Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

KU LEUVEN | universiteit hasselt

2014•2015
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: elektronica-ICT

Masterproef
Software-Defined Networking for Multi-Camera Systems

Promotor :
Prof. dr. ir. Luc CLAESEN

Promotor :
prof. dr. CHIEN CHEN

Martijn Rymen
Proefschrift ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

KU LEUVEN | universiteit hasselt

2014•2015
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: elektronica-ICT

Masterproef

Software-Defined Networking for Multi-Camera Systems

Promotor :
Prof. dr. ir. Luc CLAESEN

Promotor :
prof. dr. CHIEN CHEN

Martijn Rymen
Proefschrift ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

universiteit hasselt | KU LEUVEN

國立交通大學
National Chiao Tung University

Hasselt University & KU Leuven
Faculty of Engineering Technology

Software-Defined Networking for Multi-Camera Systems

by

Martijn Rymen

Promotors:
Prof. dr. ir. L. Claesen, Hasselt University (BE),
Prof. dr. C. Chen, National Chiao Tung University (TW)

Thesis submitted to obtain the degree of
Master of Science in Engineering Technology
Electronics and ICT Engineering

January 2015 - Made in Taiwan

Foreword

4th of April 2014... An email from Prof. dr. ir. Luc Claesen: "There is a possibility to go to National Chiao Tung University for an internship. Please think about it.". After some thinking, I decided to choose for the adventure and to go to Taiwan. This was not only a great academical opportunity, but also for me as person a great opportunity to experience a whole new culture.

At first, I want to thank Hasselt University and National Chiao Tung University, for giving me the opportunity to work on this great project. Especially, I want to thank Prof. dr. ir. Claesen and Prof. dr. Chen for the great support they have given me during this project. When I encountered a problem, I always could go to them to help me out.

Beside my two promoters, I want to thank the two teams I have worked in. The team in Diepenbeek, dr. ing. Andy Motten and dr. Wang Yimu, who helped me with my research on multi-camera systems. Beside them, I had great colleagues at the laboratory of NCTU. They were always present when I needed help on the SDN part of my thesis, or when I needed help to get something.

Finally, I want to thank my family, especially my parents, who supported and encouraged me to take this chance. However, I can imagine it was very emotional to miss me during these three months. And also, a word of thanks to all my friends for giving me a great farewell party and to keep in touch with me, 9507km away from home.

Martijn Rymen, January 2015

Contents

Foreword iii

List of Figures ix

List of Tables xi

Abbreviations xiii

Abstract xv

Abstract (NL) xvii

1 Introduction 1
- 1.1 Situation . . . 1
- 1.2 Problem Definition . . . 2
- 1.3 Objectives . . . 4
- 1.4 Proposed Method . . . 5

2 Multi-camera Networks 7
- 2.1 Introduction . . . 7
- 2.2 Processing . . . 8
  - 2.2.1 Centralized Processing . . . 8
  - 2.2.2 Distributed Processing . . . 8
- 2.3 Wireless Communication Modules . . . 9
- 2.4 Embedded Middleware . . . 9
- 2.5 Classification . . . 12
  - 2.5.1 Camera Types and Applications . . . 12
  - 2.5.2 Topologies . . . 13
  - 2.5.3 Communication and Storage . . . 16
  - 2.5.4 Communication with Other Sensors . . . 16
  - 2.5.5 Today . . . 16
  - 2.5.6 Future . . . 16

3 Software-Defined Networking 17
3.1 Introduction . . . 17
3.2 OpenFlow . . . 17
3.2.1 OpenFlow Components . . . 19
3.3 OpenFlow for Wireless Networks . . . 20
3.3.1 OpenFlow for Wireless Sensor Networks . . . 20
3.3.2 Sensor OpenFlow . . . 21
3.4 Video over a Software-Defined Network . . . 23
3.5 3D Teleimmersion over SDN . . . 29

4 Topology for Multi-Camera SDN 31
4.1 Introduction . . . 31
4.2 Camera . . . 31
4.3 Controller . . . 33
4.4 Switch . . . 33
4.5 Server . . . 33
4.6 User Device . . . 34

5 Use Case 35
5.1 Introduction . . . 35
5.2 Structure . . . 35
5.2.1 Zone Structure . . . 36
5.2.2 Camera Clusters . . . 37

6 Use Case Development 39
6.1 Introduction . . . 39
6.2 Network Controller . . . 39
6.2.1 Ryu . . . 39
6.2.2 Floodlight . . . 40
6.3 Controller Module . . . 42
6.3.1 REST API . . . 43
6.3.2 Database . . . 46
6.3.3 Camera Module . . . 49

7 Results and Discussion 55
7.1 Introduction . . . 55
7.2 Mininet . . . 55
7.2.1 Limitations . . . 56
7.2.2 Topology . . . 56
7.3 Zone Locator . . . 58

7.4 Flow Tests 58
7.4.1 Ping Test 58
7.4.2 Video Test 61

8 Conclusion **63**

Bibliography **65**

A Wireshark Output for Video Tests **67**

List of Figures

1.1 The basic idea behind the use case 3

2.1 Middleware layers [19] 10
2.2 The centralized topology 13
2.3 The clustered topology 14
2.4 The distributed topology 15

3.1 The OpenFlow model 18
3.2 The architecture for a Software-Defined Wireless Sensor Network [12] . . . 21
3.3 An overview of the architecture for VSDN [6] 23
3.4 The architecture of a host [6] 25
3.5 The architecture of a VSDN switch [6] 26

4.1 The overall topology 32

5.1 The use case 36

6.1 RYU architecture 40
6.2 Floodlight diagram [4] 41
6.3 The structure of the created controller module 42

7.1 The topology used 57

List of Tables

6.1 Camera table 47
6.2 Event table 47
6.3 Data members of camera 49

7.1 Ping test one 60
7.2 Ping test two 60
7.3 Ping test three 61

A.1 Output of Wireshark 70

Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
DPID	Data Path ID
HTTP	HyperText Transfer Protocol
JDBC	Java DataBase Connectivity
JSON	JavaScript Object Notation
LAN	Local Area Network
MAC	Media Access Control
OS	Operating System
QoS	Quality of Service
REST	Representational State Transfer
SDK	Software Development Kit
SDN	Software-Defined Networking
SOF	Sensor OpenFlow
UDP	User Datagram Protocol
UWB	Ultra-Wide Bandwidth
VSDN	Video Software Defined Networking
WAN	Wide Area Network
WSN	Wireless Sensor Network

Abstract

Intensively interacting multi-camera systems can have the requirement to be dynamically reconfigurable and to have an easy control of the network. Therefor, this work investigates the possibilities of Software-Defined Networking (SDN). SDN offers a flexible way of managing networks while proving a great scalability. In SDN the data plane and control plane are decoupled, enabling this dynamic control.

First, multi-camera networks and SDN have been researched. Afterwards, a use case is defined to validate the concept of using SDN in a multi-camera network. This use case is a soccer field, where it can be employed in demand driven 3D image calculations. For this reason, links needs to be created between adjacent cameras, allowing them to work jointly on common tasks. These links need to be created dynamically, because a tracked object can move over the entire field. This demands a 3D image from other locations on the field.

To create this use case, a network controller is used. This allows Java programming and the development of a REST API. With this API it is possible to alter the location of a tracked item, while the controller module can connect to a SQL database to retrieve the cameras that have to work together on the task. To prove that the controller is only forwarding the necessary traffic and blocking the other traffic, the network is emulated with Mininet. In this way, superfluous traffic is avoided on the network. This concludes that SDN is a dynamic way to control a multi-camera network.

Abstract (NL)

Intensief interopererende multicamerasystemen kunnen de eis hebben om dynamisch herconfigureerbaar te zijn en hebben een makkelijke netwerkcontrole nodig. Dit werk beschrijft de mogelijkheden van *Software-Defined Networking (SDN)*. SDN biedt een flexibele methode aan om een netwerk te beheren met daarnaast een grote schaalbaarheid. Om deze dynamische controle mogelijk te maken, is in SDN de data laag van de controle laag losgekoppeld.

Dit werk schetst eerst de theorie rond multicameranetwerken en SDN. Vervolgens wordt een *proof of concept* ontwikkeld om het concept van SDN voor multicamera netwerken aan te tonen. Dit is een voetbalveld, waarin het de vraag kan zijn om 3D-beelden te berekenen. Voor deze berekeningen is informatie vereist van twee of meer naburige camera's. Hiertussen dient een verbinding te worden opgezet, zodat ze kunnen samenwerken. Deze verbindingen kunnen dynamisch worden gecreëerd, omdat een object over het veld kan bewegen.

Om deze *proof of concept* te ontwikkelen wordt een netwerkcontroller gebruikt. Dit laat toe de Java-ontwikkeling te doen en de ontwikkeling van een REST-API. Met deze API is het mogelijk om de locatie van een object door te geven, waardoor de controller in een SQL-gegevensbank de benodigde camera-informatie kan opvragen. Om aan te tonen dat overbodig verkeer van het netwerk wordt geweerd, wordt het netwerk geëmuleerd door middel van Mininet. Dit alles toont aan dat SDN een dynamische methode is om een multicameranetwerk te beheren.

Chapter 1

Introduction

1.1 Situation

In the team of prof. Claesen at Hasselt University, research is focused on multi-camera video architectures, where hundreds of cameras have to communicate and work closely together.
Deep-level synchronization and communication allows cameras to exchange information between the nodes and perform a distributed processing. In the camera, not only video processing takes place, but also recognition of certain features. These features can be exchanged between the camera nodes, allowing them to work together on one common task. These tasks can range from feature recognition up to real-time 3D depth estimation, calculation and view interpolation. Instead of concentrating these tasks on one central server, the processing needs to be done in a distributed and easily reconfigurable way.

At NCTU, in the team of prof. Chen, research is ongoing about Software Defined Networking and also about the multi-camera networks as described above.
Software Defined Networking (SDN) is a centralized method of controlling a network. Especially when going to big, complex networks this can give huge advantages for the network administrator to control the network. The network control can be done dynamically and direct, without the need to keep the vendor in mind.

1.2 Problem Definition

The problem with current multi-camera systems is that the network is often not easy to rescale and control. The current technology does not allow much and easy scaling, making it for the network administrator hard to control. Beside this, the addition of new cameras could lead to a lot of work and a period of downtime for the whole network.

So, why not going over to a Software Defined Network and making the combination with the multi-camera networks? Software Defined Networking offers the advantage that it is rather efficient to control the network; it can be done dynamically and direct. Besides that, the controller can make decisions on where the traffic needs to go in a certain case, without an input from an end user.

The use case that is used in this work are cameras surrounding a soccer field, this is illustrated in figure 1.1. The cameras are all connected in a cluster to a switch, while these switches are on their turn connected to another switch, making communication between them possible.

The application needs an input of a position of a football on the field. This information is sent to the SDN controller, which is deciding which cameras have the ball in their view angle. For this reason, the controller also needs to know the location of the cameras next to the field.
The controller makes a path between these two cameras, because it is assumed that there are only two cameras that have the ball in their sight. The packets from those two cameras are sent to each other, so they can cooperate on, for example, 3D depth estimations and so on.
The packets coming from the other cameras are dropped, because they are not necessary in the calculations needed in this application.

The big challenge in this project is that geographic location, that is given by the input to the application, has to be mapped on a network topology.
The locations are sent through an API to the controller. This controller needs to map the geographic locations of the cameras and the football to the network topology that is used.

In the simulations, there are some predefined locations of a football, which are chosen randomly. Afterwards, the controller creates the path between the two cameras that have to work together in order to have a communication path between two cameras.

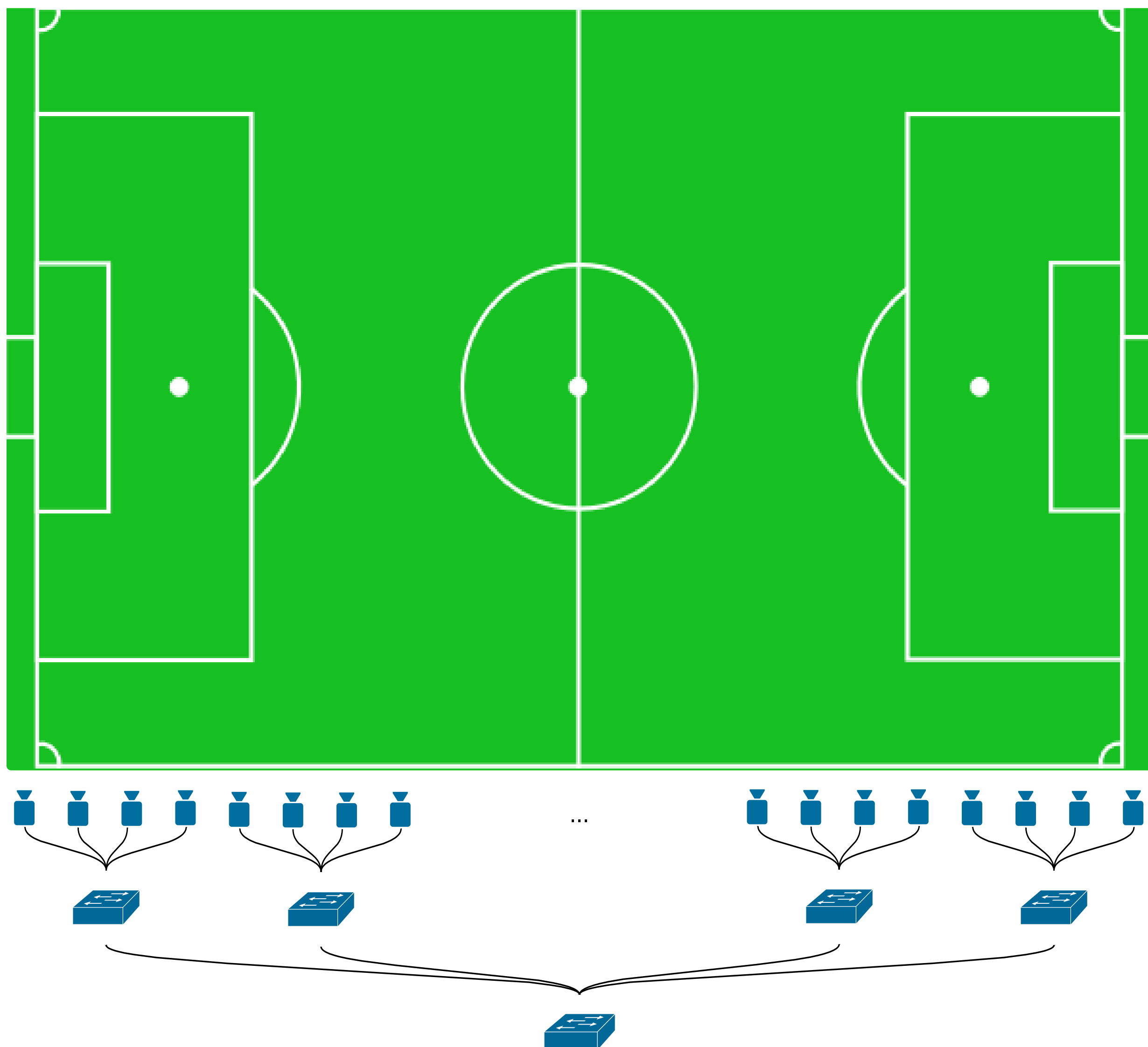

Figure 1.1: The basic idea behind the use case

1.3 Objectives

The main objective is the investigation of the use of Software-Defined Networking in a dynamically configurable multi-camera environment. Three sub-objectives are necessary to achieve this main objective.

