Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Vergleich und Analyse geläufiger CEP Systeme

Jonathan Göggel

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. K. Rothermel
Betreuer/in:	M. Sc. Henriette Röger
Beginn am:	3. Juli 2017
Beendet am:	3. Januar 2018
CR-Nummer:	C.2.4 D.2.2 D.2.5 H.2.4

Kurzfassung

Heutzutage werden komplexe Anfragen in Echtzeit auf großen Datenmengen ausgeführt. Immer mehr Daten fallen an und das Interesse diese in Echtzeit zu analysieren steigt. Die Performance eines Systems ist ein enorm wichtiger Faktor. Momentan setzen besonders große Firmen wie Google, Amazon und Netflix CEP-Systeme ein, um effizient Nutzerdaten zu analysieren und dem Anwender daraufhin Empfehlungen vorzuschlagen. Die aktuell verfügbaren CEP-Frameworks verhalten sich jeweils unterschiedlich und haben unterschiedliche Ziele.

Bisherige Auswertungen fokussieren sich nur auf jeweils ein Framework und optimieren dieses. In meiner Arbeit werden verschiedene Frameworks gegenübergestellt und untersucht, wie flexibel sie angesteuert werden können und inwieweit sie zur Laufzeit detaillierte statistische Werte liefern können. Des Weiteren wird eine API entworfen, die ermöglicht verschiedene CEP Frameworks anzusprechen und somit standardisiert den Parallelisierungsgrad und somit die Performance eines CEP-Systems zu verbessern. Durch die Standardisierung ist es auch möglich die Performance bei CEP-Systemen mit mehrere CEP-Frameworks zu regeln.

Im ersten Teil der Ausarbeitung werden verschiedene Frameworks verglichen und untersucht inwieweit sich diese für eine zentrale Ansteuerung eignen. Im zweiten Teil wird ein Interface definiert und zum Evaluieren beispielhaft ein Adapter für ein CEP-Framework erstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das effiziente Verarbeiten von Datenströmen ist heutzutage enorm wichtig. Die darin enthaltenen Daten werden beispielsweise durch Nutzertransaktionen auf Webseiten oder durch Messwerte von Sensoren kontinuierlich erzeugt. Anbieter wie Amazon, Twitter, Google und Netflix arbeiten daran, die Daten in Echtzeit zu analysieren, um Erkenntnisse über das Nutzerverhalten zu gewinnen.

Zum Verarbeiten von Datenströmen wird Complex Event Processing (CEP) verwendet.

CEP verhält sich hierbei wie eine „umgekehrte“ Datenbank. Bei einer Datenbank sind die Daten fest gespeichert und die Anfragen werden bei Bedarf getätigt und beantwortet. Bei CEP sind hingegen die Anfragen fest in der Topologie und den Operatoren festgelegt. Die Daten halten sich jedoch nur kurz im System auf.

Da sich die Ansprüche an ein CEP-System teilweise stark unterscheiden, haben sich viele verschiedene CEP-Frameworks herausgebildet. Die Frameworks unterscheiden sich zum Beispiel in der Effizienz, der Einfachheit, der Konfiguration und der Kompatibilität zu anderer Software. Teilweise unterscheiden sie sich auch in der Mächtigkeit der Operatoren.

Die unterschiedlichen Frameworks nutzen keine standardisierte Darstellung für Ereignisse und auch keine für die Konfiguration der Operatoren beziehungsweise der Topologie. Die unterschiedlichen Frameworks haben jedoch gemeinsame Probleme. Eines ist hierbei eine effiziente Zuteilung der Ressourcen für die einzelnen Operatoren. Ein wichtiger Faktor ist hierbei die Latenz, da die Echtzeiteigenschaft von CEP-Systemen in vielen Bereichen benötigt wird. Da dieses Problem für alle Frameworks gilt soll auch eine Lösung erstellt werden, welche frameworkübergreifend angewendet werden kann.

Um dies zu realisieren müssen die verschiedenen Frameworks auf eine einheitliche Weise angesprochen werden. Dafür werden Informationen über die Latenz der einzelnen Operatoren eines CEP-Systems bestimmt und aufgrund dieser Informationen Modifikationen getroffen um die Latenz zu verbessern. Dies erfolgt meistens durch das Erhöhen der Anzahl an Instanzen eines Operators. Das Bestimmen der Änderung des Parallelisierungsgrads in Abhängigkeit der Latenzinformationen wird von einem zentrales System umgesetzt, welches momentan am Institut entwickelt wird. Die vorliegende Arbeit beschäftigt sich mit dem Bestimmen der Latenzinformationen aus verschiedenen Frameworks und der Möglichkeit den Parallelisierungsgrad eines Operators zu ändern.

Als Schnittstelle zum System des Institutes wird eine API definiert, welche dann standardisiert auf die verschiedenen Frameworks zugreifen kann. Für die verschiedenen Frameworks muss daraufhin je ein Adapter entwickelt werden, welcher die Funktionen der API auf dem gewünschten CEP-Framework ausführt.

In Abbildung 1.1 ist ein beispielhafter Aufbau eines solchen CEP-Systems mit externer Steuerung abgebildet. Das CEP-System besteht aus den beiden Frameworks Heron und Parse. Auf der linken

Seite ist die Quelle A zu sehen, an welcher die Daten das CEP-System betreten. Daraufhin werden die Operatoren nacheinander ausgeführt. Am Ende verlässt der Datenstrom das CEP-System an der Senke B.

Die einzelnen Operatoren teilen jeweils im Splitter die Ereignisse des Datenstroms an die einzelnen Instanzen auf. Die Ergebnisse der Operatorinstanzen werden am Merger wieder zusammengeführt. Durch den Grad der Parallelisierung der Instanzen der Operatoren wird der Durchsatz des Operators gesteuert.

Die Metrikinformationen der einzelnen Operatoren werden im Controller gesammelt. Dieser kann auch den Parallelisierungsgrad der Operatoren bestimmen. Der Client bietet eine Schnittstelle, um darauf zuzugreifen.

Durch einen solchen übergreifenden Ansatz ist es auch möglich CEP-Systeme zu steuern, die unterschiedliche CEP-Frameworks zusammen verwenden.

Ziel der Steuerung ist es, mit begrenzten Ressourcen auf einem Servercluster die Ressourcen so auf die unterschiedlichen Operatoren aufzuteilen, dass ein optimaler Durchsatz bzw. eine möglichst geringe Latenz erreicht wird. Das bedeutet, dass kein Operator Ressourcen ungenutzt lässt, jedoch alle Ressourcen verteilt werden.

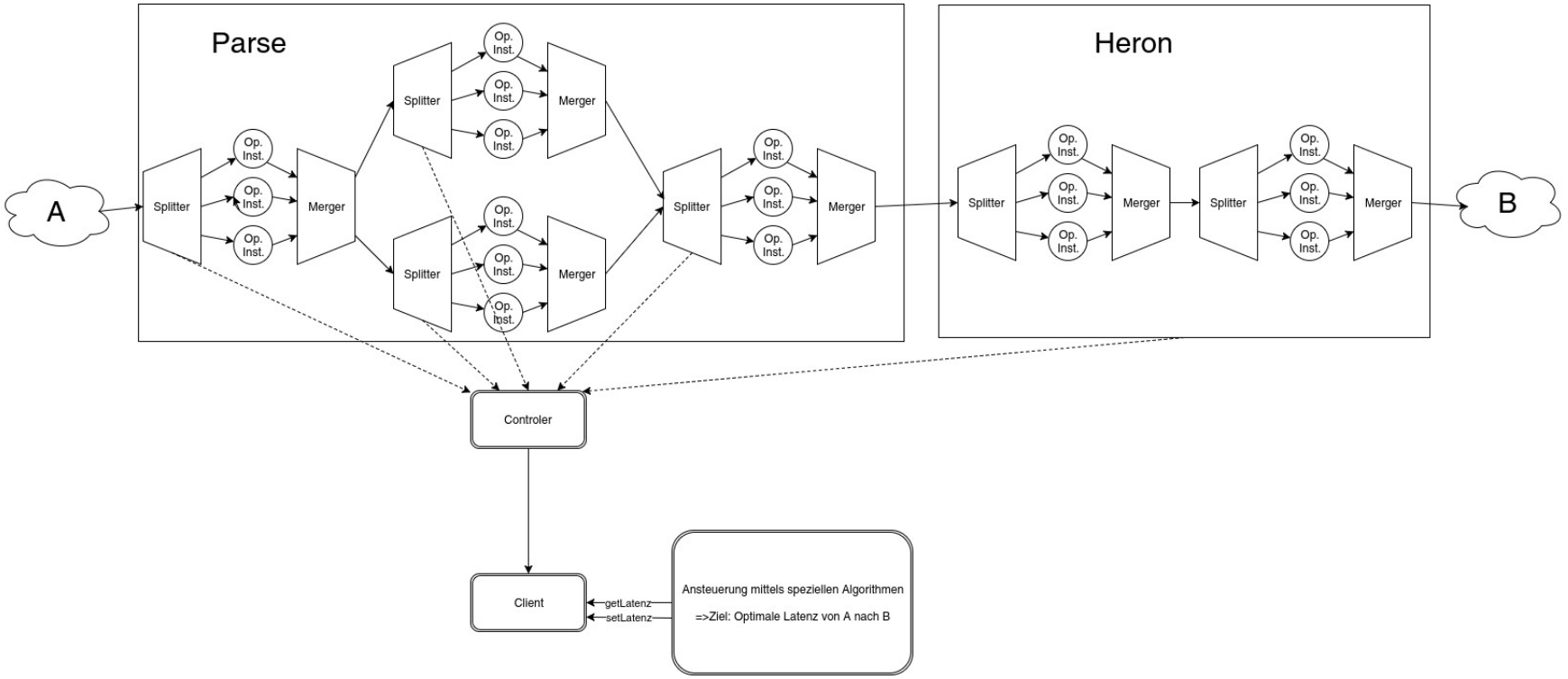

Abbildung 1.1: Konzept

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 1 – Einleitung: Gibt einen Einblick und führt das Thema der Arbeit ein

Kapitel 2 – Definitionen: Definiert die wichtigsten Begriffe für das Verständnis dieser Arbeit

Kapitel 3 – Bestehende Arbeiten: Zeigt die vorhandenen Arbeiten im Bereich auf

Kapitel 4 – Die verschiedenen CEP-Frameworks: Vergleicht die bekanntesten Frameworks in Bezug auf das zu erstellende Interface

Kapitel 5 – Latenz: Beschreibt an welchen Stellen und auf welche Weise die Latenz bestimmt werden kann

Kapitel 6 – Interface: Definiert die Anforderungen für das Interface, definiert dieses und beschreibt die Erstellung von Adaptern für das Interface

Kapitel 7 – Beispielimplementierung: Beschreibt eine Beispielimplementierung eines CEP-Systems

Kapitel 8 – Verwendung des Interfaces und Evaluation: Evaluiert das Interface inklusive der Adapter

Kapitel 9 – Zusammenfassung und Ausblick: Fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor

2 Definitionen

Im Folgenden werden die wichtigsten Begriffe zum Verständnis dieser Arbeit und von CEP-Systemen erläutert.

2.1 Datenstrom

Ein Datenstrom (engl. data stream) ist eine potenziell unendliche Folge an Daten (siehe Abbildung 2.1). Im Falle von CEP sind die Daten einzelne Ereignisse. Ein Datenstrom wird linear abgearbeitet.

Beispiel: Ein Temperatursensor sendet alle zwei Sekunden die aktuelle Temperatur.

Die Alternative zu einem Datenstrom ist die Stapelverarbeitung (Batch). Dabei sind die Daten auf einem Speicher vorhanden und werden gemeinsam verarbeitet. Latenz ist hierbei nicht entscheidend.

Eine Zwischenvariante ist Microbatch. Dabei werden viele sehr kleine Batchbereiche als Datenstrom verarbeitet. Somit kann eine geringe Latenz gewährleistet werden. Die Daten werden jedoch jeweils durch die Stapelverarbeitung verarbeitet.

