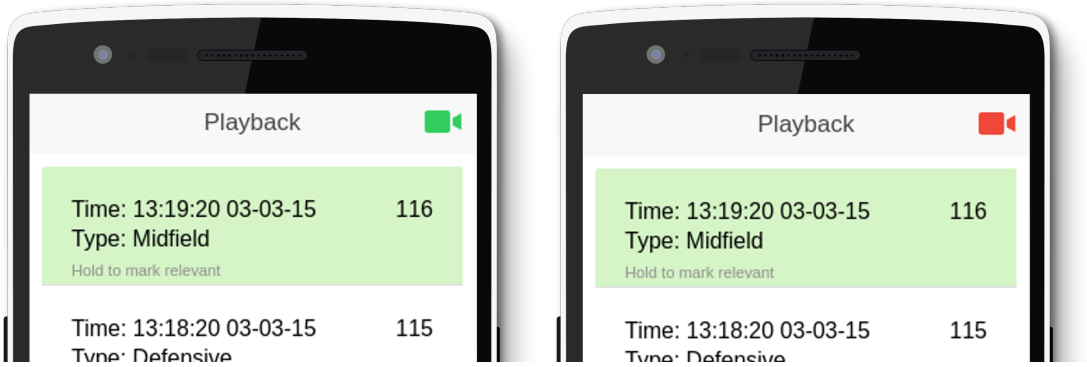


(b) Socket.IO vs HTTP POST distribution plot

Figure 4.22: HTTP POST vs Socket.IO for 1000 posts without keep-alive

For the purpose of using the app as a remote controller for video playback where low latency and reliability is important, a WebSocket connection is the best choice, especially

considering actions with a high frequency message exchange, such as zooming[8]. However, keeping a WebSocket connection constantly open requires some data due to heartbeat signals between the client and server. When annotating events, e.g., during a match, keeping a WebSocket connection to the Playback Server open is unnecessary. As such, the WebSocket connection will only be established when the user enters the playback tab. Nonetheless, the user should be able to mark events even though the connection the the Playback Server is offline, as this action is only dependant on the Web API. As such, the registered events view in the playback tab, as well as the show event info view is available, but the user is notified of a missing connection to the Playback Server through an icon, as shown in figure 4.23.



(a) Socket connection online **(b)** Socket connection offline

Figure 4.23: Status of socket connection to Playback Server

There may be possible situations where the user moves from the playback tab to the register tab or other parts of the application and then back again in a short span of time. In these cases, we made some calculations whether the WebSocket connection should be terminated when leaving the playback tab or kept open for a period of time before closing. For the Socket.IO WebSocket implementation, the heartbeat interval is 25 seconds, where each heartbeat uses 211 bytes. This amounts to 506 bytes per minute in heartbeat data. Opening and closing a Socket.IO WebSocket connection costs 4637 bytes and 687 bytes, respectively, for a total of 5324 bytes. Thus, keeping the WebSocket connection open for 10 and half minutes costs as much in network data as opening and closing a new connection. Therefore, it is more efficient to keep a WebSocket connection open if we expect the user to submit a new command during this interval. The expected use case of the remote controller is during half-time breaks and instant analysis during training sessions. Thus, the remote controller usage is likely concentrated in a short period of time, e.g., 10-15 minutes during the half-time break, and in sporadic 10 minute segments during a training session. Therefore, to be well within the expected use interval, but limit unnecessary data usage, the WebSocket connection will terminate after 15 minutes of idling, i.e., the time elapsed since the user last issued a playback command.

[8]A message is sent every 200 ms if the user is continuously moving the zoom slider

4.6 Drawing

With the standardization of HTML5, two new exciting features have been introduced: the support of `<video>` as an element, eliminating the need for third-party browser video players like those based on Adobe Flash or Microsoft Silverlight, and the `<canvas>` element that can be used to draw graphics [68–70].

Drawing as a teaching tool is used in many situations and can help to demonstrate and illustrate important points. In soccer it can be used to explain tactical and strategic goals that can aid players and personnel to gain a better understanding of a situation.

With the Bagadus App, this can be done on the training field or in the dressing room during the half-time break by having one or more devices streaming video through Playback System, and the app controlling that video. By pressing the draw button in the remote controller section, the user is presented with the current frame of video playback. In this view, the user has the ability to draw on the image using touch, and having it appear in real-time at clients streaming through the Playback System. To get the largest drawing area as possible we force the app to go into landscape mode, and hiding the menu bar at the bottom, this makes it easier to draw with precision and accuracy. The process of capturing the current video frame, scaling it and sending to the Bagadus App, as well as forwarding drawing instructions from the app to clients streaming video is discussed in chapter 5.

In the previous section, we concluded that a WebSocket connection is the best choice for situations where low latency and reliability is important, with the zoom action used as an example. The action of drawing is very much alike in terms of message frequency. When drawing, touch events are continuously fired. For each, the app emits 4 drawing coordinates, i.e., `x` and `y` for the start and stop points, since the last touch event. These coordinates are relative to the canvas they are drawn on, and must be scaled for correct display at clients. This is discussed in section 5.7.2.

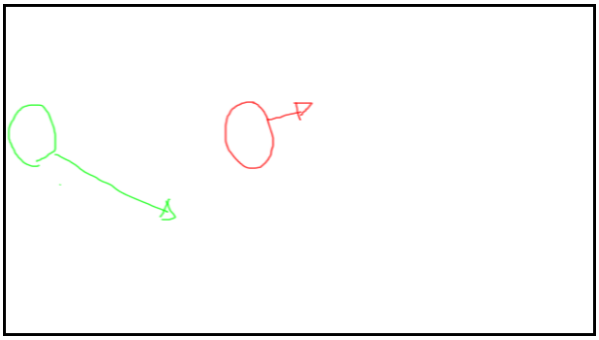
The draw functionality utilizes two canvases, one that displays the captured image, as shown in figure 4.24a, and one that displays the actual drawing with a transparent background, as shown in figure 4.24b. The drawing canvas is superimposed on the image canvas, as shown in figure 4.24c. Using two canvases, allows us store states efficiently, by only storing the drawing data, and not the actual image data, as shown in figure 4.24.

The separation of the two canvases enables us to minimize the memory usage associated with the drawing feature. To support functionality for undo and redo, we have to continuously save the states of the drawing as the drawing progresses. Our initial implementation only used one canvas, which held both captured screenshot and the drawing, this was later replaced with the two canvas solution because of poor performance. To keep track of the current, and previous states of the drawing, we save each state as base64 encoded string of the image. The size of the string increases with the complexity of

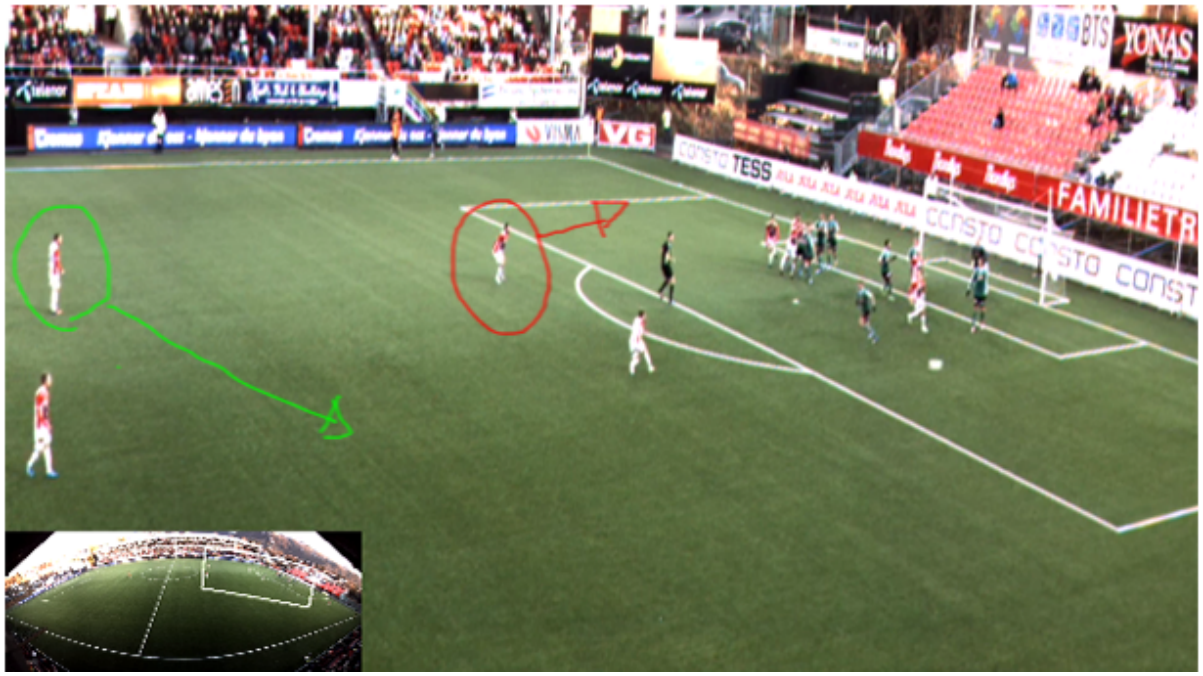
the drawing.

To illustrate, the encoded string of the image shown in figure 4.24c has a size of 526 kilobytes, while the encoded string only containing the drawing shown in figure 4.24b is about 20 kilobytes. Storing big strings in memory and having to quickly swap between large images when undoing or redoing, can produce unnecessary latency and slow behaviour on mobile devices with limited memory.

(a) Canvas containing the screenshot



(b) Canvas containing drawing



(c) Drawing canvas superimposed on screenshot canvas

Figure 4.24: Drawing and screenshot canvas

We define a draw state as the moment the user stops drawing, i.e., the touch-end event fires. On the next touch-start event, the current state is saved to an *undo array*. At the same moment, a `save_draw_state` message is sent to the Playback Server, which forwards it to clients streaming video, who store the states in the same way. Section 5.7.2 further elaborates drawing for playback clients. This provides the app user with functionality for undo and redo actions. When undoing a draw state, the current state saved to a *redo array*, an image is popped off the *undo array* and inserted into the canvas. When redoing a draw

state, the current state is saved to the *undo array* and an image popped off the redo array and inserted into the canvas. This functionality is useful for, e.g., when a coach wants to switch between draw states to explain something different, or made a mistake while drawing.

By having the Bagadus App and the clients streaming video individually storing states, we reduce the amount of redundant data that would have to be sent if the app should emit the complete draw state information.

4.7 Synchronizing device time

A synchronized time across the system is crucial when capturing notational data. The Bagadus App must be synchronized with the video subsystem for the latter to retrieve accurate video segments, as these are based on the timestamp included with the notational data from the app.

When launching the application, a timestamp t_c is captured from the mobile device's internal clock. A request is then sent to the server to get an equivalent system timestamp t_s. We make the assumption that the route is fairly symmetrical, i.e., that the request uses the same time to reach the client as the time the response takes to get back. The network time offset is thus calculated by dividing the round trip time RTT by $\frac{RTT}{2} = OWTT$ to get an estimated one-way time from client to server. The delta value is calculated by subtracting the server time and the one-way trip time from the mobile client time.

Fluctuating round trip times are a problem, something we particularly experienced at the Norway vs Bulgaria match at Ullevaal Stadium 13.10.2014 with over 14 000 spectators in the arena clogging the mobile network. As such, the client makes several requests until a good enough round trip is found. The lower the round-trip time, the more accurate the clock offset will be, as the network latency is the most uncertain element and would thus impact the network-latency-adjusted clock offset the most. We put the accepted round trip time at 500ms, which gives an error margin of up to almost half a second (if the round trip was significantly asymmetrical, the request could use 450ms one way, and 50ms the other), which should be acceptable for registered events. In a worst case scenario, the end-user would need to extend or trim the event after tagging.