The first sub-objective of the work is an investigation of the possibilities of SDN. At this moment multi-camera networks are not easy to reconfigure, for that reason the application of Software-Defined Networking is investigated. The main objective is to create a topology that can be used in a practical situation.

Besides that, another objective is to investigate if it is easily possible to scale the architecture, in terms of the amount of cooperating cameras. Multi-camera networks can consist of more than a hundred camera nodes. But when, for example, new camera nodes is added, the network becomes larger, and the architecture needs to rescale.

Finally, a use case will be developed. As described before, this is a soccer pitch. The network controller should be able to know the camera nodes and their location on the network. Using an API, an application is developed that gives the location of a ball on the field to the controller. This controller enables a flow between the cameras that are cooperating on the specific job.

1.4 Proposed Method

The next paragraphs show the roadmap that is used to create this work.

First of all, it is important to build up a theoretic background on the concept of Software-Defined Networking and also on the multi-camera networks. In this way, there is knowledge on the state of the art. Besides that, it is investigated what is already done and what can be useful for future research.

Secondly, it is necessary to get used to the experimentation software and the SDN controller. Mininet is used to simulate the network and the connections between camera nodes en switches. Floodlight is used as a controller for the Software-Defined Network.

Afterwards, the use case is created. This use case is a proof of concept on the application described. A module for the Floodlight SDN controller is developed. This modules provides an API for the ball location. On this base the Floodlight controller can install new rules on the switches to control the traffic. Besides that, a simulated topology is created in Mininet.

Chapter 2

Multi-camera Networks

2.1 Introduction

For the history of cameras we have to go far back in time. Leonardo Da Vinci was the first person to publish a clear description of the *camera obscura*[1] (Latin for darkroom), but the principle exists since 390 BCE. This darkroom is a space that is totally obscured, with a small hole in a wall. On the wall facing the hole it is possible to see a view from the outside world. Since the 16th century these *camerae obscurae* were used by painters and illustrators. They used it for the projection of a view on a thin piece of paper. After the development of chemical processes, cameras were more and more used for photography and later also for film.

Nowadays, cameras have a larger field of application. They are not only used to film movies are series, but also for video conferencing, security, monitoring and many more applications. In the recent years, many new developments have taken place in the field of multi-camera networks. A major development in the previous years is the use of multi-camera networks. Cameras no longer work alone, but are also going to collaborate on certain tasks. Collaboration also means that there needs to be communication in between the cameras.

In this chapter, the processing of images in these networks is discussed, as well as the middleware on the cameras and a classification that is made at Hasselt University.

[1] http://www.geog.ucsb.edu/ jeff/115a/history/cameraobscura.html

2.2 Processing

As described in [19], processing is an important thing for many applications of a camera network. Multiple cameras can be used to do the coverage for the monitoring of large areas. Also, multiple cameras can be used to obtain a different angle or view on the same area. To process these images, you can either use a distributed or a centralized approach. The difference between these two approaches will become clear in the next two subsections.

2.2.1 Centralized Processing

In this model, the images from all cameras are gathered by a central processor. By combining all these images, a central processing can be done with one central algorithm. Because of the huge requirements of communication bandwidth and video processing, real-time streaming from many cameras to one central processing unit is difficult in practice.
In general, there are two approaches for video processing. The first approach is the top-down approach. First the models are developed for this. Afterwards, these models are fitted to images. The second approach is the bottom-up approach. In this approach various features are extracted from images. These features are combined within an image and then across different camera views.

2.2.2 Distributed Processing

When an area that needs to be covered increases, also the amount of cameras required will increase. The advantage of distributed processing is that the processing and communication requirements in a distributed approach could be nearly constant. In centralized processing these requirements could increase when the number of cameras increases.
A new trend are gossip-based models. This means that each node repeatedly randomly contacts another node to exchange some information. Hybrid processing is likely to offer the best practical solution. In this approach the information of the cameras is combined centrally and the data processing is distributed across groups of cameras.
This hybrid processing can be mapped back on the bottom-up approach, described in 2.2.1. The amount of processing in the central node is greatly reduced. This is essential when working in environments where real-time and low latency performance is required.

2.3 Wireless Communication Modules

As discussed in [19], standard cameras have some sort of video output. Smart cameras also have requirements regarding the control and data communication. Especially when these cameras are used in a camera network, this communication is important. For practical reasons, wireless communication can be preferred, but we have to keep in mind that low power consumption is very important. On the counterpart, low power means low bandwidth and brings up a shorter communication range. For this reason it will be impossible to do real-time video streaming for multi-camera systems in the near future. In other low-power communication environments, ZigBee and Bluetooth can be seen as highly relevant. But for multi-camera systems the bandwidth is too small.

In [3] Ultra-wide Bandwidth (UWB) technology is described for video transmission. This could be also useful in multi-camera systems, because of the large amount of bandwidth that will be needed for the transmission of video. The WiMedia PHY specification can offer a bandwidth of 1Gbps, which is enough for the transmission of video from the cameras used in the multi-camera networks assumed in this work.

2.4 Embedded Middleware

In the case of multi-camera systems, the embedded middleware is an abstraction layer of the hardware devicesFigure 2.1 shows a very general partitioning of the embedded middleware into different layers of abstraction. [19]

The *Operating System (OS)* is the base of every middleware. In the operating system the drivers for the hardware can be found, but there are also basic mechanisms to access these devices. The OS provides also process and thread management and communication between processes.

The *host infrastructure layer* provides a mechanism that packs low-level system calls into modules that can be reused. This layer also enhances communication and concurrency mechanisms. It is also the first step towards a portable and platform-independent middleware.

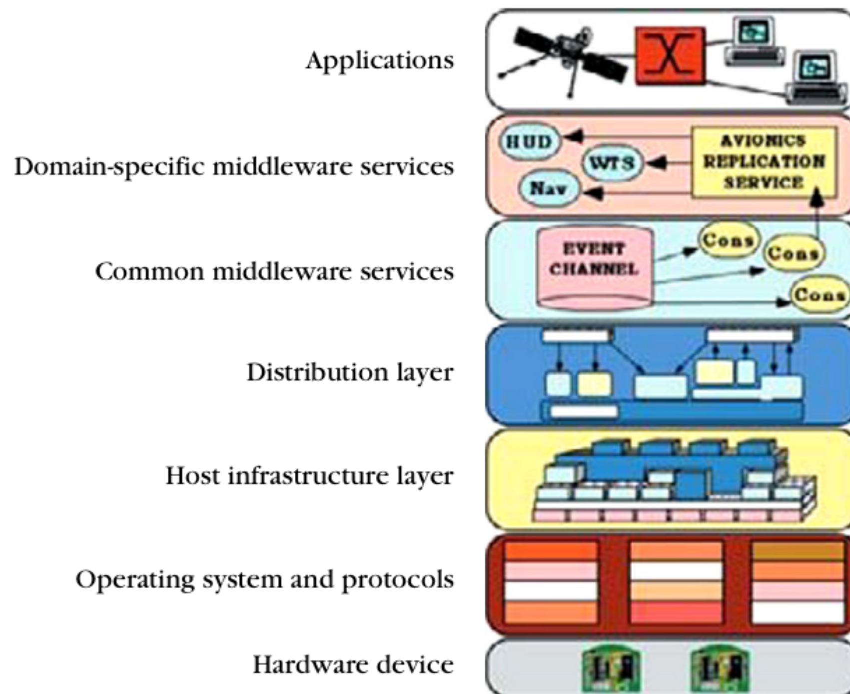

Figure 2.1: Middleware layers [19]

The *distribution layer* provides the integration of multiple hosts in a network in a distributed system. This layer also defines higher-level models for distributed programming. This layer offers the possibility for developers to develop distributed applications like stand alone applications.

The *common middleware services* enlarge the previous layer. This is done by the definition of domain-independent components and services. These can be easily reused in applications and make the development of these applications less complex. These components can be used for database connectivity, threading, logging, resource management,...

The *domain-specific layer* is in charge of providing services to applications in a specific domain. The goal of these services is to simplify the development of applications.

The *application layer* is the highest level of this architecture. The individual applications are implemented using services from lower layers.

The embedded smart camera networks lie in between general-purpose computers and wireless sensor nodes regarding available resources. As a result of this, the middleware must find a trade-off between platform independence, programming language independence and the introduced overhead.

The smart cameras are intended to process the images close to the camera sensor. So sophisticated image processing algorithms must run here. The middleware must support the developers to simplify the development of applications and the integration of this application.

In big camera networks (i.e. comprising of hundreds to thousands of cameras), many tasks have to be executed. This is nearly impossible to be set by an user, so the user needs to program rules what to do in case of an event. The camera system itself needs to assign those tasks to a certain camera, based on rules set by an user. Sometimes cameras can not perform the task on their own and so they have to collaborate with a neighborhood camera.

2.5 Classification

A preliminarily classification of multi-camera systems is defined at Hasselt University.

2.5.1 Camera Types and Applications

Multi-camera systems can be classified according to:

1. Fixed cameras
2. Moving cameras
3. Stereo cameras
4. Omni directional cameras
5. Panoramic cameras

Out of these cameras, it should be possible to get information from multiple overlapping views. From the camera information, it should also be possible to do 3D image reconstruction and free viewpoint interpolation.

For the cameras, it is the intention to use multiple views of cameras. These cameras could be cameras for television, movies, cellphones, on road intersections, railroad crossings,.. One special application could be to use information from cameras, located fully around cars, trucks, trains, buses,... These cameras could be used for driving assistance or other assistance and safety applications.

In the case of cars/trucks/buses the information from these cameras can be brought together with camera information from cameras at road intersections. This could help to avoid accidents with people in the dead corner.
In railroad applications, these cameras could be used to avoid crashes with people who are walking on the rails. The information from the trains' cameras should be used together with the information of the cameras on railroad crossings. The combination of this information can help to let a train stop when a person is seen on the rails.

2.5.2 Topologies

In the next three sections, three ways of processing are discussed. First of all, centralized processing is introduced, followed by an approach where the processing is done in clusters. Finally, a fully distributed topology is presented.

Centralized

Figure 2.2 shows the topology that represents a camera network with centralized processing. In this topology, the camera transmits its images to a central server, where the images are processed. Everything is done on this central server, making the workload very high.

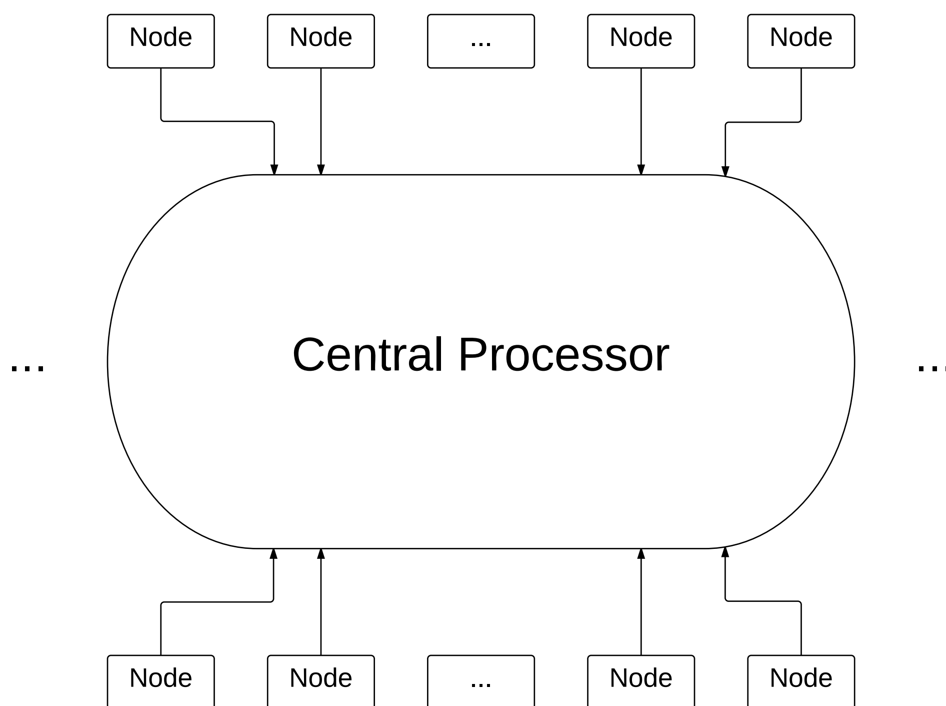

Figure 2.2: The centralized topology

Clustered

Figure 2.3 shows the topology that represents a camera network with clustered processing. Here, the server is no longer central, but there are several servers and each is serving a certain amount of cameras. This offers the advantage that the workload can be divided over more processing units. Each unit can have, when needed, very specific algorithms for the connected cameras. The cluster processors are connected with each other, so they can share some information when it is needed.

