The Omniscope - Multimedia Streaming and Computer Vision for Applications in the Virtuality Continuum

Gerald Melles[1]

Abstract: Researching applications within the Virtuality Continuum (VC) is a process involving combinations of many different technologies. Media streaming and computer vision in particular are important aspects of many VC applications. This paper introduces the Omniscope library as a way to integrate both in an efficient, user-friendly and extensible manner. It achieves this by combining GStreamer and OpenCV in a C/C++ library as well as a plugin for the Unity IDE.

Keywords: virtuality continuum; VR; streaming; computer vision; OpenCV; GStreamer; Unity

1 Introduction

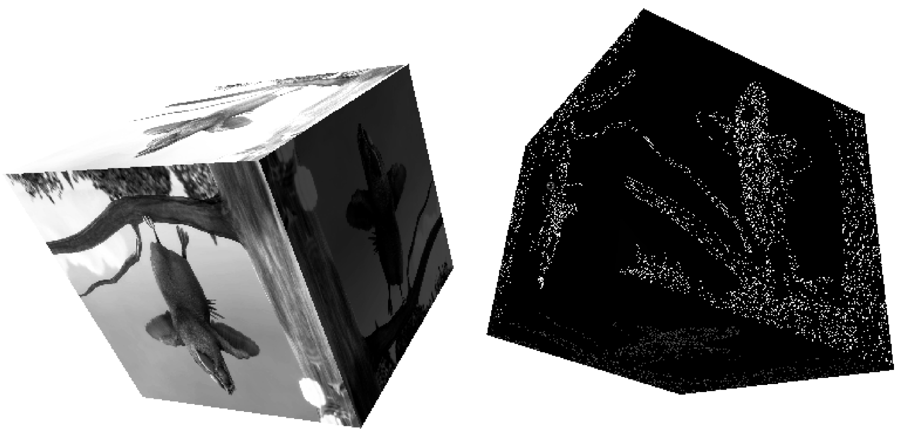
Fig. 1: Omniscope Plugin rendering and transforming a Full-HD video using edge detection

Multimedia streaming is a common requirement in systems within the virtuality continuum (VC) which encompasses Augmented, Virtual, Mixed and Blended Reality (cf. [MK94]). Among other aspects, media streams are often used to transport the required data for visual and auditory stimuli to and from peripheral devices.

[1] Hamburg University Of Applied Sciences, Fakultät TI-I, Berliner Tor 5, 20099 Hamburg, Deutschland
contact@geraldmelles.com

VC applications also make extensive use of computer vision algorithms, such as in camera-based tracking and motion capture systems. In [Ng17] for instance, Nguyen et al. propose a new and less obtrusive design for markers similar to QR-Codes and suggest their use in VR applications.

VC research projects often need combinations of both of these aspects, for example to take the user's gaze into account to render in a foveated manner, such as in [Lu17]. There is no lack of projects and proposals in this intersection, but their tooling tends to be either purpose-built for a specific use case or proprietary.

Generic multimedia streaming and computer vision libraries exist, in both proprietary and open source variants. The main challenge lies in their integration into the game or simulation engine at the core of the VC application. This is exacerbated by requirements concerning speed, efficiency and extensibility, the maintenance of platform independence and the support for a range of data formats and devices. This may go some way toward explaining why there is no generic and open solution. Computer vision algorithms in particular often remain inaccessibly embedded within proprietary software and hardware products. Rapidly prototyping a VC application combining the two as well as maintaining the ability to change their internal workings is not supported by software solutions that are available today.

The Creative Space for Technical Innovations (CSTI) at the Hamburg University of Applied Sciences (HAW) primarily uses the Unity IDE as the simulation engine for developing their applications. This proprietary IDE is one of only a few viable options for the development of state-of-the-art VC systems, especially when developing a new simulation engine is not an option (i.e. due to the time and effort required). It can be extended through scripting and plugins in a variety of programming languages. At the time of writing, Unity is one of the most commonly used IDEs for VR and AR simulations, with its main competitors being other proprietary products like the Unreal Engine and Cryengine. In part, this dominance of proprietary engines may be related to the need to support the peripherals and platforms unique to VC systems, such as head-mounted displays, wands and motion tracking systems, which are also proprietary themselves.

Both the IDE's built-in media streaming support and its open source plugins lack a way to efficiently interface them with computer vision libraries, in addition to providing only very limited support for formats and encodings. There are also no comprehensive open source integrations of computer vision libraries themselves.

This paper presents the Omniscope library and plugin for Unity as a means of making the research and development of media streaming and computer vision based applications in the virtuality continuum faster, easier and more efficient. It was developed as a link between the two fields: A C/C++ library combining the GStreamer and OpenCV libraries. It is intended to make media streaming and computer vision features as accessible and customizable as possible for researchers into VC systems at the CSTI and elsewhere and can be integrated into Unity using the IDEs native plugin architecture. Plans exist to adapt it for other VC development environments, such as the Unreal Engine.

Fig. 1 shows the Plugin rendering an MP4 video[Bi08] onto cubes in a Unity scene. On the right cube, an edge detection algorithm was applied before rendering (an implementation of the algorithm proposed by John Canny in [Ca86]).