In situations where no round trip time faster than 500ms is found, we use a simple algorithm to calculate an average round trip time and average clock offset between mobile client and server. Equation (4.1) shows this algorithm. The delta time between the current value and the previous value is calculated, and then multiplied with a constant weighting factor δ and the result added to the current value. 10 round trips are made, where the algorithm is applied every iteration for the round trip time *and* the clock offset - calculating an average of both. The result every iteration, T_{est}, is carried on into the next iteration as,

T_{prev}, creating an average.

$$\begin{aligned} \Delta_T &= T_{prev} - T_{curr} \\ T_{est} &= T_{curr} + (\delta \times \Delta_T) \end{aligned} \tag{4.1}$$

A value δ close to 1 makes the weighted average immune to single, long round trips. We chose a value of $\delta = 0.8$, as we do not want single, long round trips to affect the result considerably.

The resulting round trip time, found by either being shorter than 500ms or by calculating the average, is then divided by two to get the assumed one-way trip time $\frac{RTT}{2} = OWTT$. This network time offset is added to the clock offset to get the network-latency-adjusted clock offset between the mobile client and the server, which is then saved on the mobile client, and added to timestamps when registering events. The calculated offset is valid for one session, i.e., while the app is running, and is recalculated through this synchronization operation each time the app is relaunched.

4.8 Authentication

Traditionally, services that require authentication enforce strong password policies, such as having a length of at least 8 characters, a mix of uppercase and lowercase letters and symbols to protect passwords against brute-forcing. Such passwords may be difficult to type on mobile devices. In addition, prompting the user to enter a username and password every time an app is opened is time-consuming and may limit usability. As such, mobile operating systems provide services for storing user data on the device, e.g., iOS Keychain [71] and Android AccountManager [72]. These act as centralized registries of a mobile user's online accounts. They allow for functionality such as sharing data across applications, letting the OS decide when to synchronize data, e.g., uploading app data when a network connection becomes available, and storing authentication information. However, these services are only available through native APIs. As such, plug-ins are needed for hybrid apps to communicate with them. Furthermore, the extra functionality is not deemed necessary for the Bagadus App, as we only wish to store authentication data. Therefore, we use a simpler method of storing user authentication in the app.

Conventional web authentication uses cookies to store authentication for a session or long-term. We experienced difficulty using persistent cookies in Bagadus App, as they were cleared when closing the app, acting as session cookies. As such, we investigated saving the cookie information manually. When a domain responds with a `Set-Cookie` HTTP header after a successful authentication request, the cookie key and value can be saved in the device browser's `localStorage`, a type of web browser storage introduced with HTML5. However,

due to security concerns, browsers do not allow setting cookies for different domains than the current domain. As an hybrid app is served from a `file://` URI, setting a cookie's domain to, e.g., `https://alfheim.bagadus.no` is not allowed. Thus, the cookie cannot be stored longer than a session, i.e., while the app is running.

To circumvent this restriction, we use a token-based authentication scheme. This is identical to authenticating with a cookie, except for using custom headers and thus bypassing cookie-header restrictions. The Web API accepts a username and a SHA-256 hashed password, and returns a token to the client. This token is then stored in the app, and subsequently used for every request to the Web API.

Any data stored on the client cannot be guaranteed to be kept secret, as the phone e.g., could be stolen or the user could sell their phone without clearing wiping data. The token could be encrypted, but any encryption key would be available through the app's JavaScript files. As such, the tokens have an expire date for added security. Once it expires, the user will have to authenticate with their username and password again to receive a new token.

4.9 Evaluation

Some of the available data is currently not utilized in the Bagadus App, but can provide information that is useful for future development of the system, and is therefore purposely left in. For example, the player objects of which you register an event on, contain their default player position. This can be used for automatic positional-based retrieval of registered events e.g., all events that is registered on the midfielders.

The plausibility of using speech recognition for annotating events was briefly investigated, but we conclude that more extensive research must be done in order for it to be a feasible and practical solution for the end-user. The various, possible ways of approaching this feature was deemed too broad and time-consuming to investigate in this thesis.

In this chapter, the frameworks and protocol implementations tested, such as Socket.IO, have been used as-is. Further tweaking of parameters, such as heartbeat interval length and HTTP headers, is possible, and may lead to different results.

Because the Bagadus App can be used in training sessions, as well as in match situations, one feature that can be implemented is the ability to set the app in "match mode" or "training mode". When used in a training session, the user typically wants to show the players something immediately, eliminating the need to prioritize and filter out events. Having the training mode on could remove the need to mark an event as relevant before playing it.

4.10 Summary

In this chapter, we presented and evaluated different aspects of development in web applications, native applications and hybrid applications in regards to platforms and performance. We concluded that a hybrid application is the most suited approach for the Bagadus App, although this approach can have some disadvantages in the responsiveness of certain graphical user interfaces. Further, we found that a drag-and-drop interface require more process power than a click based interface. Drag-and-drop is also slower and more prone to errors than a click based interface. We found through user testing, that a click based interface is the preferred way to carry out simple tasks similar to the annotation process in the Bagadus App where the average time for one annotation was 0.7 seconds, well within our maximum limit of 5 seconds. We also investigated the possibility of using speech recognition for capturing annotations, but found that additional research must be conducted in order for it to be a feasible solution.

When the frequency of messages sent and received is high, we found that a WebSocket implementation yields better results then HTTP in regards to data use. And the that the results are reversed when the frequency is low. We suggested that when posting larger data, the total data size sent and received by HTTP and WebSocket converge, while posting small data files result in better results using WebSocket.

The WebSocket protocol provides the most stable results concerning the response time, and do not affect the battery life in any notable way. We have described how drawing can be achieved in a way that minimize the amount of memory usage by using two canvases, and how we synchronize time across the system. Finally, we evaluated the Bagadus App and discussed some features that could improve the usability. With our design and implementation of the Bagadus App, as described in this chapter, we have introduced a way for the application to be fast, cost effective, intuitive and available for the end-user.

In the next chapter, we will see how we enable instant playback of our annotated events from the app, through the Playback System. We will discuss the technology and motivation, and present an overview of existing components and how these are connected to our main contributions: the Bagadus App, the Playback Server and the playback client.

Chapter 5

The Playback System

The Playback System is what ties the Bagadus App together with the video subsystem of Bagadus, as described in section 2.1. It allows users to stream virtual view video connected to annotated events through a simple front-end web client, and provides an interface for the Bagadus App to control the video streams in terms of manipulating the video source and changing the virtual view perspectives. It also allows the app to perform additional analytical techniques, such as drawing on video.

In the following chapter, we will explain the structure of the Playback System, it's features and discuss design and implementation details. We will first discuss the motivation behind the system, existing components the Playback System depends on and interacts with, and then go into detail on our main contribution - the Playback Server and the web playback client - and evaluate the presented work.

5.1 Motivation

We quickly realized that the Bagadus App needed a back-end layer to efficiently integrate it with the rest of the Bagadus system, and for it to provide the functionality we wanted. For a pure annotation approach, as with the Muithu app, the Web API to the Bagadus database would be sufficient. However, for features such as controlling playback of events and drawing on video, additional infrastructure supporting the app was needed.

Direct streaming from the virtual viewer via an application such as VLC would be possible, but we wanted a more dynamic streaming experience with the possibility of selecting different streams to watch based on who was controlling a stream from the Bagadus App. By using the streaming engine optimized for streaming HTTP, which was developed by Martin Alexander Wilhelmsen in his thesis [73], using a web browser for playback is a great fit. A web application for viewing playback makes for an easy approach to watching streams from all kinds of devices.

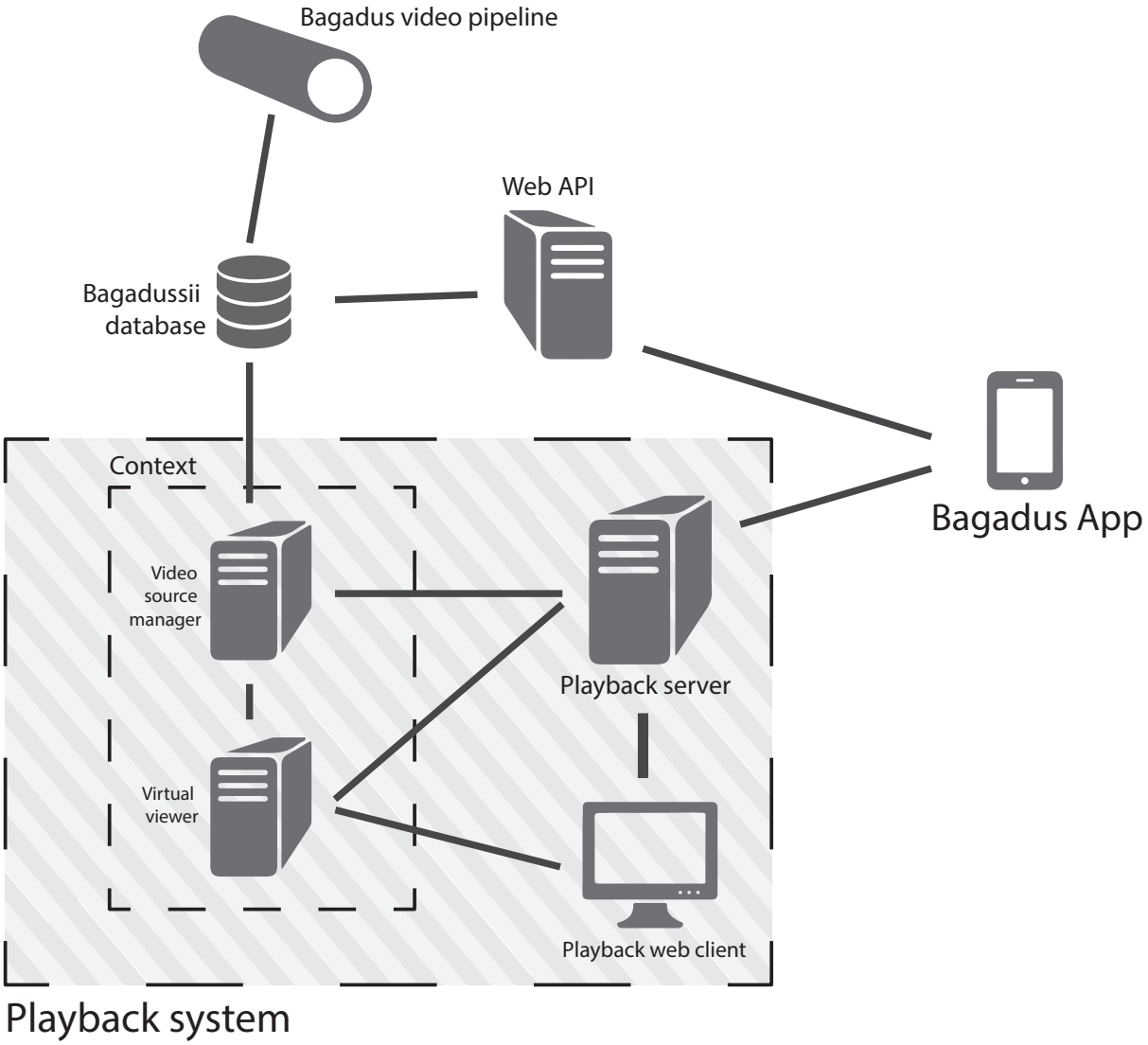


Figure 5.1: Overview of the playback system and surrounding components

5.2 Overview

The Playback System consists of several interacting components. Some are existing parts of Bagadus that we use unchanged, and some are created or adapted to our scenario. In particular, the Playback Server and the playback client components have been developed by us, while the Web API and the Video Source Manager have been developed and adjusted in accordance with our requirements with help from members of the Bagadus project. Figure 5.1 shows an overview of the components and their interactions.

The following is a list of components in the Playback System:

Bagadussii database The database containing all information regarding events, players and video.

Web API The web API towards the Bagadussii database which allows the app to query for and post notational data.

Virtual Viewer The virtual viewer described in [17, 73], which provides virtual views that are controllable through a WebSocket.

Video Source Manager A wrapper around the virtual viewer connecting event data from the database to the viewer as well as extending the viewer by providing functionality to alter the input source.

Playback Server A server that allows video control from the app as well as handling playback of video to web clients.

Web playback client The front-end web app controlled by the Playback server allowing clients to watch a virtual viewer stream.