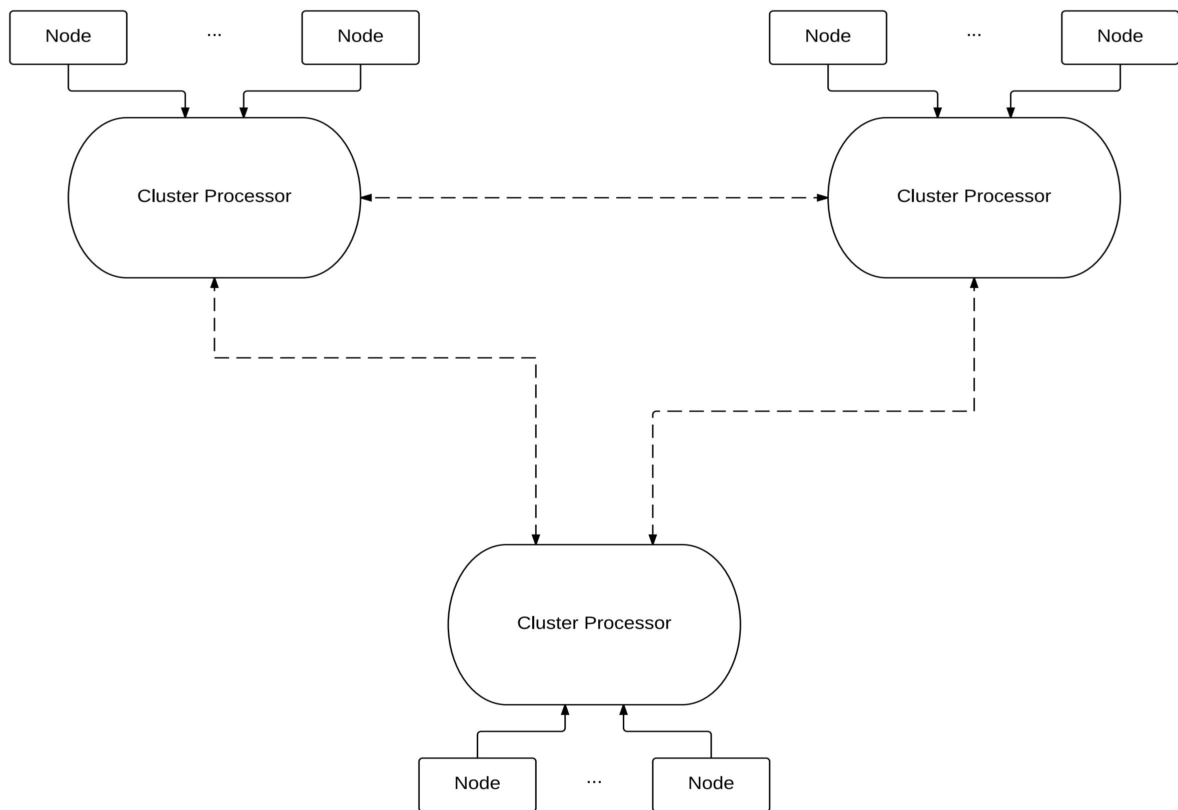

Figure 2.3: The clustered topology

Fully-distributed

Figure 2.4 shows the topology that represents a camera network with distributed processing. The processing takes place on the camera itself. Hereby, the workload is divided over every camera. These cameras need to communicate with each other in order to exchange certain features that can be necessary for the other cameras.

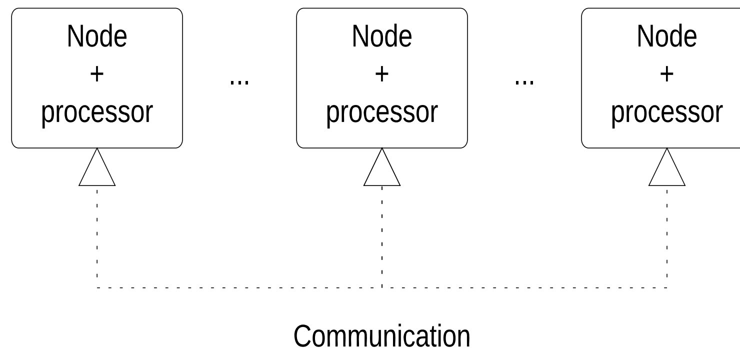

Figure 2.4: The distributed topology

2.5.3 Communication and Storage

For the communication, it is necessary to keep in mind that there can be a need to store the images from the cameras. This provides a first bottleneck, because the storage media must be capable to store this data. To make the communication between the camera and the storage media and also between the cameras more efficient, *Software-Defined Networking* can be used.

2.5.4 Communication with Other Sensors

In a practical multi-camera networks it can be desirable to communicate with other sensors in the neighborhood. Based on the information from the cameras and the sensors, the most efficient decision can be taken.

2.5.5 Today

In todays multi-camera applications the camera nodes do not communicate with each other. The processing and storage is often centralized. This means that huge bandwidth is consumed to communicate with the central server. To process these images, brute force algorithms are needed. These algorithms are widely applicable, but on the other hand they result in slower processing speed, because all exceptions need to be included in the algorithm.

2.5.6 Future

In the future, there will be communication between multiple intelligent cameras. These intelligent cameras will feature an on-board processing, so it would be possible to have an object feature detection on the camera. Such a camera needs to know its own location, so in communication with other cameras, it would be possible to only communicate with neighboring cameras. Because less information is sent, the required bandwidth will also decrease. Sensor fusion would be nice too, so information from other sensing devices can be used for the feature detection on the cameras. Off course in the processing, some features are really necessary: stabilization of the camera images, automatic calibration of those cameras, HDR processing and super resolution.

Chapter 3

Software-Defined Networking

3.1 Introduction

Software-Defined Networking is an architecture used to manage large, complex computer networks. In these networks, from time to time it can be needed to re-policy or to re-configure some network settings. Software-Defined Networking means that the data plane and the control plane are decoupled. This allows flexible and efficient management and operation of the network, because everything can be done with software. The control plane makes decisions where traffic needs to go, while the data plane forwards the traffic.

In this chapter, first of all the OpenFlow protocol is discussed. This protocol enables the communication between a network controller in software and a network device in hardware. Afterwards the different components of an OpenFlow network are introduced. The next sections handle about the usage of OpenFlow for wireless (sensor) networks and also about a special OpenFlow version for sensor networks. The last two sections introduce the usage of SDN for video streaming and 3D Teleimmersion.

3.2 OpenFlow

OpenFlow is a protocol that makes it possible to introduce the concept of SDN in software and hardware as well. [22] A main advantage of using OpenFlow is that it is possible to use existing hardware to design and analyze new protocols. Open source codes are used to control SDN controllers and switches. OpenFlow is a flow-oriented protocol and abstracts switches and ports to control the flow.

The OpenFlow architecture consists of three main components. [14] The first main component is a piece of software, named *the controller*. The function of the controller is to manage the collection of switches that control the network traffic. It is responsible for manipulating the flow table of a switch. An OpenFlow switch consists of a group table, multiple flow tables and a channel. Through an OpenFlow protocol, communication with the controller takes place over a secured channel.
The OpenFlow switches are connected to each other via OpenFlow ports. Figure 3.1 shows the model that is used in an OpenFlow architecture.

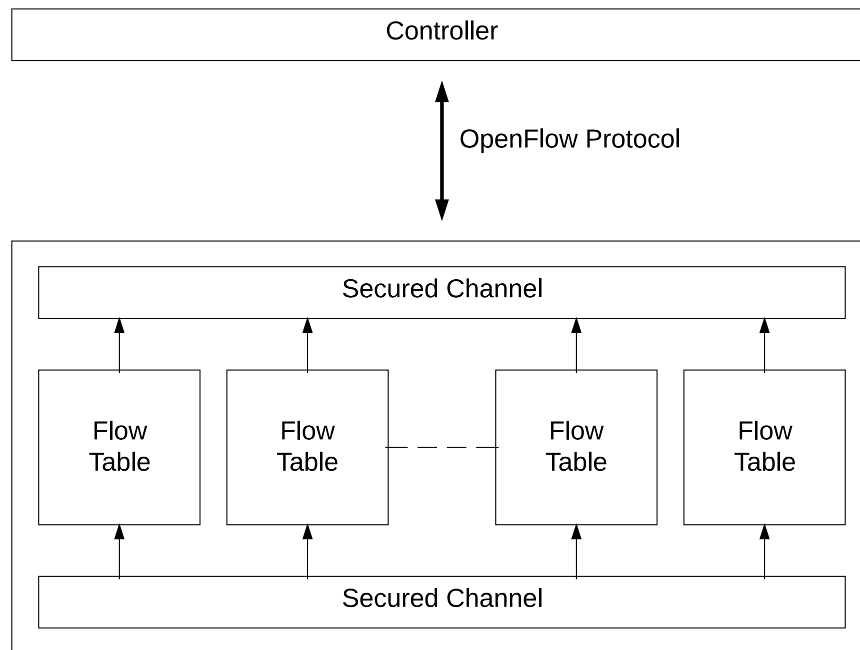

Figure 3.1: The OpenFlow model

Initially, the data flow tables of OpenFlow routing devices contain no information. In the table some fields are provided to contain information about source IP address, Quality of Service (QoS) type,... There are also some fields to contain information about the direction of different ports, called packet fields. The action field is used to contain information about packet forwarding or reception. Based on new incoming data packages, this table can be filled. When there is no matching entry in the data flow table, the package is forwarded to the controller. This controller decides what to do with the new package, either to drop it or to make a new entry in the data flow table. This new entry describes how to deal with similar packets in the future.

SDN has the capability to program multiple switches simultaneous, but still suffers from conventional complexities of distributed systems. This includes packet droppings and delay of the control packets. With new protocols as OpenFlow, SDN is easier to implement. In the control plane, the routing table is generated. In the data plane, decisions are made where the packets should be sent to, based on the flow table.

In this standard, a central controller is assumed. This can be difficult in large-scale networks. It is also difficult to use in large, wireless applications. To overcome this issue, a distributed system can be used, wherein all controllers keep in touch to have a global view.

3.2.1 OpenFlow Components

In the next sections, the different components of an OpenFlow network is discussed. [22]

Switches

Conventional switches do have to take routing decisions, they have to decide where a packet needs to go. In SDN, switches only gather and report the network status and process the packets based on the rules in a flow table. An open source protocol is defined to monitor and change the flow tables in different switches and routers. The compliant OpenFlow switches contain at least three elements:

1. Flow table(s): containing which action must take place when a known packet is received.
2. Communication channel: a link between the controller and the switch, which transports packets and commands.
3. The OpenFlow protocol: enables the communication between and OpenFlow controller and switches/routers.

Controllers

The controller is the most important component. The most complex point is located here. The controller can update entries in the data flow table. Two kinds of controllers exist. Static controllers establish a static data path, while dynamic controllers create a dynamic data path.

Flow-entries

For each entry in the flow table an action is defined. In OpenFlow three actions are supported:

1. Forwarding packets to a port
2. Encapsulation of these packets and forwarding to a controller
3. Dropping these packets

3.3 OpenFlow for Wireless Networks

Managing wireless networks is more challenging than the management of wired networks. [21] This is because wireless infrastructure is more hybrid and complicated. Another problem in the control of wireless networks is that different wireless products have their own specification of the the data link and the physical layer. This makes reconfiguration quite difficult. Now, Openflow can be used to let the MAC layer operation take place on virtual machines. Thanks to SDN reconfiguration and flexibility is much higher. Another advantage of OpenFlow is that policies can be easily updated.

3.3.1 OpenFlow for Wireless Sensor Networks

In multi-camera systems, a node is often seen as a sensor in a wireless sensor network (WSN). In [22] it is described that these products are harder to (re)program because of vendor-specific software development kits (SDK). Reprogramming is hard because, nowadays, it is needed to be done on every sensor individually. In big networks this is not realistic, because it is possible that there are a large amount of nodes in the network.

Over-the-air programming is already possible, but often these solutions are vendor-specific. This results in the use of different operating systems and programming languages. It would be better if there was a possibility to configure the network controller with an universal protocol. It may be possible to decouple the hardware of the sensor and the embedded network protocols. Then users do not have to worry anymore about the data forwarding in each sensor. They just have to program the flow table of the controller.

This is the place where SDN and OpenFlow come in. SDN and OpenFlow can forward sensor data based on a specified flow table and its rules. It is easy to change these rules. This can be achieved by a change of the controller configuration. Thanks to the use of an universal operating system, programming can be done with standard scripts.

3.3.2 Sensor OpenFlow

In [12], Sensor OpenFlow (SOF) is described as a standardized protocol between the data and the control plane for sensor networks. In the data plane, the sensor nodes can be found. These are performing flow-based forwarding of data packages. In the control plane, the network controller can be found. Just like in standard SDN, the controller combines all the network intelligence. Figure 3.2 shows the architecture of a SDN WSN for cameras.

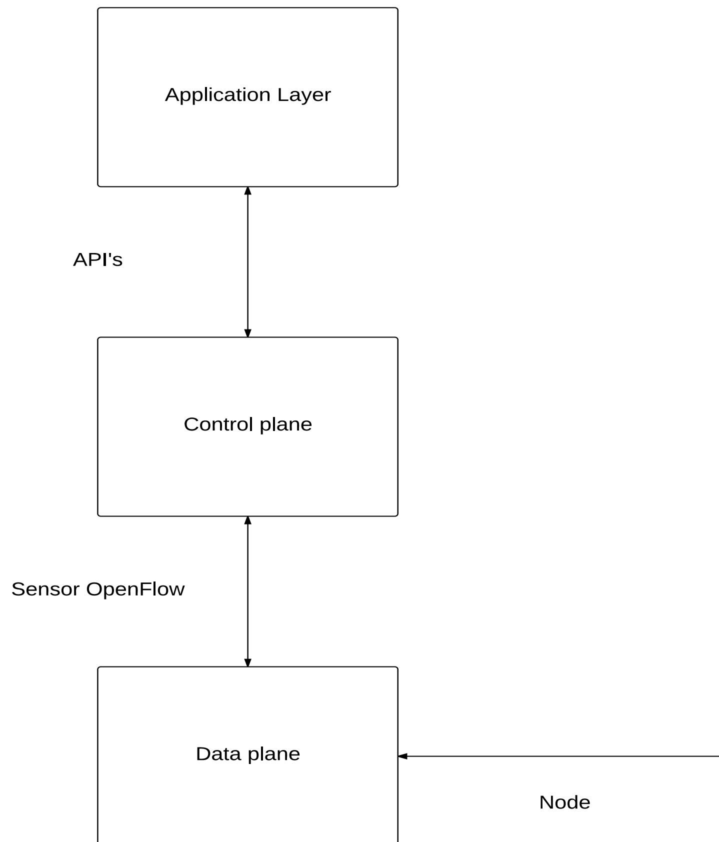

Figure 3.2: The architecture for a Software-Defined Wireless Sensor Network [12]

Challenges

The next enumeration shows six challenges that are present with *Sensor OpenFlow*. Solutions to these challenges are given in [12].