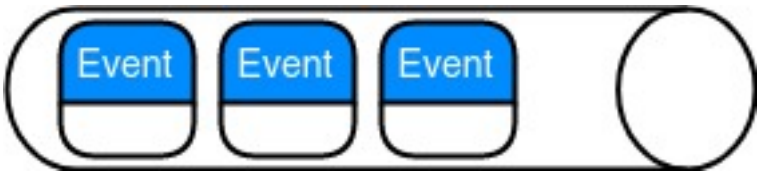

Abbildung 2.1: Datenstrom mit Ereignissen

2.2 Ereignis

Ein Ereignis (engl. event) beinhaltet verschiedene Informationen: meistens einen den Zeitpunkt, zu dem es entstand, und zum anderen weitere Informationen wie zum Beispiel Identifikation, Messwert eines Temperatursensors oder ein Einzelbild eines Videostreams. Oft besitzen Ereignisse auch einen Typ, der beschreibt welche Informationen das Ereignis beinhaltet.

Die Informationen von einfachen Ereignissen, die beispielsweise direkt aus Messwerten entstehen, haben für sich alleine betrachtet meist einen sehr geringen Informationsgehalt.

2.3 Komplexe Ereignisse

Ein komplexes Ereignis ist ein spezielles Ereignis, welches aus einfacheren Ereignissen erzeugt wird. Aus komplexen Ereignissen können weitere komplexe Ereignisse erzeugt werden. Dabei wird meist der aktuelle Kontext berücksichtigt. Dies sind zum Beispiel die Häufigkeit innerhalb eines Zeitraums oder zuvor auftretende Ereignisse. Die enthaltenen Informationen sind aussagekräftiger als die ursprünglichen Ereignisse.

Beispielsweise kann ein Feuer erkannt werden wenn innerhalb von 10 Sekunden drei verschiedene Sensoren eine Temperatur über 60° Celsius liefern. In Abbildung 2.2 sind oben die drei ursprünglichen Ereignisse zu sehen und unten das daraus resultierende komplexe Ereignis.

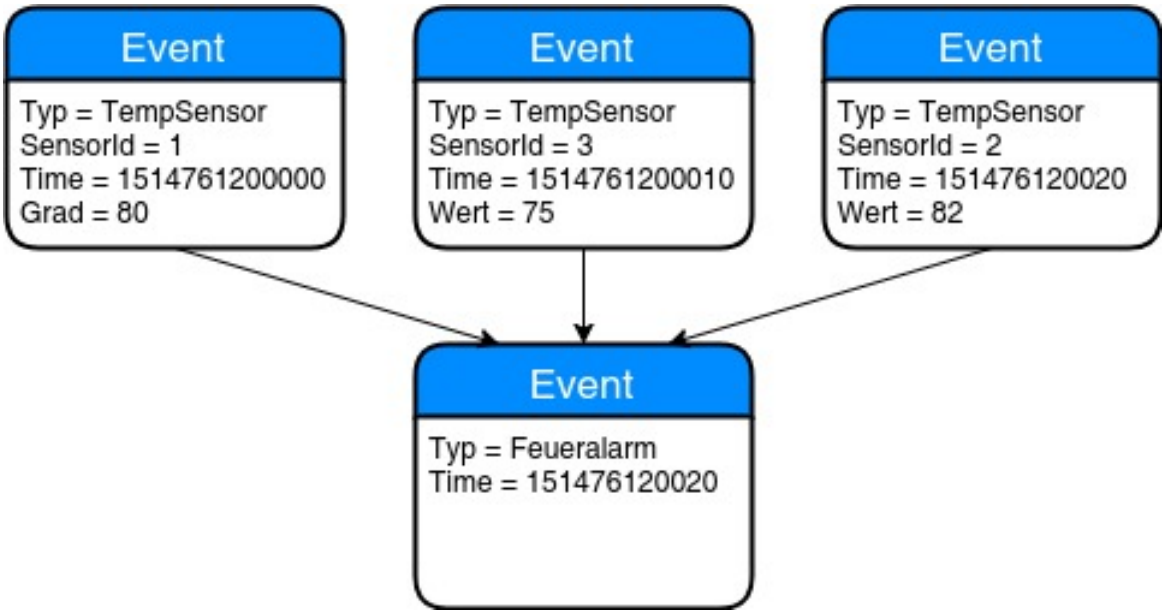

Abbildung 2.2: Komplexes Ereignis

2.4 Complex Event Processing (CEP)

Complex Event Processing (CEP) analysiert und wertet einfache Ereignisse aus um somit komplexe Ereignisse zu erzeugen. CEP ist der Vorgang, bei dem aus (sehr vielen) einfachen Ereignissen (relativ wenige aber aussagefähige) komplexe Ereignisse erzeugt werden.

Dabei werden zum einen für die weitere Analyse irrelevante Ereignisse herausgefiltert und zum anderen die relevanten Daten kombiniert und interpretiert.

2.5 Operator

Ein Operator definiert, wie mit bestimmten Ereignissen umgegangen werden soll. Ein Operator verarbeitet erhaltene Ereignisse und erzeugt daraus neue komplexere Ereignisse und gibt diese Ereignisse weiter.

Dabei werden aus vielen einfachen Ereignissen wenige oder gar keine komplexe Ereignisse erzeugt.

Ein Operator kann zum Beispiel das Minimum aus den Messwerten der letzten 30 Sekunden bestimmen, oder nur dann ein Ereignis an die Quelle senden falls in den letzten 30 Sekunden mehr als 3 Messwerte einen bestimmten Schwellwert überschritten haben.

2.6 CEP-System

Ein CEP-System führt Operatoren auf einem Datenstrom von Ereignissen aus. Der Datenstrom kommt an der Quelle in das CEP-System. Innerhalb des CEP-Systems können mehrere Operatoren ausgeführt werden. Dazu gehören beispielsweise das Verwerfen von nicht benötigten Ereignissen und das Erstellen von komplexeren Ereignissen.

Aufgrund der neu erstellten komplexen Ereignisse können Operatoren am Ende weitere Aktionen außerhalb des CEP-Systems auslösen und den Anwender über den aktuellen Zustand oder über Gefahren informieren. Es ist auch möglich einen Datenstrom an der Senke auszugeben, damit weitere CEP-Systeme die Ereignisse weiterverarbeiten können.

Zur Implementierung von CEP-Systemen existieren viele unterschiedliche Frameworks.

2.7 Topologie

Damit ein CEP-System mehrere Operatoren ausführen kann müssen diese miteinander verbunden werden. Dabei ist die Senke des einen Operators die Quelle für den anderen. Dadurch entsteht ein gerichteter, normalerweise azyklischer Graph.

Dieser Graph wird als Topologie bezeichnet. Die einzelnen Operatoren werden oft als verteiltes System auf verschiedenen Rechnern im Netzwerk ausgeführt.

2.8 Parallelisierung

Um den Durchsatz eines CEP-Systems zu steigern beziehungsweise die Latenz zu senken, kann ein Operator parallelisiert werden. Dabei werden von einem Splitter die Ereignisse an mehrere Instanzen des Operators verteilt. Nachdem die Operatoren ausgeführt wurden, werden die Datenströme der verschiedenen Instanzen durch einen Merger wieder kombiniert.

Der Splitter und der Merger sind aus logischer Sicht Teil des Operators. Physisch muss zwischen Splitter, Instanz und Merger unterschieden werden.

2.9 Fenster

Da der Eingabedatenstrom kein Ende besitzt muss der Bereich auf denen die Operationen ausgeführt werden begrenzt werden. Dafür werden Fenster (engl. windows) genutzt. Fenster enthalten eine gewisse Anzahl von Ereignissen. Diese können beispielsweise über eine feste Anzahl von Ereignissen, oder über einen festen Zeitbereich festgelegt werden. Ein Operator verarbeitet alle in einem Fenster zusammengefassten Ereignisse. Ein konkretes Ereignis ist normalerweise in mehreren Fenstern enthalten. Die Fenster gleiten quasi durch den Datenstrom, wobei die Geschwindigkeit festgelegt werden kann.

Im Beispiel in Abbildung 2.3 liefert ein Datenstrom Ereignisse mit aufsteigenden Zahlen. Die Fenstergröße beträgt „3“ und das Fenster gleitet immer um „2“ Ereignisse. Dann sind im ersten Fenster die Ereignisse „1,2,3“, im zweiten Fenster die Ereignisse „3,4,5“, im dritten Fenster die Ereignisse „5,6,7“ und so weiter.

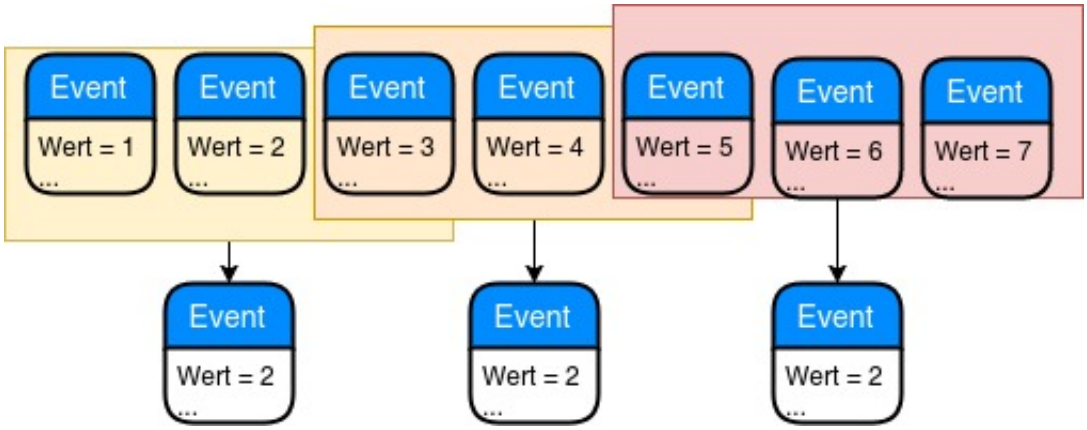

Abbildung 2.3: Fenster

3 Bestehende Arbeiten

Im Umfeld der CEP-Systeme gibt es eine Vielzahl von Veröffentlichungen, die sich mit den unterschiedlichsten Aspekten beschäftigen. Für diese Bachelorarbeit werden einige näher untersucht. Zuerst werden Veröffentlichungen beziehungsweise Arbeiten untersucht, deren Schwerpunkt auf dem Vergleich verschiedener CEP-Frameworks liegt. Danach werden Veröffentlichungen beziehungsweise Arbeiten vorgestellt, welche sich mit der Ansteuerung von Frameworks beschäftigten.

3.1 Vergleiche von CEP Frameworks

Um geeignete CEP-Frameworks für die weitere Analyse zu finden, werden hier zuerst Arbeiten untersucht, die verschiedene Streaming-Frameworks miteinander vergleichen.

3.1.1 A Performance Comparison of Open-Source Stream Processing Platforms

In diesem Paper [LLD16] werden die CEP-Frameworks Storm, Flink, und Spark Streaming miteinander verglichen und schwerpunktmäßig die Unterschiede in der Performance untersucht.

Außerdem werden die Unterschiede zwischen Streaming, Stapelverarbeitung und Microbatchverarbeitung untersucht und auf die Fehlertoleranz der einzelnen Frameworks eingegangen.

3.1.2 Processing Flows of Information: From Data Stream to Complex Event Processing

Dieses Paper [CM12] enthält eine grundlegende Einführung in das Konzept von Datenströmen und CEP-Systemen. Es werden über 30 verschiedene Streaming-Frameworks miteinander verglichen. Der Schwerpunkt liegt insbesondere auf den unterschiedlichen Datenformaten der Ereignisse und auf der Mächtigkeit der Operatoren.

Das Paper ist aus dem Jahre 2012; deshalb sind die neueren CEP-Frameworks leider nicht enthalten.