2 Architecture

Audio and video streams flow back and forth between servers, the VC simulation system and its various peripheral devices. A large number of standards, formats and encodings are in use, both free and proprietary. A few streaming frameworks seek to support as many of these as possible, both by endeavoring to be agnostic to the nature of the data they handle and by utilizing plugin architectures for components like encoders/decoders to ease their integration. One such framework is GStreamer, which is largely platform-independent and included in most Linux distributions.
OpenCV is a well-established computer vision library. Its open source nature has made it a popular basis for research into computer vision, particularly of feature detection algorithms. While it does already offer an integration with GStreamer, the Omniscope library instead offers its own in order to allow for better extensibility (i.e. other streaming framework integrations) and more complete playback support.
The Omniscope plugin links with both GStreamer and OpenCV dynamically. This is largely to accommodate the possibility of the user requiring all of the libraries' modules to have a specific (e.g. non-proprietary) licensing model and to enable its use in applications without licensing issues.[2]
Similarly, the plugin's functionality is exported to Unity using C-style bindings in a dynamically loaded library.[3]
The plugin's pipeline architecture with elements, sources and sinks is similar to that of GStreamer. Like GStreamer, the Omniscope library also largely abstracts from its elements' implementations. This way, elements are also more easily added to or removed from a project, regardless of their dependencies.
The Omniscope project currently consists of six C/C++ subprojects: Five internal modules and a standalone application. These subprojects are intended to be built using the GNU GCC Compiler (cf. [GN18]) for Linux-based systems or cross-compiled for Microsoft Windows using MinGW-w64 (cf. [Mi18]). Additionally, the project contains C# scripts for easier access to the library from within the IDE.

- Omniscope Common Module
- Omniscope GStreamer Module
- Omniscope OpenCV Module
- Omniscope Core Module
- Omniscope Unity Module
- Omniscope Standalone application

[2] Note that all GStreamer plugins used by the Omniscope plugin are licensed under the LGPL and OpenCV under the 3-clause BSD license.
[3] The Omniscope Unity module itself contains some code by Unity Inc., under the MIT license.

Omniscope Common contains header files with the pipeline elements' base interfaces and a custom thread pool implementation.
Omniscope GStreamer contains element specializations for GStreamer pipelines and elements. It provides support for GStreamer's gst-launch command line syntax, which is the recommended way for the user to extend the Omniscope plugin's streaming functionality without having to resort to working with its source code. It is required that the user supply at least one appsink or appsrc element in the pipeline definition as these are used for the exchange of samples between GStreamer and other pipeline elements.
Omniscope OpenCV consists of elements offering stream capture and processing implementations using OpenCV. It can be extended with custom sample analysis and processing modules. In its simplest form, a new sample processor can be implemented by providing a new override to a single function taking and returning a generic media sample instance (which may contain several frames and additional information). By default, sample processing functions are automatically executed asynchronously using a thread pool.
The Omniscope Core provides the pipeline itself: an API for instantiating and connecting sources, sample processors and sinks, manipulating their state (e.g. playing, pausing and seeking) and accessing the resulting media samples and analysis results.
The Omniscope Unity module adapts Unity's low-level native rendering plugin support (in C++) to permit rendering captured and optionally processed frames directly to existing target textures. This low-level access to Unity's rendering APIs makes the Omniscope much faster than passing frames up to Unity's C# scripts for rendering would be. It is also capable of interfacing with different graphics architectures and maintains the same platform independence as the rest of Omniscope's modules. In addition, any analysis results of sample processors can also be accessed through it. It is nevertheless recommended to use the provided C# scripts to facilitate communication between the Omniscope library and Unity. These use the Unity IDE's component system to provide a graphical UI (cf. 'Usage').
The project also contains a simple standalone application built upon the other modules (excluding the Omniscope Unity module). This primarily serves as a means to ease development and testing of new features but could also be used for the development of standalone streaming and computer vision applications.

3 Usage

Most of Omniscope's complexity can be hidden behind a graphical UI. The user may interact with the plugin in four ways (ascending by flexibility and descending by ease of use):

- Using Omniscope's UI (in Unity)
- Using Omniscope's C# API (within Unity's scripting system)
- Using Omniscope's 'C' style linkages
- Extending Omniscope's modular architecture in C/C++

The first option consists of an extension to Unity's component inspector. It offers input fields and buttons (no programming is required) and is intended to be used primarily for simple audio and video playback and the use of pre-built frame processors. The use of GStreamer's gst-launch syntax adds additional flexibility.
The second option is intended for developers who either want to have programmatic control over the plugin's behavior at runtime (e.g. for pausing playback or seeking) or to use the built-in computer vision algorithms for frame-by-frame analyses. For example, it could be used to apply a feature detection algorithm as a sample processor and receive the relative positions of detected features (such as eyes and faces) once per render cycle.
If the C# API does not suffice or if the plugin is to be used outside the Unity IDE, the user may also use the plugin's 'C' style linkages.
Lastly, as mentioned above, the Omniscope's architecture has been designed with extensibility in mind. Its central features are abstracted through interfaces and abstract base classes, easing the development of new elements, such as frame processing or stream capture implementations.
The Omniscope's focus lies on video and vision, but it also offers limited support for other media such as audio streams and subtitles to accompany video playback.