5.3 Data layer

For retrieving video produced by the video subsystem of Bagadus, the Playback System connects to the Bagadussii database through the Web API. These are the same components the Bagadus App is dependant on for getting player and event data as well as registering notational data.

5.3.1 The Bagadussii database

The data storage of the Bagadus system is a PostgreSQL database, which handles persistent storage of players, event types, coaches etc. as well as video. While running, the video pipeline continuously outputs three-second segment long video clips. The paths to these files are saved to the database. When a registered event is inserted into the database, the database generates a view by pairing video segments with the timestamp from the registered events. Figure 5.2 shows an example of such a view, where `event_id` is the registered event, `key` is the ID for each video sequence and `file_uri` the path to the file. This is how the Bagadussii database connects notational data with the video subsystem.

event_id	key	file_uri
107	15027	/media/til_recordings_2014/files/2014-09-28_18/0950_2014-09-28_183230.202898.h264
107	15028	/media/til_recordings_2014/files/2014-09-28_18/0951_2014-09-28_183233.202706.h264
107	15029	/media/til_recordings_2014/files/2014-09-28_18/0952_2014-09-28_183236.202662.h264
107	15030	/media/til_recordings_2014/files/2014-09-28_18/0953_2014-09-28_183239.202725.h264
209	112507	/media/til_recordings_2014/2013-11-03/first/0035_2013-11-03_18:01:12.794293000.h264
209	113451	/media/til_recordings_2014/2013-11-03/first/0036_2013-11-03_18:01:15.793287000.h264
209	113452	/media/til_recordings_2014/2013-11-03/first/0037_2013-11-03_18:01:18.793282000.h264
209	113453	/media/til_recordings_2014/2013-11-03/first/0038_2013-11-03_18:01:21.793325000.h264
209	113454	/media/til_recordings_2014/2013-11-03/first/0039_2013-11-03_18:01:24.793370000.h264

Figure 5.2: A view from the database showing the connection between registered events and video from the Bagadus pipeline.

The database provides access control depending on the user logged in, such that, e.g., the head coach of TIL's user only has access to information regarding his team, like players, registered events and video. This is done by generating database views based on the requesting user's unique key.

5.3.2 Web API

In order for the Bagadus App to communicate with the data layer of the system, a web interface to the Bagadussii database is needed. The Web API is a Representational State Transfer (REST) API built with Python, with endpoints after our specification. The Web API is responsible for authenticating users of the Bagadus App, as detailed in section 4.8, and providing the app with data.

The API queries the Bagadussii database for information using the token that the client provides. The token authenticates the user to the database, and only relevant information to that user is returned. All information is returned in JSON format, which is easily parsed on the app, due to it being developed in JavaScript. Information is exchanged through HTTP GET and HTTP POST verbs. All communications to the API is encrypted with Transport Layer Security (TLS), to avoid send the sensitive authentication information in clear text.

The Web API has the following communication endpoints for serving clients:

/ Authenticate a user. Being authenticated is needed for all other endpoints.

/players Returns a list of players.

/events Returns a list of event types to register annotations of. This endpoint also supports POSTing new event types for saving to.

/register-player-event Register a player event, i.e. a combination of an event, a player and a timestamp.

/register-group-event Register a group event, i.e., a combination of an event, a group and a timestamp.

/registered-events Returns a list of registered events, i.e., a combination of a player or group, an event, a timestamp and other metadata such as thumbnail images of the event subtracted from panorama video of the event.

/registered-event-mark Marks a registered event as relevant for playback.

/registered-event-unmark Unmarks a registered event as relevant for playback.

/time Returns a Unix epoch timestamp.

5.4 Virtual Viewer

In section 2.1.4, we briefly introduced the Virtual Viewer. It allows for generating personalized, virtual views from a stitched panorama. The raw video files of the entire $4450x2000$ panorama capture from a full 90 minutes soccer match requires more than 6GB compressed H.264 video, and must therefore be further compressed to be able to stream over an Internet connection without buffering delays. In addition, a high-end CPU or GPU is required to decode the large stream. It may as well be more productive for analysts to be able to focus on specific areas of interest instead of being restricted to a panoramic overview of the entire pitch. Thus, a solution is to execute the delivery pipeline server-side, streaming highly custom views to the client by pairing a streaming engine with the virtual view generator [19] as presented in [17, 73]. The video delivery system is designed to take OpenGL input from a source application and stream it via HTTP to clients, while sending user input back to the source application with minimal latency. The pipeline is shown in figure 5.3.

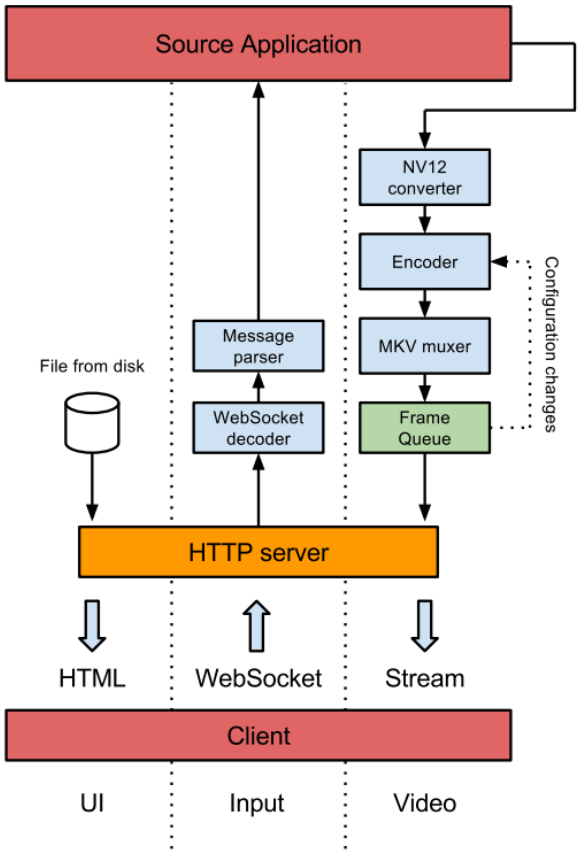


Figure 5.3: Block diagram of the delivery pipeline

In our use case, the Virtual Viewer acts as the source application, passing on OpenGL output to the delivery pipeline, which converts the output to YUV, encodes it using NVENC - Nvidia's H.264 hardware-accelerated GPU video encoder - muxes it into the MKV multimedia

format and streams it to the client over HTTP. At the same time, the delivery system also forwards commands to the viewer from a client. The Virtual Viewer provides an HTML web interface for streaming and controlling the video. The client streaming video and the client controlling the virtual view does not necessarily have to be the same, as we show in section 5.6.

⟨*message*⟩ ::= ⟨*command*⟩

⟨*command*⟩ ::= ⟨*coord*⟩ (‘=’ | ‘>’ | ‘<’) ⟨*value*⟩ ;*

⟨*coord*⟩ ::= ⟨*xcoord*⟩ | ⟨*ycoord*⟩ | ⟨*fcoord*⟩

⟨*xcoord*⟩ ::= x

⟨*ycoord*⟩ ::= y

⟨*fcoord*⟩ ::= f

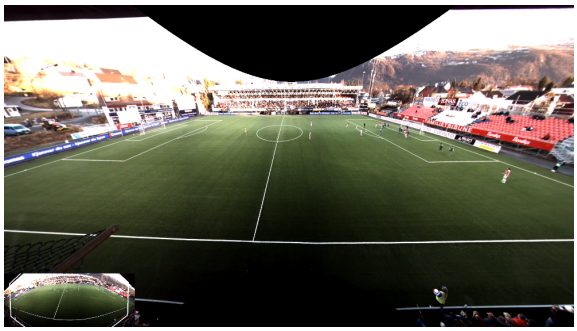
⟨*value*⟩ ::= [0-100]

Figure 5.4: Grammar for controlling stream through a WebSocket

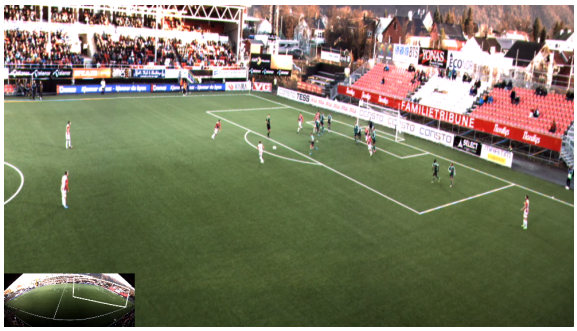
Client commands are handled through a WebSocket connection and parameters passed from a client when opening a new stream. The client, e.g., a browser, performs an HTTP GET request to the system, passing along a unique ID it generates itself, as well as parameters for bit rate, resolution and the size of the preview thumbnail in the stream, e.g., `/stream/4e06703610ab0a59?width=1920?height=1080?rate=5500`. The delivery system then returns a continuous stream matching the client's specifications, which the browser progressively downloads and plays through a HTML5 `<video>` tag. The WebSocket connection to the delivery system is used for controlling the virtual view, i.e., panning and zooming in the video. Figure 5.4 shows the grammar for the available actions. All coordinate values are normalized from 0 to 100, where x is the horizontal value, y is the vertical value and f is the zoom value. Setting the x value to 0, will pan the view to the far left, and setting it to 100, will pan the view to the far right. As shown in figure 5.5a, having the x and y value both set to 50, will center the view.

Each stream from the Virtual Viewer is independent, meaning each virtual camera view can be controlled separately from any other view. This is done by having a dedicated *stream socket* - i.e., a WebSocket connection - for every video stream. Figure 5.5 shows an example

of this, with two views simultaneously generated from the same panorama video source, but with different view coordinates.



(a) x=50, y=50, f=0



(b) x=80, y=20, f=30

Figure 5.5: Two different virtual views generated from the same panorama video source. The preview thumbnails in the lower left corner of the images show the virtual views in the context of the panorama.

5.5 Video Source Manager

The Video Source Manager is a manager for the Virtual Viewer. This component is responsible for providing source input to the Virtual Viewer, i.e., stitched panoramic video, as well as starting the viewer, and thus acting as a wrapper. As the viewer simply takes continuous panoramic video to generate virtual views of and pass on to clients, there is no functionality for modifying the source input.

For the end-user of the Bagadus App to be able to manipulate video playback, a source manager for the viewer is needed. The Video Source Manager implements functionality for this by providing an API through a WebSocket connection, so that a client can control and manipulate the source input to the Virtual Viewer.

As described in section 2.1, stitched panoramic video from the video subsystem pipeline is saved to disk continuously in three-second, raw H.264 video segments while the camera rig is recording. Paths to the stored video is saved in Bagadussii database. The Video Source Manager fetches video files from the database, based on what registered events to play, and feeds these to the viewer as source input. The Video Source Manager loops the input until a new command is received. Figure 5.6 illustrates the flow of such an operation.

The following is a description of functionality the wrapper provides.

Go to event Play video associated with an event. See figure 5.6 for details on the operation.

Pause/unpause/toggle pause Pause, unpause the video playback. When pausing, the manager feeds the same frame continuously to the viewer. When unpausing, the

manager will continue playback of an event.

Rewind event Skip to the beginning of the current event.

Skip forwards/backwards Skip two seconds forward or backward in the current event. Playback can be paused or playing.

Next/previous event Skip to the next/previous event in the playlist. A playlist contains events that have had a special flag set. This feature is discussed under section 4.5.

More/less head or tail Trim or extend an event by three second segments. This is due to the video pipeline outputting video in segments of this length, as described in section 2.1. Head represents the start of the event and tail the end. For example, an *extend_head* command will prepend the event length with three extra seconds of video.