1. Data plane: creating flows. OpenFlow assumes to match the source and destination based on IP addresses. Sensor Networks are often *data-centric*. This means that the acquisition of data is more important than the source of the data. The challenge for SOF is to create a flow between source and destination.

2. Control plane: SOF Channel. Generally the SOF channel needs to provide TCP/IP connectivity, because this is mostly not available in a WSN. This connectivity is needed in order to have reliable end-to-end communication.

3. Overhead of traffic control: the communication between the sensor network and the Sensor OpenFlow has to be hosted *in band*. This means that the communication runs over the same network as the normal traffic. The traffic control needs to be well thought allowing control messages to pass through all the time.

4. Traffic generation: for normal SDN, end-users are considered as a peripheral. Sensor nodes behave like end-users, because they generate data packets.

5. In-Network Processing: in sensor networks the data needs to be processed in real-time. This because network resources need to be saved.

6. Backward & Peer Compatibility: backward compatibility is needed in order to protect a company to do big investments. With peer compatibility, it is meant that the SOF networks are compliant with the standard OpenFlow networks.

3.4 Video over a Software-Defined Network

The possibilities to send video over a Software-Defined Network (VSDN) are discussed in [6]. Modern network architectures install a single path from source to destination, but this might not be the most optimal path between them. For example: *Integrated Services* is a QoS architecture that specifies elements along a path and allows to reserve resources for the grant of end-to-end QoS. The network should select the best path among the paths that are present, to improve the performance of the application. Current network protocols often have problems to calculate the right path. This results in the fact that the calculated best path from source to destination can be a path with congestions. The result is a negative impact on the video playback of an end-user. The selection of the path should be more adaptive to a change of the conditions of the network.

To solve the issue of the wrong selection of an optimal path, in [6], SDN is proposed as a possible solution. An architecture is proposed: this allows the possibility for an application to request video services. In figure 3.3 the structure of the VSDN can be seen.

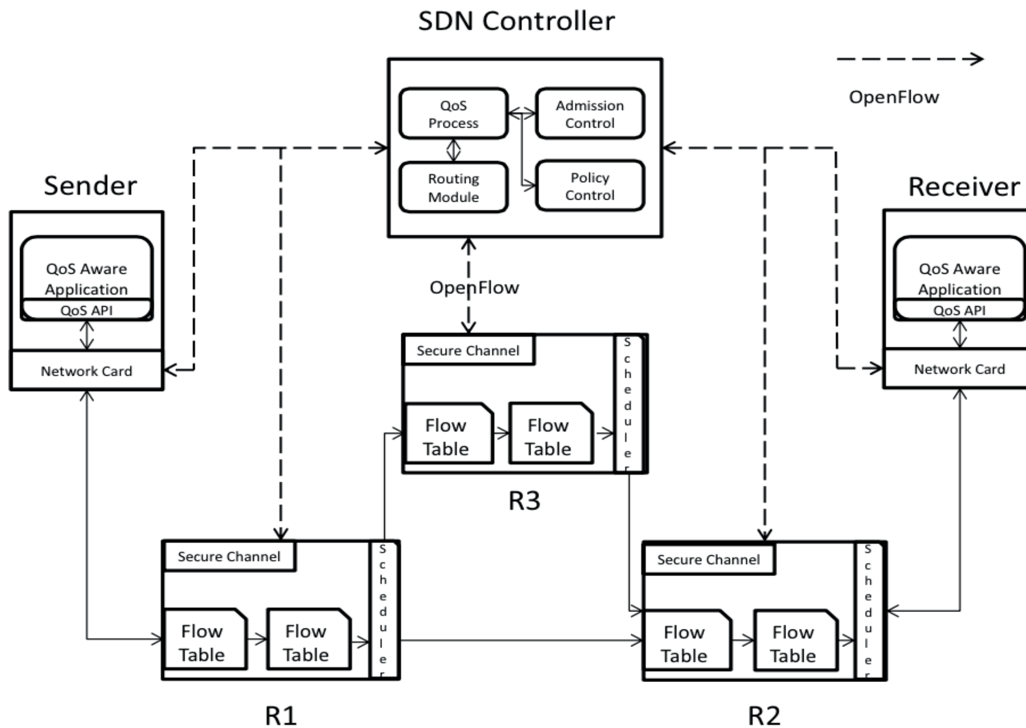

Figure 3.3: An overview of the architecture for VSDN [6]

The SDN controller has some different modules to fulfill the needs in a VSDN. The architecture consists, beside the controller, of a sender, receiver and OpenFlow switches. The controller consists of different parts present:

1. The policy control
2. The admission control
3. The routing module and topology monitor
4. The Video QoS Process

The parts are different compared to a common SDN controller, they are discussed briefly in the next paragraphs.

The first part of the controller is the *Policy Control*. It is centralized in the controller to provide consistency between all devices. Network administrators can send commands how new traffic should be handled. These policies are translated to be mappable to the network configuration. There is also a *resource monitor* to monitor the resources present in the network. In this way, flow decisions can be made with the instantaneous load in mind.

The second part in the controller is the *admission control*. When there is a request on an interface, admission control will be performed on the router located on the border between one service provider and another. If the resources are available on the path between the source and destination, VSDN will process a path. The admission control must manage the available resources and removes them from the pool of resources when in use.

The *routing module* is used to calculate the path between a source and a destination. The main responsibility of this module is to return a list of paths that meet the amount of resources needed. A *Topology Monitor* is also present, which updates the configuration of the network when there is a change in the topology.

The last part of the controller is the *Video QoS Process (VQP)*. This element is considered the main element of the topology. The controller must be able to handle the sender, receiver and error messages. The VQP must maintain the correct state of a session, when failing it needs to generate an error. A valid request from a sender will store a new session in a database, containing the needed information to identify a flow. If the receiver accepts a request or reservation, this receiver will send a request message to the controller to certify that reservations are allowed. When this is true, the VQP will request the possible paths

and receive a subgraph. This subgraph can be used to configure a physical device. After this configuration, the receiver receives a message confirming the programing. Now, sender and receiver are able to communicate over the established path. When the communication finishes, the session can be finalized and the network resources can be released.

In the host (fig. 3.4), there is no big difference compared to another host. There is a slicing layer present, enabling different service providers to share the same hardware. A packet classifier is present to determine which packet belongs to which flow and the packet scheduler is responsible to check if the packets are in ordering to the agreed service before these are transmitted over the network.

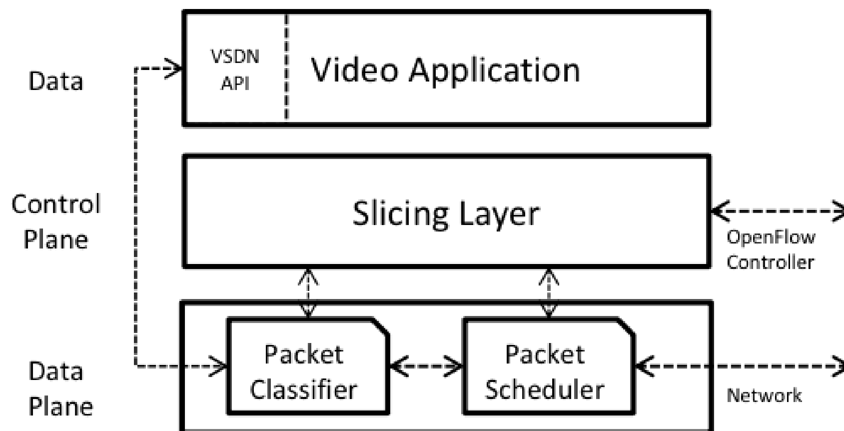

Figure 3.4: The architecture of a host [6]

Also, a special VSDN switch (fig. 3.5) is described in [6]. Such a VSDN switch is OpenFlow enabled. The controller emits Set-Queue actions to each VSDN switch in a path. This action initiates the queues on the switches in order to handle the traffic that is coming. Afterwards it will issue Flow-Add actions to each switch in the path. When one of these two actions fails, an error message must be send. The Set-Queue request will cause a switch to create a weighted fair queue and a traffic shaper based on traffic specifications that are obtained from the controller.

When a packet enters the switch, it will be queued. Based on the rules it will be dropped or it will continue until there is a matching flow table entry. When there is a match, the packet will be forwarded to the matching port. To fulfill the end-to-end QoS, changes are needed to OpenFlow. These are described in [6].

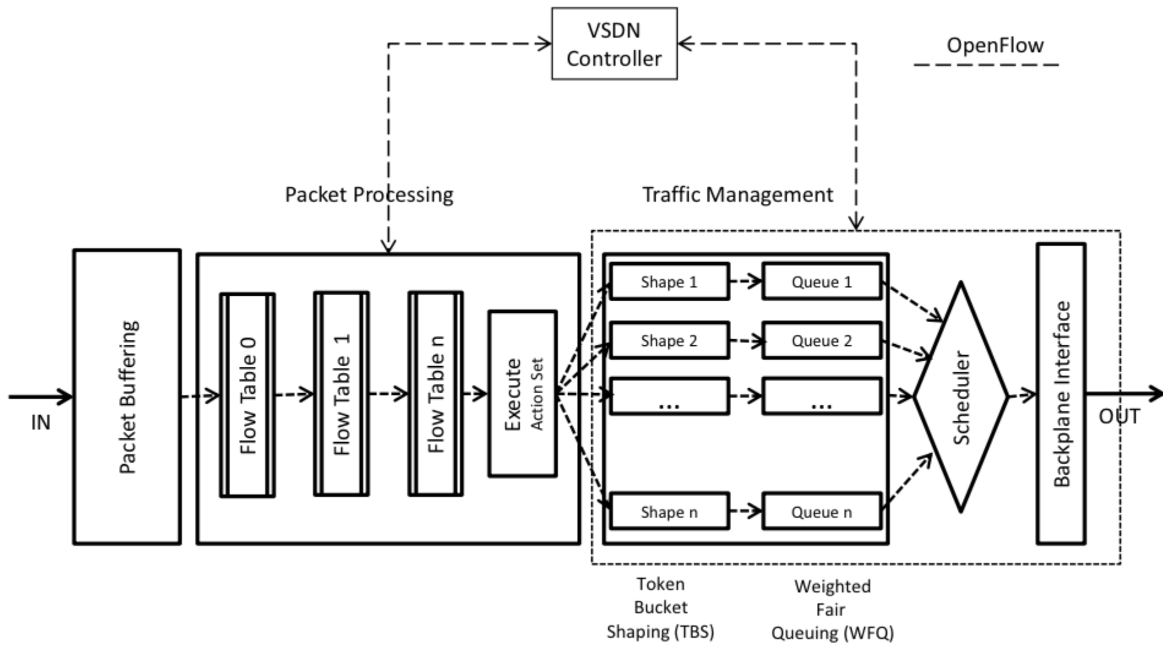

Figure 3.5: The architecture of a VSDN switch [6]

Protocol

In [6], six actions are discussed that can take place in a VSDN:

1. *Sender's actions*: when a sender makes a call, a SessionID is returned, which is used to remove the session when the resources can be released. The request is forwarded to the controller, in order to check if a similar request already exists. If so, the controller responds with *invalid request*. If it does not exist, the controller creates a new session and the packet is forwarded to the receiver.

2. *Receiver's actions*: the receiver gets requests from a sender to process a request, and checks is it has to accept the request. The SessionID is stored when accepting a request in order to remove a session from the network when finished. The receiver will reserve resources, based on the traffic description of the sender. This can be done when the request is authorized by the controller. This controller will also inform the receiver with the route that is installed. This route is used to inform the sender and to install the path when the sender correctly receives the request.

3. *Removing Reservation*: by calling a remove request, reservations from sender, receiver or the network can be performed. This can also be performed implicitly by the timer expiration from the OpenFlow switches and hosts. When the request is verified, sessions and flow entries are removed.

4. *Controller Actions*: the controller receives a request from the receiver. When the controller receives these requests, it takes care about admission and policy control. The policy control determines if the receiver can make the necessary reservations and admission control determines if the resources are available. After the clearance is given, the controller searches for a path between source and destination. When the path is determined, the controller instructs the switches and hosts to adjust their flows. The controller then forwards messages back to receivers.

5. *VSDN switch actions*: the switch in a VSDN network is a dumb device. It takes instructions from the SDN controller regarding forwarding or dumping of packets. It also forwards received requests from a sender to the controller. The controller performs the necessary actions and returns a flow entry to the switch when a new flow needs to be created.

6. *VSDN Host actions*: the host in a VSDN takes instructions from the controller. It configure its flow table as instructed when a request message is received. The host has to perform instructions like add, delete or QoS queuing received from the controller. If it is unable to complete the operation, it sends an error.

API

The API described in the paper allows hosts to request QoS from the network. VSDN offers a simple and consistent interface, without the need to know the details of the API. The sender can request a video QoS by calling *requestQoS(v, d, p)*. This can be three kinds of video service: *CIF, ED & HD*. The receiver can accept this request by calling *acceptQoSRequest(s)* and remove sessions with *removeQoSRequest(s)*. The application receives information about the network by calling *processRequest(s, d)*.

QoS Mapping

There are two service specifications for time-tolerant and time-intolerant video services: *Controlled Load (CL) & Guaranteed Service(GS).* The VSDN framework is using the GS specification. The applications on the host only specify the type of video, but not the flow specifications. The VSDN controller knows these video types and has to convert them to the specification in order to send OpenFlow messages. The mapping between video type and the guaranteed service can be found in [6].

