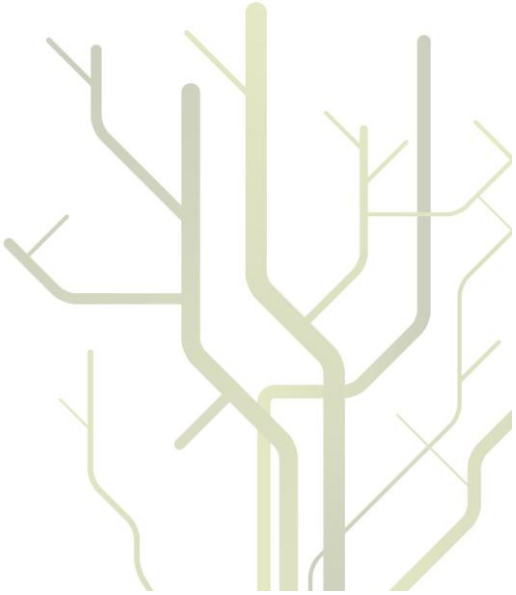
UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Real-Time Annotation of Video Streams Using Staged Processing

Øyvind Holmstad

INF-3981
Master's Thesis in Computer Science
June, 2011



Abstract

Real-time media rich applications rely on live streams of rich and accurate meta-data describing the video content to provide personal user experiences. Unfortunately, the general amount of video meta-data today is often limited to titles, synopsis and a few keywords.

A wildly used approach for extraction of meta-data from video is computer vision. It has been developed a number of different video processing algorithms which can analyse and retrieve useful data from video. However, the computational cost of current computer vision algorithms is considerable.

This thesis presents a software architecture that aims to enable real-time annotation of multiple live video streams. The architecture is intended for use within media rich applications where extraction of video semantics in real-time is necessary. Our conjecture was that staging video processing in levels will make room for a more scalable video annotation system. To evaluate our thesis we have developed the prototype runtime Árvádus.

Our experiments show that staged processing can decrease the computation time of meta-data extraction. The evaluation of the architecture suggests that the architecture is applicable in a wide range of domains where extraction of meta-data in real-time is necessary

Acknowledgements

I would like to thank my supervisor Dr. Håvard Johansen for his great advice during the planning, implementation, experimentation and writing process of this thesis. You're a patient man. I also want to thank my co-advisor Professor Dag Johansen for his excellent feedback throughout the process. You're a true source of ideas, inspiration and motivation.

I would like to thank Jinlin Guo and Dr. Cathal Gurrin from Dublin City University for providing insight in the field of computer vision. I would like to thank Svein Arne Pettersen as a representative for Tromsø IL and Ottar Dahle from ZXY Sports Tracking AS for their cooperation regarding the ZXY Sports Tracking System. I would also like to thank Joseph Hurley and rest of the iAD group for their input on how to practice proper scientific work.

A special thanks go to my friends and colleagues Erik Bræck Leer and Johan Grønvik for valuable discussions and constant feedback throughout the process.

Finally, I would like to thank my girlfriend, my friends and my family for their support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

According to Cisco [1], Internet video will account for over 57% of all consumer Internet traffic by 2014. They also state that it would take over two years for one person to watch the amount of video that will cross global IP networks every second in 2014. The importance of real-time video is also growing. By 2014, Internet TV will constitute over 8% of consumer Internet traffic, and ambient video will constitute an additional 5% of consumer Internet traffic.

Rich and accurate meta-data are important to organization of video data. Even more interesting is how rich semantic data about videos and video streams can be used for unique services like automated video editing, personalized video search, and complex recommendation engines. Yet, the general amount of video meta-data today is often limited to titles, synopsis and a few keywords. Accurate video annotations and detailed descriptions of arbitrary events in videos is the exception, not the rule. It is possible to let humans annotate videos manually, but this is time consuming and tedious, and hence not a proper solution in the situation we are now facing with massive growth of video. We need to automate the process.

In many contexts external sources of information about a video stream is available, yet it is not formalized as meta-data. For instance, subtitles, sensor data and social network data can be processed to pinpoint interesting events in a video stream. Another example is found for sports videos, where live text commentary is readily available on the Internet [2].

A wildly used approach for extraction of meta-data from video is audio and image analyses. The field of processing images is called computer vision, and it encompasses forms of computer automated tasks like detecting events in video, controlling processes through vision and extracting semantics. Computer vision is about letting the computer extract information from an image and perceive it like we humans do.

It has been developed a number of different video processing algorithms which can analyse and retrieve useful data from video. Feature extraction

can be used to detect features as color, corners or points, and even complex ones like texture and shape. By analysing and combining the low level features of thousands of images with known content, it is possible to train classifiers that recognize high-level concepts like animals and aeroplanes. Although the accuracy of high-level classifiers and detectors is of varying quality [3], it is a promising field. However, the computational cost of current computer vision and machine learning algorithms is considerable. While some simple meta-data extraction can be done in real-time, most algorithms have a computational cost too high to enable real-time classification. When we bring multiple video streams in to the mix, it is no longer trivial how to process them in real-time.

1.1 Problem definition

This thesis shall develop and study aspects of a software architecture that enables real-time annotation of multiple live video streams. The architecture is intended for use within media rich applications where extraction of video semantics in real-time is necessary. A working prototype applying the architecture will be developed and evaluated in a scientific context.

1.2 Interpretation

Our thesis is that staging video processing in levels will open for a more scalable video annotation system. We conjecture that using simple and cheap processing as filters for more heavy processing will decrease the total computational cost and enable real-time annotation. To explore this idea, we will devise an architecture and build a prototype runtime that enables such chaining of processing elements.

We will use the runtime to implement two applications that are dependent on real-time analyses: a video surveillance application and a soccer video application. In the surveillance application we will investigate how cheap video processing may trigger heavy video processing, and measure the impact on speed and accuracy. In the soccer application we will investigate how real-time sensor data can be used to trigger video processing. The applications will form the basis for the evaluation of the architecture.

1.3 Context

This thesis is a part of the Information Access Disruption (iAD) project. The iAD Center focuses on core research for next generation precision, analytics and scale in the information access domain. With DAVVI [2] iAD has explored this in a video context, where the idea is to "*integrate existing video delivery systems with search and recommendations systems and*

social networking systems" [2] in order to "*provide a personalized, topic-based user experience, blurring the distinction between content producers and consumers*" [2]. This thesis focus on extracting meta-data from videos in real-time, which could be considered an extension to the annotation side of DAVVI.

Tromsø IL (TIL) and ZXY Sports Tracking AS (ZXY) is two of the iAD industry partners. TIL is a top soccer club in Tippeligaen, the Norwegian Premier League, and has recently installed the ZXY Sports Tracking System on their home ground Alfheim stadion. Part of this thesis is the first step in exploring how to build next generation multimedia services by combining ZXY sensor data with video.

1.4 Methodology

The final report [4] of the ACM Task Force on the Core of Computer Science divides the discipline of computing into three major paradigms.

- *Theory:* Rooted in mathematics, the approach is to define problems, propose theorems and seek to prove them in order to determine new relationships and progress in computing.
- *Abstraction:* Rooted in the experimental scientific method, the approach is to investigate a phenomenon by stating hypothesis, constructing models and simulations, and analyzing the results.
- *Design:* Rooted in engineering, the approach is to state requirements and specifications, design and implement systems that solve the problem, and test the systems to systematically find the best solution to the given problem.

For this thesis it is most appropriate to use the design process rooted in engineering. We have posed a specific problem, and will systematically design and build a prototype to solve it. After finishing the prototype we will systematically test the system and evaluate it in the light of the given problem.

1.5 Outline

The next chapter will cover the basics of video and computer vision. It will present systems utilizing staged processing, and describe DAVVI, a next generation multimedia entertainment platform, in detail. Chapter 3 will cover our general model and define staged processing in the context of our domain. Chapter 4 will devise an architecture for real-time annotation of live video stream. In chapter 5 we describe Árvádus, a runtime applying our

architecture. Chapter 6 will describe our experiments and evaluate Árvádus. In chapter 7 we conclude our works and findings, and outline possible future work.

Chapter 2

Background

2.1 Video

Video is basically a series of images (or frames) stored and viewed in order. However, video is not stored as a group of single images. Video is stored in container files whose specification describes how the different data elements and meta-data coexist within the file. Different container formats include AVI, MKV and MP4.

The resolution of a video refers to number of pixels used to represent one image of it. In other words, the size of the two-dimensional pixel array describing each frame of the video. The resolution of different videos vary, but some standards are defined such as 720x480 (DVD), 1280x720 (720p), 1920x1080 (1080p/Full HD).

A video in the 720p resolution has got $1280 * 720 = 921600$ pixels per frame. Each pixel is traditionally represented with 3 bytes (RGB), so the size of each raw frame is $921600 * 3\text{B} = 2764\text{kB}$.

The number of frames viewed per second is referred to as framerate. A typical video contains approximately 24 frames per second (fps) to enable smooth playback. A video with 24 fps with the length of 1 minute thus has $24 * 60 = 1440$ frames. If the video is in the 720p resolution it uses 2764kB per frame. This totals to $2700\text{kB} * 1440 = 3,7\text{GB}$. That is a lot for only 1 minute of video excluding sound, and the reason we use compression.

2.1.1 Video compression

Compression is the process of reducing the size of the video data by removing redundant information. We typically refer to video compression as video encoding. Similarly, the process of decompressing a video is referred to as decoding. The algorithm used to encode and decode a video is referred to as a codec.

There is a wealth of different codecs designed for video compression, among them are DivX, Xvid, h264 and VC-1. Common for many of them

is that they are built on MPEG standards[1].

The MPEG standards is developed by the Motion Picture Experts Group and defines an extensive set of standards for audio and video compression and transmission. MPEG techniques establish the protocols for compressing, encoding and decoding video data, but not the encoding methods themselves. With MPEG it is not the encoder that is standardized, but the way a decoder interprets the data. Over time, encoding algorithms can change and improve, yet compliant decoders will still understand them. This is a result of keeping the standard less strict. It does not define the structure and operation of the encoder, hence implementers can supply encoders using proprietary algorithms. Decoding can be done because MPEG compliant video files contain enough meta data to interpret the video correctly.

Many of the most widely used codecs use the same basic techniques to encode video. MPEG compression utilizes a combination of two different compression schemes: spatial and temporal. Spatial compression reduces the quantity of the data by removing redundant information within the image, like when we compress a bitmap to a JPEG. Temporal compression compares the change between frames over time, and stores only the changes. A more detailed description of how this works can be found in Weise and Weynand's How Video Works [5].

2.2 Computer vision

Visual perception is the ability to interpret information and surroundings from the effects of visible light reaching the eye. This skill comes natural for human. For computers, visual perception is more difficult. Computer vision is about letting the computer dig into the world of images and perceive them like we humans do. Researchers in computer vision have been developing mathematical techniques for recovering the three-dimensional shape and appearance of objects in imagery, but we still have long way to go before machines can interpret images with same precision as the human visual system.

It has been developed a flora of different video processing algorithms which can analyze and retrieve useful data from images. Feature extraction can be used to detect features as color, corners or points, and more complex ones like texture and shape. The features can then be combined to understand more high-level concepts like recognizing faces and objects using advanced techniques.

[1] http://mpeg.chiariglione.org/

2.2.1 Classification and detection

In the context of processing video for meta-data, classification and detection (recognition) may be the most interesting computer vision field. Knowing what type of objects are present in a frame can tell us a lot about what the video is about. We divide between classification and detection. Classification tells if an image contain any instances of a particular object class (cars, people, dogs etc.), while detection tells where the instances of a particular class in the image are located (if any). The task is considered one of the most challenging in the field of computer vision. The reasons for this are straight forward: The real world constitutes of a vast amount of different objects, which all can be viewed from different angles and views, appear in different poses and occlude one another. Furthermore, the complex variation within each object class (e.g., dogs) where shape and appearance differ greatly (e.g., breeds), make it hard to create methods to perform accurate matching.

Despite the challenges, the field is maturing, and producing better results every year. The PASCAL Visual Object Classes (VOC) challenge [3] is an annual benchmark in visual object category recognition and detection, which review and evaluate the state-of-the-art methods in both classification and detection. The numbers presented from the 2009 challenge show that precision varies greatly between classification and detection, and between the type of objects classified. While the average precision is close to 60% for finding a car in an image (classification), the detection precision for cars is around 30%. The classifiers works best at finding people. The average precision is around 75% in average for all classifiers, and close to 90% for the best. On the opposite site of the scale, we find that the classifiers only find "potted plants" with a precision of 15% in average.

One of the more successful examples of recognition is face detection. It has found its way into many consumer applications like Picasa[2] and iPhoto[3], and is also used by many digital cameras to enhance auto-focus. According to Yang et al. [6], face detection techniques can be categorized as feature-based, template-based and appearance-based.

Feature-based techniques attempt to find the locations of distinctive facial features such as the eyes, nose, and mouth, and then verify whether these features are in a plausible geometrical arrangement. In template-based techniques, a standard face pattern is manually predefined. Given an input image, correlation values between the standard patterns and the face contour, eyes, nose, and mouth are computed independently. The existence of a face is determined based on the correlation values. Contrasted to the template-based method where templates are predefined by experts, the "templates" in appearance-based methods are learned from examples

[2] http://picasa.google.com/

[3] http://www.apple.com/ilife/iphoto/

in images. Most appearance-based approaches rely on training classifiers using sets of labeled training examples, each marked as belonging to one of two categories. Among these approaches we find Support Vector Machines (SVMs), Neural Networks, and Clustering.

2.2.2 Support Vector Machines

Support Vector Machines (SVM) are a set of machine learning training methods that analyze data and recognize patterns. It is considered a useful technique for data classification, and is often used to train image classifiers. In other words, we train SVMs to answer questions like "does this image contain a car or not?"

The standard SVM [7] is a binary classifier, which signifies that it takes some input data and predicts which of two possible classes it is a member of. The prediction is based on a model built with an SVM training algorithm. The algorithm is given a set of training examples, each marked as belonging to one of two classes, and then proceeds to search for a series of maximum margin separating planes in feature space between the different classes. In other words, the SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

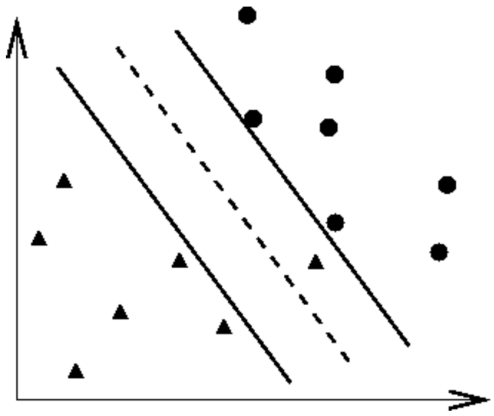


Figure 2.1: A simple SVM model in two dimensions

The points mapped into space represent the image, but how they are derived from it vary widely. Some classifiers use color as the main *feature*. Other use edges, curves, and line segments. Point features are also widely used, but how they are extracted, and thus what is defined as an interesting point, vary between different techniques. Some of the most widely used

are Scale-Invariant Feature Transform (SIFT) [8] and Speeded Up Robust Features (SURF) [9].