3.1.3 Elastic Stream Processing in the Cloud

In diesem Paper [HSD13] geht es um die dynamische Resourcenzuteilung in Cloudsystemen. Dazu sind für jedes CEP-System bzw. jede Cloud-Konfiguration jeweils eigene Anpassungen nötig. Dabei müssen zum Teil auch die Algorithmen der CEP-Systeme für die konkrete Cloudlösung optimiert werden.

Anfangs wird der Aufbau einer Streamverarbeitung aufgezeigt und dabei die verschiedenen Arten von Fenstern erklärt. Außerdem werden verschiedene Datenstromverarbeitungsprogramme verglichen. Dabei wird auf die unterschiedlichen Arten der Realisierung von Operatoren eingegangen. Die Vorteile und der Fokus der einzelnen CEP-Frameworks werden ausführlich aufgeführt.

Es wird ein Konzept entwickelt, bei dem die Ereignisse priorisiert werden können und bei dem unnötige Ereignisse verworfen werden können.

Die Untersuchung der Performance konzentriert sich auf die Operatoren, die Datenraten der Cloud, die Domain-Struktur, die Topologie und auf die Queries.

Wichtige Aspekte beim Aufteilen des Datenstroms in der Cloud sind das lokale Ausführen einiger Aktionen um die Latenz zu senken und das Behalten des Kontextes für komplexere Anfragen.

3.2 Ansteuerung von CEP Frameworks

Um einen Adapter für das Framework zu entwickeln, muss das Framework die Möglichkeit der Regulation bieten. Außerdem müssen auch Metriken für das CEP-System erfasst werden. Deshalb wird im Folgenden betrachtet, wie diese Aufgaben in anderen Arbeiten gelöst wurden.

3.2.1 DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams

Da die anfallenden Datenmengen meist nicht zuverlässig vorhergesagt werden können, ist eine dynamische Ressourcenzuweisung enorm wichtig [FDM+15]. Für das Messen der Operatoren (zum Beispiel Latenz) wird ein pullbasiertes Verfahren verwendet. Die einzelnen Operatoren messen ihre eigene Latenz. Ein zentraler Mess-Operator sammelt diese Informationen (pull) und bereitet diese auf. Zur Aufbereitung der Messdaten werden diese aggregiert und geglättet. Die Aggregation fasst hierbei die verschiedenen Instanzen eines Operators zusammen. Das Glätten reduziert ein Rauschen, welches zu unerwünschten Reaktionen führen könnte. Um den Overhead zu senken muss nicht bei jedem Ereignis gemessen werden. Die Implementierung dieses Entwurfes wurde mit Storm durchgeführt.

3.2.2 A preventive auto-parallelization approach for elastic stream processing

[KLL17] Der Schwerpunkt dieses Papers liegt darin, die zukünftige Last vorher zu sagen. Dazu werden Metriken entwickelt, anhand derer der Parallelisierungsgrad der Operatoren angepasst werden kann.

Des Weiteren wird der genaue Zeitpunkt zum Anpassen dieses Parallelisierungsgrads untersucht um die Ressourcen möglichst effizient zu nutzen.

Zur Umsetzung wird Storm verwendet.

3.2.3 Dhalion: Self-Regulating Stream Processing in Heron

[FAG+17] Dhalion ist ein System, das auf Heron aufbaut und den Durchsatz verbessern soll. Besonderer Augenmerk wird auf die Interpretation der Messwerte und die darauf folgende Reaktion gelegt. Dabei soll sich Dhalion selbst regulieren, damit größere Schwankungen beziehungsweise ein Aufschaukeln von Problemen verhindert werden. Auch sollen eventuell auftretende Probleme automatisch behoben werden.

4 Die verschiedenen CEP-Frameworks

Dieses Kapitel beschäftigt sich mit den in der Wissenschaft am meisten verbreiteten CEP-Frameworks und vergleicht diese. Außerdem wird darauf eingegangen inwieweit die CEP-Systeme von außen kontrolliert, gesteuert und überwacht werden können.

4.1 Übersicht

Im Folgenden werden die in der Wissenschaft verbreitetsten Streaming-Frameworks vorgestellt und deren Fokus beschrieben.

4.1.1 Apache Storm

Die Ziele von Storm [TTS+14] sind Skalierbarkeit, Belastbarkeit, Erweiterbarkeit, Effizienz und einfache Administration. Die CEP-Regeln werden durch eine Topologie festgelegt. Dies ist ein gerichteter, azyklischer Graph, der aus Spouts (Quelle) und Bolts (Operatoren) besteht. Spouts liefern neue Daten in die Topologie. Bolts verarbeiten eintreffende Daten weiter. Die Spouts und Bolts werden in einzelnen Workerprozessen ausgeführt. Diese laufen auf Workernodes, welche von Zookeeper verwaltet und gestartet werden. Durch einen Supervisor je Knoten besitzt das System eine hohe Ausfallsicherheit. Der Supervisor besteht aus dem Mainthread und einem Eventmanagerthread. Der Mainthread sendet einen Heartbeat um anzuzeigen, dass er noch ausgeführt wird. Der Eventmanagerthread ist für Anpassungen bei einer Änderung der Topologie zuständig. Der Processeventmanagerthread überwacht den Heartbeat und kann bei einem Absturz den Mainthread neu starten. Änderungen der Topologie können über Nimbus übernommen werden. Nimbus besitzt jedoch keine Ausfallsicherheit. Wenn Nimbus abstürzt, dann läuft das System weiter; es ist allerdings nicht mehr möglich die Topologie zu ändern.

4.1.2 Twitter Heron

Twitter Heron [KBF+15] wurde entwickelt da Storm in der Größe von Twitter nicht mehr gut genug skaliert. Twitter möchte in Echtzeit die Nutzeraktivitäten verfolgen und darauf reagieren. Dies ist beispielsweise das Liefern der passenden Werbung. Deshalb wurden Skalierbarkeit, Performance und die Möglichkeiten des Debuggens verbessert.

Heron ist API-kompatibel zu Storm. Somit können für Storm geschriebene Anwendungen auch mit Heron ausgeführt werden. Aus Optimierungszwecken wird allen Workern gleich viel RAM zugewiesen. Des Weiteren wurde Nimbus durch Aurora ersetzt. Da Zookeeper bei großen Anwendungen der Flaschenhals war, wurden die Heartbeats als Dienst ausgelagert.

Um ausführlichere Informationen über das System zu erhalten bietet Heron einen Metrics Manager an, der viele zusätzliche Abfragen über das System ermöglicht. Somit können Messwerte wie Latenz, Fehler, CPU-Auslastung und Arbeitsspeicherauslastung für das gesamte System und auch für die einzelnen Instanzen bestimmt werden.

4.1.3 Parse

Das Framework Parse [MKR15] [MTR] ist ein in Java geschriebenen CEP-Framework. Die Operatoren werden hierbei direkt in Java implementiert. Es bietet flexible Möglichkeiten neue Fenster zu erstellen. Beispielsweise kann dies durch Steuerereignisse geschehen. Des Weiteren kann der Parallelisierungsgrad der Operatoren mittels der "Queuing Theory"(QT) automatisch je nach Auslastung angepasst werden.

4.1.4 Apache Spark

Apache Spark [Apache Spark] [ZS17] [LLD16] entstand in einem Forschungsprojekt der University of California in Berkeley. Es ist eine allgemeine Engine zum effizienten Verarbeiten großer Datenmengen. Es besteht aus den Bereichen SQL und DataFrames, Streaming, Machine learning und GraphX. Dadurch können komplexe Analysen mit SQL und Streams verbunden werden.

Spark Streaming ist fehlertolerant und stellt bei Abstürzen den alten Stand wieder her. Dies ist wichtig wenn Operatoren einen Zustand besitzen. Durch das Wiederherstellen dieses Zustands kann somit die Wiederherstellungszeit des Systems verringert werden.

Spark Streaming wird mittels micro-batching realisiert. Die Batchgröße liegt bei 0.5 Sekunden. Durch die exactly-once Semantik wird auch bei Performanceproblemen immer das richtige Ergebnis geliefert.

4.1.5 Apache Flink

Apache Flink [CKE+15] [ZS17] [LLD16] realisiert Datenstrom- und Stapelverarbeitung in einer gemeinsamen Engine. Dazu existiert eine allgemeine Streamingruntime, auf der sowohl eine DataSet API mit batchoptimierten Funktionen als auch eine DataStream API mit streamoptimierten Funktionen läuft.

Die Batchverarbeitung wird als eine Spezialform des Streamings interpretiert. Ein Batch ist dabei genau ein Fenster und hat somit genau einen Anfang und ein Ende.

Um fehlertolerant zu sein wird das Prinzip des ABS (Asynchronous Barrier Snapshotting) genutzt. Um einen Snapshot zu erstellen gibt es einen speziellen Typ von Ereignis, der in den Inputstream eingefügt werden kann.

Falls nun beispielsweise ein Knoten ausfällt, kann anhand der Snapshots der Zustand der Operatoren rekonstruiert werden. Somit gehen keine Ereignisse verloren.

Flink nutzt ebenso wie Storm, einen Heartbeatmechanismus um Ausfälle von Knoten zu erkennen.

4.1.6 Esper

Esper [CM12] ist ein CEP-Framework aus dem Jahr 2006. Die Topologie und die Operatoren werden durch die Event Processing Language (EPL) konfiguriert. Dies ist eine SQL-ähnliche Sprache. Dabei werden beispielsweise die Schlüsselwörter „SELECT“, „WHERE“ und „FROM“ wie von SQL gewohnt verwendet. Zusätzlich existieren weitere Möglichkeiten, beispielsweise zum Festlegen der Fenster.

Bei der Gratisversion erfolgt die gesamte Ausführung auf einem Knoten. Mit dem kostenpflichtigen EsperHA ist auch die Verwendung eines Clusters möglich.

4.1.7 MillWheel

MillWheel [ABB+13] ist ein CEP-Framework, welches von Google benutzt wird. Es bietet eine sehr geringe Latenz. Des Weiteren ist auch die Fehlertoleranz eine Stärke von MillWheel. Das bedeutet, dass beim Ausfall eines Knotens oder einer Kante, das System weiter laufen kann. Das sind beispielsweise Ausfälle eines Rechners mit allen darauf laufenden Instanzen oder auch Ausfälle des Netzwerks.

Der aktuelle Zustand wird jederzeit durch feingranulare Checkpoints festgehalten. Dabei wird jedes gesendete Ereignis bestätigt. Ein Ereignis besteht aus einem Tripel: Zeitstempel, Schlüssel und Wert; dabei kann der Wert auch komplexe Datenstrukturen enthalten.

Eine weitere Eigenschaft von MillWheel sind Qualitätszusicherungen. Beispielsweise sind die Daten jederzeit konsistent und die Reihenfolge der Ereignisse wird immer eingehalten.

4.1.8 Amazon Kinesis

Amazon Kinesis [Amazon Kinesis] ist eine Cloud-Infrastruktur zum Verwalten von Datenströmen in Echtzeit. Es besteht aus den Bereichen Kinesis Firehose und Kinesis Analytics. Firehose lädt Streams in andere Dienste von Amazon, beispielsweise den Amazons Speicher S3 oder Kinesis Analytics. Mit Kinesis Analytics können komplexe Anfragen an einen Datenstrom mittels SQL-ähnlichen Abfragen erstellt werden. Durch die Integration in AWS (Amazon Web Service) können weitere Dienste von AWS einfach eingebunden werden. Es ist jedoch auch möglich, die Analyse der Streams mit anderen Frameworks zu erweitern, beispielsweise mit Apache Spark. Des Weiteren ist Kinesis extrem skalierbar und kann die verwendeten Ressourcen bei Bedarf automatisch anpassen.

4.1.9 Apache Gearpump

Apache Gearpump [Apache Gearpump] ist noch ein sehr neues Projekt, das momentan in der Version 0.8 vorliegt. Es wurde stark von Akka [Akka] geprägt und möchte vorhandene Streamingframeworks verbessern. Es ist beispielsweise mit Storm-Anwendungen kompatibel.