4 Conclusion

The need for well-integrated, efficient, open and extensible streaming and computer vision support for the research and development of VC applications has been established. The Omniscope project has been introduced as a solution, combining the GStreamer and OpenCV libraries and optionally integrating them with the Unity IDE. Its uses in VC applications include playing back multimedia streams from a variety of sources (both video and audio), transforming the visual frames - for example to show detected edges in a stylized way - and extracting other data through computer vision algorithms, such as the location of detected faces or tracking markers. It is being used in the CSTI and steadily improved and extended.

5 Further Research

While the Omniscope is already in use by researchers in the CSTI, formal and exhaustive analyses of its speed and efficiency have yet to be performed.
The Omniscope could also serve as a means of integrating other media based features with VC applications, such as three-dimensional audio processing based on the acoustics of a virtual space. Related work is being done at the HAW's wave field synthesis lab, for example exploring the use of acoustics in redirected walking (cf. [NF16]).
The capture of multimedia streams and their processing by computer vision algorithms are also only some of the aspects of networked VC systems. These systems' communication with various peripherals, local software and hardware landscapes and remote services provides many other challenges worthy of research. Of particular interest to researchers in the CSTI

is the integration of embedded systems and smart environments in VC systems. To this end, a number of projects have sprung up, such as the CSTI middleware (cf. [Ei17]).

References

[Bi08] Big Buck Bunny, original by the Blender Foundation (peach.blender.org), official mirror by Janus B. Kristensen (http://bbb3d.renderfarming.net), last accessed 01.07.2018).

[Ca86] Canny, J.: A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(6):679–698, Nov 1986.

[Ei17] Eichler, Tobias; Draheim, Susanne; Grecos, Christos; Wang, Qi; von Luck, Kai: Scalable Context-Aware Development Infrastructure for Interactive Systems in Smart Environments. In: Fifth International Workshop on Pervasive and Context-Aware Middleware 2017 (Per-CAM'17). Rome, Italy, October 2017.

[GN18] GNU Compiler Collection (official site): https://gcc.gnu.org/.

[Lu17] Lungaro, Pietro; Tollmar, Konrad; Mittal, Ashutosh; Valero, Alfredo Fanghella: Gaze- and Qoe-aware Video Streaming Solutions for Mobile VR. In: Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology. VRST '17, ACM, New York, NY, USA, pp. 85:1–85:2, 2017.

[Mi18] MinGW-w64 - GCC for Windows 64 & 32 bits (official site): https://mingw-w64.org.

[MK94] Milgram, Paul; Kishino, Fumio: A Taxonomy of Mixed Reality Visual Displays. In: IEICE Trans. Information Systems. volume vol. E77-D, no. 12, pp. 1321–1329, 12 1994.

[NF16] Nogalski, M.; Fohl, W.: Acoustic redirected walking with auditory cues by means of wave field synthesis. In: 2016 IEEE Virtual Reality (VR). pp. 245–246, March 2016.

[Ng17] Nguyen, Minh; Tran, Huy; Le, Huy; Yan, Wei Qi: A Tile Based Colour Picture with Hidden QR Code for Augmented Reality and Beyond. In: Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology. VRST '17, ACM, New York, NY, USA, pp. 8:1–8:4, 2017.

Fundamentals of Real-Time Data Processing Architectures Lambda and Kappa

Martin Feick, Niko Kleer, Marek Kohn[1]

Abstract: The amount of data and the importance of simple, scalable and fault tolerant architectures for processing the data keeps increasing. Big Data being a highly influential topic in numerous businesses has evolved a comprehensive interest in this data. The Lambda as well as the Kappa Architecture represent state-of-the-art real-time data processing architectures for coping with massive data streams. This paper investigates and compares both architectures with respect to their capabilities and implementation. Moreover, a case study is conducted in order to gain more detailed insights concerning their strengths and weaknesses.

Keywords: Software architecture, Big Data, real-time data processing, Lambda and Kappa architecture

1 Introduction

The internet is a global network that is becoming accessible to an increasing number of people. Therefore, the amount of data available via the internet has been growing significantly. Using social networks for building communities, distributing information or posting images represent common activities in many people's daily life. Moreover, all kinds of businesses use technologies for collecting data about their companies. This allows them to gain more detailed insights regarding their finances, employees or even competitiveness. As a result, the interest in this data has been growing as well. The term *Big Data* is used for referring to this data and its dimensions.
As Big Data has progressively been gaining importance, the need for technologies that are capable of handling massive amounts of data has emerged. In this paper, one technology of interest is the so-called Lambda architecture that was introduced by Nathan Marz in 2011 [Ma11]. Its introduction was motivated by the purpose of beating the popular CAP theorem [Br00]. The CAP theorem states that a shared distributed data system is incapable of guaranteeing Consistency, Availability and Partition tolerance at the same time. Instead, only two constraints can at most be enforced. Marz emphasizes that the architecture does not rebut the CAP theorem but simplifies its complexity to allow for more human-fault tolerance when developing shared data systems.
The Lambda architecture has enjoyed broad attention which has led to numerous people