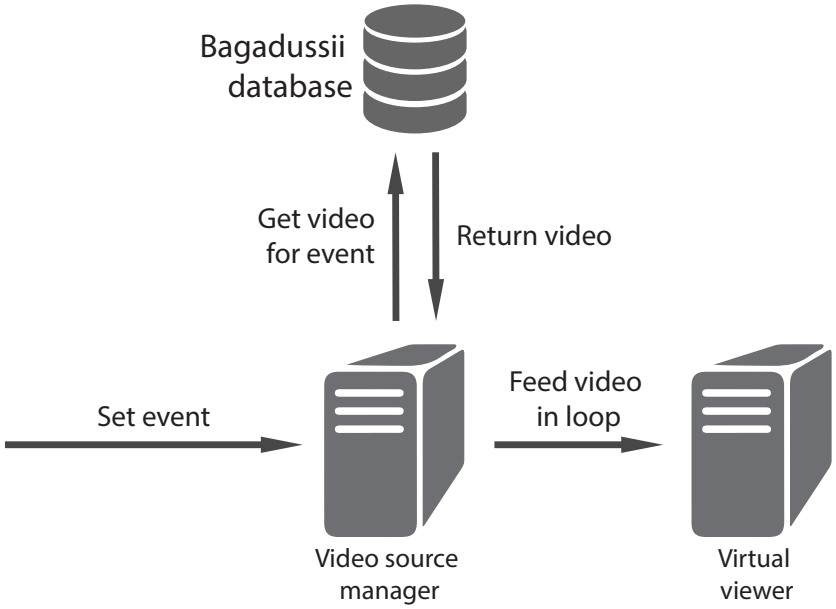


Figure 5.6: Command to wrapper to set event to play. The manager retrieves video from the database and pipes the video in a loop to the viewer.

An instance of the Video Source Manager and an instance of the Virtual Viewer runs in the same context, as the manager executes the viewer as a sub-process and provides video to it. As such, all virtual views are generated from the same panorama video, and any change in the source input will propagate through all clients streaming from the Virtual Viewer. To stream from multiple video sources concurrently, additional manager/viewer instances must be started. Each instance requires two successive ports - the first for the Virtual Viewer and the second for the Video Source Manager.

5.6 Playback Server

In this section, we will explain the Playback Server and it's role in the Playback System. An overview of the Playback Server the components it interacts with is shown in figure 5.7. The server acts as a layer between the Bagadus App, the Video Source Manager and the Virtual Viewer, as well as serving HTML/CSS/JavaScript files to the playback web clients. The app works as a controller of streams and video source data, deciding on what the Virtual Viewer should serve in terms of video streams from registered events, panning and zooming in virtual views, as well as drawing on top of video streams. Client instances of Bagadus App will be for clarity's sake referred to in this chapter as *controller clients*, and similarly, client instances of the playback web client are referred to as *playback clients*.

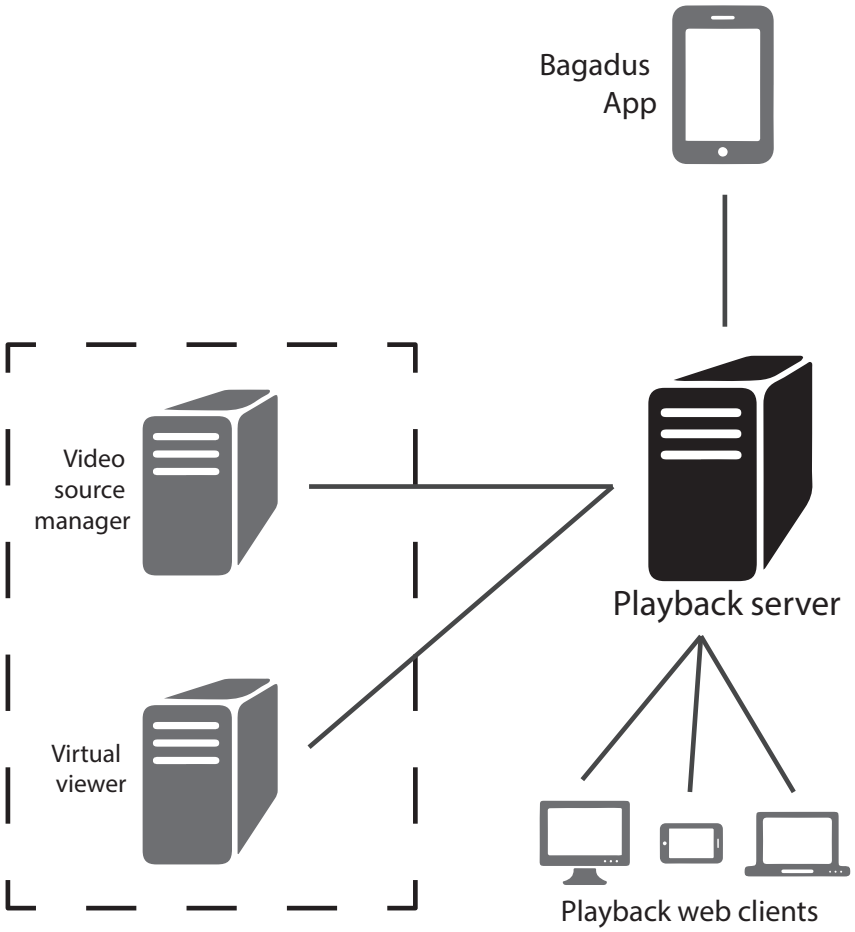


Figure 5.7: Overview of playback server and the components it interacts with

To connect the different components in the Playback System with the Bagadus App, we needed a unified API for it to interact with. A mobile device can be limited in terms of network stability, as our tests in chapter 4 showed. If the app were to connect directly to the Playback System components, without the Playback Server, several connections from the app would need to be established and kept open. This leaves much of the Playback System's functionality prone to failure, as the mobile device is the least reliable component

in the system. The app could crash due to a device failure, the device could lose its network connection, run out of battery etc., losing the current state of the system. A safer and more stable alternative is to have an intermediate interface, in the form of the Playback Server, between the Bagadus App and the rest of the Playback System. By having the Playback Server maintaining the current state of clients and components, we minimize the risk of incorrect states between the Playback System components and clients, as well as the amount of connections the app must keep open, and thus the amount of data transmitted through a possibly unstable mobile network connection.

As such, the app keeps one WebSocket connection open to the Playback Server and makes direct requests to the Web API as needed. We investigated proxying Web API requests from the app through the Playback Server as well, to eliminate all other network connections needed, but concluded with it being unnecessary, as the Web API is a REST API and thus stateless. Direct contact with the Web API also allows the app to register notational data even if the Playback Server is down.

The following is a list of needed features and criteria for the Playback Server in order to achieve the requirements of the Instant Replay Analytical Subsystem:

Controlling playback clients The server should provide a means for clients to stream video, controlled by the app, from the Virtual Viewer.

Playback of registered events The server should provide the app with the needed functionality for playing back video from notational data to clients via the Video Source Manager.

Create and manipulate virtual views The server should keep the current state of playback clients and streams, as well as provide an interface for the app to manipulate virtual views, e.g., by panning or zooming.

Draw on virtual views The server should accept draw commands from the app, which it should relay to the appropriate playback clients. The playback clients should superimpose the drawing on a virtual view video.

Near real-time Response time should be as close to real-time as possible when using the app as a remote controller for virtual views. This includes actions such as changing video source (i.e., switch event), changing a virtual view perspective or drawing on video. A major delay in these actions would make response and efficient analysis more difficult - especially in time critical situations such as half-time analysis.

5.6.1 Selected technology

The server is built upon the Node.js[1] platform with the ExpressJS[2] framework. Node.js was chosen for being open-source, cross-platform, as well as being based on JavaScript. By using JavaScript, the code base of the server and the Bagadus App are thus based on the same language, making for easier and more consistent development between the two components. ExpressJS is a web application framework simplifying the process of building web APIs, handling client requests and serving static HTML files for the playback clients. All communication with Bagadus App and with playback clients happens over a WebSocket connection working underneath the Socket.IO library. We chose Socket.IO instead of a simple WebSocket implementation for it's extended functionality for separating connected clients into namespaces, unification between client and server side APIs as well as included reconnection logic, leading to a less error-prone and time-consuming implementation of the WebSocket protocol than using a more naive and manual approach.

As Socket.IO is a custom implementation of the WebSocket protocol, it cannot communicate to clients running standard WebSocket implementations that purely supports the RFC 6455 specification [29]. The Virtual Viewer and the Video Source Manager uses a standard WebSocket implementation, and thus we use the *ws* [74] WebSocket implementation library for Node.js for communication between the Playback Server and these components. We chose the *ws* library as it conforms to the IETF specification and benchmarks very well in terms of speed [75].

The Playback Server acts as a communications hub, directing commands from the Bagadus App to other components in the Playback System, as well as providing callbacks to the app. All connections to and from the Playback Server are handled over WebSocket connections. The event-driven nature of Node.js couples nicely with the also event-driven WebSocket protocol, making for a more consistent implementation. By listening to incoming events and emitting events to clients over persistent sockets, the Playback Server keeps the real-time nature of the Bagadus system intact, as well as accurately maintaining client's states.

5.6.2 Handling clients and state

The Playback Server keeps an updated list of all clients, both controller and playback clients, connected to the server. Connections to controller and playback clients are managed using the Socket.IO WebSocket implementation, allowing for bidirectional communication.

When the Bagadus App successfully connects to the server, it passes its credentials, i.e., ID, name and resolution, to the server which stores the socket connection and credentials

[1] https://nodejs.org/
[2] http://expressjs.com/

in an in-memory data layer. The server then notifies playback clients of the connected controller client. The credentials are used by the server and the playback clients to identify the controller client. The resolution of the mobile device is needed for generating a screenshot for the drawing action described under section 4.6 and section 5.6.4.

The Playback Server serves the HTML files for the playback clients. When a client is done loading the interface, it connects to the server, where likewise as the controller clients, the socket connection and credentials are saved as objects in-memory. The playback client can select a controller to follow, i.e., listen to a controller's commands, by sending a `join_controller` message and including a controller ID in the payload. The server then registers the client to the specified controller's namespace.

5.6.3 Video playback

In the initial implementation of the Playback Server, video playback was based on static, rendered video. Annotations registered by the Bagadus App through the Web API returned 16 second video clips in three angles: the full panorama, the home team's penalty area and the away team's penalty area. Playback clients would stream video directly from the Web API, by progressively downloading the video clips and playing them. The downside to this approach is the restrictions in manipulating the video. To change the view angle, the client would need to download a new clip, and wait until it is finally downloaded before seeking everywhere in the video is possible.

One of the functional requirements for the system were that manipulation of video should be possible, e.g., pause, skip, extend and trim video. With the initial approach of serving exported video clips to clients, this would be a challenging task. If a user would want to, e.g., extend the video clip with three more seconds of footage at the end, a completely new video clip would need to be exported and served to clients. Instead, we opted to use the Virtual Viewer described in section 5.4, to enable this functionality, and generally give a better control over video playback.

Using the Virtual Viewer for controlling video streams

To use the Virtual Viewer to generate streams, we need a means to handle multiple streams controlled from the Bagadus App. In the Virtual Viewer implementation described under section 5.4, the web interface is both the receiving part of the stream, as well as the one controlling the virtual camera movement of the stream. For the Instant Replay Analytical Subsystem, these two connections are handled separately, where the playback client is a simple web client streaming video, and the app acts as a the controller of the virtual view. This is done by having the Playback Server maintaining the state of video streams.