2.2.3 Computational cost

To explore how computational intensive a typical image classifier is, we conducted an experiment. The classifier, developed at Dublin City University, is built using LIBSVM [10], a popular open source library for Support Vector Machines. The classifier is trained with TRECVID 2010 data [11], and uses both color layout, scale color, edge histograms and SURF features to form the vectors that the model is built on. The classifier in question evaluates if an image contains a waterscape or not.

We tested the classifier on 10 different images and computed the average time used for classification. The images were classified in four different resolutions, all of them commonly used for video: 1920x1080, 1280x720, 863x480 and 427x240. The tests were run on a computer running Windows 7 on a Intel Core 2 Quad Q6600 processor with 4 cores running at 2,4Ghz, 8MiB L2 cache and 4GiB of memory. The results can be seen in figure 2.2:

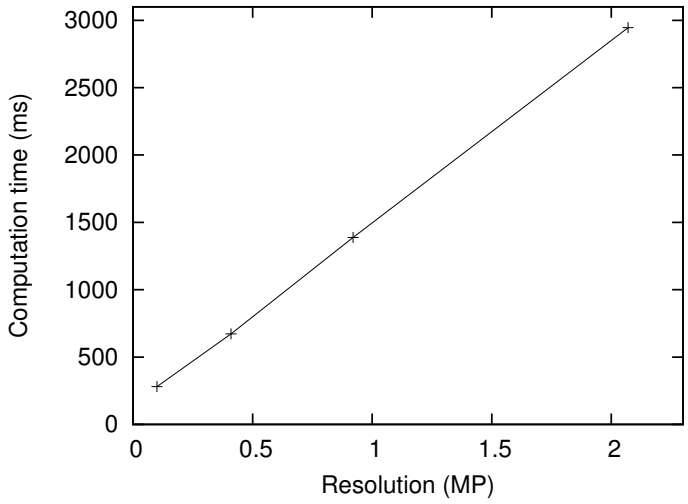


Figure 2.2: Computational time of Waterscape classifier.

Figure 2.2 show how the computational time for classifying an image grows with the size of the image in megapixels (MP). The figure shows that the computational time grows linearly with the number of pixels. While evaluating if an image contains a waterscape or not takes 281 ms in average for an image in 240p (427x240), it takes almost 3 seconds for an image in Full HD resolution (1920x1080). As described in section 2.1, a typical video stream has 24 frames per second. In order to apply the classifier to every frame in real-time we would need it to process each frame in less

then $1/24 = 0.041$ seconds. As we cannot do it faster than 0.28 seconds for the lowest resolution, it is infeasible to process every frame of a video stream. In other words, the computational cost is too high to enable real-time classification.

2.3 Staged processing

Staged processing is the concept of processing data in multiple stages. Rather than performing all processing in one big operation, the processing elements are put into independent stages that work on the data individually in sequence. The concept is well explored, and brings many advantages that many systems have utilized.

A pipeline is a commonly used abstraction where processing elements are arranged in chains so that the output of one element is the input of the next. The technique is widely used in different applications and on different abstraction levels. Instruction pipelines are used to increase instruction-level parallelism on the processor, graphics processing units utilize pipelines to increase the speed of 3D graphics rendering, and Unix pipes are used to feed the output of one process as the input to the next one. Unix pipelines can be utilized both programmatically and through the command line interfaces, and are commonly used to combine simple Unix commands to accomplish more complex tasks with ease. Figure 2.3 illustrates a Unix pipeline.

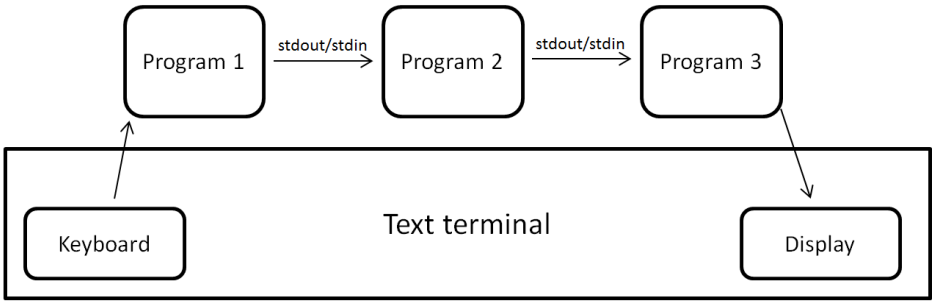


Figure 2.3: Unix pipeline.

SEDA (Staged Event-Driven Architecture) [12] is an architecture designed to enable high concurrency, load conditioning, and ease of engineering for Internet services. SEDA utilizes staged processing to allow applications to adjust dynamically to changing load. In SEDA, a stage is a self-contained application component consisting of a an event handler, an incoming event queue and a thread pool. The event queue is filled with incoming events, and the stage threads operate by pulling a batch of events off of the incoming event queue and invoke the application-supplied event handler. The event

handler processes each batch of events, and dispatches zero or more events by enqueuing them on the event queues of other stages. Each stage is managed by a controller that is in control of resource allocation. The controller continuously adjust and tune the behavior to keep the application within its operating regime. Adjustments is done to the number of threads executing within the stage, and the amount of events processed each iteration is fine-tuned.

Staging of processing elements makes it easier to distribute computations. Sawzall [13], a system for doing analyses, aggregation, and extraction of statistics on very large data sets, exploits the inherent parallelism in having data and computation distributed across many machines. Since the data records Sawzall seek to process is located on many machines, the system separates the process into two phases (or stages). In the first phase each record is evaluated individually on nearby machines. In the second phase the results are collected and aggregated. Both phases are distributed over hundreds or even thousands of computers. Figure 2.4 illustrates the overall flow of Sawzall.

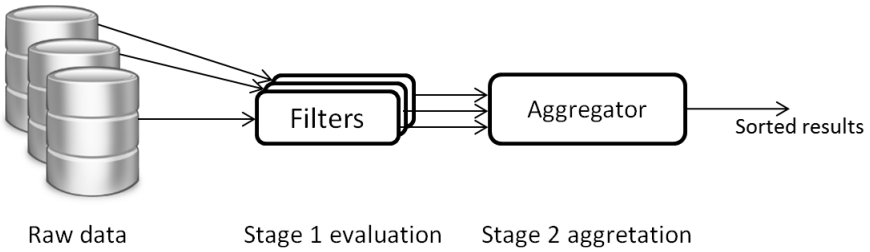


Figure 2.4: Flow of filtering, aggretating and collating in Sawzall.

MapReduce [14] is a software framework by Google for processing large data sets on distributed clusters of computers. The framework is intended for solving certain kinds of distributable problems, using a restricted programming model to make it easy to parallelize and distribute computations and to make such computations fault-tolerant. The computation performed on the data set takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. The Map function processes a key/value pair to generate a set of intermediate key/value pairs, and the reduce function merges all intermediate values associated with the same intermediate key. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. The MapReduce programming model is widely acknowledged, and has many implementations. Hadoop [15] is an open-source implementation of MapReduce, and Phoenix [16] is an implementation of MapReduce

for shared-memory systems.

2.4 DAVVI

DAVVI [2] is a multimedia entertainment platform which aims to provide a personalized, topic-based user experience blurring the distinction between content producers and consumers. It delivers multi-quality video content in a torrent-similar way, while providing a highly personalized user experience. Through applied search and advanced personalization and recommendation technologies end-user can efficiently search and retrieve highlights in a customized manner. DAVVI has been demonstrated in the domain of sports video, using a soccer example.

DAVVI utilizes an adaptive streaming approach for dissemination of the video streams, an approach utilized successfully by systems like Move Networks[4], Microsoft's Smooth Streaming[5] and Apple's HTTP Live Streaming[6]. The input video streams are chopped into discrete media objects (segments) and transcoded into different qualities. The two-second segments are self-contained video clips, which makes it possible for clients to request arbitrary segments and play them out in any order. By encoding the segments into different quality levels, the clients can continuously adapt the playback to the current network condition. Delivery is done over HTTP as a long series of small progressive downloads.

To analyze and annotate video, DAVVI has a set of extraction tools. The main meta-data source is TV broadcasting and newspaper cites that provide *live text commentary web pages* for soccer video. The unstructured commentary text are crawled and parsed by semi-automatic crawlers in real-time, and converted to structured annotation meta-data.

Based on the meta-data aggregation, video annotation and indexing, users can query for a broad range of events using keywords found in the live text commentary. Users of DAVVI can for instance query using specific keywords like "volley", "sliding tackle" and "offside". When searching for "sliding tackle Steven Gerrard" the video annotations are used to return a playlist presenting each relevant event with an event description, a video object identifier and a time interval. The playlist is sent to the video dissemination system which retrieves the respective video segments.

The playlist is presented to the user as a list of thumbnails taken from the corresponding video interval along with traditional meta-data like which game it is taken from, date, teams and result. The search results may be played back one by one using a play-all button or by generating a personalized video sequence using drag-and-drop.

[4] http://www.movenetworks.com/

[5] http://www.iis.net/download/SmoothStreaming

[6] http://www.apple.com/quicktime/extending/resources.html

Chapter 3

System Model

This chapter will describe our general conceptual model, define what we mean by staged processing, and elaborate on how it conforms to the model.

3.1 General model

An annotation system is a system that takes one or more data streams as input and produces annotations describing the streams as output. In our model, we define a *world* as the event the streams are representing. The type of data in the incoming streams may be of different types, like video, audio or sensor data. We define each stream as a *view* on the *world*, as shown in figure 3.1. In its simplest form the *world* may be a movie, with the video stream as the only *view*. In a more complex form, the *world* may be a soccer match, with a set *views* being the video streams from cameras placed around the field, while other *views* may be GPS location streams giving the positions to the players. The output of the system is consistent annotations, describing the content of the *world*.

The job of the annotation system is to extract meta-data from incoming

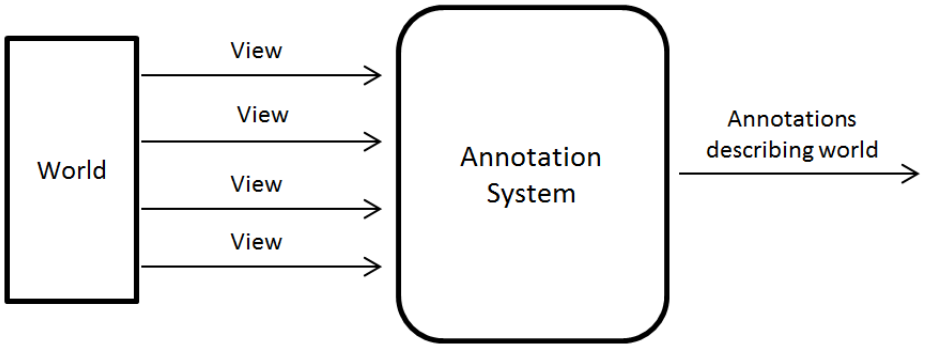


Figure 3.1: The general model of video annotation.

Table 3.1: Example of stream types and some possible sample types.

Data stream	**Samples**
Video	Frame, 24 last frames, every third frame, audio
Audio	1 min of audio, 1s of audio
Subtitle	Last word, last sentence, 10 last sentences
Social media (Twitter)	Last tweet, 10 last tweets on hashtag
Sensor(GPS)	Average location (minute), location each second

data streams to create annotations. The data streams may be of different types like image data, audio data, sensor data, and social media data. The components doing the actual processing cannot work on the stream in its entirety, only single data points in the stream. We describe such a data point in a stream as a *sample*. How a sample is defined for a particular stream is dependent on what processing should be done. Table 3.1 presents some examples of data streams and sample types.

To formalize the handling of the annotations we have decided to describe them in an event model. An *event* is defined as a notable occurrence at a particular point in time. In the context of this thesis, time is represented as a *timeline* from a defined starting point until a defined ending point. We define the collection of events residing on this timeline as a *sequence of events*. Well defined sequence of events are in turn used to describe the content of a *world*. By describing the meta-data within an event model, we also the define the processing elements as *event detectors*. An event detector's role is to analyse the samples for events.

What is regarded as an event in a *world* is domain specific. What is considered notable in one *world* may be irrelevant in another. In a surveillance application, movement in itself may be considered as a notable occurrence, whereas this is irrelevant in a soccer video where there is a lot of movement. Vice versa, soccer specific events like goals or cards are only interesting in the soccer domain, and certainly not in a surveillance application. Figure 3.2 illustrates an example of a timeline with belonging events for a soccer video and a movie. Notice how the type of events differ between the two domains.

Although an event conceptually belongs to a particular point in time, it actually maps down to the timestamp of the sample it was extracted from. More coarse grained samples causes a more coarse grained timeline. If a sample contains data stretching over a time interval, the timestamp is set to the start of the interval.

Figure 3.2: Sequence of events on a timeline.

3.1.1 Formal definition

Given the above, we define an annotation system as a system $Z = \{V, D, E\}$ where $V = \{v_1, v_2, \ldots, v_n\}$ is an unordered set of n incoming views representing our world W, $D = \{d_1, d_2, \ldots, d_n\}$ is a set of event detectors, and $E = \{e_1, e_2, \ldots, e_n\}$ is an ordered set of n outgoing events describing what is happening in W. Each data stream v_i is represented by a set of samples $S = \{s_1, s_2, \ldots, s_n\}$ which are processed by some event detector e_i, and each sample s_i has a timestamp t_i that maps down to the timeline T of the world W. Each event e_i also has a timestamp mapping to the timeline T of world W, corresponding to the timestamp of the sample s_i it was extracted from.

3.2 Staged processing

Staged processing is the concept of processing data in multiple steps. Rather than performing all processing in one big operation, the processing elements are put into independent stages that work on the data individually in sequence. By examining systems like MapReduce [14] and Sawzall [13] we know that dividing the computation into stages makes it easier to distribute the processing across many machines. We want to exploit this property in our video annotation architecture. In addition we want to see if we can exploit the division of processing elements by letting the results of preceding stages decide if there should be processing in the following stages. The question is if we can maintain the accuracy for video event detection while decreasing the computational cost associated with it. The concept is illustrated in figure 3.3.