Impact of VSDN on SDN for Multi-Camera Networks

This VSDN can be useful in the work about SDN for multi-camera networks, because it offers a solution to send video over a Software-Defined Network with keeping the QoS in mind. This last is especially useful when going to a solution where end-users want to see different views. Beside this, the cameras can request QoS with the described API, allowing them to communicate to each other over the best path.

3.5 3D Teleimmersion over SDN

In [5], 3D Teleimmersion (3DTI), a technology that supports full-body interaction in virtual environments is described. Applications are in the field of rehabilitation, collaborative dancing, online gaming and video conferencing. The technology consists of real-time multi-stream environments, where distributed participants interact with each other in a virtual space. In this space, the streams from multiple 3D cameras are fused with several microphones and other sensors.

Current multi-stream 3DTI still has some challenges to overcome. This is because of the high interactivity requirement and the large demand for processing and network resources. Participants can also frequently switch their view orientation in the virtual space, requiring frequent updates of multi-stream content distribution. Current 3DTI struggles with heavy overheads and had very complex management methods.

With the interactivity of SDN, network components have become easily accessible and manageable. Applications can use this to offload certain functionalities from the data plane network layer, improving the complexity and efficiency of the data plane. To use SDN in a 3DTI environment, OpenSession has been developed. This is a session-network cross-layer management control protocol for managing multi-stream flows in 3DTI.

OpenSession partially offloads stream multicast functionality of the application to the SDN switches. If more than one participant wants a certain view, the source only sends one copy of the local stream to the switch. The switch handles the forwarding of this stream to multiple receivers, reducing processing load at the application side and also the usage of bandwidth in the network. The end-to-end delay in the network is also improved because the stream is forwarded on only one location.

This split of the data plane introduces several challenges.

1. Application layer multi-stream semantics are lost in the data plane. Without the semantics, the SDN switch has problems with forwarding streams according to the topology. To overcome this problem OpenSession has a mechanism for assistance to the switches to differentiate the packets through multiple layers.

2. With OpenFlow, SDN provides a fixed interface to configure the data plane of a network. Because the 3DTI multi-stream overlay topology in the application layer must be mapped on the network layer, a *mapping algorithm* was developed to perform this mapping.

3. 3DTI often needs updates of the configuration of the data plane because of the changes in views. To do this fast and seamless, the updates must be performed consistently across all the participants. This helps to avoid interference. For this, OpenSession promises a *coordinated route update protocol*.

In the paper [5], OpenSession has been implemented using OpenFlow switches and a Floodlight SDN controller. OpenSession was tested in a real 3DTI application setup with four participants who were distributed within a small geographical location. The results show that OpenSession improves the multi-stream 3D streaming performance. End-to-end delay is reduced (proportional to the round-trip time) on local networks. Also the bandwidth used and the processing load in the application host is reduced proportional to the number of 3D streams.

Impact of 3DTI on SDN for Multi-Camera Networks

The work done in [5] can be useful to incorporate in SDN for multi-camera networks when there is a need to broadcast 3D images. By offloading the multicast functionality to the SDN switches, a lot of bandwidth can be saved for the initial transmission of normalized images between cameras. Because the stream is multicasted by the SDN switches, the processing load becomes smaller. This involves that the processing can be done on a camera rather than on a powerful server.

Chapter 4

Topology for Multi-Camera SDN

4.1 Introduction

In the scope of this work, a basic topology, shown in figure 4.1 is created which can be used in a multi-camera system where the network is software-defined. The basic idea behind this topology is scalability and flexibility. This chapter briefly discusses the different elements in figure 4.1, starting with the camera node. This node is followed by the SDN network controller, the SDN switch, a server for caching and finally an user device.

4.2 Camera

The used cameras, are IP cameras. This means that they send their information over an ethernet based network. Therefore the cameras are connected to a switch. There is no need to connect them all to the same switch, because, in this case, it is possible to cover a bigger area with the cameras.

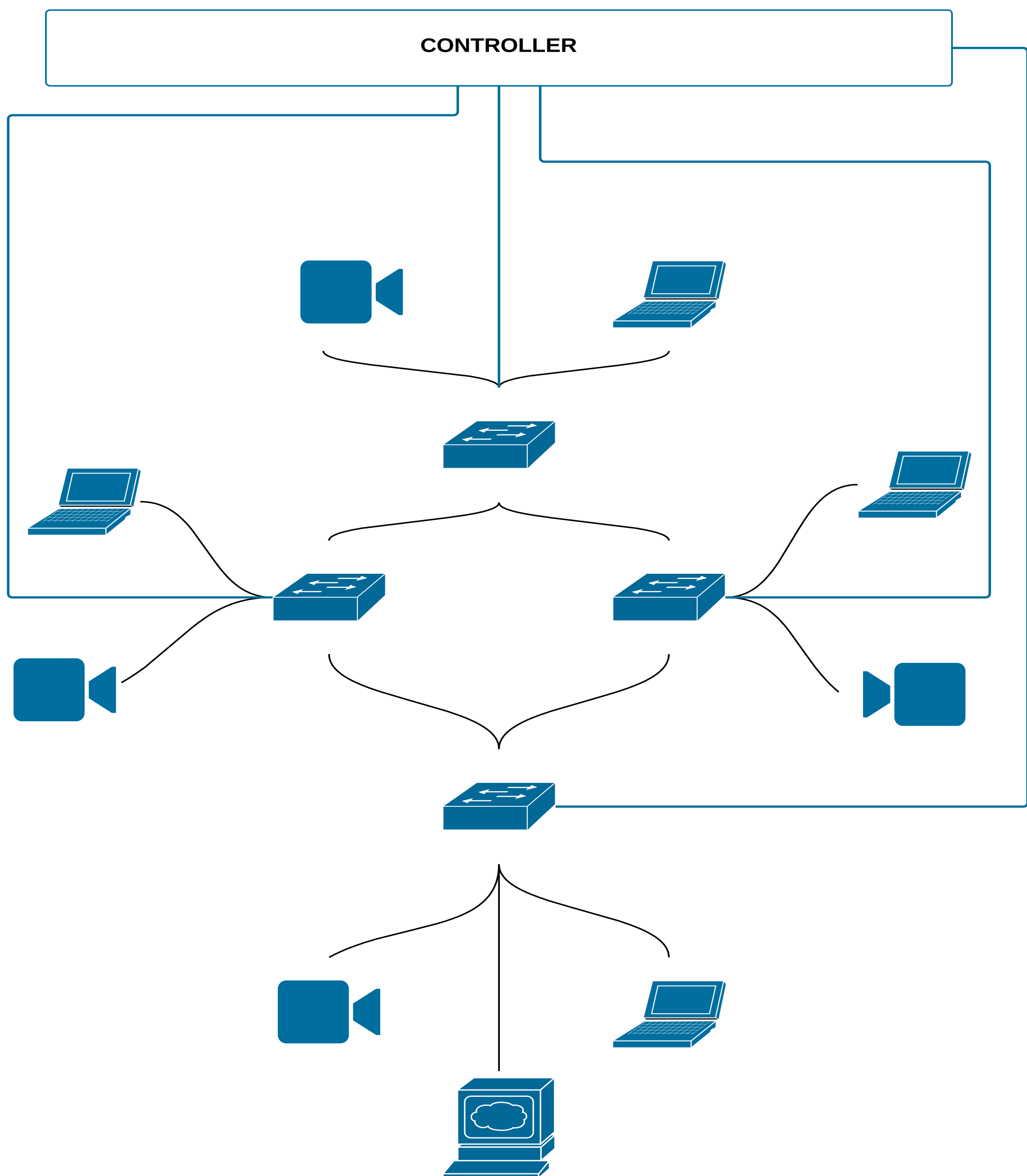

Figure 4.1: The overall topology

4.3 Controller

The controller is responsible to do the routing of the traffic. The controller needs to decide what path a certain image has to follow when it needs to be delivered to a requesting host. The controller knows the topology of the network. But in this case, not only the topology is important, it would be nice if the controller could also know what the physical location is of a host and a switch. On this base, it can make the routing decisions keeping in mind what would be the shortest path between a camera node and a host.

4.4 Switch

The switches are OpenFlow-enabled switches. These are described in section 3.2.1. This means that these switches are controlled by an OpenFlow controller, described in 4.3. These switches are receiving images from a camera and forward them on the network. There are also user devices and servers connected to a switch. It is the controller that decides what is the appropriate flow for the traffic between these switches.

4.5 Server

There is one server included in the topology. This server can be connected to any switch. The main purpose of the server is to cache the information that comes from the cameras. In theory this results in the delivery of smoother images to a host. The server can also be used for the storage of images or the features that are wanted from the cameras. In the future these servers will also be able to do image processing. So it can be possible to calculate an interpolated view out of two or more adjacent views.

4.6 User Device

The user devices can be connected to any switch that is included in the topology. A user device needs to communicate with the controller so the controller can know the physical location of the user device. There are many possible user devices, the only thing needed is that they can show the received images. It can for example be possible to use a pc, laptop, tablet, smartphone,... as an user device to watch the images from the camera.

Chapter 5

Use Case

5.1 Introduction

To proof the concept of the use of SDN in a Multi-camera network, this use case is developed. The purpose of this use case is to create a practical situation where a Software-Defined Network can be used. For that reason a soccer field is chosen. A soccer game is an ideal situation where 3D images can be captured. These images can be sent to an end user. In the future it is planned to do the calculations, necessary to create the 3D images, in the camera systems themselves. These cameras need to have the technology on board to fulfill this task. To do these calculations it is necessary to have the normalized images from at least two cameras, covering a common area. This two cameras can cooperate like the human eyes to do 3D image calculations. For this reason, a flow needs to be created between at least two neighboring cameras. In this way images from one camera can flow to a second camera, permitting this camera to do the necessary calculations. When the results are available, these images are sent to a client where the images can be used i.e. for broadcasting,... That is why a flow needs to be created from cameras to a client. The aim in this use case is to develop a network, enabling the creation of a 3D images of a ball on a certain position on a field.

5.2 Structure

Figure 5.1 shows the top view of how this use case can be developed in a real-life situation. In chapter 5.2.1 it is explained why the soccer field is divided into different zones. Chapter 5.2.2 describes why the cameras are clustered.

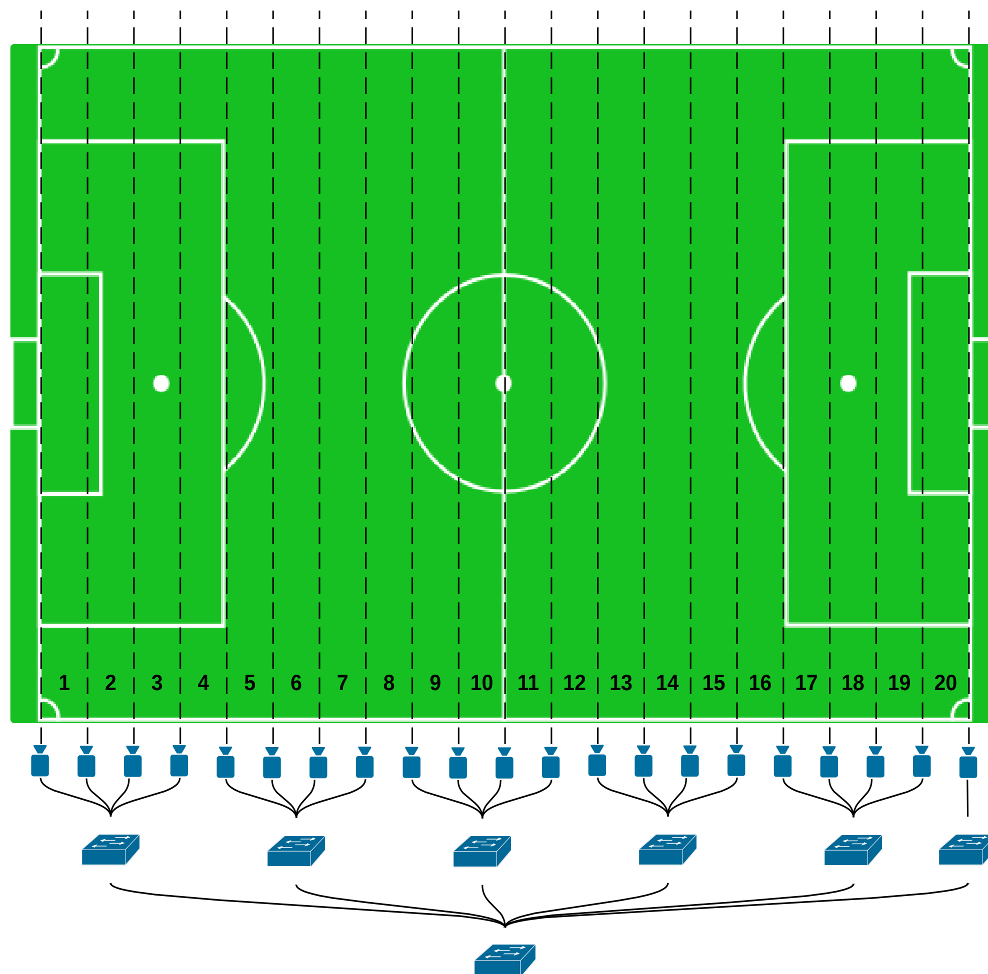

Figure 5.1: The use case

5.2.1 Zone Structure

To create 3D images it is necessary to have at least two cameras, which cooperates on one task. The difficulty is to decide which cameras need to cooperate to calculate the 3D image of the ball on the field. For that reason a *zone structure* is introduced. The soccer field is divided in a certain amount of regions, with a camera on each border. Every camera covers one zone on his left side and one zone on his right side.

An exception is made for the cameras located at the end of the field, these are only covering the zones included in the soccer field. The cameras all have the ability to cover the whole depth of the field.

If a ball enters a certain zone, the network controller needs to know which cameras are covering that zone. Then it can create a link between the two cameras which are covering that certain zone. In this way, traffic can be sent from one camera to an adjacent camera, which is responsible for the calculation of the 3D images.