Die Operatoren können zur Laufzeit als Graph konfiguriert werden. Zukünftig soll auch eine SQL-ähnliche API existieren.

4.2 Gegenüberstellung bekannter Streaming-Frameworks

Im Folgenden werden die zuvor vorgestellten Frameworks direkt miteinander verglichen. Untersucht werden die innere Struktur der Ereignisse, die Zusammenfassung der Ereignisse in Fenster und die Fehlertoleranz bei Systemausfällen. Anschließend werden für die weitere Analyse geeignete Frameworks bestimmt.

Tabelle 4.1 zeigt eine Übersicht über die verschiedenen CEP-Frameworks.

Tabelle 4.1: Gegenüberstellung geläufiger CEP-Frameworks

	Storm	**Heron**	**Parse**	**Flink**	**Esper**
Aufbau Topologie	bsw. Java	bsw. Java	nur ein Operator (Java)	bsw. Java	EPL
Ereignis Typ	Tupel	Tupel (...api.tuple)	ererbte Klasse von „Event“	Tupel	Klasse, Bean, XML
Fenster	Zeit, Anzahl	Zeit, Anzahl	Zeit, Anzahl, Weitere	Zeit, Anzahl	Zeit, Anzahl, Weitere
Fault-tolerant	Ja, Heartbeat	Ja, Heartbeat	Nein	ABS, Heartbeat	in EsperHA
Metriken	Metrics Consumer	Metrics Manager	Processing-Latency-Statistics-Monitor		-
Skalierbarkeit zur Laufzeit	Ja,Topologie ändern	Ja,Topologie ändern	Ja, Instanz hinzufügen		-
Eigenheit		effizient für große Systeme, basiert auf Storm	Flexible Abfragen mittels Java	Vereint Batch und Stream	klein, für lokale Anwendungen

4.2.1 Topologie

In den meisten Frameworks kann die Topologie innerhalb der Frameworks programmiert werden. Eine Ausnahme bietet hierbei Esper. Dort wird keine explizite Topologie festgelegt, sondern diese stattdessen durch die EPL indirekt beschrieben.

Parse bietet momentan noch keine direkte Möglichkeit an, eine Topologie aufzubauen. Hierfür müssen die Quellen und Senken des Operators manuell verbunden werden.

4.2.2 Ereignisse

In den verschiedenen Frameworks sind Ereignisse unterschiedlich implementiert.

Bei Millwheel besteht jedes Ereignis aus einer Datenstruktur mit Zeitstempel, Schlüssel und Wert. Dabei hat der Schlüssel den Typ String, der Wert hat eine beliebig komplexe Struktur.

Bei Heron, Storm und Flink wird ein Ereignis in einer Klasse des Typs „Event“ gespeichert. Diese beinhaltet ein Tupel worin die Werte des Ereignises gespeichert werden. Dabei kann das Tupel die Elemente unterschiedlicher Datentypen enthalten.

Ein Ereignis bei Esper hat eine Key/Value Struktur. Es kann aus einer XML-Struktur, einer Klasse oder durch ein CEP-Statement erstellt werden.

4.2.3 Fenster

Alle aufgeführten Frameworks unterstützen sowohl Fenster für einen festzulegenden Zeitraum als auch Fenster mit einer festzulegenden Anzahl an Ereignissen.

Das Parse Framework unterstützt zusätzlich noch weitere Arten von Fenstern. Die Kriteren zum Erzeugen neuer Fenster können frei programmiert werden. Beispielsweise können bei bestimmten Steuerereignissen neue Fenster beginnen.

4.2.4 Fehlertoleranz

Das allgemeine Ziel der Fehlertoleranz ist es, auf den Ausfall von Knoten zu reagieren und das System vor Datenverlust zu schützen. Auch ein Leistungseinbruch und ein Systemausfall sollen verhindert werden. Fehlerhafte Werte innerhalb eines Ereignisses werden hier nicht betrachtet.

Die Verarbeitung einer CEP-Anfrage wird auf verschiedene Knoten verteilt. Falls einer dieser Knoten ausfällt ist darauf zu achten, dass das System weiterhin effizient funktioniert.

Parse bietet keine Möglichkeit der Fehlertoleranz an.

Die anderen Frameworks verwenden unterschiedliche Strategien zur Lösung des Problems.

Millwheel nutzt feingranulare Checkpoints. Dabei wird zu jedem Zeitpunkt gespeichert, bei welchem Knoten sich ein Ereignis gerade befindet. Falls der Zielknoten ein Ereignis nicht verarbeiten kann, weil er überlastet oder ausgefallen ist, kann das Ereignis erneut gesendet werden.

Flink nutzt ABS (Asynchronous Barrier Snapshotting). Hierbei wird in regelmäßigen Abständen der Zustand der Operatoren und das zuletzt verarbeitete Ereignisse der Quelle gespeichert. Bei Fehlern können die Operatoren wiederhergestellt werden und die „verlorenen Ereignisse“ erneut verarbeitet werden.

Storm und Heron verwenden einen Supervisor je Knoten. Dieser prüft ob der Knoten noch reagiert und kann ihn bei Bedarf neu starten.

Esper bietet Fehlertoleranz nur in der kostenpflichtigen EsperHA Version. Eine genauere Funktionsweise der Strategie von EsperHA überschreitet den Scope dieser Arbeit.

4.2.5 Überwachen und Steuern

Flink bietet zum Abfragen von Metriken eine REST-API. Diese unterstützt unter anderem auch das Abfragen der Latenz. Zum Erfassen dieser Informationen werden spezielle Ereignisse in das System gesendet und die Daten an der Senke gesammelt. Ein Anpassen der Resourcen ist momentan nur möglich, indem das System kurzzeitig gestoppt wird.

In Storm und Heron ist es möglich mittels einer REST-API Metriken über das System abzufragen. Diese werden von einem Metrikmanager erfasst. Auch ein Ändern des Parallelisierungsgrads von Operatoren ist zur Laufzeit möglich.

In Parse ist eine Möglichkeit zum Abfragen der Latenz implementiert. Für eine Anpassung des Parallelisierungsgrads der Operatoren sind noch Änderungen im Code notwendig.

Kinesis bietet viele Möglichkeiten zum Erfassen von Metriken. Einfache Abfragen im Minutentakt sind kostenlos möglich. Andere Anfragen müssen separat bezahlt werden.

4.3 Fazit

Die hier untersuchten Frameworks sind in ihrer Funktionalität vergleichbar und potentiell für die weitere Analyse geeignet.

Heron ist sehr effizient bei der Verarbeitung von Ereignissen, weit verbreitet und gut dokumentiert. Daher eignet es sich für die weitere Analyse.

Parse ist ein leichtgewichtiges Framework, dessen Quellcode verfügbar ist und bei Bedarf einfach verändert werden kann. Es bietet flexible Möglichkeiten für Fenster.

Als drittes Framework bieten sich Esper oder Flink an, da sie weitere Aspekte, wie beispielsweise die SQL-ähnliche Syntax bzw. die Integration einer Stapelverarbeitung bieten.

Storm ist der Vorgänger von Heron. Deshalb wird Storm nicht weiter untersucht.

Millwheel eignet sich aufgrund der aufwändigen Inbetriebnahme der Infrastruktur nicht gut für eine weitere Analyse, da es den Rahmen dieser Arbeit überschreiten würde.

5 Latenz

Die Latenz in einem CEP-System soll möglichst gering gehalten werden, damit die Ereignisse auch in Echtzeit verarbeitet werden. Um die Latenz eines CEP-Systems zu verringern, kann der Parallelisierungsgrad der Operatoren erhöht werden.

In diesem Kapitel werden die Voraussetzungen und Möglichkeiten der Latenzmessung eines CEP-Systems aufgezeigt.

5.1 Latenzen in einem CEP-System

Im Folgenden werden verschiedene Stellen untersucht, an denen die Latenz gemessen werden kann. Dabei wird ein spezieller Fokus auf die Parallelisierung der Operatoren gelegt. Eine beispielhafte Topologie ist in Abbildung 5.1 dargestellt.

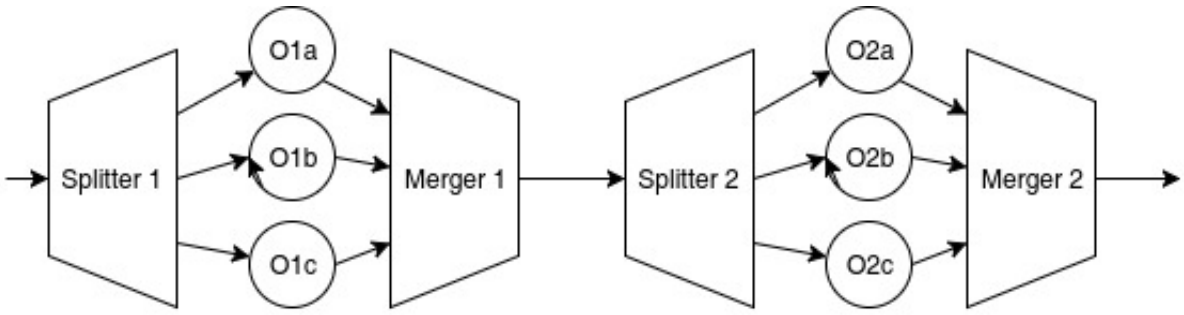

Abbildung 5.1: Topologie mit zwei Operatoren

Eine Übersicht über die verschiedenen Stellen der Latenzmessung ist in Tabelle 5.1 zu sehen. Hierbei gibt die Spalte den Start der Messung an. Das Ende kann der Zeile entnommen werden.

Tabelle 5.1: Strecken der Latenzmessung

	S1	**I1**	**M1**
S1	-	-	-
I1	aufteilen	Dauer des Operators	-
M1	Operator	Mergerstrategie	-
S2	Operator mit Netzwerk	-	Netzwerk
I2	-	Messsystem im Operator implementierbar	-
Mn	Gesamtes System	-	-

5.1.1 Anfang Instanz - Ende Instanz

Die Latenz zwischen Anfang und Ende der Instanz gibt an, wie lange die Instanz zum Bearbeiten benötigt. Dies hängt gewöhnlich von den erkannten Mustern im Datenstrom ab. Der Parallelisierungsgrad hat auf diesen Messwert keinen Einfluss.

5.1.2 Ende Merger - Anfang Splitter

Die Latenz zwischen Merger und Splitter kann beispielsweise bei einer Verbindung übers Netzwerk zu Engstellen führen. Ein höherer Parallelisierungsgrad im Operator hat darauf keinen Einfluss.

Diese Latenz betrachtet auch den Übergang zwischen zwei verschiedenen CEP-Systemen.

5.1.3 Anfang Splitter - Anfang Instanz

Die Latenz zwischen Splitter und Instanz ist stark von dem Parallelisierungsgrad der Operatoren abhängig.

Der Splitter verteilt die Ereignisse an die verschiedenen Instanzen. Wenn keine Instanz mehr frei ist gibt es hier einen Engpass. Dieser kann durch weitere Instanzen behoben werden.

5.1.4 Ende Instanz - Ende Merger

Die Latenz zwischen Instanz und Merger ist auch vom Parallelisierungsgrads des Operators abhängig, weil der Merger, je nach Strategie, auf einige Instanzen warten muss.

Durch die Erhöhung des Parallelisierungsgrads eines Operators wird diese Latenz allerdings nicht verbessert.

5.1.5 Anfang Splitter - Ende Merger

Die Latenz zwischen Splitter und Merger beinhaltet die gesamte Latenz eines Operators inklusive des Verteilens der Ereignisse an die Instanzen, des Verarbeitens in den Instanzen und des Zusammenführens am Ende.

Dieser Wert beinhaltet somit alle Komponenten, auf welche die Änderung des Parallelisierungsgrads Auswirkungen hat.