Figure 5.8 shows the flow when initiating streaming:

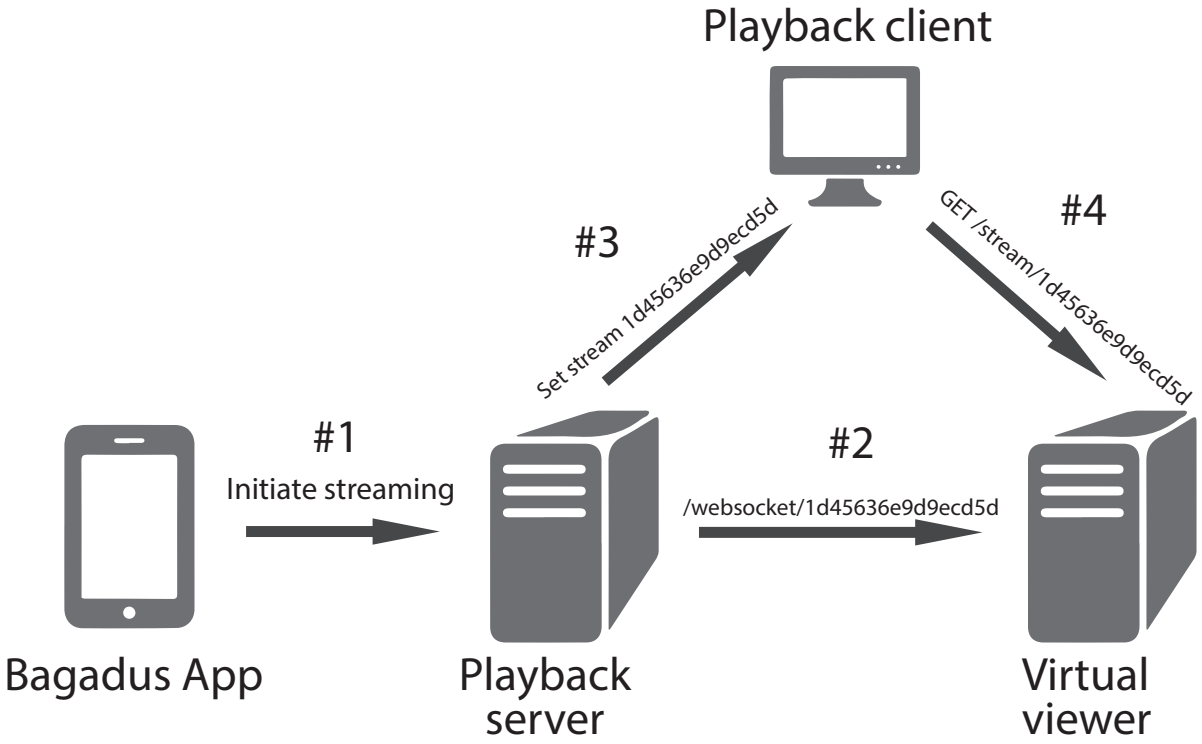


Figure 5.8: Start streaming command with one controller client and one playback client following that controller

1. A command to start streaming is sent to the server from the app.
2. The server generates an unique stream ID for the playback client that is following that controller and opens a stream socket, i.e., a WebSocket connection, to the Virtual Viewer to control it.
3. When the socket connection is established, the server notifies the playback client that it can open a stream to the viewer with the included stream ID.
4. The playback client requests it's own individual stream from the Virtual Viewer when it receives the stream ID from the playback server. This allows each playback client to select the bit rate and resolution for the stream based on the client's own configuration. For example, streaming on a mobile device over a 3G connection would need a smaller resolution and bit rate than a Full HD monitor.

Playback clients are grouped in namespaces by controller client ID. This allows the server to separate playback clients by what controller client they are following, and reduce unnecessary data traffic by not having to notify all connected clients. A playback client can easily swap which controller client to follow as it is always served an updated list of available controllers. Figure 5.9 illustrates multiple controlling clients with multiple playback clients following their namespace. It is also possible for a client to always listen to a specific controller ID, even though the controller client is inactive. This has the advantage of

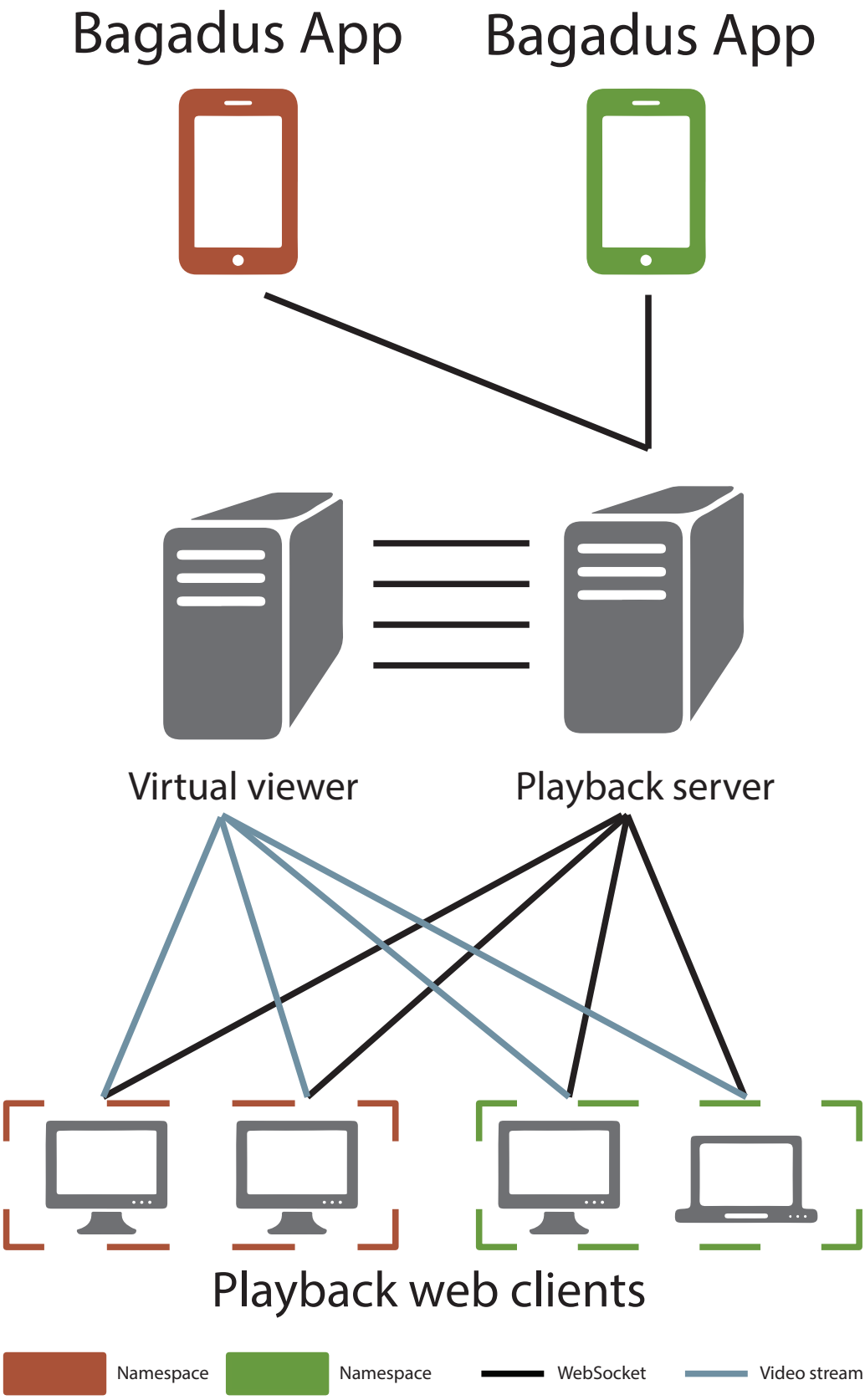


Figure 5.9: Multiple mobile devices controlling their own namespace of playback clients

permanently locking a playback client to a specific controller device, e.g., having a playback client in the locker room always listen for controlling from the head coach's mobile phone. How this is achieved is further described under section 5.7.

A command from the app centered towards controlling the virtual camera view, e.g., zooming in, is parsed, cached on the server, and forwarded to all playback clients following that controller device. Other connected playback clients are unaffected. When the controller sends a stop stream request, the server kills all stream socket connections to the viewer and notifies all playback clients currently streaming that streaming should cease.

Controlling video source input

To change or modify source input to the viewer, commands must be sent to the Video Source Manager running in the same context as that viewer. A single connection is maintained through the *control socket* - a standard WebSocket connection. It should always be open as long as the playback server is running. If the connection is lost, the playback server will repeatedly try to reconnect until a new WebSocket connection to the Video Source Manager is established. Controller clients are denied managing of video playback when a control socket connection is missing, as the video controlling is fairly useless without being able to manipulate video input.

Commands from a controller client is parsed and translated into valid messages, as described in section 5.5, for the Video Source Manager and then sent over the control socket. Video input to the viewer thus changes, which is propagated through to all playback clients streaming from that virtual viewer context.

5.6.4 Drawing on video

A functional requirement of the Instant Replay Analytical Subsystem is to be able to draw on video. We investigated different approaches to handling this. The choices regarding the Bagadus App were limited and thus the choice was simple; hybrid apps are created with HTML and the best approach to drawing with HTML is the canvas element, as discussed in section 4.6. Drawing on video can be achieved through several means: e.g., directly manipulating video, i.e. draw pixels on the video, or superimpose the drawings on the video through other means. However, it is dependant on the implementation of the playback client, i.e., if the client should be able to support drawing on top of video in some way or if the drawing must be done directly by altering pixel data in the video. As we designed and implemented a browser-based playback client, we achieved this with an HTML `<canvas>` element - the same technology used for capturing drawing from the app. The design choices of the playback client are further discussed under section 5.7.

For a convenient draw operation, users should be able to see a frame of the view they are drawing on. To achieve this, the Playback Server provides the app with a frame of the current streaming video for display in the app. Upon first connecting to the server, the app sends its display resolution, which is then cached on the server. When the app sends an `initiate_drawing` command, the Playback Server sends a `pause_stream` command to the Video Source Manager to ensure playback is paused, and performs a HTTP GET request to the Virtual Viewer with the resolution of the controller client as an argument. The Virtual Viewer returns a video stream matching the resolution. As playback is paused, the video stream consists of the same frame continuously looped. When a virtual view is opened, view coordinates for that view are defaulted to `x=50, y=50, f=50` (see section 5.4). If the controller client has changed this perspective before initiating the draw action, this virtual view will have the wrong perspective in comparison to playback clients also streaming. Thus, when opening a stream for grabbing a single frame for drawing, the server must also open a stream socket connection to send the current perspective coordinates.

To extract a frame from the video stream, we use *ffmpeg* [76]. ffmpeg is a command line tool for transcoding multimedia files, with support for HTTP streams. The video stream is piped to the an instance of ffmpeg as a child process of the server, which grabs the first frame of the stream and saves it to a temporary Portable Network Graphics (PNG) file. When the process is done, a callback is invoked. The resulting file is then read into a buffer and transmitted over the WebSocket connection to the controller client as a base64 encoded image.

This solution is not optimal, and introduces unnecessary overhead. Both opening a video stream and a corresponding stream socket, as well as running ffmpeg is needed to just extract a single video frame. To avoid a possible pipe deadlock when piping both stdin and stdout from the ffmpeg subprocess to the server process, the image is temporarily saved to disk as well. Figure 5.10 illustrates the considerable latency induced for capturing a frame. Note that the time measured is just the server time, and does not take into consideration any possible additional latency between the app and the Playback Server. As such, the estimated time for when a user requests the image until the app has received it is at minimum 6,5 seconds.

The Playback Server accepts draw commands from the controller clients, and pass these on to playback clients. The state of a drawing, i.e., its possible undo and redo actions, are handled by app and the web clients themselves. Thus, all that is required from the Playback Server, is that it relays draw commands from the controller client to the playback clients listening to that controller.

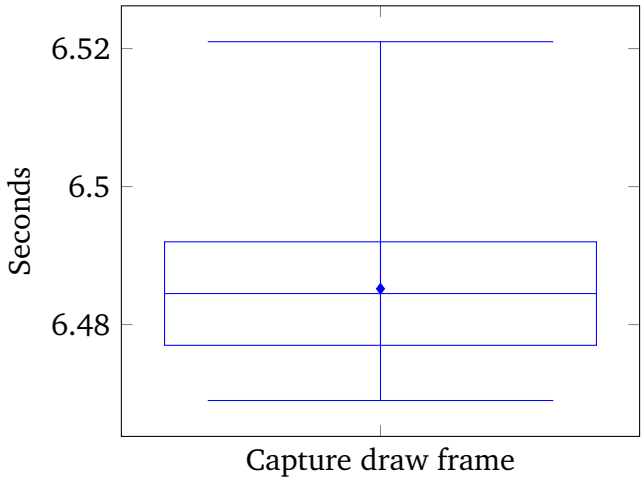


Figure 5.10: Latency induced when capturing frame for drawing from app. 30 tests were performed.