[1] Hochschule für Technik und Wirtschaft des Saarlandes, Fakultät für Ingenieurwissenschaften, Goebenstraße 40, 66117 Saarbrücken, Deutschland; e-mail: {mfeick,nikleer,mkohn}@htwsaar.de

sharing their thoughts about the technology. A considerably influential article by Jay Kreps (2014) acknowledges Marz's contribution for raising awareness about commonly known challenges of building shared data systems. At the same time, he discusses some disadvantages of the Lambda architecture [Kr14]. Consequentially, he proposes an alternative real-time data processing architecture, termed Kappa architecture, as an alternative. In contrast to the Lambda architecture, Kreps's approach is supposed to simplify any development related matters.
In this paper, we investigate the Lambda as well as the Kappa architecture, take a more detailed look at their functionalities and compare their capabilities. Therefore, the paper is divided into the following sections. Subsequently, we elaborate on the related work regarding Big Data and real-time data processing. After that, we take a more detailed look at both architectures including their workflow and implementation. Moving to section 5, we conduct a case study in which we compare both architectures[2]. This way, we are be able to analyze each architecture's strengths and weaknesses more effectively. A subsequent discussion proceeds by emphasizing significant details regarding our results. Finally, we conclude this paper's results and consider potential future work in the last section.

2 Related Work

In the related work section, we first introduce the term Big Data, and we briefly discuss the issues related to it. Afterwards, we look at *data processing solutions* as well as the term *data analytics* in order to support real-time Big Data streams.

2.1 Big Data

Over the last decade, the term Big Data became more and more relevant. Big Data has its place in almost every business area such as information technology, healthcare, education etc. However, the term Big Data does not only cover the pure size of data [Ma15, Ma11]. Instead, Big Data is composed of three standard dimensions known as the three V's, which mean *Volume*, *Variety*, *Velocity* [Ga15]. Additionally, certain companies contributed other dimensions to Big Data. For example, IBM added *Veracity* as a forth V, SAS introduced *Variability and Complexity*, and finally Oracle brought up *Value* [Ga15].
Often, the scale of data needed to support the various application scenarios is too big for a traditional database approach. Handling such an amount of data cannot be done by simply increasing the resources, because it does not consider the higher complexity and coherence of the data [Ma15, Ga15]. The next section introduces real-time data processing solutions considering all previously introduced dimensions of Big Data.

[2] The project is available on GitHub: `https://github.com/makohn/lambda-architecture-poc`

2.2 Real-Time Data Processing Solutions

Hasani et al. [Ha14b] outline that particularly the *Velocity* aspect of Big Data is difficult to handle effectively. Technologies must be able to handle real-time stream processing at a rate of millions per second. Furthermore, data streams can be collected from various sources using parallel processing. However, the goal of Big Data is to gain knowledge about the data and this is only attainable with the help of data integration and data analytics methods [Li14]. Data analytics is essentially the process of examining a data set and conclusively getting the insight/value of the data [Li14]. However, for traditional tools, it is challenging as soon as the data size extensively grows [Ha14a]. In addition, besides the most recent data, for some requests all the data is needed as it leads to more accurate outcomes [Ki15, Ha14a]. As a result, accessing information within a time limit is often not possible due to the size of the data [Li14].
A common strategy to face this challenge is to use hybrid techniques [Ki15, Ma15, Ha14a]. Abouzied et al. [Ab10] discussed HadoopDB which combines MapReduce and DBMS technologies. It is used to analyze massive data sets on very large clusters of machines [Ho12]. They present different real world applications e.g. a semantic web data application for protein sequence analysis [Ab10]. Their results show that HadoopDB is an effective platform for retaining a large data set and performing computation on it. However, HadoopDB has its limitations when using real-time data streams [Li14].
We previously introduced two general requirements of Big Data systems. First, receiving a massive real-time data stream from different sources and second, performing an analysis of this data in order to output results almost immediately [Ma15, Li14, Ha14b, Ki15]. From this point, we move on to two concrete software architectures/patterns called Lambda and Kappa that are nowadays commonly used for Big Data systems.

3 Lambda Architecture

The Lambda architecture has been given its name by Nathan Marz [Ma11], and describes a generic, scalable and fault-tolerant real-time data processing architecture. It provides a general-purpose approach to apply an arbitrary function on an arbitrary data set [Ma11]. Marz defines the most general-purpose function, as a function that takes all the existing data as input (*query = function(all data)*), and returns its results with low latency. However, calculating results to ad-hoc queries using the entire data set is computationally expensive. Therefore, the Lambda architecture uses pre-computed results (views) being able to respond with low latency [Ma15].
Figure 1 shows an overview of the Lambda architecture comprising three layers. The batch layer has essentially two functions: (1) It stores an immutable master data set and (2) is responsible for pre-computing the batch views based on this data set. The speed layer is responsible for indexing real-time views, compensating the high latency of the batch layer. In particular, due to the massive data sets in the batch layer, it takes time for the latest batch layer views to be calculated, causing a lack of availability. The speed layer is used to close

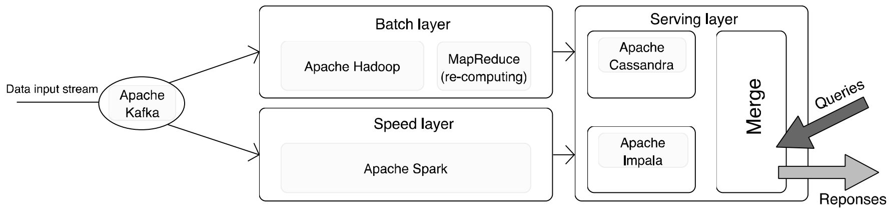

Fig. 1: The Lambda architecture and its workflow through Batch, Speed and Serving layer

this gap by providing an efficient way for querying most recent data [Ma15]. As soon as the batch layer has re-computed its views, the speed layer discards the redundant data, and hence they are provided by the batch layer views. Moreover, there are queries where most recent data and data from the batch layer are required. Therefore, the serving layer merges results from batch and speed layer views. Further, the serving layer takes care of indexing and providing the merged views, enabling easy access for the user.