5.1.6 Gesamtsystem

Die Latenz des Gesamtsystems hilft zu erkennen, ob die Änderung des Parallelisierungsgrads sich positiv oder negativ auf das Gesamtsystem auswirkt.

5.2 Synchrone Uhrzeiten

Zum Bestimmen der Latenz werden zwei Zeitstempel erfasst und deren Differenz berechnet. Dabei muss sicher gestellt werden, dass alle Uhren, die zusammengehörige Anfangs- und Ende-Zeiten bestimmen, synchron laufen. Wenn Anfangs- und Ende-Zeiten auf demselben Rechner bestimmt werden ist das automatisch gewährleistet; wenn diese Zeiten auf verschiedenen Rechnern bestimmt werden, müssen die Rechnerzeiten synchronisiert werden.

5.3 Latenz bei Fenstern

Komplexe Ereignisse hängen meist von mehreren Ereignissen ab. Das komplexe Ereignis tritt jedoch erst bei einem späteren Ereignis ein. Da die Latenzmessung diese Zeit bis zum Eintreffen des auslösenden Ereignisses nicht beinhalten soll, muss hierbei die Latenz zwischen eintreffen des letzten (auslösenden) Ereignisses und dem resultierenden komplexen Ereignis gemessen werden.

5.4 Senden von Latenzinformationen

Wenn Latenzinformationen innerhalb eines CEP-Systems erfasst wurden, müssen diese zur Abfrage und Auswertung an eine zentrale Stelle geschickt werden. Dies kann als Ereignis, im Ereignis oder direkt übers Netzwerk umgesetzt werden.

5.4.1 Als Ereignis

Die erfassten Latenzinformationen werden als Ereignis mit einem speziellen Typ weiter durch das CEP-System geschickt. Die anderen Komponenten leiten Informationen dieses Typs direkt weiter. An der Senke des CEP-Systems kommen dann alle Latenzinformationen an und können ausgewertet werden.

5.4.2 Im Ereignis

Die Latenz kann innerhalb eines Ereignisses gespeichert werden. Dabei können bei der Verarbeitung des Ereignisses die Latenzinformationen wie die anderen Werte gespeichert werden. An der Senke des Systems können diese Daten gesammelt und ausgewertet werden.

Ein Beispiel ist in Abbildung 5.2 zu sehen. Dabei wird im Operator die aktuelle Zeit erfasst und an der Senke ausgewertet. Es wurde der initiale Zeitstempel t=0 gewählt. Der Operator O1 fügt seinen Zeitstempel O1=5 hinzu. An der Senke wird aus den Differenzen der Zeitstempel die Latenz zwischen den Knoten bestimmt. Hierbei müssen alle Knoten eine synchronisierte Uhr besitzen.

Bei der Erstellung von komplexen Ereignissen im Operator werden die Latenzinformationen an das neue Ereignis weiter gegeben. Dies hat den Vorteil, dass dadurch diese Informationen nicht

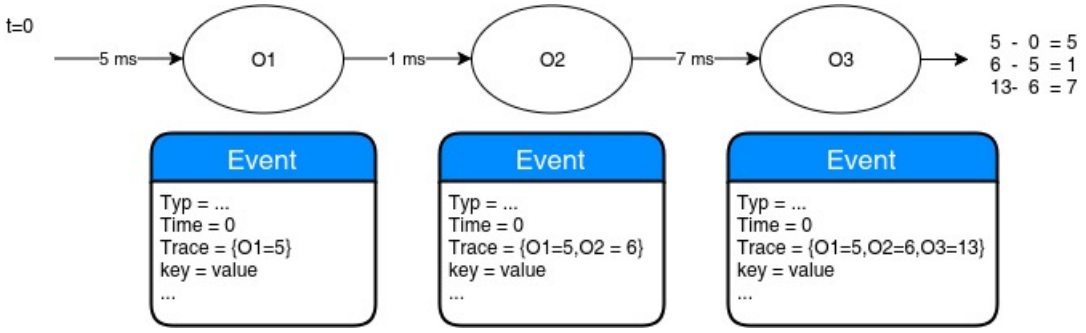

Abbildung 5.2: Latenzmessung im Ereignis

verloren gehen. Allerdings entsteht durch dieses Vorgehen ein sehr großer Overhead. Des Weiteren muss auch eine Id des Ursprungs-Ereignis mitgegeben werden um die Latenz zu berechnen.

Ein Problem dieser Methode ist, dass nicht immer alle Ereignisse in der Senke ankommen. Wenn beispielsweise ein Operator auf ein gewisses Muster (engl. pattern) wartet und bei Nichteintritt die Ereignisse verwirft, dann können nur bei Eintreten dieses Musters Latenzinformationen bestimmt werden. Alle zuvor bestimmten Latenzinformationen, die in diesem Ereignis mitgeführt wurden, gehen verloren. Dies gilt insbesondere auch für Latenzinformationen komplexer Ereignisse, die in einem vorherigen Operator erzeugt wurden, aber nicht mehr weiter gesendet werden.

5.4.3 Direkt über das Netzwerk

Die erstellten Latenzinformationen können direkt über das Netzwerk an eine zentrale Stelle übermittelt werden.

5.5 Erfassen der Latenz

Um den optimalen Parallelisierungsgrad eines Operators zu bestimmen, sind möglichst genaue Informationen über dessen Latenz nötig. Im Folgenden werden drei allgemeine Ansätze aufgezeigt, wie die Latenz in einem CEP-System ermittelt werden kann, sowie deren Vor- und Nachteile erläutert.

5.5.1 Messen in der Instanz

Das Sammeln der Latenzinformationen kann auch in der Instanz erfolgen. Hierbei wird jedem Ereignis beim Verlassen ein Zeitstempel mitgegeben, welcher am Beginn einer Instanz des nächsten Operators wieder ausgelesen wird. Aus der Differenz des Zeitstempels und der aktuellen Uhrzeit kann die Latenz bestimmt werden.

Hierbei werden synchronisierte Uhren für das gesamte System benötigt.

Problem dieser Methode ist allerdings, dass die ermittelte Latenz nicht genau der Latenz des Operators entspricht. Wie in Abbildung 5.3 zu sehen ist, beinhaltet sie den Weg vom Merger des

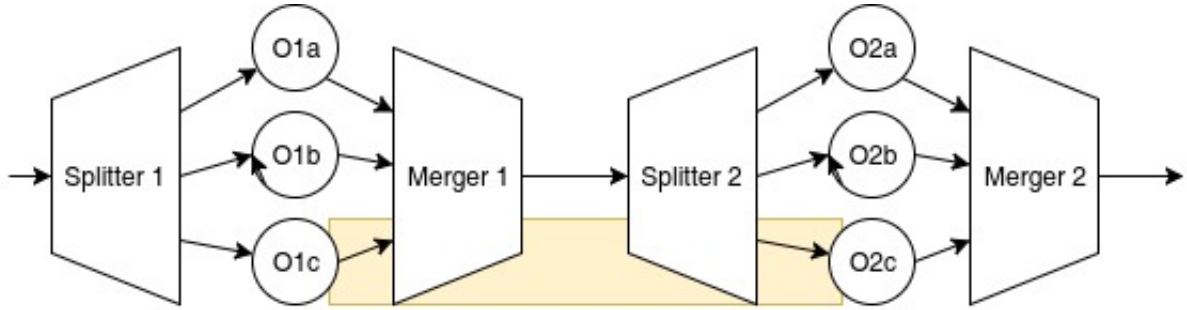

Abbildung 5.3: Latenzmessung in der Instanz

ersten Operators und den Weg des Splitters vom zweiten Operators. Somit ist es schwierig zu entscheiden von welchem Operator der Parallelisierungsgrad angepasst werden soll.

Eine Möglichkeit ist, eine geschätzte Aufteilung zwischen Splitter und Merger zu verwenden. Falls beispielsweise beim Splitter mehr Latenz zu erwarten ist als beim Merger, kann 60 Prozent des Messwertes dem zweiten Operator O2 zugerechnet werden. Die restlichen 40 Prozent werden dem ersten Operator O1 zugerechnet. Eine weitere Möglichkeit ist, durch Anpassung des Parallelisierungsgrads zu testen, welcher der beiden benachbarten Knoten den Engpass besitzt.

Das Messen in der Instanz kann bei den meisten Frameworks verwendet werden, da keine direkten Anpassungen im Framework vorgenommen werden müssen. Die einzige Voraussetzung hierfür ist, einen aktuellen Zeitstempel bestimmen zu können.

Ein weiteres Problem dieser Methode ist, dass jeder Operator angepasst werden muss. Das heißt, zum einen muss am Ende jeder Instanz der Zeitstempel gesetzt werden und zum anderen muss am Beginn jeder Instanz der Zeitstempel wieder gelesen und die Latenz weitergeschickt werden.

5.5.2 Anpassungen im Framework

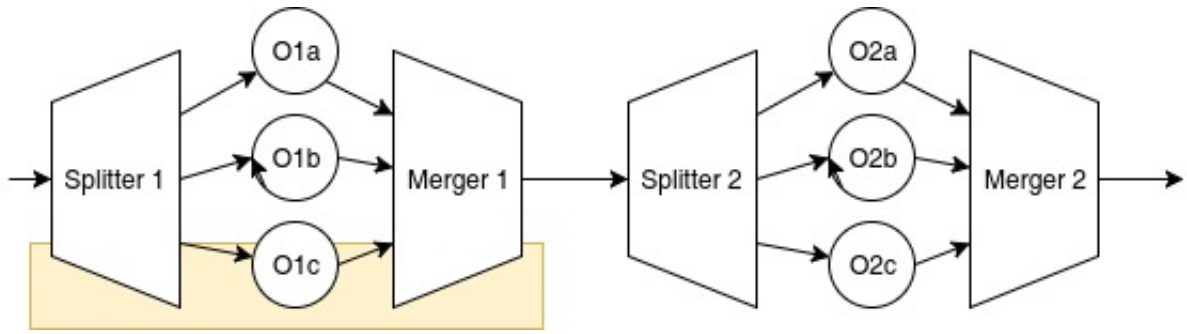

Abbildung 5.4: Latenzmessung im Framework

Die sicherste und mächtigste Möglichkeit, die Latenz zu bestimmen, ist über das Framework selbst.

Dabei müssen bei der Verarbeitung der Ereignisse am Merger und Splitter Anpassungen implementiert werden (Abbildung 5.4). Wenn die zu einem Operator gehörenden Splitter und Merger

auf dem selben Knoten ausgeführt werden, ist eine synchronisierte Uhrzeit gegeben. Ansonsten muss diese durch synchronisierte Uhren sicher gestellt werden.

Einige Frameworks wie beispielsweise Heron unterstützen dies schon und liefern eigene Latenzinformationen je Operator. Bei Open Source Frameworks ist es zudem möglich, selbst Anpassungen vorzunehmen und fehlende Messungen zu ergänzen. Diese Möglichkeit ist jedoch aufwändig und der somit entstehende Fork des Frameworks muss für Aktualisierungen immer manuell angepasst werden.

Bei proprietären Frameworks ist eine Anpassung des Frameworks oft nicht möglich.

5.5.3 Zwischenschalten von Messpunkten

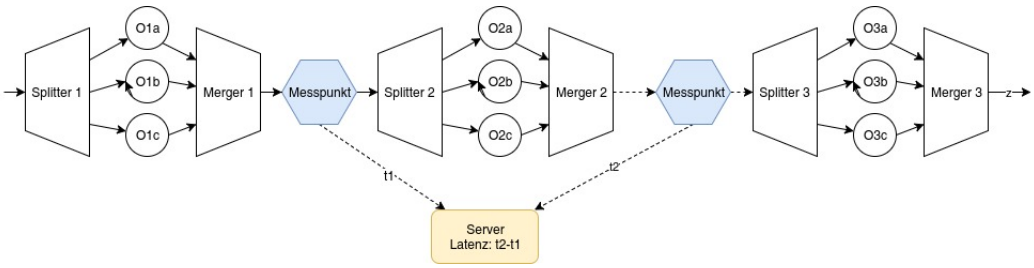

Abbildung 5.5: Latenzmessung durch externe Messpunkte

Eine sehr generische Möglichkeit ist das Zwischenschalten von Messpunkten. Hierbei wird zwischen alle miteinander verbundenen Operatoren jeweils ein Messpunkt geschaltet, durch den alle Ereignisse laufen (siehe Abbildung 5.5). Dieser sendet seine Messungen an ein zentrales Programm, welches daraufhin die Latenz errechnet. In diesem Fall wird aus den Zeiten t1 und t2 die Latenz zwischen Splitter s2 und Merger m2 gemessen. Das ist genau die Latenz des Operators O2. Hierbei müssen die Uhren aller Messpunkte synchronisiert sein.