5.2.2 Camera Clusters

The cameras generate a large amount of data (e.g. 1Gbps). For this reason, network traffic needs to be controlled, so the data coming from one camera will not disturb the whole network. Taking this in account, the choice is made to create a tree topology. In this tree, cameras are grouped per four and a group of cameras is connected to the same switch. Every *cluster switch* is connected to the one central switch, allowing communication between multiple clusters and also between a cluster and a client. This allows that the high amount of traffic that can be present in one cluster, does not interrupt the rest of the network. Traffic only has to flow through the central switch when a camera in one cluster needs to communicate with a camera, localized in another cluster. Besides that, also the traffic from this second camera to a client flows through the switch. The aim is to block the traffic coming from cameras that do not provide images necessary for the calculation of the 3D images described in 5.2.1. This helps to save some bandwidth, which can be used for the transmission of the images that are important for the 3D image calculations.

Chapter 6

Use Case Development

6.1 Introduction

This chapter discusses the development of the functionality needed to make it possible to proof the concept of the use case. In the first section, the network controller is discussed. A trade-off is made between two controllers: RYU (6.2.1) and Floodlight (6.2.2). Afterwards, in section 6.3, the description is given about the module, programmed to make this *proof of concept* working. Afterwards, the REST API, the database, the communication with the database and finally, the logic, added to the Floodlight controller, are discussed.

6.2 Network Controller

To control the network traffic flow and the rules, SDN uses a network controller. In this work a trade-off is made between RYU (6.2.1) and Floodlight (6.2.2). These two controllers are briefly explained next.

6.2.1 Ryu

The word RYU [11] comes from Japanese and means "flow". It is a component-based SDN framework. It provides software components with an API. For developers, this makes it easier to create new applications for the management and control of the networks. The framework is located in the middle between SDN applications and the OpenFlow switches. RYU communicates with the SDN applications using the API, while communication with the OpenFlow switches is done with the OpenFlow protocol. This is shown in figure 6.1.

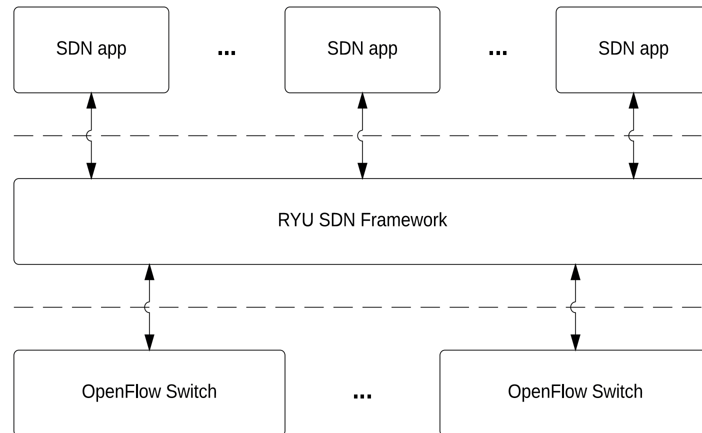

Figure 6.1: RYU architecture

In RYU, a component indicates a separation unit, providing an interface for control the state and to generate events. The communication is done by passing messages instead of directly to the unit. This communication is independent of the used language. The components included in RYU are written in Python and consists of Python thread(s) or OS process(es).

6.2.2 Floodlight

Floodlight [4] is another OpenFlow controller. It comes with a wide range of applications built on top. The basic controller has the functionality to control an OpenFlow network. The applications written on top provide functionality to serve a user.

Figure 6.2 shows relationship between the Floodlight Controller and the Module Applications. These two can communicate with each other through a Java API. Besides that, the possibility exist in Floodlight to communicate with applications written in another programming language. For this communication, a REST API is provided, allowing communication with JSON strings.

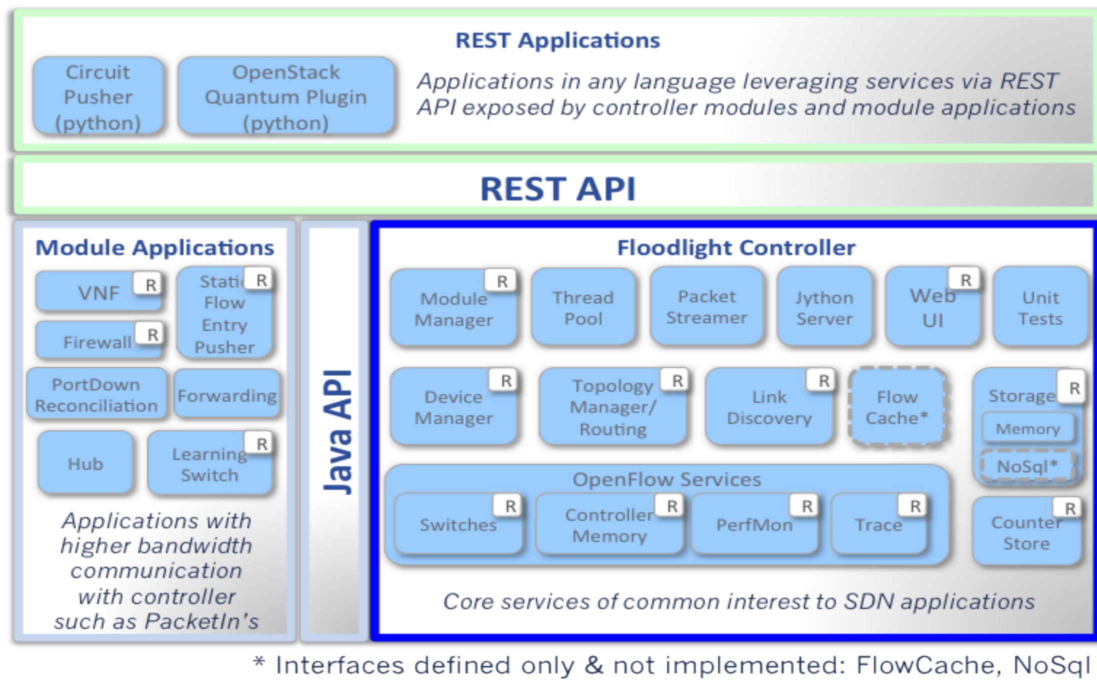

Figure 6.2: Floodlight diagram [4]

In the development of the use case, Floodlight is chosen as controller. This choice is made, because it is easy to create a new REST API. This API enables the controller to receive information from other applications, which can be written in another language. This is useful when communicating the location of a ball on the field to the Floodlight controller. The controller modules are all written in Java, a more familiar language than Python, used in RYU. The building and installation of Floodlight can be found on [8].

6.3 Controller Module

Figure 6.3 shows the structure that is used to develop the use case. The main component is the controller module. This module is responsible for the creation of the link between two camera nodes and from the second camera node to the client. To create this link, information is needed about where a ball is located on the field just as the location information of the cameras on the field. An API and a database are developed in order to retrieve and communicate this information. A ball location can be communicated to the controller module, while the controller module can connect to a database. This database contains the cameras that are covering that ball location. On this way the cameras can be communicated back to the module and a link between them can be created.

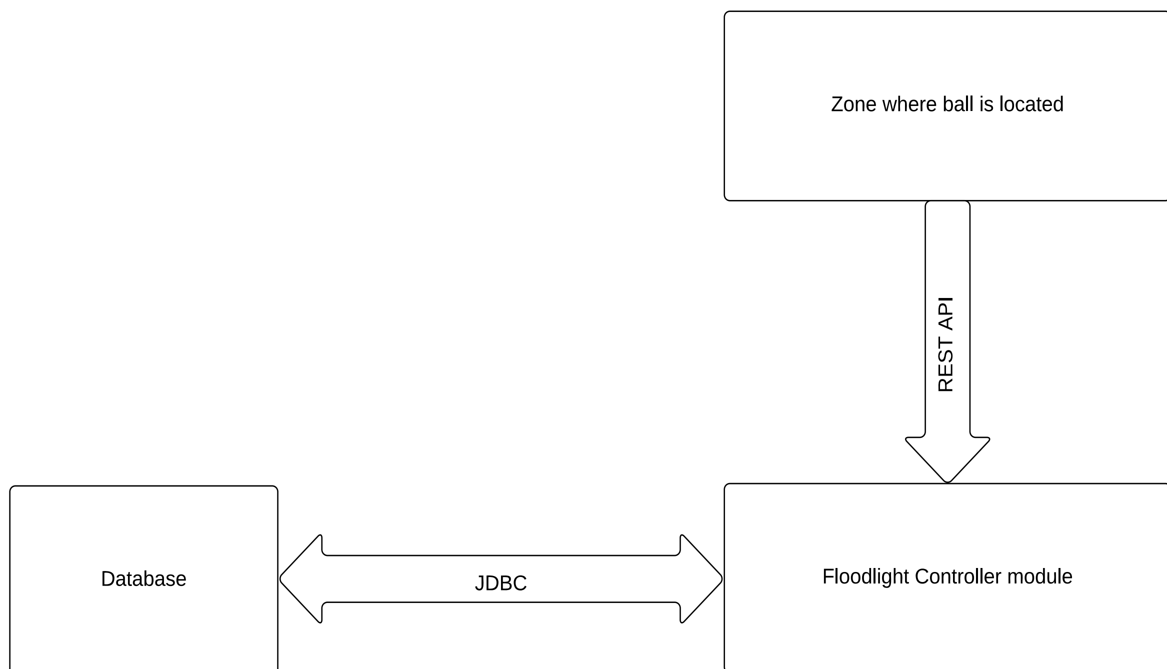

Figure 6.3: The structure of the created controller module

6.3.1 REST API

REST *(Representational State Transfer)* [7] is an architecture to create distributed software and to make the communication possible among its components. This architecture is based on six conditions:

1. Client-Server: the main target is to create an independence between a user interface and the data server. This is done so that the components can be used on multiple platforms and the components can evolve independent of each other.

2. Stateless: the server does not keep track of the state of the different clients. The requests that are sent to the server need to contain all the information that is required by the server to fulfill the request.

3. Cacheable: this means that the request can be cached on a server, so when a similar request is sent the server does not need to recalculate everything all over again.

4. Uniform interface: every service needs to have a similar interface. This makes it easier to get used to the interface used for new services. A disadvantage is that there is no space for potentially better, custom solutions, but this doesn't outweigh the big advantage of uniformity.

5. Layered system: a service only needs to know the surrounding layers. In this way the service might use other, lower level services, but the central service does not need to know how that service handles the request. In this way functionality can simply be added where and when needed.

6. Code-on-demand (optional): not all code needs to be included when starting a service. New code can be adopted when there is a need for the user.

In this use case, a REST API is used because of the uniform method of communication. The big advantage is that communication can take place between different programming languages. This is useful when a camera, with functionality programmed using another programming language than Java, needs to communicate the location of a ball on a field.

In order to make the REST API usable, it needs to be implemented in the controller module. This is done with the creation of four Java classes, described in the next subsections.

ICameraService

The ICameraService class is created to provide an overview of which functions are accessible from other modules.

In this case there are two functions who are public accessible.
The first one is the function *setBallZone*. This function is included to communicate the zone where the ball is located on a field for a certain event. Two arguments are expected as input. The fist is an integer representing the zone where the ball is located on the field. The second represents the event for which the ball zone needs to be set.
The second function is *clearAllFlowMods*. This function is included to clear the flows that are present in the flow table of the switches. This function can be useful when an error occurs, so it is possible to start the functionality again with an empty flow table.

CameraWebRoutable

The class *CameraWebRoutable* is created to register the REST API in the REST server of the Floodlight controller. This includes the creation of the domain on which the functionality of the API is accessible. First of all it sets the base path of this REST API. It also enables the access to the module that is handling the requests coming to this web address.

CameraResource

The *CameraResource* class includes the necessary functions to decode a JSON *(JavaScript Object Notation)* string that comes in on the API. Two functions are included to handle arriving JSON strings. At this moment only POST messages are handled, which should contain the JSON information that is needed to set the zone where a football is located for a certain event.

The first function is the *handlePost* function, returning a *String* over the REST API, with the status of the handling of the posted message. The function expects a JSON string as argument. First a request is done to *ICameraService* in order to know which functions of the *CameraModule* (6.3.3) can be used when handling a post.

The first thing to do is to decode the arriving JSON string. To accomplish this, the function *jsonExtractMessage* is used. This function returns an object in the form of *JsonString*. With the getters from that class, the new ball zone and the event can be retrieved. Next a try-catch statement is programmed to set the new ball zone with the *CameraModule* class.

If the received ball zone is greater than 0, the functionality of the *CameraModule* is used to set this new ball zone. When it is correctly executed, a confirmation of the setting of the ball zone is sent back over the API.

If the ball zone is equal to 0, the functionality of the *CameraModule* to clear all flow mods is executed. This choice is made because normally the numbering of the zones will start at 1. If the function is correctly executed, a status message is sent to confirm that the flows present were removed.

The last possibility is that another integer was sent through the REST API. The same functionality is used when another character is sent. In this case there will be no functionality present and only a status message is returned.

For the structured communication of information, JSON is used. To parse these JSON strings in the Java code, the toolkit *JSON.simple* is used.[1] This toolkit provides the functionality to encode and decode JSON strings. In this case only the decoding function of the toolkit is used.

A JSON string is built first with a string, containing an indicator which information is sent. In this case two data members are sent, a zone and an event. Following this indicator a colon indicates that the value of the information is sent. The comma indicates that a new data member starts.

A new JSON object is created, parsed from the incoming *jsonMessage*. Because these information fields contains string messages, the values are retrieved as a string. To use these zones, an integer needs to be parsed from this string. When these integers are correctly parsed, a new object of the form *JsonString* is created. After the creation of this element it is returned, so that the containing information can be used.

[1] https://code.google.com/p/json-simple/

JsonString

This class has been created to create a standard object wherein the information about a field zone can be stored. The setter for a ball zone requires two arguments. The first one is an integer, representing the zone where the ball is located; the second one is an integer that represents the event for which the *JsonString* is sent. Beside this, two getters are coded, so the zone and the event contained in this *JsonString* can be retrieved again.