3.1 Workflow & Technologies

As illustrated in Figure 1, all incoming data is dispatched to batch and speed layer for further processing. Since we talk about a real-time stream processing with a massive amount of data, a common technology to realize this is Apache Kafka [Du16].
Moving on to the batch layer, a standard technology to store the master data set and to perform the recomputations of the batch views is Apache Hadoop [Ha14a]. Hadoop is an open-source framework that ”allows the distributed processing of large data sets across clusters of computers using simple programming models” [Ho12]. Following the general application domains and purpose of the Lambda architecture, the master data set grows extensively over time [Ma11]. MapReduce allows the system to compute the batch views even on a large data set [Ho12]. It is composed of three steps (Map, Shuffle, Reduce) using clusters for distributed parallel computations [De08]. MapReduce aims to parallelize the mapping, shuffling and reducing steps in order to significantly improve the time complexity of computations on large data sets [De08]. Notice, that by the time the batch layer views are generated, they are already outdated as a result of the sustained real-time stream processing [Ma11]. This leads us to the speed layer that compensates the high latency of the batch layer and provides the recent data only. For instance, Apache Spark is used to implement this layer in order to reach the required performance [Ha14b]. Spark is an engine particularly developed for large-scale Big Data processing. Spark maintains Apache MapReduce’s linear scalability, and fault tolerance while improving its performance considerably.
Finally, the serving layer stores batch and speed layer views, and subsequently responds to ad-hoc queries by returning the pre-computed views. We distinguish between requests

addressed to views from batch and speed layer, and others that require the use of views from both layers simultaneously. To respond to such requests the serving layer must merge different views (see Figure 1). Generally, the amount of data in the serving layer is relatively small as it only hosts the computed views from batch and speed layer. A common technology for storing batch views is Apache Cassandra views [Ha14a]. Cassandra is a NoSQL database featuring a distributed deployment providing a high level of reliability [Ne13].

3.2 Trade-offs

Using the Lambda architecture has various advantages such as fault-tolerance against hardware failures and human mistakes. It also addresses the problem of computing arbitrary functions on arbitrary data in real-time [Ma15]. However, the software architecture pattern is highly complex and redundant. In oder to apply the Lambda architecture for a specific use case, it has to be tailored correspondingly. Moreover, the different technologies that are needed to run batch, speed and serving layer make it challenging to implement (see Figure 1). Furthermore, keeping both, batch and speed layer synchronized, increases the computational time and effort. In addition, maintaining and supporting both layers is difficult because they are distinct and fully distributed [MJ17].
In summary, the Lambda architecture achieves its goals but comes with high complexity and redundancy. The question arises whether the majority of the use cases require a batch and a speed layer or not. Before we move on to this specific question in our case study, we introduce the Kappa architecture.

4 Kappa Architecture

The Lambda architecture enjoyed comprehensive attention after it was introduced by Marz [Ma11]. Only a few years later, Jay Kreps, a principal staff engineer at LinkedIn, shared his thoughts about the Lambda architecture pointing to its naturally existing disadvantages [Kr14]. Kreps presented another approach for real-time data processing that, in contrast to the Lambda architecture, favors simplicity with respect to development related matters – the Kappa architecture. In this section, we take a closer look at the architecture's components, their functionality and point to several similarities as well as differences compared to the Lambda architecture.

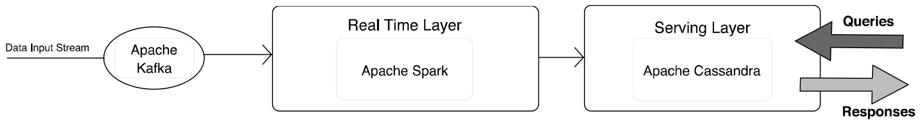

Fig. 2: The Kappa architecture and its workflow through Real-Time and Serving layer

4.1 Overview

The Kappa architecture represents another way of designing a stream processing system. Similarly to section 3, we start off by introducing the architecture's components and elaborate on their functionality. Figure 2 provides the basic outline of the Kappa architecture. Once again, the architecture requires a data stream. First, the incoming data is fed into a stream processing system, sometimes referred to as the real-time layer [Zh17]. This layer is responsible for running the stream processing jobs and providing real-time data processing. Afterwards, the data is directed into the serving layer that queries any required results. Notice that this architecture's components do not particularly differ from the Lambda architecture. Moreover, there is no need to elaborate on any further technologies that are used for implementing this architecture. That is because the Kappa architecture can be implemented by using the same open source technologies previously presented for realizing the Lambda architecture, as illustrated in Figure 2. Next, we take a closer look at the architectures differences.