Diese Möglichkeit funktioniert mit allen Frameworks, jedoch entsteht durch die zusätzlichen Messpunkte ein großer Overhead.

6 Interface

Dieses Kapitel befasst sich mit dem Interface zum zentralen Ansteuern der CEP-Systeme und der Bestimmung der Latenz. Zuerst werden die Anforderungen an das Interface definiert. Danach werden verschiedene Möglichkeiten vorgestellt, die Latenz der Operatoren zu bestimmen. Anschließend folgt die Definition des Interfaces. Danach wird die konkrete Implementierung des Interfaces beschrieben und die Anbindung zwischen dem Client, welcher die Informationen abfragen möchte, und dem CEP-System. Am Ende werden die einzelnen Implementierungen für die Adapter erläutert.

6.1 Anforderungen

Ein CEP-System kann mehrere unterschiedliche Frameworks enthalten. Sowohl die Frameworks als auch die Operatoren müssen eindeutig unterscheidbar sein und konkret angesprochen werden können.

Mit Hilfe der API soll es in einem CEP-System möglich sein die Latenz der beteiligten Operatoren abzufragen und den Parallelisierungsgrad zu ändern. Dazu müssen sowohl allgemeine Informationen zur Topologie als auch Latenzinformationen je Operator bestimmt werden können.

Um die Latenzinformationen eines Operators zu ermitteln wird die Operatorbezeichnung des jeweiligen Frameworks verwendet. Bei Verwendung mehrerer Frameworks ist darauf zu achten, das sich diese Namen nicht überschneiden.

Für jedes Framework wird ein Adapter benötigt, der die Anfragen an das jeweilige Framework weiterleitet.

Um das Blockieren der API bei Fehlern zu vermeiden wird ein Timeout für alle Anfragen benötigt. Tritt dieser ein, wird die Anfrage abgebrochen.

Die Steuerung des Frameworks wird mit einer REST-Schnittstelle umgesetzt, welche über ein Java-Interface angesprochen werden kann.

6.2 Definition Interface

Das Interface bietet eine Funktion, die eine Liste aller Operatoren aus allen Frameworks liefert (getAllOperators). Für jeden dieser Operatoren kann sowohl die Latenz abgefragt (getLatency), als auch der Parallelisierungsgrad abgefragt (getOperatorParallelism) oder geändert (setOperatorParallelism) werden.

Die Hauptdokumentation ist im Java-Interface (siehe Abschnitt 6.2). Sie ist mit Javadoc direkt aus dem Java-Interface generiert. Die Dokumentationen für die Shell (Abschnitt 6.2.5) und für die REST-API (Abschnitt 6.2.4) sind davon abgeleitet.

6.2.1 Java

6.2.2 Method summary

getAllOperators()
getLatency(String, String) gibt die Latenz des Operators zurück
getOperatorParallelism(String)
setOperatorParallelism(String, int) setzt den Parallelisierungsgrad für einen Operator

6.2.3 Methods

- **getAllOperators**

  ```
  java.util.List getAllOperators()
  ```

  - **Returns** – Eine Liste mit allen Operatoren des CEP-Systems

- **getLatency**

  ```
  int getLatency(java.lang.String type,java.lang.String operator)
  ```

  - **Description**

    Gibt die Latenz des Operators zurück

  - **Parameters**

    * `type` – Min/Max/Avg
    * `operator` – Latenz von diesem Operator

  - **Returns** – Die Latenz in Nanosekunden

- **getOperatorParallelism**

  ```
  int getOperatorParallelism(java.lang.String operator)
  ```

  - **Parameters**

    * `operator` – Der Operator von dem der Parallelisierungsgrad bestimmt wird

  - **Returns** – Parallelisierungsgrad für einen Operator

- **setOperatorParallelism**

```
boolean setOperatorParallelism(java.lang.String operator,int parallelism)
      throws java.util.concurrent.TimeoutException
```

- **Description**

  Setzt den Parallelisierungsgrad für einen Operator

- **Parameters**

  * `operator` – Der Operator an dem der Parallelisierungsgrad geändert werden soll
  * `parallelism` – Der Parallelisierungsgrad

- **Returns** – true: Erfolgreiche Änderung des Parallelisierungsgrads false: Fehler beim Ändern des Parallelisierungsgrads

- **Throws**

  * `java.util.concurrent.TimeoutException` – Das System hat zu viel Zeit benötigt. Die Anfrage wurde nicht vollständig ausgeführt. Teile sind jedoch eventuell angepasst worden.

6.2.4 REST

Tabelle 6.1: REST-API

Befehl	Parameter	Rückgabe	Beispiel
getalloperators		Array	/v1/getalloperators
getlatency	type :String operator :String	Zahl	/v1/getlatency?operator=word&type=max
getoperatorparallelism	operator :String	Zahl	/v1/getoperatorparallelism?operator=word
setoperatorparallelism	operator :String parralelism :Zahl	Boolean	/v1/setoperatorparallelism?operator=word &paralelism=5

6.2.5 Cli

Die Kommandozeile gibt die gleichen Strukturen zurück wie die REST-API. Aufrufe: interface.sh getalloperators

interface.sh getlatency FROMOPERATOR NEXTOPERATOR

interface.sh setoperatorparallelism OPERATOR PARALLELISM

interface.sh getoperatorparallelism OPERATOR

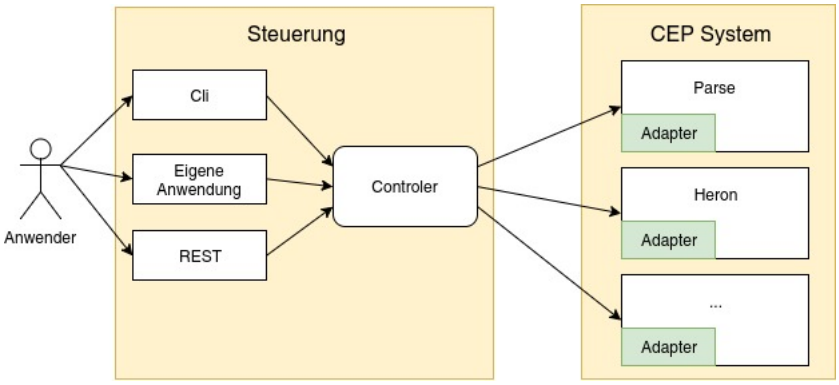

Abbildung 6.1: Struktur Interface

6.3 Verwendung des Interfaces

Um das Interface zu verwenden wird ein Programm (Controller) benötigt, welches den zuständigen Adapter für den gegebenen Operator bestimmt und diesen ansteuert. Das Konzept ist in Abbildung 6.1 zu sehen.

Dieses Programm kann über Kommandozeilenparameter (Abschnitt 6.2.5), über eine Javaklasse die den Controler direkt aufruft (Abschnitt 6.2) implementiert, oder über die REST-Schnittstelle (Abschnitt 6.2.4) angesteuert werden.

Damit der Controller die Anfragen an die zuständigen Adapter schicken kann wird eine Zuordnungstabelle (Mappingtabelle) benötigt. Diese muss beim Starten des Controllers mitgegeben werden.

Die benötigen Timeouts werfen je nach Ansteuerung unterschiedliche Fehlermeldungen. Bei der Kommandozeileneingabe wird ein Error-Code gesetzt. Bei der Verwendung der Java-Klasse wird eine Java-Exception geworfen. Bei der REST-Schnittstelle wird der Statuscode gesetzt.

6.4 Implementierung Adapter

6.4.1 allgemeiner Ansatz für neue Adapter

Ein Adapter muss über die API Abschnitt 6.2.4 ansprechbar sein und wird als eigenständiges Programm gestartet. Wenn der Adapter in Java geschrieben wird reicht es, wenn er das Interface aus Abschnitt 6.2 implementiert. Zum Starten des Adapters kann die Klasse RESTServerService genutzt werden, welche die im Interface verwendeten Funktionen als REST-API anbietet.

Je nach Framework und Anfrage ist die Kommunikation des Adapters mit dem Framework aktiv (Abfragen) oder passiv (Lauschen).

Beim Abfragen sendet der Adapter eine oder mehrere Anfragen an das Framework. Die Anfrage wird dann vom Framework beantwortet. Diese Antwort muss nun vom Adapter standardisiert werden und kann dann an den Controller gesendet werden.

Beim Lauschen öffnet der Adapter einen Port, auf den das Framework regelmäßig Latenzinformationen schickt. Der Adapter sammelt diese permanent und liefert die Ergebnisse sobald der Controller seine Anfrage sendet.

6.4.2 Heron

Bei Heron kann die Parallelisierung durch den heron-cli Befehl „heron update [cluster] [topologiename] –component-parallelism=[operator]:[parallelism]“ durchgeführt werden. Dabei kann der Parallelisierungsgrad für die einzelnen Komponenten separat eingestellt werden.

Die Latenzinformationen werden vom Metrics Manager erfasst. Dieser gibt sie an verschiedene Senken weiter. Eine davon ist die FileSink. Diese protokolliert die Metriken als JSON im Filesystem.

Eine weitere Methode zum Abfragen der Metriken ist die Heron Tracker REST API[Heron Tracker REST API]. Diese liefert Informationen über die Metriken und auch über die Topologie.

Der neu implementierte Adapter für Heron nutzt zum Setzen des Parallelisierungsgrads das heron-cli. Zum Abfragen von Informationen des Systems wird die heron-tracker-api verwendet.

6.4.3 Parse

Bei Parse müssen die zu verwendenden Instanzen manuell gestartet werden. Dabei werden sie beim Splitter angemeldet. Die Instanzen registrieren sich hierbei am Splitter und am Merger. Am Splitter kann hierbei gesteuert werden, welche Instanzen Ereignisse bekommen und welche passiv sind. Somit kann die Parallelität reduziert werden.

Parse unterstützt eigene Algorithmen zum automatischen Anpassen der Parallelisierung. Zum einen ist das ein CPU gesteuerter Algorithmus, bei dem je nach CPU-Last die Anzahl der Instanzen angepasst wird. Zum anderen ist das ein QT-Algorithmus, bei dem aufgrund von Vorhersagen der künftigen Last die Parallelisierung angepasst wird. Dadurch soll jedem Ereignis eine bestimmte durchschnittliche Wartezeit garantiert werden. Um den Parallelisierungsgrad über die Schnittstelle zu ändern, dürfen diese Algorithmen aktiviert werden. Dies ist über die Parameter beim Starten des Splitters konfigurierbar.

Der neu implementierte Adapter für Parse ist im Splitter integriert, da er direkten Zugriff auf diesen benötigt. Zum Ändern der Parallelität fügt er die passiven Instanzen zum Manager hinzu beziehungsweise entfernt diese aus ihm. Zum Abfragen der Latenz nutzt er die vorhandene statische Klasse „ProcessingLatencyStatisticsMonitor“.

Wenn beim Adapter mehr Instanzen angefragt werden, als das System zur Verfügung hat wird ein Fehler geworfen. Wenn trotzdem mehr Instanzen verwendet werden sollen, müssen diese manuell gestartet werden.

7 Beispielimplementierung

Als Beispiel für die Verwendung von CEP-Frameworks wird ein Tischfußballspiel analysiert. Durch die Implementierung können die Eigenarten der verschiedenen CEP-Frameworks erkannt werden.