In figure 3.3 we see four incoming video streams, ready to be processed for extraction of meta-data. All streams are processed in stage one. If a

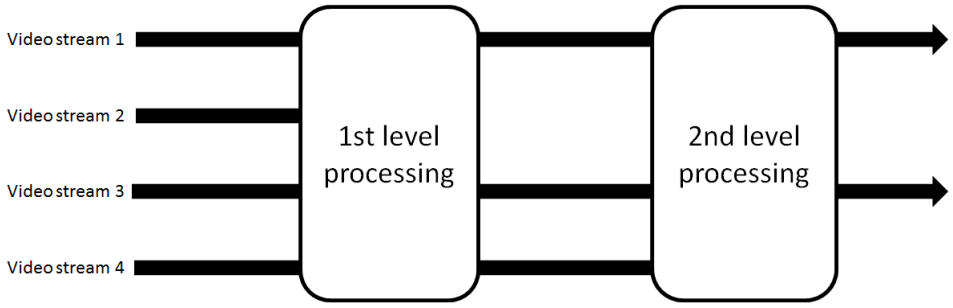


Figure 3.3: Staged processing.

stream passes the criteria in the 1st level of processing, it is passed on for 2nd level processing in the next stage. Stage 1 act as a filter. In this figure, stream one, three and four are considered interesting enough to be processed in stage two, and we save the computational cost of processing stream two in the second stage.

When looking at staged processing in the context of the general model defined in section 3.1, it is clear that the video processing takes place inside the annotation system black box in figure 3.1. Using this model, each stage, or processing element, is an event detector, and the data flowing through the stages are samples. The staging happens on the sample level. As a result, a sample s_1 of view v_1 may not reach the second stage, while sample s_2 from the same view may do so.

Chapter 4

Architecture

To evaluate our thesis that staged processing increases the scalability of a video annotation system we will devise an architecture around the principle. This chapter describes how we developed our architecture, and discusses its features.

4.1 Overview

In section 3.1 we defined an annotation system as a system that takes one or more data streams as input and produces annotations describing the streams as output. This definition forms the basis for our architecture. An application built on top of our architecture will use it as a foundation to get real-time annotations for a set of live video streams. This is illustrated in figure 4.1.

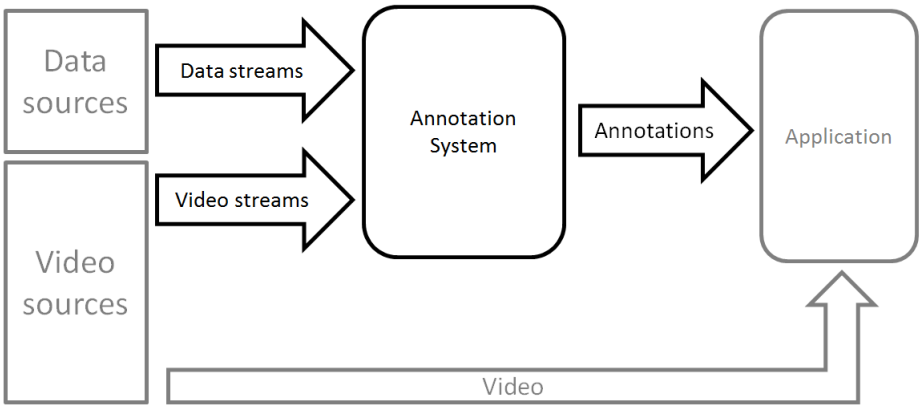


Figure 4.1: Architecture overview.

The annotation system receives stream of input data from different sources and create annotations for the application to use. Note that it is not

the annotation system that deliver video or other data to the application. This data is delivered to the application on another channel, separate for our system.

When taking the concept of staged processing (see section 3.2) into consideration we understand that we can divide the main responsibility of our architecture into two: To orchestrate the flow of samples to and between event detectors, and to deliver the resulting events to the application in real-time.

4.2 Architecture

The first step in the orchestration of sample flow is to manage the incoming data streams. This is the responsibility of the Streaming data handler (1), as illustrated in figure 4.2. The component accepts incoming data stream of different types.

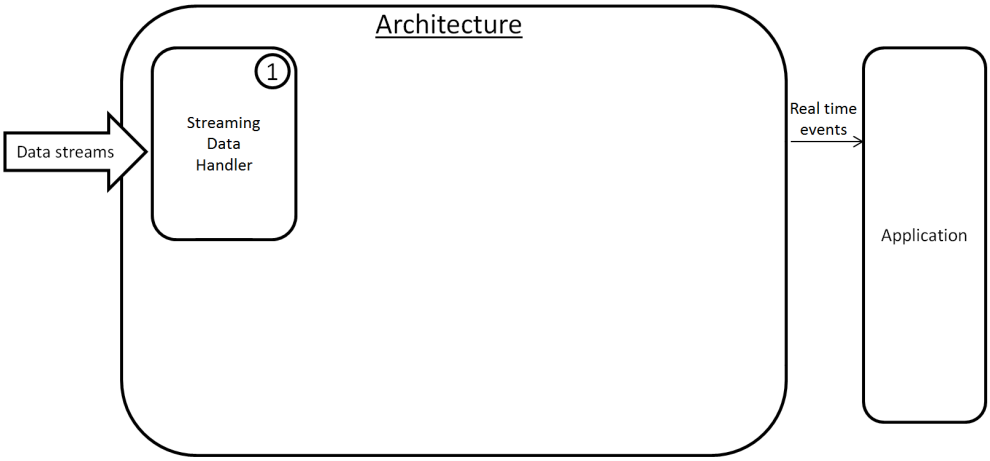


Figure 4.2: Incoming data streams are managed by the Streaming data handler (1).

The Streaming data handler continuously partitions the incoming streams into the samples required by the system's event detectors. What is considered a sample is not only dependent on the type of data stream (see table 3.1), but also on the application. For instance, a typical video has 24 frames per second, but in some cases it may suffice to process a single frame each second. In such a case the Streaming data handler makes one sample available containing a single frame, and discards the 23 remaining frames.

An important task for the Streaming data handler is to handle synchronization. Two video streams covering the same *world* need to be synchronized in order to provide the application with a consistent sequence of

events. The timestamp put on a sample in the Streaming data handler is the timestamp that will follow it and its derived events through the system.

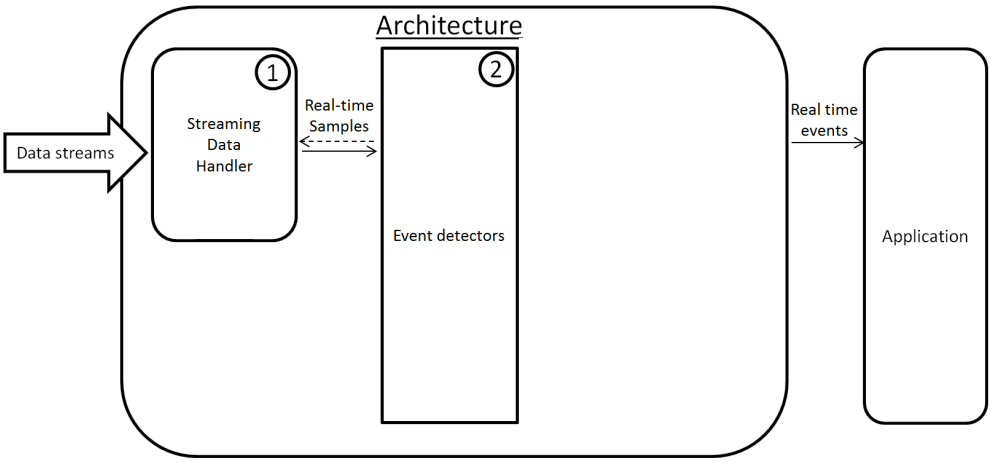


Figure 4.3: Incoming data streams are managed by the Streaming data handler (1) and passed on as samples to the Event detectors (2).

The next step in the orchestration of sample flow is to forward the samples from the Streaming data handler (1) to the Event detectors (2) in real-time, as illustrated in figure 4.3. The communication is pull-based, meaning that the Event detectors request samples when needed. We chose a pull-based communication scheme because of its simplicity, robustness and scalability properties. Minimizing the need to keep state makes the Streaming data handler less prone to failure. At the same time the Event detectors will get the samples in the speed they can process them, as they continuously request the next sample as the processing of the previous sample is finished.

When the number of streams and event detectors increase, the load on the Streaming data handler may be too high for one computer to handle. While the work of partitioning the streams into samples may be costly due to expensive video decoding algorithms, another bottle neck is the bandwidth limitations. Delivering samples from multiple high definition video streams to a group of event detectors is a heavy task.

In our architecture we address this issue by distributing the workload across multiple computers. Figure 4.4 illustrates how a load balancer is used to distribute the requests through a single entry point. For Internet services it is common to let the load balancer reply to the client without the client ever knowing about the internal distribution of workload. However, because bandwidth can be the most scarce resource in this context, this is not how the Streaming data handler operates. An event detector will connect to the load balancer once, and be redirected to the node which

handle the appropriate stream. The two will then communicate without further involvement of the load balancer.

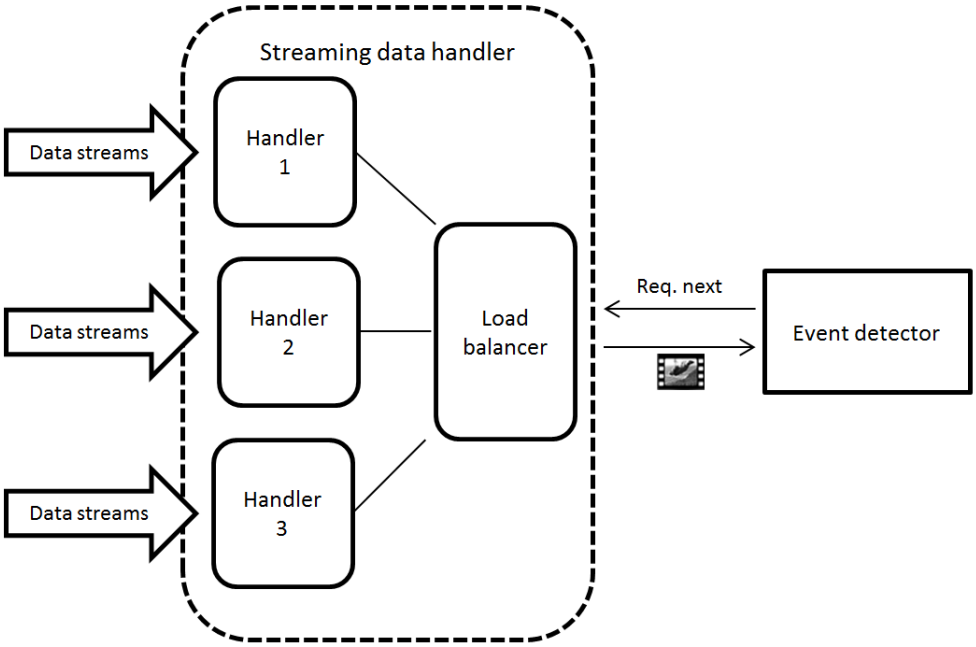


Figure 4.4: Distributing the Streaming data handler.

Event detectors are responsible for the extraction of meta-data. An event detector takes a sample as input, processes it, and may produce and publish an event as output. An application typically utilizes multiple event detectors, processing different types of data. Nevertheless, an event detector should be simple and independent, with no tight dependencies to other components or the application as a whole. It may contain state, for instance by keeping track of the last 10 samples processed, but should not rely on anything other than the samples alone. This ensures that the detector can be reused in different applications, and that the application developer can plug in and out detectors as needed.

Our conjecture was that staged processing increases the scalability of a video annotation system. In our architecture staged processing corresponds to staging of event detectors. In short, staging event detectors means that the occurrence of one event may trigger the search for others. Figure 4.5 depicts the concept of staged event detection.

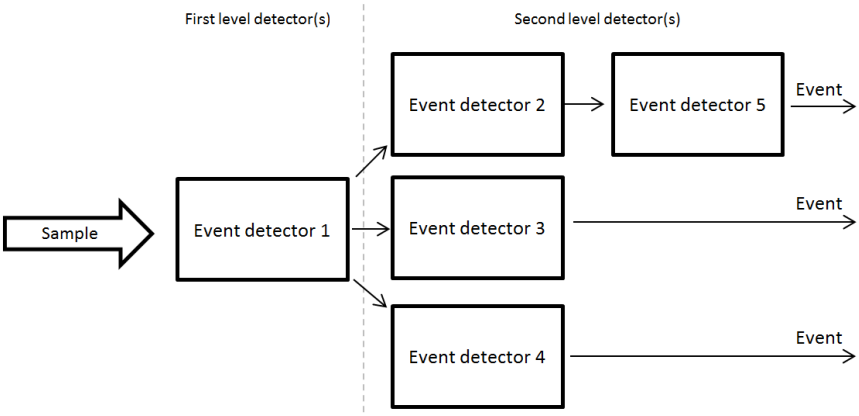


Figure 4.5: Staged event detection.

If *Event Detector 1* detects an event in the incoming sample, it triggers the following detectors (2, 3 and 4) to start processing on the current sample. We divide between two types of event detectors — *first level* and *second level*. First level event detectors continuously request samples from the Streaming data handler and process them for events. Second level event detectors do not do any work until they get triggered by first level event detectors. This implies that if the first level detectors do not detect any events, the second level detectors will not do any processing. Notice how *Event detector 5* also is considered a second level stage detector, even though it is in a third stage of the processing chain. This is because it is conceptually equal to the detectors in the second stage. It does not do any processing until its previous stage, in this case *Event detector 2*, has triggered it.