5.6.5 Time and performance

The Playback Server is designed to be hosted on the same network as the rest of the Playback System, making network latency between local components negligible. All data storage is handled in memory, eliminating common latency-inducing actions such as I/O to a database. A possible bottleneck could a high amount of concurrent WebSocket connections, e.g., many playback clients following a controller client. As controlling playback should be as responsive as possible, we have tested how much latency the Playback Server contributes to the command flow for opening virtual views streams to playback clients. The latency measured is from when the server receives an incoming request to initiate streams, until all necessary video stream objects have been instantiated and the process of opening individual WebSocket connections to each client is started. Note that this is the processing time on the server, and does not include any network latency.

The tests were performed on a machine with an Intel i7-4820K processor running at 3.7GHz and 16GB of memory. The Virtual Viewer and Bagadussii database are running on the same machine. The results, as presented in figure 5.11, shows that the server induces a small latency to the command flow when initiating streams. This is likely due to the number of video stream WebSocket connections that need to be instantiated and opened at the same time. In a possible future setting with thousands of concurrent playback clients, e.g., with sport fans streaming from home, some additional optimization may need to be introduced, but for the current implementation directed at internal analytical use with likely considerably fewer than 100 clients, we found the added latency to be negligible. Furthermore, the latency is the maximum delay added to the flow, i.e., the elapsed time before *every* WebSocket connection request has been instantiated, and connecting has started. Many WebSocket connection requests, depending on the network latency, may

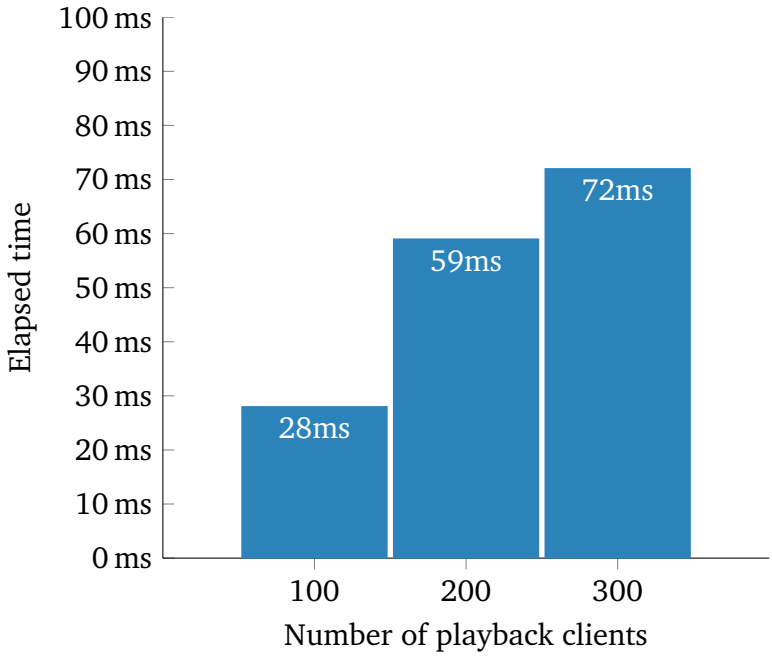


Figure 5.11: Server latency on initiate stream command

have been opened by the end of the test.

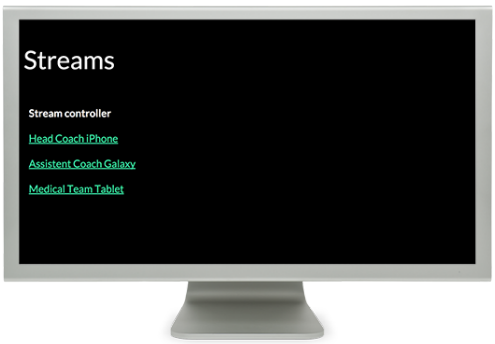
5.7 Web Playback Client

In order to present video playback, we created a HTML based web application optimized for Google Chrome. The Virtual Viewer provides H.264 encoded video over HTTP. Any software with support for HTTP streaming and decoding this format is possible to use, e.g., VLC. However, using a browser allows for easier handling of playback, i.e. changing stream source, as well as communicating with the Playback Server. Google Chrome produced the best results [73] for streaming via the delivery pipeline that the Virtual Viewer uses, and is thus the reason for our browser choice.

5.7.1 Design

The user interface of the playback client is designed to be minimal and simple to use. The only user interaction needed is to select what controller to follow. It is also possible to lock the client to one specific controller. This can be beneficial for clients that will always be listening to the same controller device, e.g., a client using the stadium big screen as a monitor can be set to permanently listen to commands from the Bagadus App on the head coach's mobile device.

The playback client files are served from the Playback Server. When a browser connects to the web page and the files are loaded, a WebSocket connection through Socket.IO is



(a) A list of available controllers to follow

(b) Following a controller - not streaming video

(c) Following a controller - video is streaming

Figure 5.12: The three available views for a playback client. The browser is running in full-screen mode for the cinematic effect.

established between the server and the client. The server is notified that a playback client is connected, and passes back a list of available controllers to follow. This list is presented to the user, as shown in figure 5.12a. The list displays the user name of the controllers. By clicking on a user name, the user is redirected to that controller's stream page, as displayed in figure 5.12b. When the page is loaded, the client sends a notification to the server over the WebSocket connection that the user is now following said controller. If the controller is currently controlling video playback, the Playback Server generates a stream ID, opens a WebSocket stream to the Virtual Viewer, sends the current virtual view perspective, and

notifies the playback client that it should connect to the Virtual Viewer with the generated ID. The client performs a HTTP GET request to the viewer, injects the response stream into an HTML `<video>` element and starts playback. Figure 5.12c shows the playback client when streaming video.

5.7.2 Drawing

When a user is drawing from the app, the Playback Server forwards the draw coordinates to the playback clients. These coordinates are relative to the display resolution of the source mobile device, and must thus be scaled to fit the display resolution of the playback client. When the Bagadus App enters the draw state, it notifies the Playback Server, which in turn emits the app's display resolution to playback clients listening to that controller. Each playback client then calculates the scale factors C_w and C_h between its own, individual resolution w_p and h_p and the app's resolution w_a and h_a as shown in equation (5.1). The scale factors are saved in the browser's memory for the rest of the session.

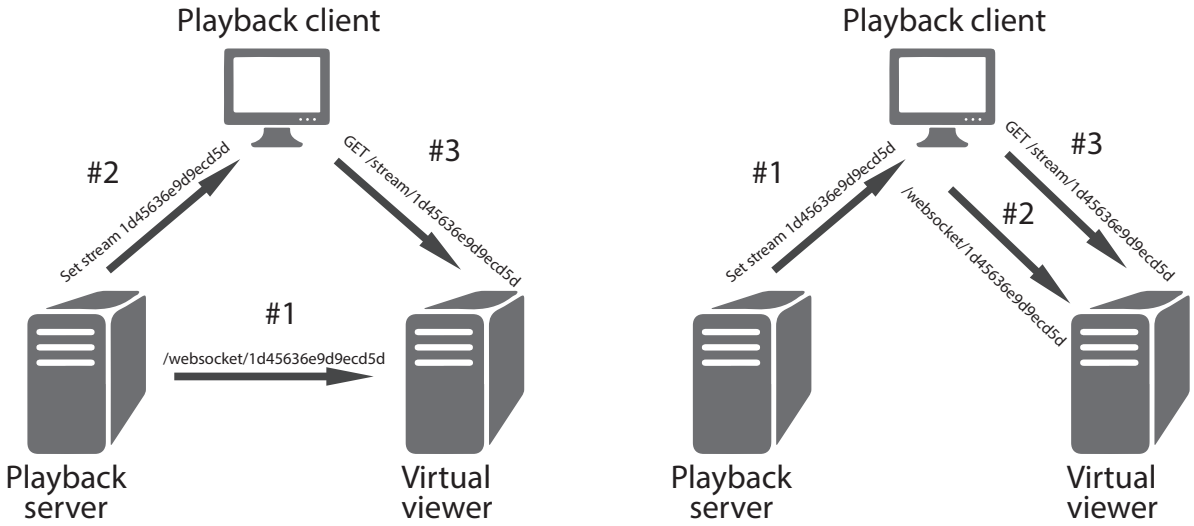
$$\begin{aligned} C_w &= \frac{w_p}{w_a} \\ C_h &= \frac{h_p}{h_a} \end{aligned} \tag{5.1}$$

When the playback client receives a draw coordinates, it multiplies the scale factors with the start and end coordinates, to get new, scaled coordinates that fits its display resolution. As the `<canvas>` element does not support scaling drawn graphics when resizing the browser window - the canvas information is simply erased - we have implemented functionality for this for added robustness. A listener fires an event when the browser window is resized, and calls a function that saves the current canvas as a PNG image, scales it according to the new canvas size, and draws it back to the canvas. However, if downscaling, the draw lines can become very thin and imperceptible. Thus, a simple algorithm calculates a line width to 2 thousandths of the width and height of the canvas combined and applies it. The line width is set to a minimum of 1 pixel as to be discernible.

Section 4.6 describes how the Bagadus App saves the drawing state in order to enable undo and redo actions. Each time the app saves the current state, it emits a `save_draw_state` message to the Playback Server, which forwards these to playback clients. Just as the app, the client keeps two last-in-first-out arrays for saving the drawing state: *undo array* and *redo array*. When the playback client receives this message, it saves the current drawing as a base64-encoded PNG image to the undo array in the browser's memory. When the app performs a redo or undo action, a message is emitted to the clients through the server. For, e.g., an undo action, the client saves the current state to the redo array, pops the top image off the undo array and inserts it in the `<canvas>` element.

5.8 Evaluation

During the design phase of the Playback System, we focused on eliminating additional network routes and keeping the playback client as simple as possible. Thus, we moved the responsibility of controlling video sockets from the playback clients to the Playback Server as shown in figure 5.13a. In the original front-end implementation for the Virtual Viewer, a client streaming video also controlled the video socket. During testing of the Playback System, we discovered the benefit of this. The Virtual Viewer keeps a buffer of ready frames to stream to clients. The bigger the buffer gets, the further behind in streaming the client is. By pinging the Virtual Viewer through the video socket, the Virtual Viewer responds with the latency the client is behind real-time in streaming. The client can then adjust the playback rate of the video to catch up buffered frames, and thus be closer to real-time streaming. In our implementation, this functionality is not as feasible anymore, as the Playback Server opens and controls the video sockets. Thus, the playback clients have no way of catching up to real-time if the stream falls behind.



(a) The current implementation

(b) An alternative implementation

Figure 5.13: The current way of controlling the video socket through the Playback Server, compared to an alternative way where the video socket is controlled by the playback client.

A possible change is to have the Playback Server constantly notifying the playback clients to increase or decrease the playback rate based on the amount of buffered frames, but this puts quite a bit of extra load on the Playback System if many playback clients are streaming. A better solution might be to move the control of video sockets back to the playback clients as shown in figure 5.13b, and route all video control commands from the app, i.e., panning, tilting and zooming, through the playback clients to the Virtual Viewer instead of directly from the Playback Server to the Virtual Viewer. This will add an extra network step, but seeing as playback clients must have a reasonably good network connection, as video

streaming is bandwidth demanding, it might not matter much.

By performing a test, we showed that the Playback Server can handle hundreds of concurrently connected playback clients and simultaneously instantiate and open stream sockets for these when streaming should commence. Furthermore, we displayed that the server can handle this with minimal latency introduced. This shows that even though the current implementation of the system will presumably be used with one controller client and at maximum a few playback clients concurrently, it can handle hundreds of simultaneous playback clients and multiple controlling clients.

In our implementation, there is only one context of the Virtual Viewer and Video Source Manager running concurrently. In effect, this means that all controller clients are manipulating the same video source at once, which may lead to confusing situations if several simultaneous clients are, e.g., changing events and skipping in video at once. For the current use of the system, i.e., for half-time breaks and during practice sessions, this was deemed to not be an issue, as there will likely only be one controlling client manipulating video at any time. However, with additional adjustments, the Playback Server could easily handle communicating with multiple contexts by connecting each context with a controller client namespace. For smooth operation of several simultaneous instances of the Virtual Viewer and the Video Source Manager, distributing to additional computers may be necessary.