4.2 Distinguishing Lambda and Kappa

Even though there are numerous similarities, considerable differences arise from the fact that the Kappa architecture passes on a batch processing system. This way, the architecture only requires one code base instead of implementing two heterogeneous systems [OA16]. As a result, development related processes like implementation, debugging and code maintenance are simplified. On the other hand, passing on a batch layer also results in the architecture to be incapable of managing computation intensive applications. This is the case with respect to large scale machine learning scenarios where a model needs to be trained [Zh17]. Furthermore, the performance of batch processing tasks in general suffers from the unavailability of a batch layer [Li17]. However, the Kappa architecture's disadvantages are not particularly problematic as Kreps suggested this approach as an alternative to the Lambda architecture valuing simplicity over efficiency [Kr14]. This means that a direct comparison of both architectures is difficult since the performance of the architecture largely depends on the use case. Therefore, the most appropriate architecture always has to be chosen based on the given application scenario.

5 Case study

In order to provide a comprehensive overview on how both, the Lambda architecture and the Kappa architecture, are implemented, we conduct a case study in the following section. Comparing both architectures, we investigate a stereotypical use case, pointing out advantages and challenges. Based on the technologies presented in subsection 3.1, we develop a *proof-of-concept* implementation. In doing so, we explain the individual technologies in more detail and explain why they are used in the particular case. Ultimately, we try to give an extensive overview of the Lambda architecture, while constantly keeping in mind the approach of the Kappa architecture, investigating structural differences.
As data source, we use Twitter's streaming API, which provides us with comprehensive data about tweets and users. These contain both unstructured data (a tweet's text) as well as structured metadata about the tweet (the tweet's ID, timestamps or included hashtags). Using this data as an input, we are aiming to analyze the hashtags according to their popularity. For the development of our software we use the multi-paradigm programming language *Scala* since it allows smooth integration of the above mentioned tools. Furthermore, thanks to its functional approach, Scala comprises a number of integrated functions allowing it to seamlessly implement technologies such as MapReduce [Up17].

5.1 Providing the data

To access the data of the Twitter streaming API, we use the Twitter4J library. This requires a corresponding registration of the app on Twitter and allows us to access a filtered stream using OAuth authorization. We use a location-based filter that uses minimum and maximum values of longitude and latitude as its range allowing us to access a broad spectrum of all tweets. Thus, we receive a large amount of tweets in very short periods of time, impeding the immediate processing of tweets as they come in. As mentioned earlier, it is reasonable to delegate the buffering of messages to a message broker, as for instance Kafka. This is mainly due to its asynchronous and message-based communication which also implies a complete decoupling of senders and receivers [Du16]. In our example, we utilize the Producer API to write tweets into a queue when they arrive and the Consumer API to read from the queue to populate batch and speed layers. Note that Kafka is usually implemented as a cluster and therefore multiple bootstrap servers can be specified to host this cluster. A cluster node, also referred to as a broker, is responsible to store messages within a specified topic. This topic is unique and can be subscribed by various consumers. In order to allow for parallelism, especially when using Kafka as a distributed cluster, topics are further divided into partitions. In order to take the sequence of incoming messages into account, each message is annotated with a timestamp. This way, messages from different partitions can later be merged together easily [Du16]. For each tweet we consider the set of hashtags. In order to enable a clear identification later on, each message sent to Kafka contains not only the hashtag's text but also the tweet's ID as well as its user's screen name and its timestamp. Since we prefer a serialized yet object-oriented format for message exchange, we convert every Hashtag object into the JSON format. This allows for high compatibility with Cassandra.

5.2 Implementing the batch layer

Now that we have provided a stream of messages, we can start processing them. First, we need to implement a consumer in order to read messages from the Kafka queue. This consumer is primarily responsible for filling the master data set. The idea here is that data is not accidentally changed or deleted, which results in a high degree of consistency [Ma15]. Based on this data, specific views are later calculated to display concrete information (such as the number of hashtag occurrences). In order to achieve a realistic processing speed, we implement the consumer in such a way that it reads multiple messages at a time from the Kafka message queue, writing them to the Cassandra database. This process is scheduled to be executed at a regular interval. The scheduling is done by implementing the consumer as an *Actor*. Actors are a basic concurrency construct in Scala, somewhat comparable to tasks in other programming languages, with the difference that actors can communicate with each other [Up17].
As previously mentioned, we want to use Cassandra as the database of our choice. This enables SQL-like queries that can be created using CQL (*Cassandra Query Language*) [Gu16]. CQL supports all common CRUD operations. As it is possible to assign tables to certain namespaces, called *keyspaces*, we define three different key spaces, for the master data set, the batch view and the realtime view. This allows us to create tables of the same name to enable uniform access to batch and speed views.
As the master data set now gets populated with new hashtags at a regular interval, we can start calculating batch views. Again utilizing a scheduler, we execute a batch job, which iterates through the whole data set, regularly counting occurrences of same hashtags. As the database grows over time, it is inadequate to sequentially count the occurrences as it is done in the speed layer. Instead, it might be reasonable to apply concurrent methods, ideally within a distributed system. One such method, MapReduce, has already been presented in subsection 3.1. After retrieving a list of hashtags from the master data set, we can distribute equally sized chunks of them to several map processes [Gu15]. Each map process then emits a key-value pair, mapping the hashtag as a key to an initial value of 1. Next, while implicitly shuffling same-titled hashtags to dedicated chunks, the reduce function sums up the values of same-titled hashtags. This way we now receive a new list of key-value pairs with a hashtag as the key and the number of that hashtag's occurrences in the data set as the value [Gu15].