7.1 Idee

Es wird ein Tischfußballspiel in Echtzeit analysiert um bestimmte Muster und Statistiken zu erfassen. Diese Statistiken sollen während des Spiels direkt ausgegeben werden. Dazu wird die Position des Balls auf dem Spielfeld in Echtzeit bestimmt. Dann müssen die Koordinaten mittels Matrixtransformation normalisiert werden. Dabei wird die Koordinate 0/0 auf die Mitte des Tisches gelegt. Die aktuelle Positionen des Balls können als Ereignisse an ein CEP-Framework weiter gegeben werden.

Folgende komplexe Ereignisse können erstellt werden:

- Wenn der Ball in Tornähe war und daraufhin verschwindet, war das ein Tor.
- Wenn sich die Richtung des Balls ändert, war das entweder ein Kontakt mit einem Spieler oder mit der Bande. Dies kann abhängig von der Position des Balls auf dem Spielfeld bestimmt werden.
- Wenn ein Ball seitlich gepasst wird und dann nach vorne gespielt wird, ist das ein Angriff.

Es können auch Statistiken über Ballkontakte oder die Zeit auf den Hälften des Tisches erstellt werden.

7.2 Probleme

Da der Ball von einer Stange oder einem Spieler verdeckt sein kann, ist es nicht zu jedem Zeitpunkt möglich, die Position des Balls zu bestimmen. Des Weiteren kann es Probleme bei der Ballerkennung durch ungleichmäßige Lichtverhältnisse geben, falls das Spielfeld nicht gleichmäßig ausgeleuchtet ist.

Um die fehlenden Werte zu erkennen, kann die Position des Balls interpoliert werden. Es ist eventuell auch möglich die Operatoren so flexibel zu implementieren, dass sie auch trotz fehlender Werte korrekt arbeiten. Des Weiteren könnte eine zweite Kamera mit einer anderen Perspektive helfen. Es ist auch möglich einen Ball anderer Farbe zu nutzen um den Kontrast zum Spielfeld zu verstärken.

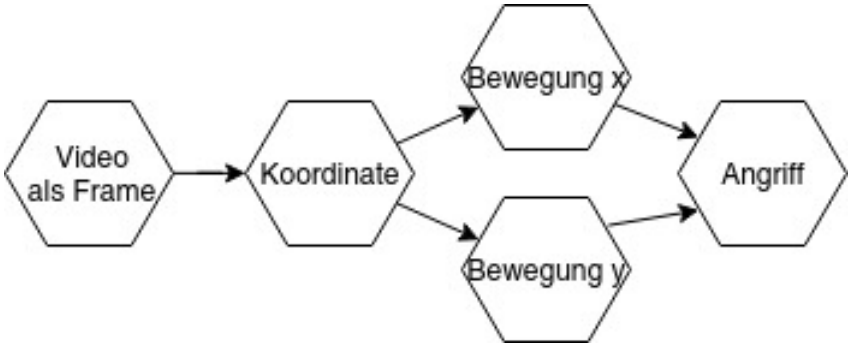

Abbildung 7.1: Aufbau Beispielimplementierung

7.3 Konzept

Um die Möglichkeiten von CEP zu zeigen, bietet sich das Erkennen von einem Angriff an, da er eine gewisse Komplexität besitzt und mit mehreren Operatoren implementiert werden kann. Des Weiteren ist die Tischposition für dieses Muster nicht so wichtig und muss deshalb nicht genau konfiguriert werden.

Um das Angriffsmuster zu testen, wird eine Videosequenz verwendet, die nur aus Angriffen besteht. Damit das Beispiel nicht zu komplex wird, werden die Koordinaten nicht normalisiert.

Die Topologie des CEP-Systems ist in Abbildung 7.1 zu sehen. Zuerst wird das Video in einzelne Frames unterteilt und diese an den ersten Operator gesendet. Dieser bestimmt aus dem Bild die aktuelle Ballposition. Diese wird in Fenstern der Größe zwei dann an zwei weitere Operatoren geschickt. Diese erkennen die horizontale beziehungsweise vertikale Bewegung des Balls und schicken diese weiter an den letzten Operator. Dieser erkennt den Angriff wenn zuerst eine Bewegung auf der x-Achse und direkt darauf eine Bewegung auf der y-Achse empfangen wird.

7.4 Umsetzung

Zum Erkennen der Ballposition wird OpenCV verwendet. Es bietet die Möglichkeit Videostreams zu analysieren. Zur Verwendung muss die native Bibliothek des Frameworks vom Operator geladen werden.

OpenCV wandelt zuerst das Video in einzelne Frames, welche als Base64 codiert im CEP-System weiter verarbeitet werden .

Danach wird dieses Frame mittels OpenCV analysiert um die Position des Tischfußballs im Spielfeld zu bestimmen.

Zum Erkennen der Richtung einer Bewegung wird der Winkel zwischen dem Ideal und der tatsächlichen Richtung bestimmt. Ist dieser kleiner als $\pi/8$” wird die Bewegung erkannt.

Listing 7.1 Heron Topologie

```
builder.setSpout("frames", new FramesSpout(Duration.ofMillis(0),
       videopath), spouts);
builder.setBolt("coords", new CoordsBolt(hmin, smin, vmin, hmax, smax, vmax),
       bolts).shuffleGrouping("frames");
builder.setBolt("attackx", new AttackBolt(new double[]{1,0},"x")
      .withWindow(BaseWindowedBolt.Count.of(2))).shuffleGrouping("coords");
builder.setBolt("attacky", new AttackBolt(new double[]{0,1},"y")
      .withWindow(BaseWindowedBolt.Count.of(2))).shuffleGrouping("coords");
builder.setBolt("attackxy", new AttackxyBolt("x","y")
      .withWindow(BaseWindowedBolt.Count.of(2)))
      .shuffleGrouping("attackx").shuffleGrouping("attacky");
builder.setBolt("print4", new PrinterBolt(), 1).shuffleGrouping("attackxy");
```

7.5 Implementierung in Heron

Bei Heron wird mittels des „TopologyBuilders“ die Topologie beschrieben. Dabei werden die Spouts und Bolts der Topologie zugewiesen und jeweils die Ressourcen verteilt. Zum Hinzufügen eines Spouts beziehungsweise Bolts wird ein neues Objekt erstellt. Dieses bekommt einen Namen um auf dessen erzeugte Ereignisse zugreifen zu können. Bei Bolts wird zusätzlich angegeben von welcher Komponente der Eingabedatenstrom stammen soll. Dieser wird mit einer Groupingmethode gesetzt. In diesem Fall ist dies „shuffleGrouping“. Es ist auch möglich mehrere Datenströme an eine Komponente zu binden.

Die Topologie für dieses Beispiel ist in Listing 7.1 dargestellt.

Es wird die Spout „frames“ und die Bolts „attackx“, „attacky“, „attackxy“ und „print“ verwendet. Für die Fenster mit der Größe 2 wird die „shuffleGrouping“ Methode verwendet, da sie immer die direkt aufeinanderfolgenden Ereignisse in ein Fenster zusammenfasst und jedes Fenster an genau eine Instanz geschickt wird.

Das Mergen der Datenströme von „attackx“ und „attacky“ funktioniert in Heron gut und dabei wird die Reihenfolge beibehalten.

Bei der Implementierung der Komponenten müssen diese vom passenden Basistyp abgeleitet werden. Mit der Funktion „declareOutputFields“ werden die Schlüssel der Ereignisse festgelegt. Im „prepare“ wird ein „OutputCollector“ mitgegeben, über den Ereignisse gesendet werden können. Neue Ereignisse werden im Bolt von der Funktion „execute“ produziert.

Die Ergebnisse können mit einem „PrinterBolt“ ausgegeben werden.

Damit der Parallelisierungsgrad zur Laufzeit geändert werden kann, ist es wichtig, beim Starten von Heron darauf zu achten, dass in der „packing.yaml“ als packing Algorithmus „FirstFitDecreasingPacking“ verwendet wird. Mit Round Robin wird dies momentan nicht unterstützt.

Listing 7.2 Parse Topologie

```
java -jar merger.jar 4202 localhost 4211 localhost 4221
java -jar merger.jar 4212 localhost 4231
java -jar merger.jar 4222 localhost 4231
java -jar merger.jar 4232

java -jar splitter.jar 4201 -1 tu 2 1 rr 0 0 0 0 0 0 none 2 0 100 1 4435
java -jar splitter.jar 4211 -1 tu 2 1 rr 0 0 0 0 0 0 none 2 0 100 1 4436
java -jar splitter.jar 4221 -1 tu 2 1 rr 0 0 0 0 0 0 none 2 0 100 1 4437
java -jar splitter.jar 4231 -1 tu 2 1 rr 0 0 0 0 0 0 none 2 0 100 1 4438

java -jar instance.jar localhost 4201 foosball2coords 1 2 false 100 10 10
      localhost 4202
java -jar instance.jar localhost 4211 foosball2attackx 1 2 false 100 10 10
      localhost 4212
java -jar instance.jar localhost 4221 foosball2attacky 1 2 false 100 10 10
      localhost 4222
java -jar instance.jar localhost 4231 foosball2attackxy 1 2 false 100 10 10
      localhost 4232

java -jar source.jar localhost 4201 true fooseballframes attacks.avi 100 100 true
```

7.6 Implementierung in Parse

Das Framework Parse besteht aus den Komponenten „Source“, „Splitter“, „Instance“ und „Merger“. Diese besitzen jeweils eine eigene Main-Methode und können auf verschiedenen Knoten ausgeführt werden. Für dieses Szenario wurden jedoch alle Komponenten auf einem gemeinsamen Rechner ausgeführt. Zur Kommunikation untereinander werden Nachrichten über Sockets zwischen diesen Komponenten ausgetauscht.

Um eine Topologie zu starten, müssen für jeden Operator „Splitter“, „Instance“ und „Merger“ gestartet werden. Die Startkonfiguration für dieses Beispiel ist in Listing 7.2 abgebildet.

Zum Senden der Ereignisse wurde die „Source“ angepasst um mittels OpenCV die Frames zu generieren.

Das Ereignis beinhaltet hierbei die Höhe und Breite des Frames, das Frame als Base64, einen OpenCV internen Typ und die aktuelle Framenummer. Die erzeugten Ereignisse werden an den „Splitter“ weiter gegeben.

Im „Splitter“ mussten keine Anpassungen vorgenommen werden. Beim Aufruf wird Round Robin als Vergabestrategie (Scheduler) gewählt. Des Weiteren werden Fenster mit jeweils zwei Ereignissen erzeugt, die immer um ein Ereignis weitergeschoben werden.

In der „Instance“ wurden die Operatoren „foosball2coords“, „foosball2attackx“, „foosball2attacky“, „foosball2attackxy“, implementiert. Die Operatoren „foosball2attackx“ und „foosball2attacky“ sind in einer Klasse implementiert, werden jedoch mit unterschiedlichen Parametern gestartet.

Dazu mussten sie neu erstellt,als Konstanten in „util.Constants“ hinzugefügt und im „re.Manager“ aufgerufen werden.

Die Instanzen haben die Funktion „run“. Sie holt in einer Endlosschleife immer das nächste Ereignis mittels „this.manager.getEventsToBeProcessed().poll()“ und verarbeitet dieses. In „foosball2coords“ wird dort die Position des Balls bestimmt. In den anderen Operatoren wird mit der Funktion „handleReceivedEvent“ ein Fenster gebildet. Auf diesem Fenster werden dann die Werte berechnet.

Die Funktion des Operators ist in der Funktion „f“ implementiert.

Neue Ereignisse geben die Operatoren mittles „this.manager.emittedEventHook(e)“ aus.

Am „Merger“ mussten Anpassungen implementiert werden, damit dieser Ereignisse an einen beziehungsweise zwei „Splitter“ schicken kann. Dies ist für den Aufbau einer Topologie in Parse notwendig.