The current process of generating an image for the app to draw on is slow and inefficient. Opening a video stream, a stream socket, emitting the current virtual view perspective coordinates to the stream socket, saving the image to disk and then read it from disk for transmitting to the app is a costly process. A likely much more efficient solution would be to extend the virtual viewer to extract and return a single frame through a HTTP request with the client's specifications. As the virtual viewer is designed for efficient video processing and can handle hundreds of individual virtual views at once, we expect that generating a single virtual view frame from panoramic video already in memory will be a rapid and non-disruptive process.

Figure 5.14 shows the Instant Replay Analytical Subsystem in use during a test in the dressing room at Alfheim, where a laptop connected to the TV monitor shows the playback client and an iPad where the Bagadus App is installed is used for as a remote controller. The users were thrilled about the prospect of using the system in the future as it provided an easy and efficient way to preform analysis. Installation and tests of the system is also planned to take place at Ullevaal for the national team, in June, 2015.

(a) Playback client idle

(b) Playback client streaming

Figure 5.14: The Playback System in use in the dressing room at Alfheim

5.9 Summary

In this chapter, we have presented the Playback System as a part of the Instant Replay Analytical Subsystem for Bagadus. First, we presented the motivation for the system, detailing how it is necessary for supporting the functional requirements of the Instant Replay Analytical Subsystem. We then introduced an overview of the system, briefly outlining the components. The system utilizes existing components from the Bagadus system as well as components implemented by other participants in the project during this thesis. We then presented our main contribution to the Playback System: a playback server and client web interface for linking the modules in the system together. We showed that the Playback Server is necessary for tying the system together, minimize failures by moving state management from the Bagadus App to server and reduce the number of connections the app must keep open over a possible unstable mobile network connection. The playback client provides an easy and convenient manner for users to view streams controlled from the app. Finally, we evaluated our solution and discussed what could have been done differently to further improve the system.

Chapter 6

Conclusion

In this chapter, we summarize the work that has been presented in this thesis. We list our main contributions, and present possibilities and ideas for future work that can be done, before we make our final remarks.

6.1 Summary

In this thesis, we have implemented an instant analytical subsystem for Bagadus. The system is used for capturing sports notational analysis data and visualizing the data with video from the Bagadus pipeline.

In chapter 2, we presented an overview of the existing Bagadus system, and its different subsystems: video, tracking (sensor) and analytical. The video subsystem produces panorama video of five cameras by combining the individual feeds through a video processing pipeline. Another component, the Virtual Viewer, can produce virtual views from the panorama video. The tracking system captures accurate positional data of players on the pitch. The intended analytical subsystem, Muithu, supports registering notational data, but lacks integration with the Bagadus project. It uses its own camera setup, that requires a lot of manual labor to operate and to connect video with notational data. We discussed this and other limitations of using the Muithu system as the analytical subsystem for Bagadus.

Next, in chapter 3, we presented a design of a new instant analytical subsystem for Bagadus. The system is built around a mobile application, Bagadus App, and the Playback System. The former gives the user the possibility of registering notational data during a sports session, as well as controlling playback of virtual view video streams to clients through the latter. We discussed the functional requirements for the system, as well as introduced communication strategies for the components in the system.

In chapter 4, we detailed the design and implementation of the Bagadus App. We

discussed the different approaches to mobile application development, and concluded with a hybrid path being the most fitting for our purpose. An intuitive and easy user interface is crucial for the effectiveness of the app in use. Therefore, we performed a user test, which concluded with a click-based interface being the most efficient, as well as preferred, way to registering notational data. We also investigated the approach of using speech recognition to capture notational data, but concluded that more extensive research must be conducted in order for it to be a feasible solution. Other tests, regarding data usage, determined us to use the HTTP protocol for registering notational data, and the custom WebSocket protocol implementation Socket.IO for controlling playback of virtual views. In order to combat challenging network conditions, we implemented a caching functionality for the app. To achieve accurate timestamps for registered annotations, two approaches to synchronizing the mobile device time with the Bagadus system time were implemented.

Chapter 5 describes the Playback System - a system which bridges the gap between the Bagadus App and the Bagadus video subsystem. The need for this system was quickly identified, as a scalable, multi-user and direct approach between the app and the video subsystem was deemed unfeasible. We first described different components of the Bagadus system that the Playback System interacts and integrates with. To achieve the requirements of the Instant Replay Analytical Subsystem, the Playback Server was designed and implemented based on these. Combined with the Virtual Viewer and the Video Source Manager, it enables controlling of playback and manipulating of virtual views to multiple clients by multiple users of the Bagadus App. The Playback Server was found to be introducing minimal extra latency to the system.

6.2 Main Contributions

In this thesis, we have designed and implemented a new system for instant review and analysis of soccer sessions. We have shown how hybrid mobile development can be used for supporting multiple platforms with the same code base, but also identified some of its limitations in functionality, feel and responsiveness. Inspired by the principles of Muithu, we have constructed a new system consisting of the Bagadus App and the Playback System, which removes the manual labor of Muithu and enables near-instant playback with extended functionality.

The user now has the ability to annotate an event, and within seconds play back the associated video on the big screen at the stadium, in the dressing room, or at any other monitor, with full video control and drawing as an analytical aid - all though the Bagadus App.

Through user testing and surveys, we have shown that a drag-and-drop based interface is

slower and more prone to errors than a click-based interface for the purposes of the Bagadus App. The average speed of performing a task with the drag-and-drop interface was about 1.2 seconds, with a total correctness of 92%. The average speed using the click interface was 0.7 seconds, which is well within the maximum time limit we set at 5 seconds, and with a total correctness of 98%. A click-based interface was also preferred over a drag-and-drop interface by all the users who participated in our tests. By using a click based interface for the Bagadus App, we have provided a fast and accurate method for annotation.

We have compared the WebSocket and HTTP communication protocols, and conducted experiments by building two applications and two servers, one pair that used WebSocket for communication, and one pair that used HTTP. The results showed that our HTTP application produced less data traffic than the WebSocket application when the frequency of message exchange was low. In the test where the applications were configured to perform 1 posts every 5 minutes for 50 minutes, the WebSocket application produced 77% more data traffic than the HTTP application. While the application using WebSocket produced less data traffic when the message frequency was low, the HTTP application produced over 400% more data traffic than the WebSocket application in the high-frequency message test. Furthermore, we found that the application using WebSocket gave more stable results than the one using HTTP. In the test where we disabled HTTP keep-alive, the HTTP applications yielded a standard deviation of about 560% larger than the WebSocket application. In addition, we discovered that with a small message payload, the HTTP application was especially expensive in terms of payload size versus total size of data produced compared to the WebSocket application, and that as the size of a message payload increased, the two applications produced more equal results in regards to data traffic. We also observed that a WebSocket connection does not influence the battery life of mobile devices considerably. By using a WebSocket connection for the remote controller of the app, we have limited the latency and data usage, as well as provided a stable solution for controlling video.

With the Bagadus App, we have provided easy, fast and lightweight methods for performing sports analysis on multiple platforms. By combining the app with the new Playback System, we have delivered a scalable analysis platform for allowing multiple clients to stream video controlled by a mobile device. With the possibility of controlling and fine-tuning video sequences from the app, as well as enabling drawing for maximizing video analysis, we have made the Bagadus system available for easy consumption for the intended end-users. The system have been tested in use at Alfheim Stadium, where the users were more than pleased with the new opportunities that the new Instant Replay Analytical Subsystem provided.

6.3 Future Work

In chapter 4, we mention that the Bagadus App should be available through the respective app stores of each platform. Currently, the Bagadus App is not published due to delays in implementing the authentication on the Web API. We expect it to pass the Apple App Store review process, as we have followed their mandatory user interface principles.

There is currently no simple way of creating new event types or analyze notational data in detail outside of the app. A website using the Web API to present data and statistical analysis of notational data could be beneficial for both coaches and players. Players could review video from annotations registered on themselves and others. Social media functionality could allow a sports club to comment on and analyze notational data and video together. An admin user, such as a coach, could use the website to update event type information, player information and other data in the Bagadussii database.

Expanding the system to fan consumption is another possibility. Fans can act as playback clients following a controller, much as watching a TV broadcast. A more exciting possibility would be to release the app to fans, and allow them to make their own annotations - for academic reasons or simply for entertainment, adding a new dimension to consuming live sports. Through the aforementioned possible website, fans could view video of their annotations and export and share the video clips to other services such as YouTube.

The system is designed to be used to analyze and evaluate situations in soccer, but can also be used in other sports or, e.g., when choreographing a dance performance. Principles of the Bagadus system can also be a useful tool in other areas when you want to capture situations of value such as video surveillance, or in instructive or academic arenas such as surgery.

The Playback System currently only utilizes one running context of the Virtual Viewer and the Video Source Manager, as this was deemed sufficient for the current intended use, i.e., during training sessions and half-time breaks. However, the functionality of having multiple contexts controlled by the Playback Server is possible with some adjustments, e.g., by having a dedicated context for each controller client namespace. This can allow concurrent users of the Bagadus App to control video source individually without sharing the same instance of the Video Source Manager with other controllers, i.e., manipulating the same video source. Running multiple of these contexts may require distribution across additional computers.

6.4 Final Remarks

We developed the system mainly for the sports personnel at TIL, but we later also began discussions with the head coach of the Norway national soccer team, who had a different

vision of how he wanted to utilize the Bagadus App. While TIL mainly wanted to use the app in game situations for half-time evaluation and analysis, the sports personnel at Ullevaal saw it as a opportunity for use in player development during training sessions. TIL also came to this realization after a presentation at Alfheim demonstrating the functionality ordered by the staff at Ullevaal. The staff at Alfheim were really thrilled with the possibilities of the new system, stating that the "all-in-one" solution provided them with an easy and efficient way of doing game analysis. They also stated that it is crucial to be on the bleeding technological edge in soccer. In an interview featured in Norwegian Broadcasting Corporations television program, Schrödingers Katt [77], the director of sport at TIL said that he hoped Bagadus could help the team maintain their position in the top division of Norwegian soccer - and so do we.

Appendix A

Accessing the source code

Access to the source code can be given upon request.

Bibliography

[1] NTB. *Høgmo innfører videoovervåking*. 2014. URL: http://www.dagbladet.no/2014/10/31/sport/fotball/per-mathias_hogmo/landslaget/36015899/ (visited on 11/01/2014).

[2] Høynig Morten. *Inviterer Mourinho på studietur til Tromsø og TIL*. URL: http://www.dagbladet.no/2011/09/20/sport/eliteserien/tippeligaen_2011/teknologi/data_og_teknologi/18218503/.

[3] *Ever wondered what managers write on their notepads during games? Jose Mourinho shares the secret - Mirror Online*. URL: http://www.mirror.co.uk/sport/football/news/ever-wondered-what-managers-write-3005640 (visited on 03/24/2015).

[4] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vegard Berg-Johansen, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Håkon Kvale Stensland, and Pål Halvorsen. "Soccer video and player position dataset". In: *Proceedings of the 5th ACM Multimedia Systems Conference*. ACM. 2014, pp. 18–23.

[5] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Øystein Landsverk, Carsten Griwodz, Pål Halvorsen, Magnus Stenhaug, and Dag Johansen. "Bagadus: An Integrated Real-time System for Soccer Analytics". In: *ACM Trans. Multimedia Comput. Commun. Appl.* 10.1s (Jan. 2014), 14:1–14:21. ISSN: 1551-6857. DOI: 10.1145/2541011. URL: http://doi.acm.org/10.1145/2541011.

[6] ZXY Sport Tracking AS. *ZXY Sport Tracking*. 2014. URL: http://www.zxy.no (visited on 12/08/2014).

[7] Dag Johansen, Magnus Stenhaug, Roger Bruun Asp Hansen, Agnar Christensen, and P-M Høgmo. "Muithu: Smaller footprint, potentially larger imprint". In: *Digital Information Management (ICDIM), 2012 Seventh International Conference on*. IEEE. 2012, pp. 205–214.