5.3 Implementing the speed layer

While the batch layer works on the basis of the immutable master data set, the speed layer receives the stream of new data as an input. Therefore, the results of the speed layer represent only a sample of the total amount of data. Considering the Kappa architecture, the results of

the batch layer can be approximated by interpreting a batch as a limited stream. Here, one does not define a batch as a function on the entire data set, but rather as a function on an arbitrarily large recording of the stream.
In contrast to the batch layer, the retrieved data is not written to a database, but is forwarded directly to a calculation unit. This can be achieved by using Apache Spark, especially by leveraging a data structure called the *Resilient Distributed Dataset* (RDD) [KW17]. This is basically an immutable collection of data records that might reside on multiple nodes in a cluster. Each operation on a RDD requires the construction of a new RDD, memorizing the resulting hierarchy in the *RDD lineage graph*. This allows for fast computation, as data is kept in memory. Further, the concept of a `DStream` represents a continuous flow of RDDs, each representing a fixed windows of data received from the stream. A `ViewHandler` is given a `DStream`, allowing it to continuously executing fast calculations on small data chunks. Figure 3 illustrates the operating principle.
In the ViewHandler we now convert the RDD into a so-called `DataFrame`. This is a kind of wrapper that allows us to execute SQL-style queries on the RDD. The necessary methods and concepts are included in the module SparkSQL. This allows us to apply aggregate functions to the data. In particular, by grouping the hashtags, we can assign them with the number of their occurrences. Note that this is in general executed sequentially. Therefore –and in contrast to the batch layer– it is inapplicable for larger data volumes. As in the batch layer, the resulting hashtag-count pairs must be timestamped, allowing both views to be combined later on.

5.4 Implementing the serving layer

Now that we are able to perform calculations on both, batch and speed layer, we need to consider how to provide the results to the user. While being responsible for providing an easy-to-use interface for queries, the serving layer is also in charge of merging the results from the individual layers. Hence, if you want to have an exact assertion about the number of hashtags at a certain point in time, it is inevitable to compensate the batch layer's calculation latency by merging the results from the speed layer. Considering Table 1, one can see that there are overlaps of hashtags in batch view and real-time view. However, since the results of the speed layer were retrieved shortly after a football match, the hashtags correspond to the football match. For creating an interface, we use Akka to create a http server with a RESTful API, providing ordered JSON lists of hashtags as a result for queries. The

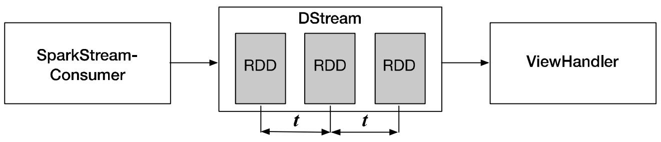

Fig. 3: Principle of the `DStream` in conjunction with a `ViewHandler`

Batch layer results (24 hours)		**Speed layer results (1 hour)**	
hashtag	**count**	**hashtag**	**count**
job	25970	FRAARG	2268
CareerArc	23256	job	1851
Hiring	19835	ARG	1643
YksBirincisiEmreP.	15614	FRA	1602
FRAARG	9813	CareerArc	1550

Tab. 1: Top 5 Hashtags in the batch view and in the realtime view after reading the twitter stream for 24 hours and 1 hour, respectively

merging is performed in Cassandra using tailored CQL queries. In particular, we consider the timestamp of our data when querying the data set. We select the batch view with the latest timestamp, storing the results into a list of hashtag objects. Further, we select all real-time views having a more recent timestamp than the batch view, also storing them into a list. We can then merge both lists in order to retrieve a new list comprising the updated hashtag objects.

6 Discussion & Limitations

While the Lambda architecture allows both high accuracy and fast processing of requests, one does this at the cost of maintaining two separate code bases and hence two complex, distributed systems. This results in some difficulties. On the one hand, both layers must be kept synchronous. If you change a particular view in one layer, the corresponding view must be adapted in the other layer as well. Further, merging in the service layer involves a certain complexity. The data must be structured in a way that efficient merging is possible. Thus, designing the database schemes to be compatible with each other is essential. Moreover, there must be a feature that allows the comparison of the data sets, such as timestamps. In addition, as the master data set grows, more hardware resources are needed in order to compensate the increase of latency while performing batch calculations.
The Kappa architecture, on the other hand, does not integrate a dedicated batch layer at the expense of accuracy. This is based on the assumption that numerous applications do not require the entire data volume, but a sufficiently large segment of the current streaming data. Nevertheless, the number of resources scales with the size of this segment. The more data you want to observe per iteration, the more memory is necessary to process the data at the same time. Figure 4 provides an overview of the architecture we implemented in the case study described in section 5. Although we have followed the approach of the Lambda architecture, the implementation can easily be transfered into a Kappa architecture by removing the corresponding components. In addition, the service layer has to be adjusted as well, since the merging of the two views is omitted. Ultimately, the choice of architecture strongly depends on the respective application and the type of data, necessitating a compromise between consistency, availability and partition tolerance.

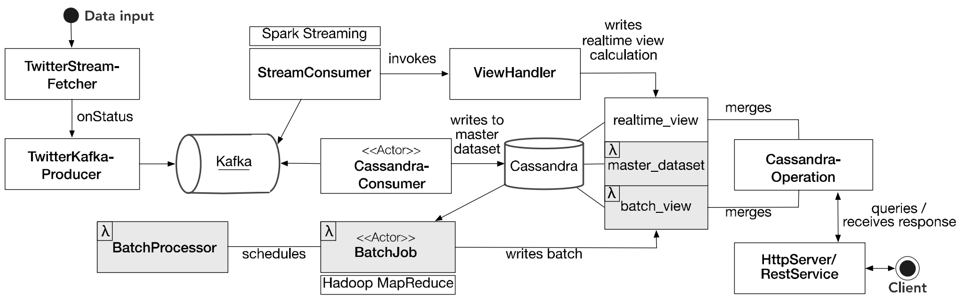

Fig. 4: Overview of the architecture, highlighting components only used in lambda architecture.

7 Conclusion & Future Work

In this paper, we have investigated the state-of-the-art real-time data processing architectures Lambda and Kappa. We started off by taking a closer look at each architecture's components, workflow and theoretical capabilities. While the Lambda architecture was capable of raising awareness about how challenging the development of a shared data system can be, its high complexity remains a considerable disadvantage. Even though the Kappa architecture improves this aspect, the architecture can only be applied for specific use case and might suffer from performance issues. We gave a brief introduction on most commonly known technologies for implementing each architecture's layers as well as the concept of MapReduce. Furthermore, we have discussed the architectures most significant differences that need to be considered when developing a shared data system. After our theoretical investigation, we used Twitter's streaming API for conducting a case study that allows us to gain more detailed insights regarding each architecture's strengths and weaknesses.
We discussed that measuring an architecture's performance with respect to a given use case might not provide particularly sensible information as the result depends on numerous factors. Consequently, future work should focus on prodiving an analysis regarding these factors for allowing an easier decision-making regarding the choice of an architecture.

8 Acknowledgments

We gladly thank Prof. Dr. Markus Esch for his continuous support during the project.

References

[Ab10] Abouzied, Azza et al.: HadoopDB in Action: Building Real World Applications. In: SIGMOD International Conference on Management of data. pp. 1111–1114, 2010.

[Br00] Brewer, Eric A.: Towards Robust Distributed Systems. PODC '00, ACM, New York, NY, USA, pp. 7–, 2000.

[De08] Dean, Jeffrey & Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113, 2008.

[Du16] Dunning, T. & Friedman, E.: Streaming Architecture: New Designs Using Apache Kafka and MapR Streams. O'Reilly Media, 2016.

[Ga15] Gandomi, Amir & Haider, Murtaza: Beyond the hype: Big data concepts, methods, and analytics. International Journal of Information Management, 35(2):137–144, 2015.

[Gu15] Gunarathne, T.: Hadoop MapReduce v2 Cookbook - Second Edition. Community Experience Distilled. Packt Publishing, 2015.

[Gu16] Gudipati, Pramod Kumar: Implementing a lambda architecture to perform real-time updates. Master's thesis, Department of Computing and Information Science, Kansas State University, Manhattan, Kansas, 2016.

[Ha14a] Hasani, Zirije et al.: Lambda architecture for real time big data analytic. ICT Innovations, 2014.

[Ha14b] Hasani, Zirije et al.: Survey of technologies for real time big data streams analytic. In: Informatics and Information Technologies. pp. 11–13, 2014.

[Ho12] Holmes, Alex: Hadoop in practice. Manning Publications Co., 2012.

[Ki15] Kiran, Mariam et al.: Lambda architecture for cost-effective batch and speed big data processing. In: Big Data. IEEE, pp. 2785–2792, 2015.

[Kr14] Kreps, Jay: Questioning the Lambda Architecture. 2014.

[KW17] Karau, H.; Warren, R.: High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark. O'Reilly Media, 2017.

[Li14] Liu, Xiufeng et al.: Survey of real-time processing systems for big data. In: IDEAS. 2014.

[Li17] Lin, Jimmy: The Lambda and the Kappa. IEEE Internet Computing, 21(5):60–66, 2017.

[Ma11] Marz, Nathan: How to beat the CAP theorem. Thoughts from the Red Planet, 2011.

[Ma15] Marz, Nathan & Warren, James: Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications Co., Greenwich, CT, USA, 2015.

[MJ17] Misra, Pankaj; John, Tomcy: Data Lake for Enterprises. Packt Publishing, 2017.

[Ne13] Neeraj, N.: Mastering Apache Cassandra. Community experience distilled. Packt Publishing, 2013.

[OA16] Ordonez-Ante, Leandro et al.: Interactive querying and data visualization for abuse detection in social network sites. In: Internet Technology and Secured Transactions. IEEE, 2016.

[Up17] Upadhyaya, B.P.: Programming with Scala: Language Exploration. Undergraduate Topics in Computer Science. Springer International Publishing, 2017.

[Zh17] Zhelev, Svetoslav & Rozeva, Anna: Big data processing in the cloud-Challenges and platforms. In: AIP Conference Proceedings. AIP Publishing, 2017.