6.3.2 Database

To store the information a trade-off is made between hard coding the information about cameras on the field in a module and between storing the information in a database alternatively. The decision is made to store the information in a database, this because the controller can then be used to globally control more than one event, or to control more than one side of the field when this is required. Another advantage is that the information in the database can be reused when a similar situation needs te be fitted with cameras.

In the next parts, the structure of the MySQL database, the connection method and the class that is used to retrieve the information from the database are discussed.

Database Structure

The database contains two different tables. The first table (table 6.1) is used to store the information about the cameras.

The second table (table 6.2) is a table, created to have certain information about the events. In this way the controller can potentially work on more than one event at a time.

Name	Type	Description
Id	Int	An integer indicating the specific element of the table it is.
Event_id	Int	An integer representing the id of an event. This id is the id that is associated to the event in the event table.
Camera_mac	Varchar	This data member must be included in the form '00:00:00:00:00:02', representing the unique MAC address of a camera that is located on a certain position on the field.
Dist_l	Int	An integer representing the distance that is covered on the left side of the camera.
Dist_r	Int	An integer representing the distance that is covered on the right side of the camera.
Zone_l	Int	An integer representing the zone that is covered on the left side of the camera.
Zone_r	Int	An integer representing the zone that is covered on the right side of the camera.

Table 6.1: Camera table

Name	Type	Description
Id	Int	An integer indicating which element of the table it is.
Name	Varchar	The name of the event.
Num_of_cam	int	An integer representing the amount of cameras that are used to cover the event.
Length	Int	An integer representing the length of the side that is covered.
Interspace	Int	An integer representing the interspace between the cameras.
BottomSwitch	Int	The amount of switches that are used to connect the cluster switches together. This is included for larger amounts of cameras, because on these bottom switches the traffic can not be blocked (described in 6.3.3).

Table 6.2: Event table

Java DataBase Connectivity

To connect to the database, API's are provided by the programming language to handle this connection. In Java, the API that is capable of doing the connection and the interactions is called Java DataBase Connectivity *(JDBC)* [9]. With JDBC you can connect to every kind of database if a driver exists. Drivers for MySQL are available on the website of MySQL. [2]

Connect

The Connect class is used to let the Java code interact with the MySQL database. As described in 6.3.2, this is being done with the JDBC API. This class contains six functions, which are explained in detail in the next paragraphs.

The first function that is included is the function to establish a connection with a MySQL database. The function *establish* first tries to load the appropriate drivers that are needed for the MySQL connection.

When this driver is loaded correctly, the connection with the database is established in a try-catch statement. This is done by the use of the *getConnection* class of the Driver-Manager. As arguments, the address of the MySQL server, the database and the username and password are needed. When this is executed successfully, it is possible to send SQL queries to the MySQL server. To end this connection, the function *disconnect* is used.

The next four functions are providing the functionality to interact with the MySQL database. The first one is *getCamera*, expecting an integer representing the zone and an integer representing the event, as input parameters. It returns an array list of cameras that are covering a certain zone of a certain event. In this new array list, the MAC address and the covered zone on the left and rigt of a camera are added. When the statement is finished, the array list is returned in order to use the information inside it.

The next two functions are implemented in the same way as the function discussed above. Except that the SQL statement and the arguments needed are different. The function *getAllCamerasEvent* returns an ArrayList containing the information of all the cameras that are used for a certain event. The other similar function is the function *getAllCameras*, retrieving all the cameras present in the table.

[2] http://dev.mysql.com/downloads/connector/j/

The last function that is implemented, is the function *getAmountOfBottomSwitches*. This function returns an integer that represents the amount of switches that is used to connect the *cluster switches* to each other.

Camera

Just as the class *JsonString*, discussed in 6.3.1, this class is created to have a standardized object that represents a camera. This object consists of three data members:

Type	Name	Description
String	Mac	The MAC Address of the camera
Int	zone_l	The zone that is covered on the left
Int	zone_r	The zone that is covered on the right

Table 6.3: Data members of camera

Beside a setter for these arguments, also three getters are made. These getters are used to retrieve a separate piece of information, MAC, zone_l, zone_r, stored in a *Camera* object.

6.3.3 Camera Module

The last class is *CameraModule*. This class is the main class for the created module. In the class, first the variables are created to use the functionality from other modules.
Many functions are auto-generated when creating this main class, from these functions, the following are adapted:

- getModuleDependencies(): returning an array list with the other modules used by this module.
- init(FloodlightModuleContext context): initializing the new variables that are created with the services used from other modules.
- startUp(FloodlightModuleContext context): executed on startup of the controller. In this case the REST API is registered and the connection with the database is established.

Another important thing is that the module needs to be registered in the Floodlight properties. This is done by the registration of the camera module into *src/main/resources/floodlightdefault.properties* and *src/main/resources/META-INF/services/ net.floodlightcontroller.core.module.IFloodlightModule*.

Besides the adaption of these functions, six new functions are created in order to provide functionality for the setting of a ball zone. These functions are discussed in detail in the next sections.

setBallZone

The most important function that is included, is the function to set the ball zone. This function is called when the REST API receives an input with location information. As input arguments the new zone for the ball and the event for which this rule needs to be set are required.

In this function first of all the MAC Address of the client is defined. On this way the other MAC-addresses will represent the camera nodes. After this declaration, the MAC addresses of the two cameras that are covering the zone are retrieved. This is done by using the getMac() function of a *Camera* object.

After this, the location of the MAC address is needed. The function *getMacLocation* is used. Here the MAC address is formatted as a *long*. After this a *SwitchPort* object is made, containing the switch DPID, the port of the switch which has the MAC address attached and the error status of the port.

Next the existing flows in the switches are deleted. This is done by executing the function *clearAllFlowMods*.

When this is executed successfully, the route between the two cameras is calculated. This is done by using the function *getRoute* from the module *Routing*. With this function, a route between the two nodes can be found. From the calculated route, the path is needed. Therefor the function *getPath* is executed, returning a list of *nodePortTuples*. This list contains the *nodeId* and the *portId* to identify switches that are included in the path.

Afterwards, the flows are installed on the designated switch(es). When the two cameras are located on one cluster, the flows only need to be installed on one switch. Then the function *installFlow* is executed just once. When the traffic needs to run between different clusters, the function needs to be executed multiple times. This is done with a for-loop. It must be done for every unique switch in the path list.

When the installation of the paths between the camera nodes is done, the path between the second camera and the client on the last switch needs to be established. This is done in the same way as the connection of cameras between multiple clusters.

When successfully, the traffic coming from the other cameras is blocked. In this way, superfluous traffic is avoided over the network. First, a set containing all the switch DPIDs is created. Secondly, the amount of switches that is used to connect the clusters is retrieved from the database. After this, an iterator is created to run through the elements of the data container. Finally the size of the set with the DPIDs is calculated, in order to know on how many switches the traffic needs to be blocked.

To block superfluous traffic, first the DPID of the switch is retrieved. After this the number of ports that are in use for that switch are retrieved. This is done because it is not necessary to block more than one port on a certain switch (in the use case for example on switch six). The function *blockTraffic* is executed to drop unwanted packets.

getMacLocation

This function requires a long integer, representing the MAC address, as an argument. The target device is an object of the form *IDevice*, which is an implementation of *IDeviceService*. An object of the form *SwitchPort* is created, by getting the attachment points from the target. Finally, this object is returned so it can be used.

installFlow

For this function, the DPIDs, ports and MAC addresses of the switches are needed as input arguments. The function creates an *OpenFlow match*, which is being used to refer to a single entry in a flow table. After the creation of the match, the input port and the source and destination MAC address are being set. Afterwards, the wildcards for the addition are set. These contain the value of the attributes for the switch. These wildcards also state that the input port, source and destination MAC have to match in order to set the rule for a certain node. When this is fulfilled the function *addFlow* is called.

addFlow

To install the flow on a switch, the OpenFlow message *OFPFC_ADD* is used to notify an OpenFlow switch that a new flow is coming. The match message, an OpenFlow message and the port of node two are needed as argument. A new OpenFlow module message and an array list, containing the necessary actions, are created. To this array list an action is added to send packets out the specified OpenFlow port. As argument only the port of the second camera node is used.

When done, the *flowMod* is written to the switch and the messages for the switches from the current thread are flushed.

clearAllFlowMods

To clear all the flows present in the flow tables, a set is created which contains all the DPIDs of the switches. Through this set runs an iterator, to clear the flow entries on all these switches. Just like in 6.3.3, an OpenFlow module message is created, with as biggest difference that the command in this case is not be the addition of a flow, but the removal of the flows. Therefor the command *OFFlowMod.OFPFC_DELETE* is used. The priority is set to 100, because it is more important the clear all the flows than to maintain the forwarding or blocking of flows that are present.

blockTraffic

The last function that is implemented is a function to block the traffic that enters from the cameras that are not used. The amount of ports that need to be blocked depends on the amount of camera nodes that are present on one switch. The match (discussed in 6.3.3), is now focused on just the input port. This input port is a port between one and the amount of ports that need to be blocked. The blockage is done by setting the output port of the traffic on these input ports to *none*. This means in the *OpenFlow message* that the following rule is added: *setOutPort(OFPort.OFPP_NONE)*. The priority is set to 1, so the flows between cameras will have a higher priority. After this, the rule is written to the switch and the traffic is blocked.

Chapter 7

Results and Discussion

7.1 Introduction

To prove that this controller does what it is expected to do, some experiments took place. Because the fact that it is not possible yet to use real hardware for the experiments, the switches, camera nodes and connections are emulated with Mininet. This is discussed in section 7.2. The main goal of these emulated experiments is to proof the concept of sending data over the Software-Defined Network, depending on the location of a ball in the football field. In the next chapters the emulated network is discussed, as well as the webpage used to emulate the switching location of a ball. Finally some test are done to show the flow of data.

7.2 Mininet

Mininet[1] is the network emulator that is used in these experiments. Mininet can run a collection of hosts, switches, routers and the links between them, on a single Linux kernel. Mininet is using lightweight virtualization to make one system look like a complete network. A host in Mininet actually runs like a real machine. You can start a SSH-session with it and run arbitrary programs on it. The running programs can send these packets to what seems to be a real ethernet interface. The processing happens in what seems to be a normal switch or router. In Mininet it is possible to customize packet forwarding and those custom SDN network designs can easily be transmitted to hardware OpenFlow switches. Mininet is an open source project and under active development.

[1]http://mininet.org

7.2.1 Limitations

Although Mininet is a very handy tool, it has some limitations. If Mininet runs on one single system, it can be necessary to balance the resources over the virtual hosts and switches. Next to that, Mininet is using a Linux kernel. This implies that it is impossible to run software depending on other OS kernels.

If in the application a custom behavior is needed, you need to write the controller yourself, because Mininet does not create it for you. The virtual hosts that you create, are by default not connected to your LAN. This means that they are not able to connect to a WAN. Also, Mininet is not a simulator and this means that measurements are based on real-time.

7.2.2 Topology

In order to do the emulations, a network topology is programmed for Mininet. This is done by describing the network in Python code. In figure 7.1 the topology used can be found. This topology consists of twenty-one cameras in order to emulate the coverage of a football field. Six switches are present to connect the clusters of cameras, one switch to connect these clusters and finally one server, which can be used for caching, storing and more.

The topology, as said before, is written in Python. First of all, the necessary imports are made from the mininet package, installed on the virtual machine. Afterwards the function *addTopology* is defined, expecting a net and a size as arguments.

To create the use case, first, the seven switches are added. When this is finished, the client is added. First the *endHost* is added, with a link to the seventh switch. This is done because the client needs the first MAC Address in order to recognize it easily. When done, the camera nodes are added in the network. This is done with a for-loop, running from one to the size of the network plus one. The links between the node and the cluster switch is created in an if-then-else statement, creating just four nodes on one switch. Finally, the links between the *cluster switches* and switch seven are created.

After this creation, the net is defined, listening for input on port 6634. The MAC Addresses of the nodes are set automatically. The amount of nodes present is defined as twenty-one.

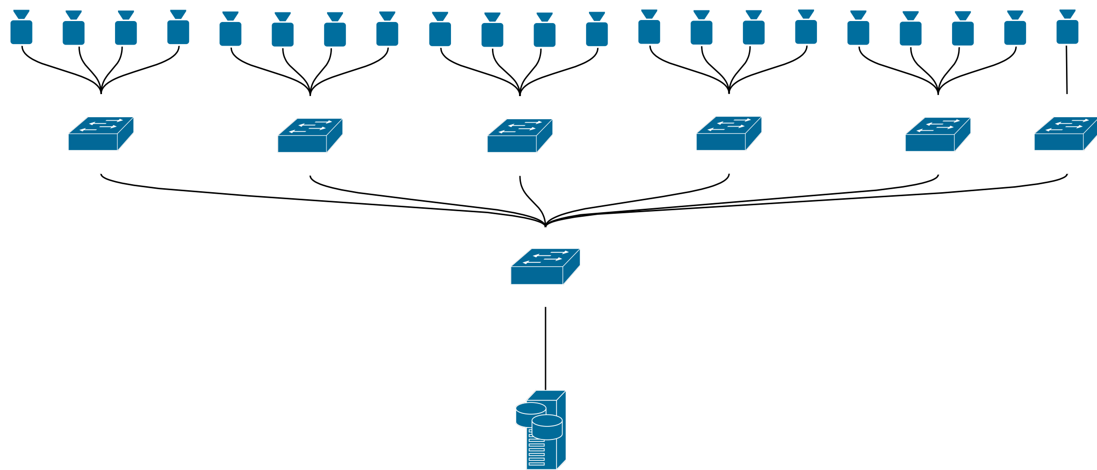

Figure 7.1: The topology used

Finally, the topology is started and a *PingAll* is sent, so the controller knows all nodes present.

7.3 Zone Locator

To prove the fast changes in the flows, a PHP page is created. This page does ten times a for-loop, sending a POST message to the controller module. In the for-loop, first, a random number is generated, representing the zone where a ball is located. This number, together with an integer to represent the event, is stored in an array, which afterwards is encoded into a JSON string.