[8] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. “Computing as a discipline”. In: *Communications of the ACM* 32.1 (1989), pp. 9–23.

[9] *Interplay Sports*. URL: http://www.interplay-sports.com/ (visited on 04/08/2015).

[10] *FOOTBALL services by Prozone Sports*. URL: http://www.prozonesports.com/subsector/football/ (visited on 04/01/2015).

[11] *Kiswe | Home*. URL: http://kiswe.com/ (visited on 04/01/2015).

[12] Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David KC Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, et al. “Bagadus: an integrated system for arena sports analytics: a soccer case study”. In: *Proceedings of the 4th ACM Multimedia Systems Conference*. ACM. 2013, pp. 48–59.

[13] Asgeir Mortensen, Vamsidhar Reddy Gaddam, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. “Automatic event extraction and video summaries from soccer games”. In: *Proceedings of the 5th ACM Multimedia Systems Conference*. ACM. 2014, pp. 176–179.

[14] Marius Tennoe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Hakon Kvale Stensland, Vamsidhar Reddy Gaddam, Dag Johansen, Carsten Griwodz, and Pal Halvorsen. “Efficient implementation and processing of a real-time panorama video pipeline”. In: *Multimedia (ISM), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 76–83.

[15] Simen Saegrov, Alexander Eichhorn, Jorgen Emerslund, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. “Demo: Bagadus an integrated system for soccer analysis”. In: *Distributed Smart Cameras (ICDSC), 2012 Sixth International Conference on*. IEEE. 2012, pp. 1–2.

[16] Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Kvale Stensland, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. “Be Your Own Cameraman: Real-time Support for Zooming and Panning into Stored and Live Panoramic Video”. In: *Proceedings of the 5th ACM Multimedia Systems Conference*. MMSys ’14. Singapore, Singapore: ACM, 2014, pp. 168–171. ISBN: 978-1-4503-2705-3. DOI: 10.1145/2557642.2579370. URL: http://doi.acm.org/10.1145/2557642.2579370.

[17] Ragnar Langseth. “Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views”. MA thesis. 2014.

[18] Basler AG. URL: http://www.graftek.com/pdf/Brochures/basler/ace.pdf (visited on 01/21/2015).

[19] Vamsidhar Reddy Gaddam, Carsten Griwodz, and Pål Halvorsen. "Automatic exposure for panoramic systems in uncontrolled lighting conditions: a football stadium case study". In: *IS&T/SPIE Electronic Imaging*. International Society for Optics and Photonics. 2014, pp. 90120C–90120C.

[20] Sigurd Ljødal. "Implementation of a real–time distributed video processing pipeline". MA thesis. 2014.

[21] *Parallel Programming and Computing Platform | CUDA | NVIDIA | NVIDIA*. URL: http://www.nvidia.com/object/cuda_home_new.html (visited on 01/22/2015).

[22] Ragnar Langseth, Vamsidhar Reddy Gaddam, Hakon Kvale Stensland, Carsten Griwodz, and Pal Halvorsen. "An evaluation of debayering algorithms on GPU for real-time panoramic video recording". In: *Multimedia (ISM), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 110–115.

[23] Espen Oldeide Helgedagsrud. "Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching". In: (2013).

[24] R Pantos and W May. "HTTP Live Streaming draft-pantos-http-live-streaming-05". In: *Apple Inc., IETF draft* 23 (2010).

[25] Thorsten Lohmar, Torbjorn Einarsson, Per Frojdh, Frédéric Gabin, and Markus Kampmann. "Dynamic adaptive HTTP streaming of live content". In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*. IEEE. 2011, pp. 1–8.

[26] The Verge. *GoPro releases Wi-Fi BacPac + Wi-Fi Remote Combo Kit for $99.99*. 2012. URL: http://www.theverge.com/2012/6/6/3066289/gopro-wi-fi-bacpac-wi-fi-remote-combo-kit (visited on 01/06/2015).

[27] *Operating system market share*. URL: http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1&qptimeframe=M (visited on 01/27/2015).

[28] J. Reschke R. Fielding. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. Internet Engineering Task Force. 2014. URL: http://tools.ietf.org/html/rfc7230 (visited on 01/22/2015).

[29] Ian Fette and Alexey Melnikov. *The websocket protocol*. Internet Engineering Task Force. 2011. URL: https://tools.ietf.org/html/rfc6455 (visited on 01/20/2015).

[30] Kaazing Corporation Peter Lubbers & Frank Greco. *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web*. 2013. URL: http://www.websocket.org/quantum.html (visited on 01/22/2015).

[31] Peter Lubbers, Brian Albers, Frank Salim, and Tony Pye. *Pro HTML5 programming*. Springer, 2011.

[32] *Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013*. URL: http://www.gartner.com/newsroom/id/2665715 (visited on 04/29/2015).

[33] *App Review - App Store - Apple Developer*. URL: https://developer.apple.com/app-store/review/ (visited on 01/29/2015).

[34] Andre Charland and Brian Leroux. "Mobile application development: web vs. native". In: *Communications of the ACM* 54.5 (2011), pp. 49–53.

[35] *J2ObjC*. URL: http://j2objc.org/ (visited on 01/29/2015).

[36] *Java SE 7 Java Native Interface-related APIs and Developer Guides*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/ (visited on 01/29/2015).

[37] *GPU Accelerated Compositing in Chrome - The Chromium Projects*. URL: https://sites.google.com/a/chromium.org/dev/developers/design-documents/gpu-accelerated-compositing-in-chrome (visited on 01/30/2015).

[38] *Apple - Press Info - iPhone to Support Third-Party Web 2.0 Applications*. URL: https://www.apple.com/pr/library/2007/06/11iPhone-to-Support-Third-Party-Web-2-0-Applications.html (visited on 01/30/2015).

[39] *Internet Archive Wayback Machine*. URL: https://web.archive.org/web/20130201000000*/http://apple.com/webapps (visited on 01/30/2015).

[40] *Apache Cordova*. URL: https://cordova.apache.org/ (visited on 02/03/2015).

[41] *Ionic Documentation - Ionic Framework*. URL: http://ionicframework.com/docs/ (visited on 02/05/2015).

[42] *jQuery Mobile*. URL: https://jquerymobile.com/ (visited on 04/08/2015).

[43] *Mobile App Development Framework. JavaScript and HTML5. Download Sencha Touch Free. | Sencha Touch | Products | Sencha*. URL: https://www.sencha.com/products/touch/ (visited on 04/08/2015).

[44] *Mobile App Development & App Creation Software - Xamarin*. URL: http://xamarin.com/ (visited on 04/08/2015).

[45] *Limitations | Xamarin Android*. URL: http://developer.xamarin.com/guides/android/advanced_topics/limitations/ (visited on 02/04/2015).

[46] *Limitations | Xamarin iOS*. URL: http://developer.xamarin.com/guides/ios/advanced_topics/limitations/ (visited on 02/04/2015).

[47] Jeff Wisniewski. "Mobile That Works for Your Library". English. In: *Online* 35.1 (Jan. 2011). Copyright - Copyright Information Today, Inc. Jan/Feb 2011; Document feature - Illustrations; Last updated - 2013-06-13; CODEN - ONLIDN; SubjectsTermNotLitGenreText - United States–US, pp. 54–57. URL: http://search.proquest.com/docview/848233523?accountid=14699.

[48] *Apps Solidify Leadership Six Years into the Mobile Revolution | Flurry*. URL: http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution (visited on 02/04/2015).

[49] Ingus Smits Gatis Vitols and Aleksejs Zacepins. "Issues of Hybrid Mobile Application Development with PhoneGap: a Case Study of Insurance Mobile Application". In: *Proceedings of the 11th International Baltic Conference, Baltic DB&IS 2014*. Tallinn, Estonia, June 2014, pp. 215–220.

[50] *WebView for Android*. URL: https://developer.chrome.com/multidevice/webview/overview (visited on 03/10/2015).

[51] *The WebKit Open Source Project*. URL: https://www.webkit.org/ (visited on 03/10/2015).

[52] *The Chromium Projects*. URL: http://www.chromium.org/ (visited on 03/10/2015).

[53] Magnus Stenhaug, Yang Yang, Cathal Gurrin, and Dag Johansen. "Muithu: A touch-based annotation interface for activity logging in the norwegian premier league". In: *MultiMedia Modeling*. Springer. 2014, pp. 365–368.

[54] Jeffrey Zeldman. *Taking your talent to the web: A guide for the transitioning designer*. New Riders Publishing, 2001, p. 98.

[55] Apache Software Foundation. *Apache Cordova Documentation Platform Support*. 2014. URL: https://cordova.apache.org/docs/en/3.4.0/guide_support_index.md.html (visited on 11/04/2014).

[56] Lilian Genaro Motti, Nadine Vigouroux, and Philippe Gorce. "Drag-and-drop for older adults using touchscreen devices: effects of screen sizes and interaction techniques on accuracy". In: *Proceedings of the 26th Conference on l'Interaction Homme-Machine*. ACM. 2014, pp. 139–146.

[57] Kori M Inkpen. "Drag-and-drop versus point-and-click mouse interaction styles for children". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 8.1 (2001), pp. 1–33.

[58] *Socket.IO*. URL: http://socket.io/ (visited on 02/04/2015).

[59] The Wireshark team. *Wireshark*. 2015. URL: https://www.wireshark.org/ (visited on 01/29/2015).

[60] *Socket.IO JavaScript framework ready for real-time apps*. JavaWorld. June 2, 2014. URL: http://www.javaworld.com/article/2358967/html-css-js/socket-io-javascript-framework-ready-for-real-time-apps.html (visited on 03/25/2015).

[61] Angular. *Protractor Documentation*. 2015. URL: http://angular.github.io/protractor/#/ (visited on 02/13/2015).

[62] *Configuring Socket.IO*. URL: https://github.com/Automattic/socket.io/wiki/configuring-socket.io (visited on 02/04/2015).

[63] Giridhar D Mandyam and Navid Ehsan. "Html5 connectivity methods and mobile power consumption". In: *Accessed January* 16 (2012), p. 2013.

[64] Eliot Estep. "Mobile HTML5: Efficiency and Performance of WebSockets and Server-Sent Events". MA thesis. Aalto University, June 2013.

[65] Zheng-Hua Tan and Børge Lindberg. *Automatic speech recognition on mobile devices and over communication networks*. Springer Science & Business Media, 2008.

[66] *Web Speech API Specification*. URL: https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html (visited on 05/14/2015).

[67] De Wet Swanepoel, James W Hall III, and Dirk Koekemoer. "Vuvuzela: good for your team, bad for your ears". In: *SAMJ: South African Medical Journal* 100.2 (2010), pp. 99–100.

[68] *HTML5 Video*. URL: http://www.w3schools.com/html/html5_video.asp (visited on 03/16/2015).

[69] *Canvas API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (visited on 03/16/2015).

[70] *HTML5 Canvas*. URL: http://www.w3schools.com/html/html5_canvas.asp (visited on 03/16/2015).

[71] *iOS Keychain Services Tasks*. URL: https://developer.apple.com/library/ios/documentation/Security/Conceptual/keychainServConcepts/iPhoneTasks/iPhoneTasks.html (visited on 05/12/2015).

[72] *AccountManager | Android Developers*. URL: http://developer.android.com/reference/android/accounts/AccountManager.html (visited on 05/12/2015).

[73] Martin Alexander Wilhelmsen, Hakon Kvale Stensland, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, and Pal Halvorsen. “Using a Commodity Hardware Video Encoder for Interactive Video Streaming”. In: *Multimedia (ISM), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 251–254.

[74] *websockets/ws GitHub*. URL: https://github.com/websockets/ws (visited on 03/18/2015).

[75] *websocket client benchmark*. URL: http://websockets.github.io/ws/benchmarks.html (visited on 03/18/2015).

[76] *FFmpeg - A complete, cross-platform solution to record, convert and stream audio and video*. URL: http://ffmpeg.org/ (visited on 04/30/2015).

[77] *NRK TV - Schrödingers katt - 09.04.2015*. URL: http://tv.nrk.no/serie/schrodingers-katt/DMPV73000915/09-04-2015 (visited on 04/27/2015).