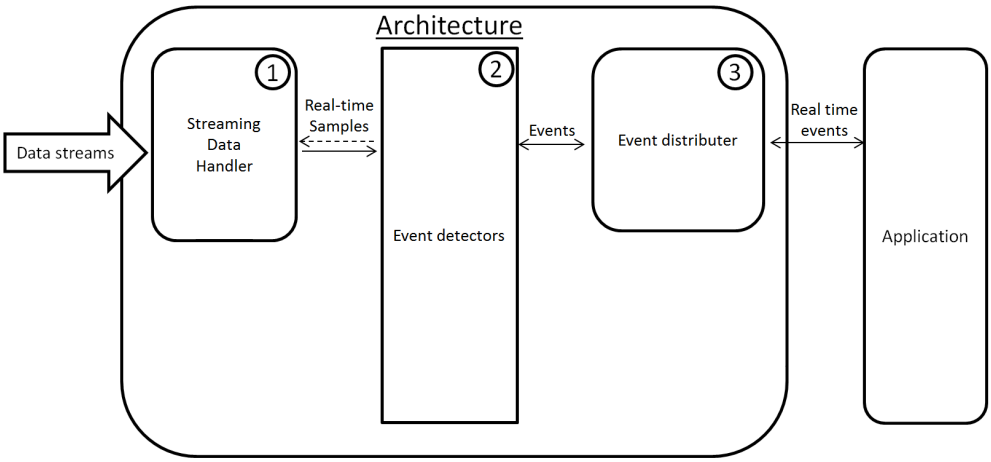


Figure 4.6: Events are passed from the Event detectors (2) to the Event distributer (3).

The resulting events are passed on to the Event distributer (3), which deliver the events to the application, as illustrated in figure 4.6.

As the two headed arrow in the figure suggests, the Event distributer (3) also delivers events back to the Event detectors (2). This is how we handle communication between the different event detectors. When *Event detector 1* triggers *Event detector 2* to start processing in figure 4.5, the communication is actually done through the Event distributer. The flow of events between the different components is designed using the publish/subscribe message pattern [17], with the Event distributer as the message broker.

With publish/subscribe the publisher of a message does not direct messages to specific receivers (subscribers). Rather, messages are characterized by topic, and published to a central event notification service providing storage and management for subscriptions. The publisher has no knowledge of what, if any, subscribers there may be. Subscribers express interest in one or more topics to the notification service, and only receives messages that are of interest, without knowledge of what, if any, publishers there are.

In our architecture, first level event detectors act as publishers, while second level event detectors act as both publishers and subscribers. Upon detecting an event, an event detector immediately publishes it to the Event distributer. The distributer forwards the event to current subscribers of this event type, which may be the application, an event detector, or both.

Staging event detectors is as simple as letting second level event detector subscribe to events by other event detectors. It is also interesting to see how an event detector can be triggered by multiple other event detectors simply by subscribing to different types of events.

Note that the sample data an event was extracted from is not contained in the published event. The main reason for this is that a second level detectors might not process the same type of sample data as the event detector that triggered it. To explain we use figure 4.5 as an example. In this set of event detectors, *Event detector 1* may work on sensor data, *Event detector 2* on twitter data, *Event detector 3* on audio data, and *Event detector 5* on video data. Forwarding the original sensor data sample to the other detectors will be an expense with no pay-off.

In addition, the application is purely interested in the event, not the sample it was extracted from. If the sample is a frame from a video stream and the application for some reason should be interested in viewing it, the application has access to it elsewhere, as illustrated in figure 4.1. Delivering sample data along side the event can thus be seen as a waste of resources.

As samples is not included in the published event, the second level event detectors need to fetch the appropriate samples elsewhere. The component responsible for delivering samples to second level detectors is the Raw data storage. The role of the Raw data storage will be explained in more detail shortly.

Figure 4.7 shows how staged event processing works using publish/sub-

scribe. To better illustrate the flow of events, each event is characterised by a color and a symbol. *Event detector 1* publishes a *red star* event, which detector 2, 3 and 4 subscribe to. *Event detector 2* publishes a *blue diamond* event which *Event detector 5* subscribes to. The detectors in the figure are chained the same way as the detectors in figure 4.5.

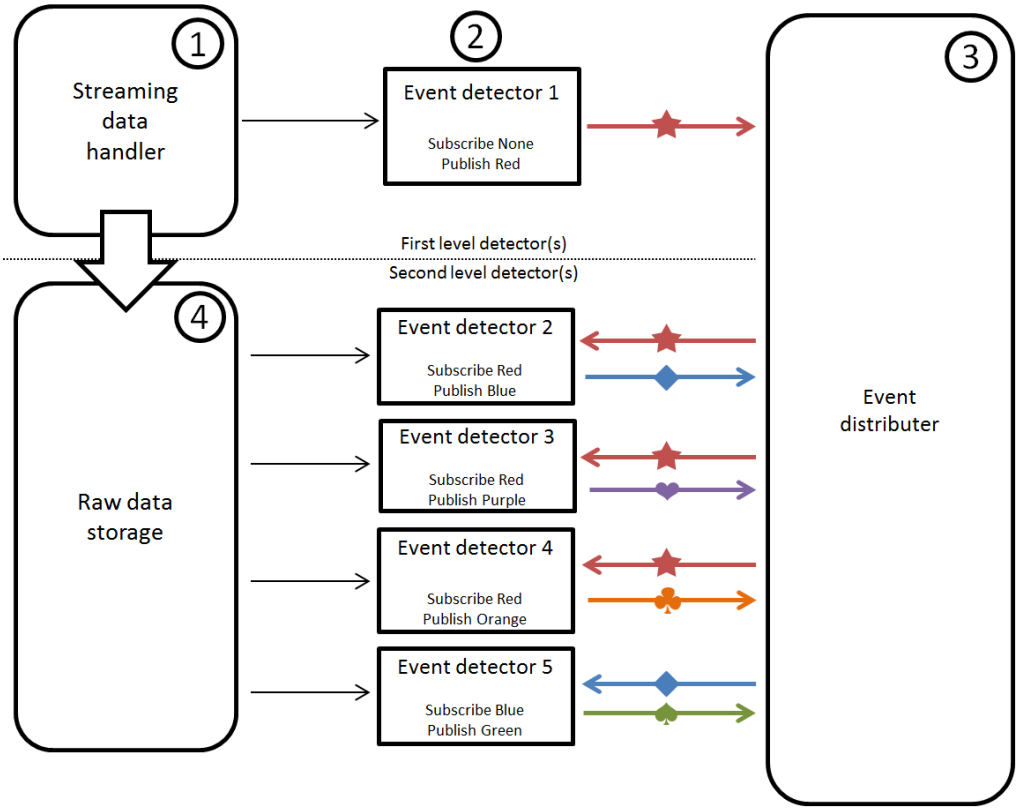


Figure 4.7: Staged event detection using publish/subscribe.

By not having to handle direct communication between event detectors it is easy to insert and remove detectors as needed, even during runtime. In the example above *Event detector 2* could crash with no consequence for *Event detector 1* which triggers it. However, it would result in no *blue diamond* events being published, leading *Event detector 5* to never doing any processing. Yet, the lack of point-to-point communication between the two detectors make *Event detector 5* run as if nothing happened.

Using publish/subscribe opens for a more scalable system. Individual point-to-point and synchronous communications lead to rigid and static applications, and makes the development of dynamic large-scale applications cumbersome. Publish/subscribe, on the other hand, opens for a more flexible configuration of the system, because we achieve loose coupling between the

different components. This simplifies the reconfiguration of the applications applying our architecture.

Another advantage of publish/subscribe is that it opens for a dynamic network topology. Distributing the event detectors across different computational nodes is just a matter of deployment, as there is no need for direct communication channels between the detectors. The communication pattern makes the system modular.

The Raw data storage (4) provides the second level event detectors with samples. All samples from the Streaming data handler (1) are passed along to the Raw data storage, as illustrated in figure 4.8. The component store all samples for a certain period of time, and delivers them to the event detectors in a pull-based fashion.

One could argue that the Streaming data handler and the Raw data storage could be a single component as they serve the same purpose — delivering samples to the event detectors on request. However, we chose to divide them, as they operate under different prerequisites.

As long as the number of data streams are constant, the load on the Streaming data handler will not change. The opposite is true for the Raw data storage which workload is variable, as a result of the unpredictable nature of the second level event detectors. Their processing is dependent on the content of the data streams, and we never know when they may request samples. To get a more predictable system we put the handling of the historic samples in a stand-alone component. This way the the indeterminable nature of the second level event detectors cannot affect the performance and latency of the Streaming data handler.

When devising an architecture like this, it is important to consider the implications of how we store and retrieve the historic streaming data. When the number of streams and event detectors grow, the way we store the historic data can affect the performance of the system. The Raw data storage might need to be distributed across nodes to handle the increased load. We have not considered the Raw data storage in a large-scale perspective in detail, but future work will include looking to the work by Hildrum et al. [18] on storage optimization for large-scale distributed stream-processing systems.

Figure 4.8 illustrates the complete architecture, including the Raw data storage (4) and the Event storage (5). While the Event distributer provides the application with access to events in real-time, we also need to supply the application with the possibility to access historic events. For this purpose the system has the Event storage (5). The component subscribes to all events published in the system, and stores them in a persistent storage.

The persistent storage is a relational database. The idea is to make it possible for the applications to form complex queries to fetch batches of related events, for instance for use in search.

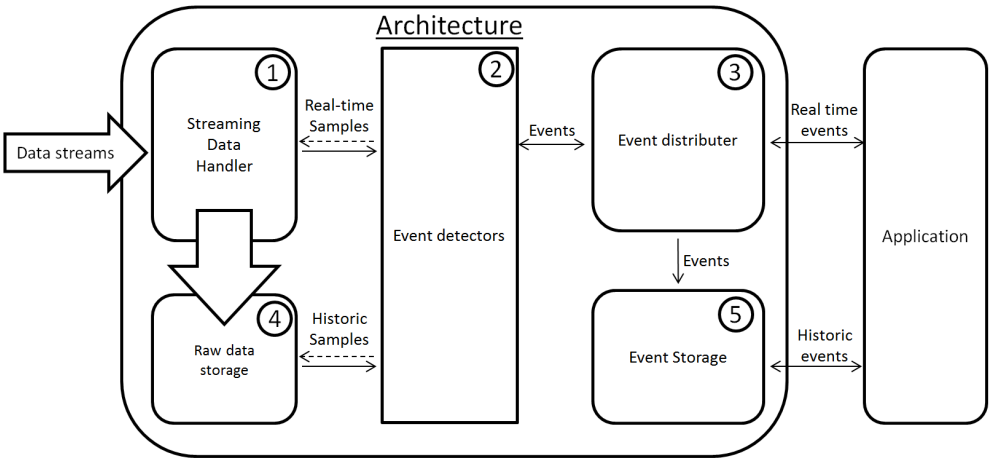


Figure 4.8: The Streaming data handler (1) passes all samples to the Raw data storage (4). The Event distributer (3) passes all events to the Event storage (5).

Chapter 5

Árvádus

This chapter describes the design and implementation of Árvádus, a prototype runtime built on the architecture described in section 4.1. The name "Árvádus " is Sami for "understanding", a reference to the underlying purpose of the runtime: to understand what is currently happening in a set of data streams. Árvádus is built to evaluate our thesis that staged processing increases the scalability of a video annotation system, hence the focus in this implementation is to create a solid foundation with enough features to enable our experiments and evaluate the architecture. Árvádus is implemented in Java 1.6.0. The runtime is illustrated in figure 5.1.

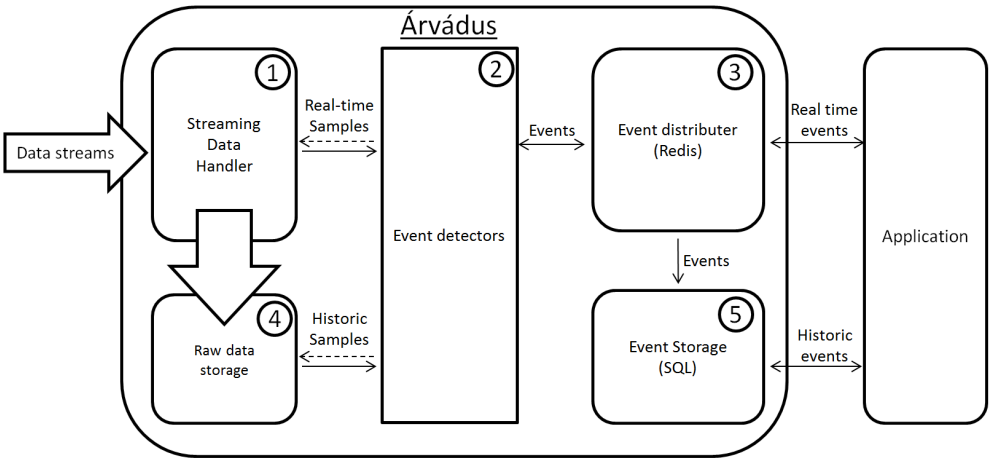


Figure 5.1: System overview.

The main components of Árvádus are:

- *(1) Streaming data handler:* This component handles the incoming data streams, and presents an interface for the event detectors to request real-time data samples. All incoming data is passed on to the Raw data storage-component.

- *(2) Event detectors:* The event detectors are responsible for the extraction of meta-data. An event detector takes a sample as input, processes it, and may produce and publish an event as output. Event detectors may be placed in stages.

- *(3) Event distributer:* This component orchestrates the flow of events through the system. It forwards the events published by the event detectors in real-time to the interested parties, which include other event detectors, the application and the event storage.

- *(4)Raw data storage:* This component stores all samples for a certain period of time. The samples are available for event detectors who require historic data for processing.

- *(5) Event storage:* This component stores all detected events. It provides the application with access to historic events.

These components will be described more thoroughly in the following sections.

5.1 Streaming data handler

The management of the incoming streams is an important task. The manner they are managed and delivered could affect both the efficiency and stability of the system. In Árvádus, the Streaming data handler manages the incoming data streams and delivers real-time samples to the first level event detectors. The communication is pull-based, meaning that the event detector requests samples when needed. Figure 5.2 illustrates the relationship between the Streaming data handler and an event detector.

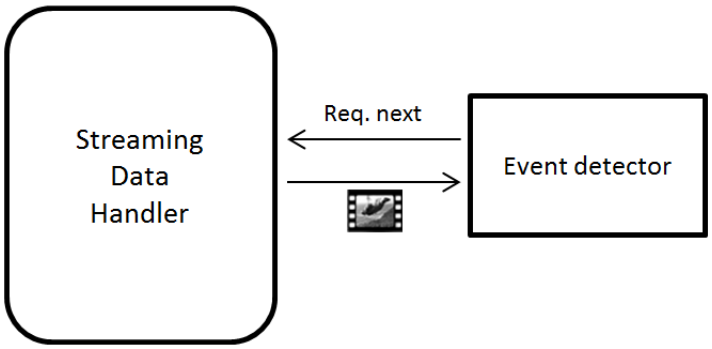


Figure 5.2: Relationship between the Streaming data handler and an event detector.

In this prototype implementation of Árvádus the outgoing sample queues are not filled with samples from live streams. For the example video streams used in the experiments we have stored the videos as a series of frames on disk. When streaming, we read a single frame at a time from disk and put them in to the queue. By using our knowledge of the framerate of the original video, we can put them into the queue at their natural speed, and thus simulate a proper video stream authentically.

Upon delivery of the samples we divide between two schemes — *best effort* and *strict*. With strict delivery every single sample is delivered to the event detector. This may lead to high latency in the system if samples appear faster than the detector can process them. With best effort this problem is avoided by always delivering the freshest sample to the event detector. If the event detector processes samples faster than they appear, it will still process every sample. However, when the processing is slow the Streaming data handler may skip delivery of some samples. This ensures that the system keeps delivering events in real-time, but we lose accuracy due to not processing every sample.

When bootstrapping the system, first level event detectors start by sending the Streaming data handler an initialization message. The message specifies what streams the event detector want samples from, and what scheme they will have them delivered with. If the initialization phase is successful, the event detector can start requesting samples.

The communication between the Streaming data handler and the Event detectors are done with a custom message protocol over TCP. A message consists of four fields: A type, a meta-data field, a field declaring the size of the payload and the payload itself. The payload is mainly used to ship binary data like video frames or audio samples. If the payload size field is `0` there is no payload attached to the message. The type is an integer defining what type of message it is. Examples of message types are initialization messages, confirmation messages, requests and sample messages. The content of the meta-data field is dependent on the message type. For instance, a sample message from a video stream contain information like *frame number*, *sequence number* and *framerate*, while a request for a frame contains a stream identifier and a frame number. The meta-data field is marshalled using JSON (JavaScript Object Notation)[1], a lightweight data-interchange format which is easy for humans to read and write and simple for machines to parse and generate. An advantage of using an open format as JSON is that we can write event detectors relatively independent of platform.

As described in section 4.1, synchronization between the different data streams is an important issue that should be handled by the Streaming data handler. However, as our experiments in chapter 6 do not involve multiple concurrent streams, this has not been implemented in Árvádus. Likewise,

[1] http://www.json.org/

as the load on the Streaming data handler is not a pressing issue in the evaluation of our thesis, the load balancing described in section 4.1 is not implemented.

5.2 Event detector

The component determining if something interesting occurs is called an event detector. An event detector takes a sample as input, processes it, and may produce an event as output. The sample type is application dependent (see table 3.1), but the output format is consistent across applications and data types. Figure 5.3 illustrates an event detector.

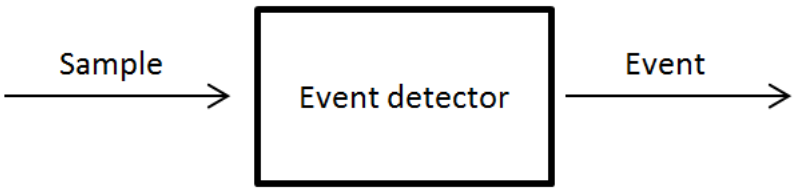


Figure 5.3: Event detector.

In the implementation of Árvádus a detector must conform to the *EventDetector* interface. The interfaces specifies one method *apply(Sample s)*, which returns an *Event* object. A *Sample* contains the sample data to be processed by the event detector, together with some meta data like *source* and *sequence id*. The *Event* class defines events. It is made to be general enough to support a wide range of event types, yet narrow enough to hold detailed information about events. The class contains the following variables:

Table 5.1: The event class

Type	**Name**
Long	timestamp
String	type
Integer	sequenceId
String	source
String	data

The *timestamp* denotes the time in which the event occurred. While it is managed as a Java Date-object during processing, it is stored as a long representing the the number of milliseconds since January 1, 1970, 00:00:00 GMT to ensure interoperability across platforms. Note that the stored time represents the timestamp of the sample the event was detected in, not the

time in which the event was detected. For some sample types the sample contains data from a time interval. In such a case, the timestamp denotes the start of the interval.

Type describes what kind of event this is. The field is application dependent, meaning that it is up to the application developer to decide what this field should include. We encourage to use event types that define both domain and type separated by a dot. Looking at the example events from illustrated in 3.2, natural event types would be names like *soccer.goal* and *movie.kiss*.

The *sequence id* identifies what sequence of events the event belongs in. A sequence may contain events from many sources, and the sources may be of different types, so *Source* is used to describe which one. For instance, if a sequence has two incoming video streams, stream one can be denoted by VIDEO1 and stream two be denoted by VIDEO2.

Data is by far the most interesting field. It is a CSV-formatted string that may contain data specific to the event type when it is needed. The field is event detector specific, but conform use of the field is encouraged. For instance, if the current event describes movement in a video, the data field may contain an indicator on how much movement there is and what area of the image the movement occurs in. Likewise, if the event is a goal in a soccer application the data field might include information about the player that scored and the team he represents.

Conceptually we divide between *first level* and *second level* detectors. The difference lies in the way the samples are requested. *First level* event detectors continuously request the freshest samples in real-time from the Streaming data handler. *Second level* detectors request samples from the Raw data storage after triggered by the events published by other detectors.

In the implementation of Árvádus this difference does not affect the way we implement the actual detectors. Instead, we wrap the detectors inside a class which handles the communication with the Streaming data handler or the Raw data storage, dependent on the current *level*. This means that we can use the same event detector as both a *first level* detector and a *second level* detector.

5.3 Event distributer

The heart of Árvádus is the Event distributer whose role is to distribute the events between the different components. The events, which originate from the Event detectors, should be passed along to other event detectors, the event storage and the application, as described in section 4.1 and illustrated in figure 5.1. The flow of events is orchestrated using the publish/subscribe message pattern, implemented using Redis[2].

[2]http://redis.io/

In principal, Redis is an key-value store, but it is often referred to as an advanced data structure server since keys can contain strings, hashes, lists, sets and sorted sets. Redis also offers simplistic publish/subscribe functionality. The Redis publish/subscribe system uses the key-value store as a central message broker which clients can publish and subscribe to. In Redis messages are published on *channels*, which subscribers in turn express interest in. For instance in order to subscribe to the channel *event.soccer.goal* the client issues a SUBSCRIBE providing the name of the channel:

```
SUBSCRIBE event.soccer.goal
```

Messages sent by other clients to this channel will be pushed by Redis to all the subscribed clients. In addition, Redis supports pattern matching. Clients may subscribe to patterns in order to receive all the messages sent to channel names matching a given pattern. For instance, a client interested in all types of soccer events may issue the command:

```
PSUBSCRIBE event.soccer.*
```

By doing this it will receive all the messages sent to matching channels like *event.soccer.goal* and *event.soccer.red_card*, but not *event.surveillance.motion*.

5.4 Raw data storage

In our architecture we defined the Raw data storage which job is to deliver historic samples to event detectors. It receives all data stream samples from the Streaming data handler, and stores them for a certain period of time. The samples may be requested by second level event detectors if they are triggered by the first level detectors. Like for the streaming data handler, the communication is pull-based.

Our implementation of the Raw data storage is limited. As we described in section 5.1, the Streaming data handler in Árvádus streams video from frames on disk. As the Streaming data handler and the Raw data storage reside on the same machine in our prototype implementation, the frames are not passed from one component to the other. Rather, the Raw data storage just access the same frames on disk as the Streaming data handler. The difference between the two components lies in the type of requests they accept. For the Streaming data handler the event detectors continuously request the next frame. For the Raw data storage the event detectors requests a specific frame.

5.5 Event storage

While the Event distributer provides the application with access to events in real-time, we also need to supply the application with the possibility

to access historic events. For this purpose Árvádus has the event storage. It consists of two components: The *preserver*, which subscribe to all events published in the system, and a *persistent storage*, where the events are stored. The event storage is illustrated in figure 5.4.

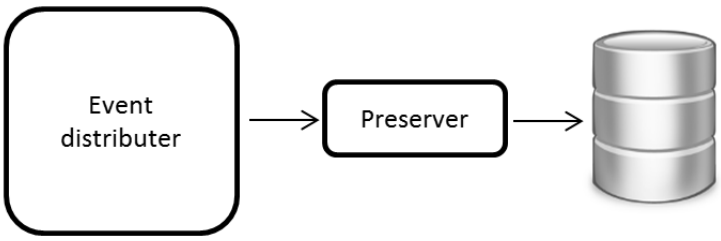


Figure 5.4: The event storage consists of the preserver and a SQL database.

The preserver subscribes to events using patterns, as described in section 5.3. This makes sure every published event is picked up and stored. The persistent storage is a SQL database. Figure 5.5 illustrates the database model, which is built around the event structure described in section 5.2. The model has five tables: Event, EventType, EventSource, SourceType and Sequence.

The Event is the central unit of the database. It contains rows of events that has been detected. The timestamp property denotes what time a particular event occurred, and the data field is used to describe it. The other fields are event type, event source and sequence, which relates to entries in the other tables.

The EventType table lists all the different event types. In addition to the name, each event type has a description and belongs in a category. Each new event in the Event table contains a reference to a event type in this table. It is the event type that decides what is contained in the *data* field in the event table.

Each event originates from a stream. In the database model a stream is represented as an event source. The list of streams are stored in the EventSource table. Because the streams can be of different types (video, audio, etc.) we have another table listing all source types.

Lastly each event is part of a particular sequence. The Sequence table lists all *sequence of events*, as described in 3.1.

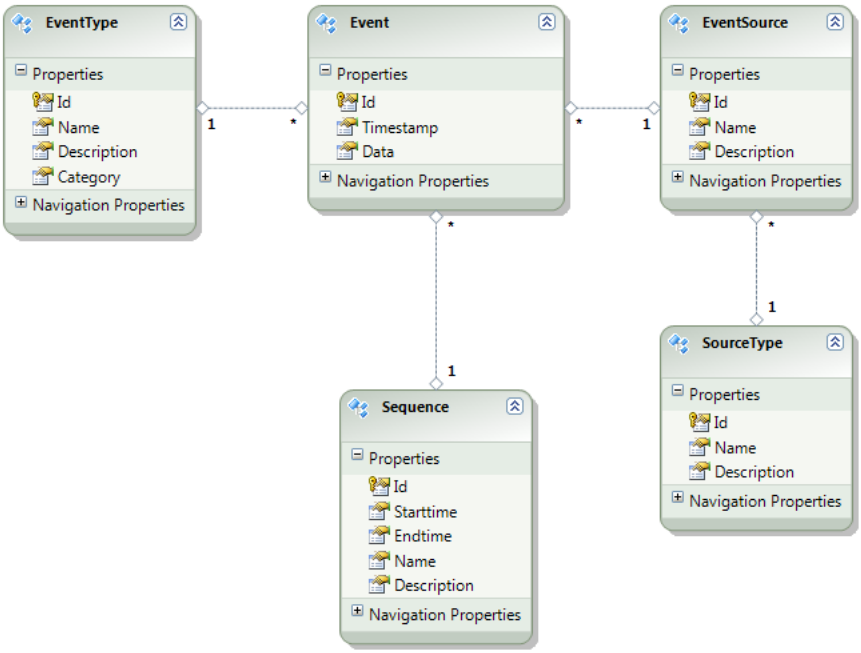


Figure 5.5: The database model.

5.6 Applicability

Árvádus is designed to be as general as possible, making it applicable to a wide range of applications in different domains. Building applications on top of Árvádus is as easy as creating new event detectors and tweaking the Streaming data handler to deliver the samples you require. When evaluating the architecture we look at two applications in two very different domains. One application process surveillance video in search for unknown people in restricted areas, the other looks at detection of notable events in soccer video.

Chapter 6

Evaluation

To evaluate the architecture devised in chapter 4.1 we have done two case studies. In both studies we implement a prototype application which utilizes the architecture in a specific domain. Both applications are built on top of Árvádus, the runtime described in chapter 5. The first application is a real-time surveillance application, where we use staged event detectors to detect faces in a set of videos. The second application is in the soccer domain, and shows how sensor data can be used to detect events which trigger video processing. Our goal is to investigate how applicable the architecture is in different application domains, and how effective the concept of staged processing is.

6.1 Case study 1: Surveillance application

An interesting domain where real-time video is a key factor is video surveillance. This is a domain where its common to monitor many video streams in parallel, where real-time delivery is critical, and where we find large potential for automation.

We consider a scenario within a huge office building where hundreds of CCTV cameras monitor the perimeter and a single guard is given the task to look for suspect activity. Fortunately, the guard has a computer system to help him. The system's primary task is to notify the guard if it detects unknown people in restricted areas.

Typically, recognizing a person in an image involves recognizing the persons face with a technique called face recognition. Face recognition is usually done by training classifiers with manually tagged images of known persons, but the training techniques used vary [19]. Before face recognition can be applied to an image, the locations and sizes of any faces must first be found. This process is called *face detection*, and is one of the most applied applications in computer vision.

To explore the value of staged processing in a video annotation context,

we have created a prototype application as a subset of the described surveillance computer system. In this prototype we apply face detection to frames from a single video stream to measure the impact our architecture has on accuracy and efficiency. The face detection is implemented using functions from OpenCV[1]. The stream is simulated from images on disk, as described in section 5.1. The prototype takes video streams as input and outputs events every time a face is detected. We have not implemented a graphical user interface for the application, as it is the processing measurements which are of importance.

6.1.1 OpenCV

OpenCV (Open Source Computer Vision) [20] is an open source library of programming functions for computer vision. The library was originally written in C, but there exits wrappers for most languages and platforms. As Árvádus is built in Java, we utilize JavaCV[2]. The library has more than 2000 optimized algorithms, and can amongst other things be used to transform images, recognize patterns, track motion in 2 and 3 dimensions and do 3D reconstruction from stereo vision.

In this prototype application we utilize OpenCV to detect faces in images. The library supplies a detection function that use Haar-like features [21] to detect features in images. The object detector takes a classifier model and an image as input, and outputs a set of rectangles marking the found objects, if any. The detector is trained with a few hundreds of sample views of a particular object (i.e., a face or a car) called positive examples, and a set of negative examples which do not contain the object. We utilize a face classifier model provided with the library named *haarcascade_frontalface_alt_tree.xml*.

6.1.2 Experiments

In this experiment we use Árvádus as the foundation to process a single video stream in search of faces. We want to investigate if the computational time spent on processing video decreases when introducing staging of processing elements.

The video stream is 3 minute long, has a resolution of 640x360 pixels, and a frame rate of 30 frames per second. It was recorded at the University of Tromsø to simulate the video stream of a static surveillance camera from a relatively busy hallway. Figure 6.1 shows an example frame from the video, where a detected face has been marked. In the terms of the general model (3.1) we consider each frame as a sample.

[1] http://opencv.willowgarage.com/
[2] http://code.google.com/p/javacv/

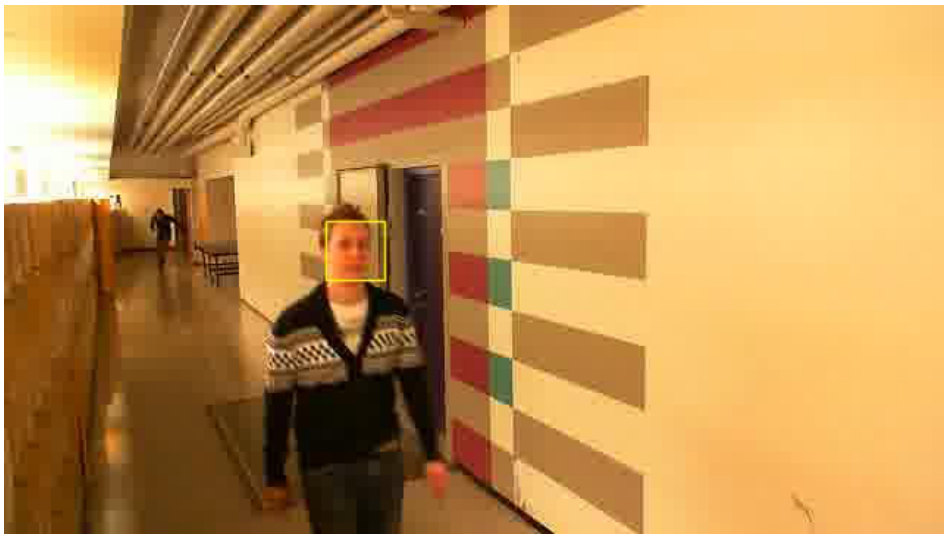


Figure 6.1: Frame 3662 from the processed video stream.

In principle, we only need one detected face of each person passing the camera to enable face recognition. However, for this experiment we keep processing every frame for faces, even though the result is that we detect faces of the same person multiple times.

First, we process every single frame of the video in search of faces. In other words, we apply face detection as a first-level event detector, and process all 5400 frames of the stream using the strict delivery scheme described in section 5.1. Next we implement the processing using two-staged processing. Our conjecture is that frames without motion is less likely to contain faces, thus processing motionless frames may prove to be a waste of time. Consequently, our event detector in stage one is a simple motion detector where we analyse the amount of motion from frame to frame.

The motion detector is implemented by measuring the absolute difference in pixels from one frame to the next. The difference is computed after smoothing out the frames using Gaussian blur. The visual effect of this gaussian blur resembles that of viewing an image through a translucent screen, and it is done to reduce image noise and thus increase the accuracy of the detector. Figure 6.2 shows two following frames and the calculated difference between them after smoothing.

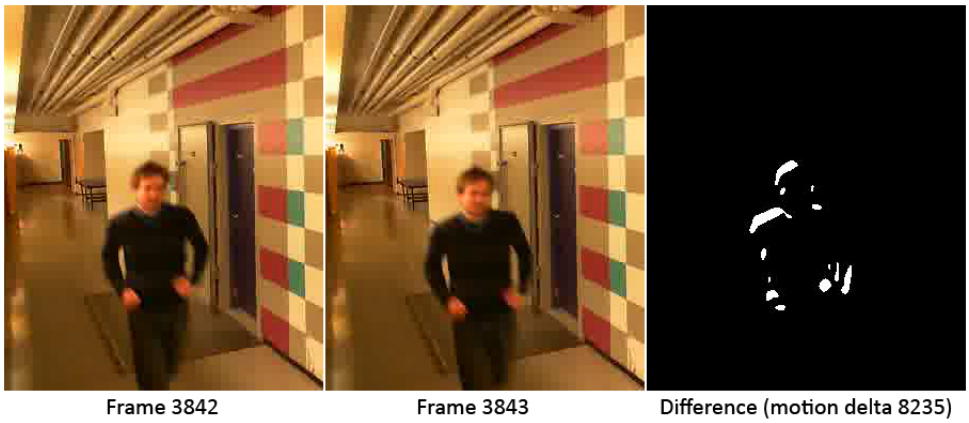


Figure 6.2: Two frames from the video stream and the difference between them. The absolute difference represented in this image is calculated to be 8235.

We refer to the difference between the frames as the *motion delta*. If a sufficient amount of pixels have changed, we publish an event which in turn triggers face detection on the current frame. We refer to the limit for what is considered a sufficient amount of pixel change as the *motion delta threshold*. As the *motion delta* is an absolute value independent of resolution, framerate and color depth, the *motion delta threshold* needs to be tweaked to each video stream accordingly.

In the experiment we look at how different thresholds affect the computation time and accuracy of the processing. Figure 6.3 show how the computational time change with the motion delta threshold.

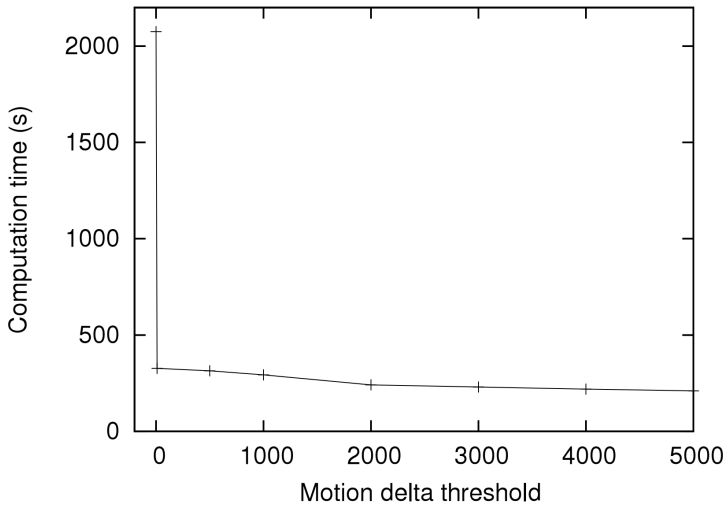


Figure 6.3: Computational time for processing 3 minutes of video.

In the line chart, the time used when processing every frame of the stream is represented by the datapoint where *motion delta threshold* is 0. As expected, processing every frame of the stream is expensive. Over 2075,9 seconds, or 34 minutes, is needed to run the face detection algorithm on the 3 minutes of video. This is not even close to meeting our real-time criteria. When filtering the stream with a *motion delta threshold* of 1, which means face detection is applied when we detect the least measurable amount of movement, the computational cost decrease drastically from 2075,9 seconds to 326,3 seconds. That is a speed-up of 84,3%.

However, increasing the motion delta threshold further does not impact the computational time to the same degree. A threshold of 500 gives a computation time of 313,9 seconds, only 12,4 seconds faster than for a threshold of 1, and with a threshold of 1000 the processing takes 293,1 seconds. The initial gain of applying a filter at all heavily outweighs the gain of increasing the threshold. Nevertheless, when comparing a threshold of 5000 with the minimal threshold of 1 the speed-up, of 116,3 seconds, is quite substantial. With this we have proved that applying staged processing with motion detection as a filter for face detection results in a considerable speed-up, but the question whether we manage to maintain the same accuracy for the face detection in stage two remains.

To explore how the accuracy change when using a motion detector as a first stage filter, we counted the number of faces detected. Figure 6.4 show how the number of faces detected change with the motion delta threshold.

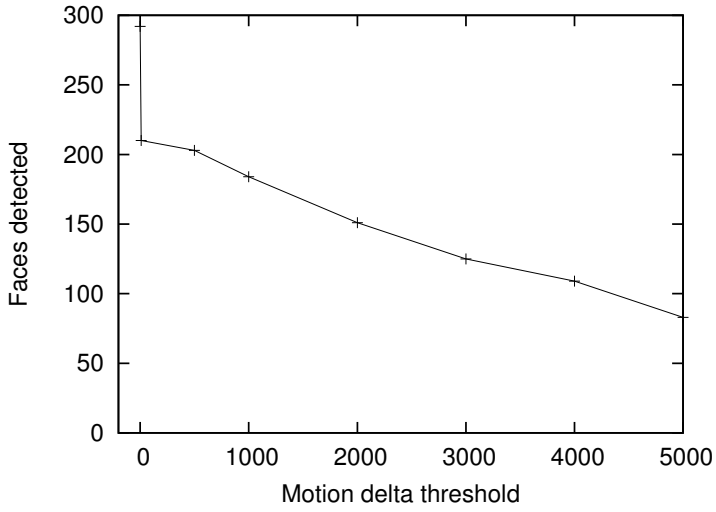


Figure 6.4: Accuracy when processing 3 minutes of video.

When processing every frame the detector found a total of 292 frames with faces. By applying the motion detector with a threshold of 1 the

number drops to 210. This corresponds to a reduction of 28,0%. This is sensible. Processing fewer frames will naturally lead to less faces detected. However, our conjecture was that frames without motion was less likely to contain faces. This appears to hold some truth if we compare the reduction in computation time with the reduction in number of faces detected. The time spent processing decreased with 84,3% while the reduction in detected faces is on only 28,0%. We do lose some accuracy, but the gain up in speed is more significant. In other words, most of the frames we filter out in stage one does not contain faces.

Increasing the threshold further leads to a linear fall in the number of faces detected. With the *motion delta threshold* set to 5000 we only detect 83 faces in the video. Again sensible, because an increased threshold leads us to processing fewer frames. However, now we save computation time by not processing frames that are likely to contain faces. Thus, the number of faces detected decrease faster, and now in correlation to the decrease in computation time.

To further evaluate how staged processing affects the accuracy, we look at the difference in the number of false positives between the two approaches. A false positive is a result that is erroneously positive when a situation is normal. In this context this means that the face detector finds and marks a face where there is no face. Figure 6.5 shows a false positive marked in frame 2228 of the video stream.

Figure 6.5: Frame 2228 from the processed video stream is a false positive.

To perform this experiment we counted the number of false positives among the detected faces. Note that we are not doing this to analyse the implementation of the face detector, but to get a better understanding on

how staged processing affect the use of it in our prototype system. Figure 6.6 illustrates the number of false positives we found with and without motion filtering in a bar chart.

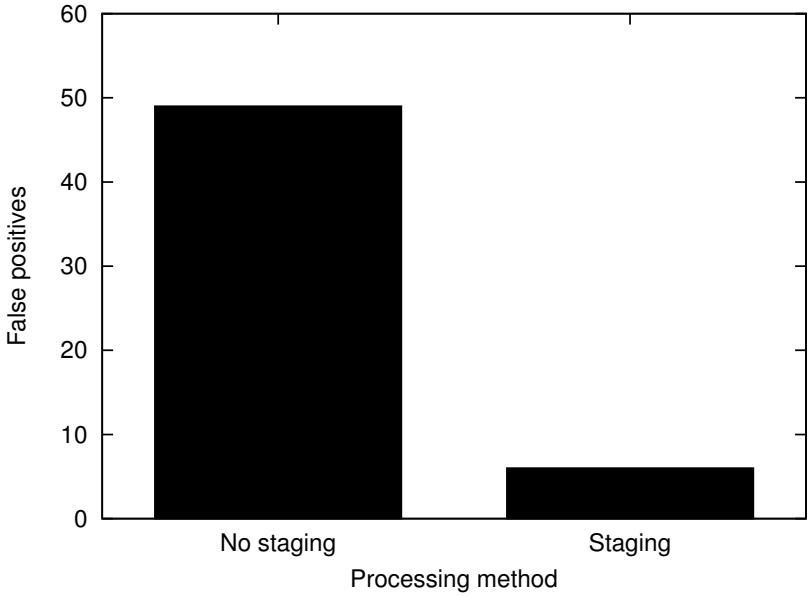


Figure 6.6: False positives.

When processing every frame the detector found a total of 292 frames with faces, and 49 of these are false positives. When staging the processing with a *motion delta threshold* of 1 we detect only 6 false positives. In other words, most false positives are detected in sections of video without motion. This supports our conjecture that frames without motion was less likely to contain faces.

The improvement in accuracy becomes even clearer by comparing the difference in percentage of false positives between the two approaches. When processing every frame 16,9% of the detected frames are false positives. When using staged processing only 2,8% are false positives. By applying this knowledge when looking at the decrease in accuracy in percent by removing the false positives from the equation, we see that the reduction in detected faces is only 16.0%.

6.1.3 Conclusion

The experiments done with the surveillance application show that applying staged processing can indeed decrease the computation time. The speed-up when applying *motion detection* as a filter for *face detection* is over 84%, while the decrease in detected faces is only 28.0%. When adding the

number of false positives into equation the reduction in detected faces is barely 16.0%. Knowing that each face is detected multiple times, this can be argued to be a good trade off.

The results indicates that staged processing can be used to reduce the computational cost associated with annotating video streams in real-time without losing too much precision. It does not prove that staged processing will work in every domain and for any case, but it shows that the architecture has some application areas where the pay-off is significant.

6.2 Case study 2: Soccer application

Sports video is another interesting domain where real-time video is a key factor. This is a domain where it is common that the same event is covered by multiple video streams, where real-time delivery is critical for the viewer, and where we find a large potential for automation. We think automatic annotations of events in sports video can be a driver for future advanced personalized user experiences [2].

We have previously investigated how we can reduce the computational cost of real-time meta-data extraction by staging cheap video processing ahead of heavy processing. However, we conjecture that we can use external sources of information as filters in stage one of the processing pipeline to increase the speed-up even further.

In the soccer domain, it has become more and more common for players to wear sensors that track their position during the game. To investigate how we can use external data to trigger video processing of sports videos, we explore two prototype applications using such soccer sensor data. The applications are built using Árvádus, and takes a ZXY Sport Tracking[3] sensor data stream as input and produces events describing the soccer game as output. We do not process any video in our experiments, rather the focus is on how we can use sensor data to trigger potential video processing.

6.2.1 ZXY Sport Tracking System

The ZXY Sport Tracking system (ZXY) is a product from the Norwegian-based company ZXY Sport Tracking AS. It is a radio-based positioning system that provides analytic information about both physical and tactical performances in real-time. Two of the top soccer clubs in Tippeligaen, the Norwegian Premier League, are currently utilizing ZXY Technology for analyses purposes: Tromsø IL and Rosenborg BK.

ZXY works by having players wear sensors that transmit data to dedicated receivers placed around the field. The sensors are all stationed in the ZXY Sports chip, which is worn by the players on a belt around the

[3]See: http://www.zxy.no

waist. The sensors continuously monitor the players' actions on the sport field, recording factors as position, heading, effort and pulse. The chip reports to the ZXY Positioning Sensor at up to 40 times per second. The ZXY Positioning Sensors, which utilize RadioEye technology from Radionor Communication AS, is able to compute the position, direction and speed of a wireless device. Hence, positioning data is also made available for analyses. All recorded data is stored in a network connected SQL database in real-time.

To explore the data, ZXY provides a PC suite and web interface. The PC Suite includes a 3D graphics user-interface, providing users with the possibility to watch an animated reenactment of the game with on-screen tools available to add visual effects like rubber banding and measurements. The web interface provides access to analytics data like meters run, effort, and other aggregated data.

For this and future projects, the iAD centre has entered a collaboration with ZXY Sports Tracking AS and Tromsø IL. We have gotten access to tracking data recorded from soccer matches played on Alfheim Stadion, and we will use them for research purposes and prototype application development.

6.2.2 Prototypes

For both of the soccer prototypes we use Árvádus to process ZXY sensor data. While we do discuss video processing for the prototypes, this has not been implemented. The goal is to investigate how we can extract semantics from sensor data in real-time in order to trigger video processing, and the focus is to show how Árvádus can be used to achieve it.

We use ZXY records from the Tippeliga match between Tromsø IL and IK Start played April 4th 2011 as the streaming data. The records only describe the players on Tromsø IL, as the player on Start did not wear the sensors. Nevertheless, by using authentic match data we can realistically explore how to process positioning data in real-time in order to create exciting applications.

Table 6.1: Example of basic ZXY snapshots for one player.

Timestamp	**PID**	**First name**	**Last name**	**X pos**	**Y pos**
19:05:48	1	Sigurd	Rushfeldt	38	14
19:05:49	1	Sigurd	Rushfeldt	39	12
19:05:50	1	Sigurd	Rushfeldt	39	11
19:05:51	1	Sigurd	Rushfeldt	40	10

The records consist of a series of snapshots describing the current status of a single player. Besides containing basic information describing the player

like name, player id and team id, the snapshot contains detailed information about current effort, pulse, position and heading. ZXY registers and stores this data 20 times each second. At the same time, the system continuously aggregates the data about each player, and computes live statistics like total distance and total effort.

Table 6.1 shows four stripped down ZXY snapshots from Tromsø IL's top scorer Sigurd Rushfeldt. The snapshots tell how Sigurd moves inside the penalty box over a 4 second period from 19:05:48 to 19:05:51 in the match between Tromsø IL and IK Start. By default ZXY registers position with high granularity in centimeter precision. However, the records we use is aggregated. Instead of 20 snapshots per player per second, we get one snapshot per player per second. The position is the average position within one second, and is represented in meter precision.

The positioning data from the ZXY sensors is stored as Cartesian coordinates, where the coordinate system has its origin in one of the corner arcs of the field. The coordinate system is adjusted to the size of the pitch of Alfheim stadion, the home ground of Tromsø IL. The pitch is 105 metres long and 68 meters wide. Figure 6.7 illustrates the coordinate system.

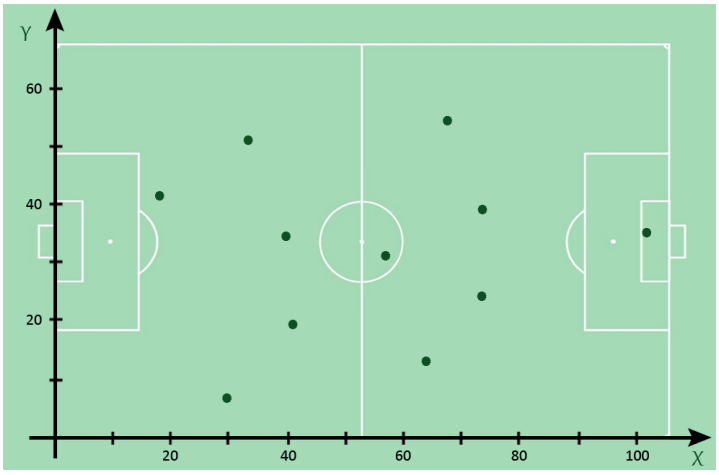


Figure 6.7: Cartesian coordinate system for player position in ZXY, scaled to the size of Alfheim stadion. The points are players.

In our system we divide between two types of *samples* delivered by the Streaming data handler: player samples and team samples. A player sample corresponds to a snapshot as described above. A team sample contains a set of player samples. The set consists of the last player sample from every active team member.

While the stream of ZXY sensor data samples can be simulated to authentically arrive each second over a 90 minute period, our experiments are done by processing the data as fast as possible. It is currently not possible

to request the next player sample for a specific player. When the event detector requests the next player sample the response is the next available sample in the stream.

6.2.2.1 Match events

For our first soccer prototype application we consider an application where a user subscribes to interesting events in a set of soccer matches. The user defines what types of event he is interested in, and will always get video from the match that currently matches his criteria best. Events are detected automatically on the fly and video showing the events are delivered to the user in real-time.

Some events that a viewer may consider of interest include goals, corner kicks, free kicks, penalties and cards, but also more loosely defined events like counter-attacks and breakaways. While we conjecture that events can be detected by recognizing certain player position patterns in the ZXY sensor data, other events may need to be extracted through video processing. One of the drawbacks of ZXY is that it does not tell anything about the ball or the referee. This is an area were video processing can compliment it.

For our proof of concept ZXY detector we have implemented a naive corner kick detector that looks for a certain pattern in the position of the players of a team. The detector takes a ZXY team sample as input and may produce a corner event as output. The ZXY team sample contains the positions of all players of one team.

The detector looks for players within a 2 meter radius of the corner arc and counts the number of players within the penalty box. If the number of players within the box is greater than three and there is a player near the corner flag, we define it as a corner. To avoid publishing an event about the same corner situation several times, the detector does not publish more than one event each playing minute. This is a reasonable assumption, knowing the normal frequency of corner-kicks. Figure 6.8 illustrates typical player positioning during a corner kick that would be detected by our event detector.

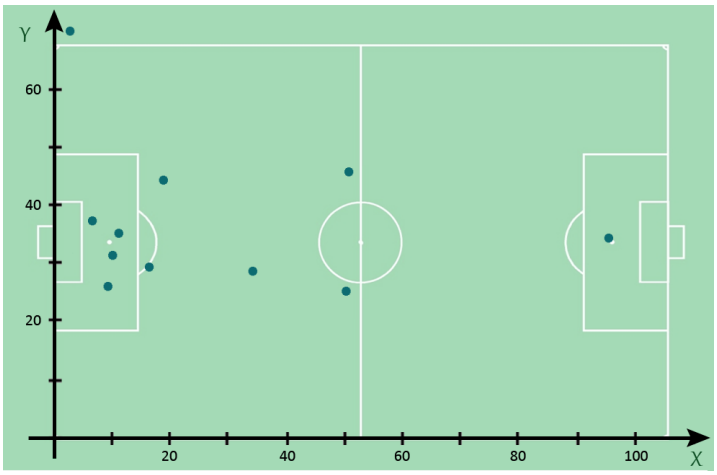


Figure 6.8: Corner kick positioning with players from a single team.

To evaluate the detector we used it to process the ZXY records from the match between Tromsø IL and IK Start. Processing the entire 90 minute match for corners took 17 seconds, proving that sensor data can be used to extract events in real-time. With the detector we are able to detect all of the 5 corners Tromsø had during the match. That said, the detector also finds 1 false positive when the Tromsø player Hans Åge Yndestad is positioned close to Tromsøs own corner arc during an offensive throw-in from Start.

We envisage using similar approaches to detect other events. Figure 6.9 illustrates three other situations where we can exploit knowledge of typical position patterns to pin point notable events. For these situations we assume we have position data for players from both teams.

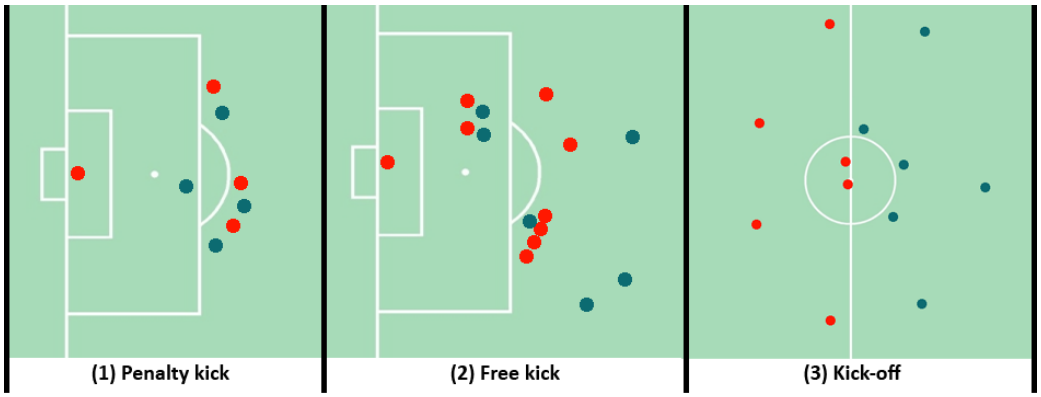


Figure 6.9: Position patterns revealing notable events.

A penalty kick (1) can be detected by looking for situations where a single attacking player is inside the penalty box and a group of players from

both teams is situated just outside it. A free kick (2) in a dangerous position is characterised by the opposing team lining up a wall of players, trying to block a potential shot by creating a human barrier. This straight line of players from the same team does not normally occur unless there is a free kick. A kick-off (3) is characterised by having two players from the same team positioned inside the centre circle and the remaining players from both teams are located on the their respective halfs.

In all of the proposed patterns we use static positions to detect events, but we envisage evaluating movement to create even more accurate event detectors. Statefull detectors that remember the last known positions of all team member can not only compute movement vector patterns to increase the accuracy of the already proposed detectors, but also create new detectors. For instance, we conjecture that we can look for sudden shift in average movement and direction to detect the beginning phase of a counter-attack.

However, some events are not detectable through ZXY sensor data. For instance, the actions of the referee is not present in the ZXY records. For that reason we envisage using a video processing algorithm to detect when the referee hands out yellow and red cards. However, on the basis of the experiment done in section 2.2.3 we understand that continuously processing the video stream in look for cards may not be practically possible in real-time.

We propose using two-staged processing, as illustrated in figure 6.10. By looking for free-kick situations (2) through a first-level event detector, we can trigger the search for cards through video processing in level two as soon as they appear. This is a good example of how sensor data can be used to trigger video processing.

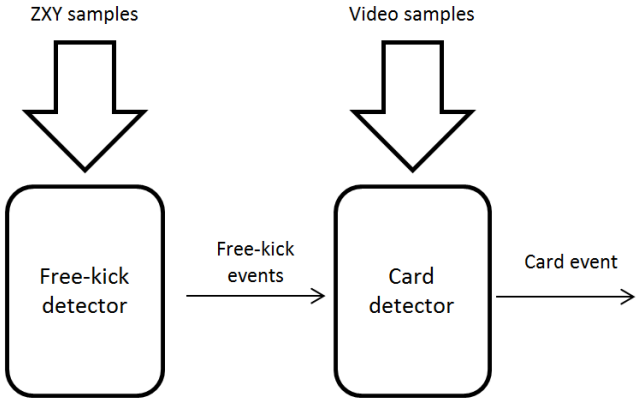


Figure 6.10: Two-staged card detection.

While we manually created the corner detector, we imagine using ma-

chine learning and SVMs (see section 2.2.2) to create more effective sensor data event detectors. The SVM can be trained to detect specific soccer situations by creating a model through a large set of positive and negative example patterns.

6.2.2.2 Player subscription

For our second soccer prototype we consider a media application where the viewer of a soccer game will get a computer produced video stream based on his preferences. The viewer can choose to subscribe to the ball or to a specific player, and will always get video from the camera that covers his interest. Which camera to view is computed on the fly, and the video is delivered to the user in real-time.

We build the application on top of Árvádus, and create a set of event detectors to help us select which stream to present to the viewer.

We envisage that Alfheim stadion has installed four static cameras along one of the sidelines. Each camera covers one zone of the field. The setup is illustrated in figure 6.11. It is natural to assume that the view of the cameras overlap, but this is not illustrated in the figure.

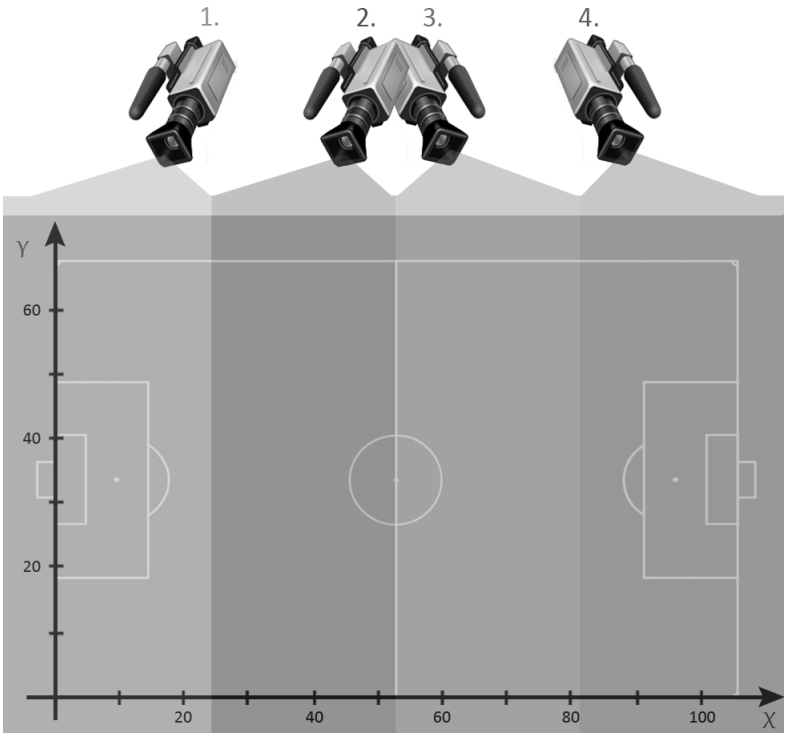


Figure 6.11: Camera setup.

If the viewer decides to subscribe to a player, we need to process the ZXY

sensor data to determine where the player is positioned. We implemented a *zone change detector* which continuously requests player samples from the Streaming data handler, process them, and may produce events that are published to the application.

The event detector knows which camera covers which zone, and keeps state about the last known zone each player was in. The previous zone is stored in a hash-map to ensure quick look-up, with the *player id* as key and the *zone number* as value. For each received player sample the detector computes the current zone of the player, and compares it with the previous zone of the same player. If the two zones are different, the detector publishes an event to the Event distributer. Figure 6.12 illustrates a zone change where the a player moves from *zone two* to *zone one*.

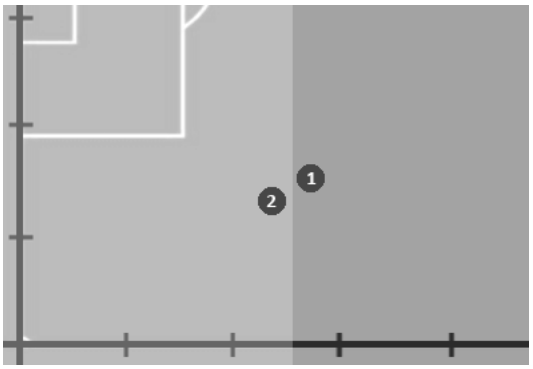


Figure 6.12: Example of zone change.

Events will be published for every player that switches zone. The event published on a zone change contains information about the current player, what zone he moved from, and what zone he moved to. The end-user application can use this to switch the video stream when the currently concerned player changes zone.

If the viewer decides to subscribe to the ball we need to adjust our approach, as information about the ball's whereabouts is not available through ZXY. Specialized computer vision techniques have been proposed to detect and track the ball in soccer broadcast video [22] [23], and the results are promising. We envisage using such techniques to implement a ball detector that processes video samples. The questions is which of the four streams to process samples from. Processing every stream may be too computationally costly. One solution is to approximate which camera zone the ball is currently in by utilizing ZXY sensor data.

To detect the ball we propose a three-staged processing pipeline, as illustrated in figure 6.13. The first-level event detector is the *zone change detector* described above. Zone change events trigger the second stage: the *important zone detector*. This detector keeps an aggregated view of all the

players' zone changes. It uses the number of players in each zone to rank the zones. The zone with most players is considered the most important, and so forth. Every time the ranking of the two top positions change, the detector publishes an *important zone change event*. The event contains data about which two of the four zones is currently considered the most important.

The important zone change events are used to trigger the *ball detector*. When triggered, the ball detector requests the last sample from what is currently considered the two most important camera zones. In other words, the zones we approximate the ball to be in. The detector proceeds to process both samples to decide which of them currently contains the ball.

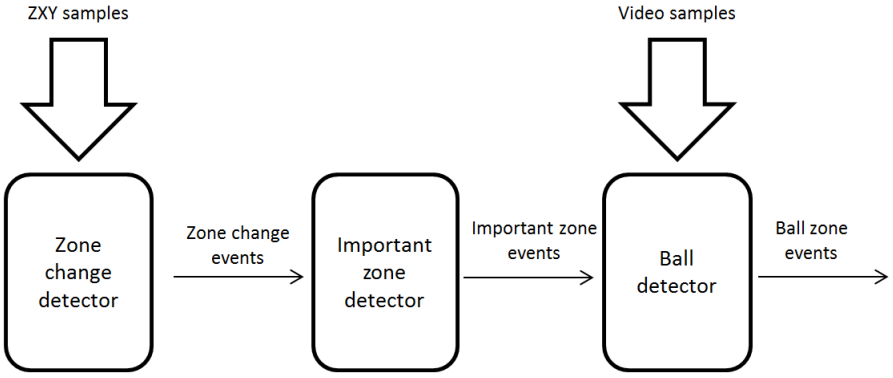


Figure 6.13: Three-staged ball detection.

This example shows that it is possible to use sensor data events to trigger processing of video data. It illustrates how we can use simple event detectors as building blocks to create more advanced event detectors. This is one of the key strengths of Árvádus. The publish/subscribe plug-in architecture makes it possible to compose advanced dependency graphs between detectors without tight programatically dependencies.

6.2.3 Conclusion

The case study done with the prototype soccer applications shows the importance of external sources when annotating video. With the use of ZXY sensor data we are able to detect important soccer events like corners, free-kicks, counter-attacks and kick-offs.

The case study illustrates the applicability of our architecture. Árvádus can be used to annotate video through sensor data as well as video data. More importantly, the prototype shows how it is feasible to use cheap processing of sensor data to trigger heavy processing of video data, and thus decrease the computational cost associated with the meta-data extraction.

One thing to take note of is that this is an application domain with large potential, where Árvádus can be used to realize interesting real-time

applications.

6.3 Summary

In this chapter we have evaluated our architecture by implementing prototype applications in two different application domains. The results of our experiments indicate that Árvádus is applicable in a range of domains where extraction of meta-data in real-time is necessary.

The experiments done with the surveillance application prove that staged processing can decrease the computation time of meta-data extraction. The speed-up when applying *motion detection* as a filter for *face detection* is over 84%, while the decrease in accuracy is barely 16.0%. The results does not prove that staged processing will work in every domain and for any case, but it suggests that our architecture has some application areas where the pay-off is significant.

The case study done with the prototype soccer applications illustrates the applicability of our architecture. Árvádus can be used to annotate video through sensor data as well as video data. More importantly, the prototype shows how it is feasible to use cheap processing of sensor data to trigger heavy processing of video data, and thus decrease the computational cost associated with the meta-data extraction.

By employing publish/subscribe as a foundation for communication between event detectors, Árvádus becomes a modular runtime that makes it easy to stage event detectors in advanced dependency chains. The design also makes it easy to develop and reuse event detectors, as they are standalone components with no external dependencies.

Chapter 7

Conclusion

This chapter will present our achievements, conclude our work, and outline possible future work

7.1 Achievements

This thesis describes and evaluates an architecture for annotation of video streams in real-time. The problem definition we defined in section 1.1 is stated below:

This thesis shall develop and study aspects of a software architecture that enables real-time annotation of multiple live video streams. The architecture is intended for use within media rich applications where extraction of video semantics in real-time is necessary. A working prototype applying the architecture will be developed and evaluated in a scientific context.

We have proposed an architecture built on the idea of staged processing. Our thesis was that staging video processing in levels would make room for a more scalable video annotation system. We conjectured that using simple and cheap processing as filters for more heavy processing would decrease the total computational cost and enable real-time annotation.

To evaluate our thesis we have built the runtime Árvádus which applies our architecture. Árvádus realizes chaining of processing elements by using a combination of the publish/subscribe message pattern and pull-based communication. Publish/subscribe is used for transport and delivery of events. Pull-based communication is used to retrieve samples for processing.

To evaluate the architecture we have performed two case studies. In both studies we implemented a prototype application utilizing Árvádus as the foundation. Both applications are media-rich applications where extraction of video semantics in real-time is necessary. The first application is a real-time surveillance application, where we use staged event detectors to detect

faces in a set of videos. The second application works on real-time sports video, and explores how we can extract annotation from sensor data and use it to trigger video processing. Our goal was to investigate how applicable the architecture is in different application domains, and how effective the concept of staged processing is.

In the evaluation of the surveillance application we investigated how staged processing affects the speed and accuracy of meta-data extraction. We wanted to see the impact of letting cheap video processing trigger heavy video processing. Our conjecture was that frames without motion are less likely to contain faces. Consequently, we created a motion detector which analyzes the amount of motion from frame to frame to act as a filter for the face detection in stage two. The results were acceptable. The speed-up when applying *motion detection* as a filter for *face detection* is over 84%, while the decrease in accuracy is barely 16.0%. The experiments show that applying staged processing indeed decreases the computation time. While it does not prove that staged processing will work in every domain and for any case, it strongly suggests that our architecture has some application areas where the pay-off is significant.

In the evaluation of the soccer application we investigated how real-time sensor data can be used to trigger video processing. Our conjecture was that we can use external sources of information as filters in stage one of the processing pipeline to increase the speed-up even further. We developed two prototypes which process ZXY sensor data for meta-data. The events generated by the the prototype shows how it is feasible to use cheap processing of sensor data to trigger heavy processing of video data, and thus decrease the computational cost associated with the meta-data extraction.

7.2 Concluding Remark

As real-time media rich applications rely on live streams of meta-data describing the video content, it is a problem that the general amount of video meta-data today often is limited to titles, synopsis and a few keywords. The goal of our work was to develop an architecture for annotation of video streams in real-time for use in such media-rich applications. Our experiments have shown that our proposed architecture can decrease the computational time associated with meta-data extraction. This is accomplished by staging the meta-data extraction in levels, using cheap processing as filters for more heavy processing. By realizing the staging through a combination of the publish/subscribe message pattern and pull-based communication, we have created a loosely-coupled and modular architecture that makes it easy to develop reusable processing elements, and to distribute them across nodes.

7.3 Future work

The implementation of Árvádus is not complete. Our focus in the implementation phase was to develop a solid foundation with enough features to enable our experiments and evaluate the architecture. While this was accomplished, more work lies ahead in order to complete the runtime.

The most important part missing is the component taking care of incoming video streams in the Streaming data handler. In the current implementation, streams are simulated using frames stored on disk. We have previously investigated [24] how to extract frames from Smooth Streaming videos in real-time, and proven it to be feasible. We envisage using a similar approach within Árvádus by utilizing FFMPEG[1] to decode the incoming video stream. We will proceed to partition the video into samples of frames or groups of frames, and queue it up for the event detectors to request.

A proper Raw data storage is also missing in the prototype implementation. In the experiments we utilized streams already present on disk, so no live storage of the samples were required. We plan storing the samples individually, but grouped by stream. A memory cache will be implemented to increase response time. Future work will also consider the Raw data storage in a large-scale perspective, looking to the work by Hildrum et al. [18] on storage optimization for large-scale distributed stream-processing systems.

We envisage that the work on Árvádus can be used to extend DAVVI [2]. DAVVI has been demonstrated in the domain of sports video using a soccer example, where the main meta-data source is TV broadcasting and newspaper cites that provide *live text commentary web pages* for soccer video. We think that Árvádus' ability to extract meta-data from different sources in real-time makes it applicable as an extension to DAVVI's suit of metadeta extraction tools. In the future we would like to combine the efforts of Árvádus and DAVVI in an application in the soccer domain. Such an application could be composed by utilizing Árvádus to annotate events in soccer video through ZXY sensor data, and to use DAVVI for the dissemination of video and as an interface to the user.

While our experiments showed the applicability of our architecture in the surveillance and sports domain, we reckon that Árvádus can be used as a foundation in a range of different domains. Future work would include exploring how Árvádus can be used to drive next-generation media applications where annotated video streams are used to create personal user experiences.

[1] http://ffmpeg.org/

References

[1] Cisco, "Cisco visual networking index: Forecast and methodology, 2009-2014," Cisco, June 2010.

[2] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, A. Kvalnes, C. Gurrin, S. Zav, B. Olstad, E. Aaberg, T. Endestad, H. Riiser, C. Griwidz, and P. Halvorsen, "Davvi: a prototype for the next generation multimedia entertainment platform," in *Proceedings of the seventeen ACM international conference on Multimedia*, ser. MM '09. ACM, 2009, pp. 989–990. [Online]. Available: http://doi.acm.org/10.1145/1631272.1631482

[3] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, pp. 303–338, September 2009.

[4] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Commun. ACM*, vol. 32, pp. 9–23, January 1989. [Online]. Available: http://doi.acm.org/10.1145/63238.63239

[5] M. Weise and D. Weynand, *How Video Works (2nd Edition)*, E. Schumacher-Rasmussen, Ed. Focal Press, 2007.

[6] M.-H. Yang, D. Kriegman, and N. Ahuja, "Detecting faces in images: A survey," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, pp. 34 – 58, 2002.

[7] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the fifth annual workshop on Computational learning theory*, ser. COLT '92. New York, NY, USA: ACM, 1992, pp. 144–152. [Online]. Available: http://doi.acm.org/10.1145/130385.130401

[8] D. Lowe, "Object recognition from local scale-invariant features," *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, pp. 1150 –1157, 1999. [Online]. Available: http://dx.doi.org/10.1109/ICCV.1999.790410

[9] H. Bay, T. Tuytelaars, and L. V. Gool, "Surf: Speeded up robust features," in *In ECCV*, 2006, pp. 404–417.

[10] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001, software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[11] A. F. Smeaton, P. Over, and W. Kraaij, "Evaluation campaigns and trecvid," in *MIR '06: Proceedings of the 8th ACM International Workshop on Multimedia Information Retrieval*. New York, NY, USA: ACM Press, 2006, pp. 321–330.

[12] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 230–243. [Online]. Available: http://doi.acm.org/10.1145/502034.502057

[13] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, and G. Inc, "Interpreting the data: Parallel analysis with sawzall," in *Scientific Programming Journal*, 2005, p. 2005.

[14] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://portal.acm.org/citation.cfm?id=1251254.1251264

[15] "Hadoop: Open source implementation of mapreduce." [Online]. Available: http://hadoop.apache.org/

[16] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24. [Online]. Available: http://portal.acm.org/citation.cfm?id=1317533.1318097

[17] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[18] K. Hildrum, F. Douglis, J. L. Wolf, P. S. Yu, L. Fleischer, and A. Katta, "Storage optimization for large-scale distributed stream-processing systems," *Trans. Storage*, vol. 3, pp. 5:1–5:28, February 2008. [Online]. Available: http://doi.acm.org/10.1145/1326542.1326547

[19] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, "Face recognition: A literature survey," *ACM Computing Surveys*, vol. 35, pp. 399–458, 2003. [Online]. Available: http://dx.doi.org/10.1145/954339.954342

[20] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[21] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, 2001, pp. I–511 – I–518 vol.1.

[22] D. Liang, Y. Liu, Q. Huang, and W. Gao, "A scheme for ball detection and tracking in broadcast soccer video," in *Advances in Multimedia Information Processing - PCM 2005*, ser. Lecture Notes in Computer Science, Y.-S. Ho and H. Kim, Eds. Springer Berlin, 2005, vol. 3767, pp. 864–875.

[23] J. Yu, Y. Tang, Z. Wang, and L. Shi, "Playfield and ball detection in soccer video," in *Proceedings of the 3rd international conference on Advances in visual computing - Volume Part II*, ser. ISVC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 387–396. [Online]. Available: http://portal.acm.org/citation.cfm?id=1779090.1779135

[24] O. Holmstad, "Extracting frames from smooth streaming videos," December 2010.

Appendix A

CD-ROM

The included CD-ROM contains the source code for Árvádus and the video frames utilized in the experiment. ZXY data is available on request only.