To send the POST messages to the REST API, *cURL* is used. cURL is a software project to transfer data through various protocols.[2] First the connection is opened to the REST API. After this, the request is set to *POST* and the transfer string is added. This transfer string is used to indicate that a return message comes over the API. To finalize the message, the HTTP headers are set. When this is finished, the command is executed and the result is stored in the variable *result*. Afterwards the loop is going to sleep for five seconds, so only after five seconds the loop is executed again for a new randomized zone.

7.4 Flow Tests

To prove the functionality of this network, two kinds of test are performed. The first one, discussed in 7.4.1, is a test of the reachability of two nodes when the network rules are applied. The second test, discussed in 7.4.2, is a test to perform video streaming over the Software-Defined Network, when the rules are applied.

7.4.1 Ping Test

The first executed test tests the reachability of the hosts when a certain rule is applied. Because the use of the *Zone Locator (7.3)* would not allow to have enough time to perform the test, the POST message is sent with Postman[3], an extension to test API's in Google Chrome. The reachability of hosts is then tested with the *ping* command executed between two hosts.

For the first test (table 7.1), the message {*"zone":"1", "event":"1"*} is posted. This creates a path between camera one and two and between camera two and the client. The first

[2] http://curl.haxx.se/
[3] http://www.getpostman.com/

command sent was *h1 ping h2*, performing a ping between camera one and camera two. The reachability between camera two and the client and between camera three and four are also tested. This last one does not work, because there is a rule installed on the switch to drop the packets coming in from these cameras.

For the second reachability test (table 7.2) a path is created between camera four and camera five. This means that traffic goes through the central switch to flow from one camera cluster to another one. Also the connection between camera five and the client is initialized. The POST message sent is {*"zone":"5", "event":"1"*}. As can be seen, the connection between camera four and five works, just like the connection between camera five and the client. But camera seven can not reach camera ten, because of the traffic blockage.

The last test executed in this part is a test of the connection between camera twenty and camera twenty-one, covering the last zone of the use case (table 7.3). The controller is notified by the message {*"zone":"20", "event":"1"*}, that a connection between camera twenty and twenty-one needs to be established, just like the connection between camera twenty-one and the client. In the test it is shown that the camera nodes can communicate with each other. Camera twenty-one can communicate with the client, but other cameras can not reach each other.

Source	Destination	Packet Loss	Time
h1 (10.0.0.2)	h1 (10.0.0.2)	0 %	0.490 ms 0.074 ms 0.066 ms 0.086 ms
h2 (10.0.0.3)	endHost (10.0.0.1)	0 %	1.050 ms 0.173 ms 0.094 ms 0.092 ms
h3 (10.0.0.4)	h4 (10.0.0.5)	100 %	unreachable unreachable unreachable unreachable

Table 7.1: Ping test one

Source	Destination	Packet Loss	Time
h4 (10.0.0.5)	h5 (10.0.0.6)	0 %	2.110 ms 0.079 ms 0.099 ms 0.076 ms
h5 (10.0.0.6)	endHost (10.0.0.1)	0 %	1.290 ms 0.050 ms 0.061 ms 0.080 ms
h7 (10.0.0.8)	h10 (10.0.0.11)	100 %	unreachable unreachable unreachable unreachable

Table 7.2: Ping test two

Source	Destination	Packet Loss	Time
h20 (10.0.0.21)	h21 (10.0.0.22)	0 %	1.760 ms 0.093 ms 0.088 ms 0.087 ms
h21 (10.0.0.22)	endHost (10.0.0.1)	0 %	1.630 ms 0.073 ms 0.076 ms 0.072 ms
h1 (10.0.0.2)	h1 (10.0.0.2)	100 %	unreachable unreachable unreachable unreachable

Table 7.3: Ping test three

7.4.2 Video Test

Another test that is done, is a test of video transmission. To transmit video, VLC media player[4] is used. An User Datagram Protocol (UDP) stream is created between a source and a neighboring destination. To make to load on the virtual machine not very high, two times two hosts are chosen to do the test.

The first two hosts are host one and host two. These two were chosen because they are on the same *cluster switch*. The second two hosts are host four and host five. These are both on different *cluster switches*. This implies that flow rules have to be installed on the two designated switches for a cluster and on the connecting switch on the bottom.

To analyse the flow of data packages on the network, the network protocol analyzer *Wireshark*[5] is used. With this tool it is possible to track packages on the network and to analyse them.

Table A.1, which can be found in appendix A, shows the analysis of the network packages. In this table only the packages interesting for this specific application are maintained.

[4]http://www.videolan.org/vlc/
[5]https://www.wireshark.org/

In the first part of the table the data can be found with no blockage on the network. This means there is no ball zone set and all the traffic can flow without any interruption. As can be seen (between the time 10:30:28.1 and 10:30.28.2), streaming packages are just forwarded from host four (10.0.0.5) to host five (10.0.0.6) and from host one (10.0.0.12) to host two (10.0.0.3).

At the time 10.30.38, the ball zone is set to zone number one. This triggers host four to search for a destination for his images with destination host five. Because of the port block, these packages can not be delivered. Host four uses the Address Resolution Protocol (ARP) to resolve the MAC address of the host. Meanwhile the images from host one to host two can still be forwarded, because there is a link between those two cameras.

As can be seen in the table, at the time 10:30:49, the packages from host one to host two can not longer be delivered. This is because the ball zone is now set to zone four. This includes that the flow rules need to be installed on the two *cluster switches* and on the bottom switch. This includes that the images from host four can pass through to host five.

When a ball returns to zone one, the network controller makes a connection between host one and host two again and the other traffic is blocked. This can be seen because the fourth host is searching for the fifth host to deliver the video packages, but this is not possible because of the blockage.

In the last part of the table it can be seen that the both cameras are no longer able to transmit the video packages, because the zone is now set to zone twenty. This implies that traffic can only run between host twenty and host twenty-one.

The conclusion of this test is that the network controller is not only working on small packages like previously in the ping test, but also on larger packages like video streaming packages. This includes that the network controller, like proposed in this work, can be used in demand-driven applications with higher bandwidth needs.

Chapter 8

Conclusion

This work discusses the possibilities of Software-Defined networking for multi-camera systems. It is clear that SDN offers a dynamic way to manage large-scale networks. In the final version of the use case, it is shown that the Floodlight SDN controller can create a path between nodes, based on the location of a ball on a soccer field. This path is useful when there is a need to calculate 3D images. Beside this, it also does not matter how many cameras are used to cover the field. This can go from two cameras (but a very large view angle is needed), to more than hundred cameras. For the controller it does not matter, because it can just create a path between two cameras covering a zone on demand. This work offers the first insights in the possible usage of a Software-Defined Network for Multi-Camera Systems, but still lots of other things are possible.

First of all, extensions to the use case presented in this work are possible. It can be useful when there is a client-side page where an user can add a camera to a certain event, instead of just putting it in a database. This application must have the capabilities to find out which zone is covered by a certain camera, based on the distance between the cameras. Another possible extension is to not only track a ball, but also follow a certain player or the referee,... In that case, the API present must be rewritten in order to be able to receive these messages. Also the controller module needs to be rewritten, because there is not only a path between two cameras needed, but also a path to communicate which player needs to be followed by the intelligent cameras. This last thing can also be used for security applications, for example on a music festival where the security service wants to track a suspicious person.

Secondly, there can be a need to add other sensors together with the cameras. This could be for example useful to monitor traffic on the roads. For example, a sensor could be used to measure the speed of a certain vehicle. With Sensor OpenFlow (discussed in 3.3.2), there could be a possibility to communicate with the multi-camera system in order to track the car that is driving too fast and help the police to catch it.

A third thing that could be an extension of the current proposal, is another camera configuration. For example, cameras could be placed in a triangle, offering the possibility to make a full 3D image of an object that is located in the middle of the triangle. The biggest challenge here is the processing on board of the cameras. The creation of a path between the cameras will not be that hard, because it is something what the controller will do, based on the rules that are programmed and present.

A fourth and last extension to this project is in the field of a service provider - user relation. For example, in the case of a football field: *why only offer the images that are present?* With the usage of SDN it might be possible to calculate *virtual views*, which are interpolated views between two or more cameras. This might be useful when an end user wants to be the director himself. With an application for the end user, he may scroll just next to the field to follow something that is of special interest to him. This can be for example useful when there is a discussion about if a player is offside or not. The Software-Defined Network controller needs to create a fast and feasible path between the cameras and the user, also keeping in mind that more than one user might want to see a view, but this can be a different view. On this way, a path can be chosen that offers the end user a good experience.

The conclusion is that Software-Defined Networking is a dynamical and flexible way to control a network. This is useful in environments where multiple cameras need to cooperate on a certain task, as is shown in the developed use case. There is a nice future for the usage of Software-Defined Networking in Multi-Camera systems, but this work is only the first step in its application.

Bibliography

[1] C.-H. Lin, D.-N. Yang, C.-C. Lin, and W. Liao, "Multicast Group Management for Multi-View 3D Videos in Wireless Networks," 2014. arXiv:1409.8352v4.

[2] Y.-N. Y. Ting-Yu Ho and D.-N. Yang, "Multi-View 3D Video Multicast for Broadband IP Networks," 2014. arXiv:1410.3977v1.

[3] S. Lee, Y. Jeon, S. Choi, M. S. Han, and K. Cho, "Gigabit uwb video transmission system for wireless video area network," *Consumer Electronics, IEEE Transactions on*, vol. 57, pp. 395 – 402, May 2011.

[4] "Project floodlight." http://www.projectfloodlight.org, November 2014.

[5] A. Arefin, R. Rivas, R. Tabassum, and K. Nahrstedt, "Opensession: Sdn-based cross-layer multi-stream management protocol for 3d teleimmersion," in *16th International Conference on Network-Based Information Systems*, pp. 44–51, September 2013.

[6] H. Owens and A. Durresi, "Video over software-defined networking (vsdn)," in *16th International conference on Network-Based Information Systems*, Conference Publishing Services, 2013.

[7] K. Aerts, *SOA and Cloud Computing.* UHasselt/KU Leuven FIIW, 2013.

[8] B. Salisbury, "How to build a floodlight controller module." http://networkstatic.net/tutorial-to-build-a-floodlight-sdn-openflow-controller-module/, November 2011.

[9] K. Aerts, *Database programmatie met Java en C sharp.* UHasselt/KU Leuven FIIW, 2012.

[10] Y.-C. Chen, D.-N. Yang, and W. Liao, "Efficient multi-view 3d video multicast with depth image-based rendering in lte networks," *Globecom 2013 - Wireless Networking Symposium*, pp. 4427 – 4433, 2013.

[11] *RYU SDN Framework*. https://itunes.apple.com/be/book/ryu-sdn-framework-english/id828730590?l=nl&mt=11: Ryu Project Team, February 2014.

[12] T. Luo, H.-P. Tan, and T. Q. S. Quek, "Sensor openflow: Enabling software-defined wireless sensor networks," *IEEE Communications Letters*, July 2012.

[13] A. Motten, *Multi-Camera Computational Video Architecture*. PhD thesis, Universiteit Hasselt, Agoralaan D, 3590 Diepenbeek Belgium, 2013.

[14] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 493–512, 2014.

[15] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *IEEE Communications Surveys and Tutorials*, vol. PP, pp. 1–27, June 2013.

[16] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys and Tutorials*, vol. 16, pp. 1617 – 1634, June 2014.

[17] "Software defined networking." http://en.wikipedia.org/wiki/Software-defined_networking, 2014.

[18] "Openflow." http://en.wikipedia.org/wiki/OpenFlow, 2014.

[19] H. Aghajan and A. Cavallaro, *Multi-Camera Networks - Principles and Applications*. ISBN 13: 978-0-12-374633-7, 30 Corporate Drive, Suite 400 Burlington, MA 01803: Elsevier - Academic Press, 2009.

[20] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: A survey," *IEEE Communications Magazine*, pp. 24 – 31, November 2013.

[21] C. J. Bernados, A. D. L. Oliva, P. Serrano, A. Banchs, L. M. Contreras, J. C. Zuniga, and H. Jin, "An architecture for software defined wireless networking," *IEEE Wireless Communications*, pp. 52 –61, June 2014.

[22] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network (sdn) and openflow: From concept to implementation," *IEEE Communications Surveys and Tutorials*, vol. 16, pp. 2181 – 2206, May 2013.

Appendix A

Wireshark Output for Video Tests

Time	Source	Destination	Protocol	Length	Info
...					
10:30:28.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Program Map Table (PMT)
10:30:28.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:28.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:28.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:28.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:28.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:28.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:28.1	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:28.1	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:28.1	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:28.1	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:28.2	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent

Time	Source	Destination	Protocol	Length	Info
...					
10:30:38:5	10.0.0.2	10.0.0.3	MPEG TS	1358	
10:30:38.5	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:38.6	Mac Cam 4	Mac Cam 5	ARP	42	Who has 10.0.0.6? Tell 10.0.0.5
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:30:38.4	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
...					
10:30:39.6	Mac Cam 4	Mac Cam 5	ARP	42	Who has 10.0.0.6? Tell 10.0.0.5
10:30:39.6	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
...					

Time	Source	Destination	Protocol	Length	Info
10:30:48.0	Mac Cam 1	Mac Cam 2	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
10:30:48.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:48.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:48.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
...					
10:30:49.0	Mac Cam 1	Mac Cam 2	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
10:30:49.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:49.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:49.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
10:30:49.1	10.0.0.5	10.0.0.6	MPEG TS	1358	Source port: 58018 Destination port: search-agent
...					
10:31:11.5	Mac Cam 4	Mac Cam 5	ARP	42	Who has 10.0.0.6? Tell 10.0.0.5
10:31:11.6	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:11.6	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:11.6	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:11.6	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:11.6	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:11.6	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent

Time	Source	Destination	Protocol	Length	Info
10:31:20.8	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:20.8	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:20.8	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:20.8	10.0.0.2	10.0.0.3	MPEG TS	1358	Source port: 36639 Destination port: search-agent
10:31:20.7	Mac Cam 4	Broadcast	ARP	42	Who has 10.0.0.6? Tell 10.0.0.5
10:31:21.0	Mac Cam 1	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2

Table A.1: Output of Wireshark

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Software-Defined Networking for Multi-Camera Systems

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2015**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Rymen, Martijn

Datum: **15/01/2015**