Für die Latenzmessung muss die Latenz im Operator bestimmt werden und dann mit „InstanceMain.getMonitoringWindow().measurementEventProcessing“ gesetzt werden.

7.7 Vergleich und Fazit

Da beide Implementierungen in Java umgesetzt wurden, war es möglich, einen gemeinsamen Programmcode zu verwenden. Jedoch ist ein großer Teil der Implementierung direkt abhängig von einem direkten Zugriff auf die Ereignisse, beziehungsweise auf die Fenster. Diese sind jeweils anders zu implementieren. Zusätzlich muss in Parse die Latenzmessung manuell im Operator ergänzt werden, während Heron diese Daten automatisch bestimmt.

Des Weiteren kann in Heron die Topologie und die Zuweisung der Fenster direkt in der Hauptklasse konfiguriert werden. Bei Parse muss dies jedoch über das Hintereinanderschalten mehrerer Systeme umgesetzt werden.

Heron ist momentan in aktiver Entwicklung; deshalb ist die Dokumentation nicht immer vorhanden beziehungsweise korrekt. Dafür ist es in Heron einfach eine Topologie aufzubauen und die Operatoren zu implementieren. Es eignet sich gut für größere aufwendige Topologien und auch um schnell eine Topologie zu entwickeln.

Parse ist an einigen Stellen noch sehr fehleranfällig. Beispielsweise treten Exceptions auf wenn zusätzlich zu den Ereignissen eine Leerzeile geschickt wird. Auch wenn die Ereignisse ein Komma oder ein Minus enthalten treten Probleme auf. Das Aufbauen einer Topologie ist eher umständlich und durch die hohe Anpassbarkeit muss im Operator viel Programmcode für die Ereignisverwaltung geschrieben werden. Es eignet sich gut wenn viel angepasst werden muss. Leider ist es von der Laufzeit deutlich langsamer als Heron.

8 Verwendung des Interfaces und Evaluation

In der Ausarbeitung wurde gezeigt, dass es möglich ist, ein Interface zu definieren und zu verwenden, dass den Anforderungen aus Kapitel 6 entspricht. Aufgrund des allgemeinen Ansatzes können auch für weitere Frameworks Adapter erstellt werden und somit weitere Frameworks unterstützt werden.

Beim Messen der Latenz muss die Synchronisation der Uhren beachtet werden. Es kann schnell zu Performanceproblemen kommen. Somit ist eine Anpassung innerhalb des Frameworks oft die bessere Lösung um die Effizienz und Korrektheit zu gewährleisten.

8.1 Vorteile dieser Schnittstelle

Durch diese Schnittstelle kann jedes beliebige CEP-System, welches aus unterstützten Frameworks besteht, einheitlich angesteuert und optimiert werden. Dies ist auch dann möglich, wenn das CEP-System aus unterschiedlichen Frameworks besteht.

Die Unterstützung der Frameworks und die Logik der Ressourcenzuteilung sind getrennt und jeweils insoliert erweiterbar. Wenn ein neues Framework unterstützt werden soll, muss nur ein weiterer Adapter implementiert werden. Die Systeme die das Interface nutzen, können dieses dann direkt nutzen. Wenn eine neues System beziehungsweise eine neue Strategie zur Ressourcenzuteilung verwendet werden soll, funktioniert diese sofort auf allen unterstützten Frameworks.

8.2 Möglichkeiten ohne diese API

Im Folgenden werden drei Möglichkeiten aufgezeigt, welche Alternativen es zu der Idee eines solchen Interfaces gibt.

8.2.1 Keine Ressourcenzuteilung

Wenn auf die Ressourcenzuteilung vollständig verzichtet wird, dann werden zwangsläufig die Ressourcen nicht effizient genutzt.

8.2.2 Ansätze anderer Paper nutzen

Dabei wird die Ressourcenzuteilung automatisch, oft anhand von Vorhersagen, angepasst. Dies funktioniert jedoch nur für genau dieses Framework. Bei der Verwendung eines anderen Frameworks muss somit diese Komponente ausgetauscht werden. Eine frameworkübergreifende Ressourcenzuteilung ist nicht möglich.

8.2.3 Eigene Lösung entwerfen

Wenn eine eigene Lösung entworfen wird, die es auch ermöglichen soll, die Ressourcen frameworkübergreifend zuzuteilen, müssen für die verschiedenen Frameworks viele Sonderfälle eingeführt werden um die Frameworks nativ anzusprechen. Dann ist jedoch eine nachträgliche Erweiterung aufwendig, da viele Stellen angepasst werden müssen.

Alternativ kann auch eine Möglichkeit mit Vererbung für die unterschiedlichen Frameworks gewählt werden. Dies entspricht jedoch dann dem Ansatz der vorliegenden Arbeit.

8.3 Fazit

Es ist somit zu erkennen, dass dieser Ansatz für eine allgemeine, strukturbasierte Möglichkeit zum Anpassen der Ressourcen über Frameworks hinweg benötigt wird.

9 Zusammenfassung und Ausblick

In dieser Arbeit wurden zu Beginn die verschiedenen Veröffentlichungen zur Ansteuerung von CEP-Frameworks und zum Vergleich existierender CEP-Frameworks untersucht. Dabei beschäftigen sich viele Paper mit der Ansteuerung genau eines Frameworks.

Hierbei wurde festgestellt, dass Potenzial für einen allgemeinen frameworkübergreifenden Ansatz existiert.

Um einen Adapter für ein Framework zu entwickeln, muss das Framework die Möglichkeit der Regulation bieten. Außerdem müssen auch Metriken für das CEP-System erfasst werden.

Dabei wurden die verschiedensten Aspekte verglichen. Es fanden sich hierbei keine Vergleiche mit den Schwerpunkten Ansteuerung und Bestimmung von Latenzinformationen.

Deshalb wurden zur Implementierung und zum Entwurf eines Interfaces die verschiedenen in der Wissenschaft am meisten verbreiteten CEP-Frameworks auf diese Aspekte verglichen.

Des Weiteren wurden der Aufbau von Topologien, die Struktur von Ereignissen, die Unterstützung von Fenstern und die Fehlertoleranz bei Systemausfällen untersucht.

Bei der Analyse wurden Heron und Parse für die weitere Analyse ausgewählt. Dabei stellte sich heraus, dass Heron sehr effizient bei der Verarbeitung von Ereignissen, weit verbreitet und gut dokumentiert ist. Parse ist ein leichtgewichtiges Framework, dessen Quellcode verfügbar ist und bei Bedarf einfach verändert werden kann. Es ist ziemlich flexibel.

Es wurde ein Interface entwickelt, welches mehrere CEP-Frameworks innerhalb eines CEP-Systems unterstützt und die Anfragen durch eine Zuordnungstabelle weiterleitet.

Zur Implementierung der Adapter wurden die Voraussetzungen untersucht. Dazu gehören die Stellen, an denen innerhalb eines CEP-Systems die Latenz gemessen werden kann sowie die Bedeutung von synchronen Uhrzeiten und das Verhalten der Latenz bei Fenstern. Weiterhin wurden die Stellen, an denen in ein CEP-System eingegriffen werden kann um die Latenz zu erfassen, und die Arten um Latenzinformationen zu senden, analysiert.

Aufgrund dieser Analyse wurde festgestellt, das für die Adapter für Heron und Parse jeweils die existierende Latenzmessung des Frameworks verwendet werden soll. Daraufhin wurden die Adapter entsprechend implementiert.

Um die Eigenarten von Heron und Parse zu untersuchen wurde ein Beispiel implementiert. Dabei wurde deutlich, dass eine Topologie in Parse aufwendig zu implementieren ist. In Heron ist eine Umsetzung einfacher, da das Framework gut strukturiert und übersichtlich ist.

Ausblick

Das hier implementierte Interface kann nun von einer externen Software genutzt werden, um ein CEP-System auf Latenz zu optimieren. Hierfür sind weitere Verfahren und Algorithmen notwendig.

Des Weiteren können zusätzliche Adapter entworfen werden um weitere CEP-Frameworks zu unterstützen.

Falls zur Ansteuerung weitere beziehungsweise andere Latenzinformationen benötigt werden, können diese mittels weiterer Adapter implementiert werden.

Literaturverzeichnis

[ABB+13] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle. „MillWheel: Fault-tolerant Stream Processing at Internet Scale“. In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), S. 1033–1044. ISSN: 2150-8097. DOI: 10.14778/2536222.2536229. URL: http://dx.doi.org/10.14778/2536222.2536229 (zitiert auf S. 23).

[Akka] URL: http://akka.io (zitiert auf S. 23).

[Amazon Kinesis] URL: https://aws.amazon.com/kinesis/ (zitiert auf S. 23).

[Apache Gearpump] URL: https://gearpump.apache.org/releases/latest/index.html (zitiert auf S. 23).

[Apache Spark] URL: https://spark.apache.org (zitiert auf S. 22).

[CKE+15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. „Apache flink: Stream and batch processing in a single engine“. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (zitiert auf S. 22).

[CM12] G. Cugola, A. Margara. „Processing Flows of Information: From Data Stream to Complex Event Processing“. In: *ACM Comput. Surv.* 44.3 (Juni 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: http://doi.acm.org/10.1145/2187671.2187677 (zitiert auf S. 17, 23).

[FAG+17] A. Floratou, A. Agrawal, B. Graham, S. Rao, K. Ramasamy. „Dhalion: Self-regulating Stream Processing in Heron“. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), S. 1825–1836. ISSN: 2150-8097. DOI: 10.14778/3137765.3137786. URL: https://doi.org/10.14778/3137765.3137786 (zitiert auf S. 19).

[FDM+15] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, Z. Zhang. „DRS: dynamic resource scheduling for real-time analytics over fast streams“. In: *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. IEEE. 2015, S. 411–420. DOI: 10.1109/ICDCS.2015.49 (zitiert auf S. 18).

[Heron Tracker REST API] URL: https://twitter.github.io/heron/docs/operators/heron-tracker-api/ (zitiert auf S. 37).

[HSD13] W. Hummer, B. Satzger, S. Dustdar. „Elastic stream processing in the cloud“. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 3.5 (2013), S. 333–345. DOI: 10.1002/widm.1100 (zitiert auf S. 18).

[KBF+15] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja. „Twitter heron: Stream processing at scale“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, S. 239–250. DOI: 10.1145/2723372.2742788 (zitiert auf S. 21).

[KLL17] R. K. Kombi, N. Lumineau, P. Lamarre. „A Preventive Auto-Parallelization Approach for Elastic Stream Processing“. In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, S. 1532–1542. DOI: 10.1109/ICDCS.2017.253 (zitiert auf S. 18).

[LLD16] M. A. Lopez, A. G. P. Lobato, O. C. M. Duarte. „A performance comparison of Open-Source stream processing platforms“. In: *Global Communications Conference (GLOBECOM), 2016 IEEE*. IEEE. 2016, S. 1–6. DOI: 10.1109/GLOCOM.2016.7841533 (zitiert auf S. 17, 22).

[MKR15] R. Mayer, B. Koldehofe, K. Rothermel. „Predictable low-latency event detection with parallel complex event processing“. In: *IEEE Internet of Things Journal* 2.4 (2015), S. 274–286. DOI: 10.1109/JIOT.2015.2397316 (zitiert auf S. 22).

[MTR] R. Mayer, M. A. Tariq, K. Rothermel. *Real-time Batch Scheduling in Data-Parallel Complex Event Processing*. Techn. Ber. Technical Report 2016/04. University of Stuttgart. DOI: 10.1109/JIOT.2015.2397316 (zitiert auf S. 22).

[TTS+14] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy. „Storm@Twitter“. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, S. 147–156. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595641. URL: http://doi.acm.org/10.1145/2588555.2595641 (zitiert auf S. 21).

[ZS17] A. Y. Zomaya, S. Sakr. *Handbook of Big Data Technologies*. Springer, 2017 (zitiert auf S. 22).

Alle URLs wurden zuletzt am 28. 12. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift