BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Vergleich von Streamingframeworks: STORM, KAFKA, FLUME, S4

vorgelegt von

Eduard Bergen

Matrikel-Nr.: 769248

dem Fachbereich VI – Informatik und Medien –
der Beuth Hochschule für Technik Berlin vorgelegte Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
im Studiengang
Medieninformatik-Online (Master)

Tag der Abgabe 28. Oktober 2014

1. Betreuer	Herr Prof. Dr. Edlich	Beuth Hochschule für Technik
Gutachter	Herr Prof. Knabe	Beuth Hochschule für Technik

Kurzfassung

Mit der enormen Zunahme von Information in unterschiedlichen Quellen wie Sensoren (RFID) oder Nachrichtenquellen (RFD newsfeeds) wird es schwieriger Informationen beständig abzufragen. Um die Frage zu klären, welcher Rechner am häufigsten in einem Netzwerk frequentiert wird, werden unterstützende Systeme notwendig. An dieser Stelle helfen Methoden aus dem Bereich des Complex Event Processing (CEP). Im Spezialbereich Stream Processing von CEP wurden Streaming Frameworks entwickelt, um die Arbeit in der Datenflussverarbeitung zu unterstützen und damit komplexe Abfragen auf einer höheren Schicht zu vereinfachen. In dieser Master Thesis geht es um einen Vergleich zwischen den Streaming Frameworks: Apache Storm, Apache Kafka, Apache Flume und Apache S4. In der Thesis werden Prototypen entwickelt, um Messwerte zu erfassen und zu vergleichen.

Abstract

With the tremendous growth of information in various sources such as sensors (RFID) or news sources (RFD newsfeeds) it is more difficult to query information continously. To clarify the question, which machine is most commonly frequented in a network, supporting systems are beeing necessary. At this point methods in the field of Complex Event Processing (CEP) help to solve this issue. In the special area of stream processing of CEP, streaming frameworks were developed to support and simplify the work of complex queries in a higher abstraction level. This master thesis involves a comaprison between the streaming frameworks: Apache Storm, Apache Kafka, Apache Flume and Apache S4. In the thesis prototypes are developed to capture and to compare measured values.

Inhaltsverzeichnis

Kapitel 1

Einführung

> Social media streams, such as Twitter, have shown themselves to be useful sources of real-time information about what is happening in the world. Automatic detection and tracking of events identified in these streams have a variety of real-world applications, e.g. identifying and automatically reporting road accidents for emergency services. [MMO+13]

Im Internet steigt das Angebot zu unterschiedlichen Informationen rapide an. Gerade in Deutschland wächst das Datenaufkommen, wie die Studie der IDC [Dig14, S. 2-3] das zeigt, exponentiell. Dabei nimmt ebenfalls das Interesse an wiederkehrenden Aussagen über die Anzahl bestimmter Produkte, die Beziehungen zu Personen und die persönlichen Stimmungen zueinander zu. So wird in [Dat14] eine interaktive Grafik zum Zeitpunkt der Ansprache zur Lage der Union des Präsidenten der USA angezeigt. Je Zeitpunkt und Themenschwerpunkt wird in der Ansprache zeitgleich die Metrik Engagement zu den einzelnen Bundesstaaten aus den verteilten Twitternachrichten berechnet ausgegeben.

In der Infografik [Jam14b] von Josh James, Firma Domo wird ein Datenwachstum von 2011 bis 2013 um 14,3% veranschaulicht. Es werden unterschiedliche Webseiten vorgestellt. Dabei werden unterschiedlichen Arten von Daten, die pro Minute im Internet erzeugt werden gezeigt. In der ersten Fassung [Jam14a] waren es noch 2 Millionen Suchabfragen auf der Google-Suchseite [Goo14]. Die zweite Fassung gibt über 4 Millionen Suchanfragen pro Minute an. In Facebook [Fac14] konnten in der Fassung mehr als 680 Tausend Inhalte getauscht werden. In der zweiten Fassung werden mehr als 2,4 Millionen Inhalte pro Minute getauscht.

Um die Sicherheit bei Verlust einer Kreditkarte zu erhöhen und gleichzeitig die höchste Flexibilität zu erhalten, gibt es im Falle eines Schadens bei der von unterschiedlichen Orten gleichzeitig eine unerwünschte Banküberweisung stattfindet, für die Bank die Möglichkeit, die Transaktion aufgrund der Positionserkennung zurückzuführen [SÇZ05, S. 3, K. Integrate Stored and Streaming Data].

Mit steigenden Anforderungen, wie in der Umfrage [Cap14, S. 8] durch schnellere Analyse, Erkennung möglicher Fehler und Kostenersparnis dargestellt, und damit einem massiven Datenaufkommen ausgesetzt, kann die herkömmliche Datenverarbeitung [CD97, S. 2, K. Architecure and End-to-End Process] durch das Zwischenlagern der Daten in einem Datenzentrum keine komplexen und stetigen Anfragen zeitnah beantworten [MMO+13, S. 2 K. Related Work: Big

Data and Distributed Stream Processing]. Damit müssen Nachrichten, sobald ein Nachrichteneingang besteht, sofort verarbeitet werden können. Allen Goldberg stellt in [GP84, S. 1, K. Stream Processing Example] anhand eines einfachen Beispiels Stream processing, zu deutsch Verarbeitung eines Nachrichtenstroms ausgehend von loop fusion [GP84, S. 7, K. History] vor. Da Allen Goldbergs Beschreibung, zu Stream processing, in die Ursprünge geht, soll ein einfaches Modell eines Stream processing Systems für die weitere Betrachtung als Grundlage dienen.

So wird in [AAB+05, S. 2, K. 2.1: Architecture] die distributed stream processing engine Borealis vorgestellt und als große verteilte Warteschlangenverarbeitung beschrieben. Die Abbildung [AAB+05, S. 3, A. 1: Borealis Architecture] zeigt eine Borealis-Node mit Query processor in der Abfragen verarbeitet werden. Eine Borealis-Node entspricht einem Operator, in dem laufend Datentupel sequentiell verarbeitet werden. Mehrere Nodes sind in einem Netzwerk verbunden und lösen dadurch komplexe Abfragen. Damit die Komplexität, die Lastverteilung und somit die Steigerung der Kapazität für die Entwicklung von neuen Anwendungen vereinfacht werden, wurden Streaming Frameworks entwickelt. Streaming Frameworks stellen auf einer höheren Abstraktion Methoden zur Datenverarbeitung bereit.

Bisher werden einzelne Streaming Frameworks separat in Büchern oder im Internet im Dokumentationsbereich der Produktwebseiten vorgestellt. Dabei werden vorwiegend Methoden des einzelnen Streaming Frameworks erläutert und auf weiterführenden Seiten vertieft. Als Software Entwickler wird der Nutzen für die Streaming Frameworks nicht sofort klar. Zum Teil sind die Dokumentationen veraltet, in einem Überführungsprozess einer neuen Version oder es fehlt ein schneller Einstieg mit einer kleinen Beispielanwendung.

In dieser Arbeit wird eine Übersicht mit Einordnung und Spezifikation über die einzelnen Streaming Frameworks Apache Storm [Mar14c], Apache Kafka [KNR14a], Apache Flume [PMS14a] und Apache S4 [S413c] gegeben. Dabei werden außerdem die Streaming Frameworks diskutiert und verglichen.

Die vorliegende Arbeit ist in Drei Teile aufgebaut. Im ersten Teil erfolgt eine theoretische Einführung in die Grundlagen sowie die Vorstellung und Anwendung von Kriterien an einem Referenzmodell. Im zweiten Teil werden die Streaming Frameworks und ein praxisnaher Anwendungsfall vorgestellt, sowie Messungen durchgeführt. Auf der Basis der Erkenntnisse, die in den ersten beiden Teilen gewonnen wurden, werden im dritten Teil die Kernaussagen zusammengefasst sowie ein Ausblick gegeben.

Das erste Kapitel führt mit praktischen Beispielen in das Thema ein. Im zweiten Kapitel werden Grundlagen geschaffen und verwendete Fachbegriffe erläutert. Zudem wird die eingesetzte Technologie vorgestellt, eingeordnet und zusammengefasst. Im dritten Kapitel findet eine Vorstellung und Anwendung von Kriterien an einem Referenzmodell statt. Dabei werden unter den gewonnen Grundlagen weitere Fachbegriffe eingeführt. Für die weitere Betrachtung der Streaming Frameworks, wird abschließend das Referenzmodell bewertet. Kapitel Vier stellt die einzelnen Streaming Frameworks vor. Im Anhang wird jeweils eine Installationsanleitung und ein kurzes Startprogramm als Quelltext abgelegt. In Kapitel Fünf wird die Systemarchitektur vorgestellt, Anforderungen, der Algorithmus und das Vorgehen bei der Implementierung der Prototypen werden basierend auf den Erkenntnissen aus Kapitel Vier vorgestellt. Das sechste Kapitel stellt die Projektstruktur vor und dokumentiert die einzelnen Prototypen. In Kapitel Sieben wird die Messumgebeung vorgestellt und die Ergebnisse aus der Messung dargestellt, diskutiert und bewertet. Anschließend werden die Streaming Frameworks verglichen und Erkenntnisse werden gezogen. Das letzte Kapitel greift die Erkenntnisse aus Kapitel Sieben auf

und führt in die Schlussbetrachtung ein. Abschließend wird zusammengefasst und ein Ausblick gegeben.

Kapitel 2

Grundlagen

Im folgenden Kapitel werden die Begriffe Event, Stream, Processing aus der Informatik im Bereich der verteilten Systeme erläutert und in einen Zusammenhang zu Streaming Frameworks gebracht. Dabei wird ein Grundkonzept für eine streambasierte Nachrichtenverarbeitung gestellt. Im weiteren Verlauf und maßgeblich in Kapitel 4 wird stets auf das Grundkonzept Bezug genommen. In der Einführung wurde die stream processing engine Borealis [AAB+05] als ein einfaches Modell eines Stream processing-Systems erwähnt. Zuerst werden im Unterkapitel 2.1 die wesentlichen Fachbegriffe vorgestellt. Anschließend wird im Unterkapitel 2.2 ein Zeitbezug zu verwandten Technologien gegeben und die Streaming Frameworks aus Kapitel 4 werden eingeordnet. Das Kapitel 2 endet mit einer Zusammenfassung.

2.1 Grundbegriffe

Ein großer Teil der verwendeten Grundbegriffe sind in [TvS07] definiert. An dieser Stelle werden nur die wesentlichen Grundbegriffe vorgestellt. Ein verteiltes System wird von Andrew S. Tanenbaum und Maarten van Steen in [TvS07, S. 19, K 1.1] als *„[...] eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen.“* Verteilte Systeme bestehen also laut [TvS07] aus unabhängigen Komponenten und enthalten eine bestimmte Form der Kommunikation zwischen den Komponenten. Informationen werden zwischen Sender und Empfänger über ein Signal ausgetauscht. Dazu hat Claude E. Shannon in [Sha48, S. 2, A. 1] ein Diagramm eines allgemeinen Kommunikationssystems vorgestellt. In der genannten Abbildung wird das Signal in einem Kanal codiert übertragen. Dabei ist das Signal einem Umgebungsrauschen ausgesetzt. Durch Einsatz geeigneter Kodierverfahren in Übertragungsprotokollen können Übertragungsfehler festgestellt und behoben werden. Im schlimmsten Fall wird eine fehlerhaft übertragene Nachricht zum Beispiel innerhalb des Transmission Control Protocol (TCP) auf Open Systems Interconnection Model (OSI)-Modell Schichtebene 4 in [Uni94, S. 40, K. 7.4.4.6 Data transfer phase] neu übertragen. Der Kanal ist das Medium in [Sha48], um die Nachricht zu übertragen. Tanenbaum und van Steen beschreiben in [TvS07, S. 184, K. 4.4.1] ein kontinuierliches Medium[1] gegenüber einem diskreten Medium[2], als zeitkritisch zwischen Signalen. Shannon beschreibt in [Sha48, S. 3 und S. 34] ein kontinuierliches System mit folgendem Zitat:

[1] kontinuierliches Medium: Temperatursensor

[2] diskretes Medium: Quelltext

> *„A continuous system is one in which the message and signal are both treated as continuous functions, e.g., radio or television. [...] An ensemble of functions is the appropriate mathematical representation of the messages produced by a continuous source (for example, speech), of the signals produced by a transmitter, and of the perturbing noise. Communication theory is properly concerned, as has been emphasized by Wiener, not with operations on particular functions, but with operations on ensembles of functions. A communication system is designed not for a particular speech function and still less for a sine wave, but for the ensemble of speech functions.“*

Ein Stream oder ein Datastream ist somit eine Folge von Signalen. Einem Signal entspricht ein Event und die Anwendung von Funktionen findet im Processing statt. Somit ist Event stream processing eine Signalfolgenverarbeitung in einem kontinuierlichen Medium. Weiterhin soll in diesem Zusammenhang von Event stream processing oder abgekürzt ESP gesprochen werden.

Zu Streams wird eine Paketierung von unterschiedlichen Substreams in Audio, Video und Synchronisierungsspezifikation verstanden. In [TvS07, S. 191] wird ein Beispiel für die Kompressionsverfahren 2 und 4 für Audio und Video Übertragung der Motion Pictures Expert Group (MPEG) gezeigt. Durch den Verbund unterschiedlicher Algorithmen zur Komprimierung der Substreams werden paketierte Streams bereitgestellt. Paketierte Streams im der MPEG werden in dieser Arbeit nicht näher betrachtet.

Weiterhin beschreibt Muthukrishnan in [Mut10] (2010) mehrere Forschungsrichtungen in Datastreams. Darunter werden *„[...] theory of streaming computation [...], data stream management systems [...], theory of compressed sensing [...]“* [Mut10, S. 2, Absatz 2] aufgezählt. Die Forschung in *streaming computation* konzentriert sich auf geringe Zugriffszeiten während mehrfachem Zugriff auf permanent ankommenden Datennachrichten. Mit einem *data stream management system* soll ein Zugriff, durch Einsatz von speziellen Operatoren auf nicht endende Datenquellen möglich sein. Und in der *theory of compressed sensing* wird nach geringen Zugriffsraten zum Aufteilen in Signalmustern unterhalb der *Nyquist*-Rate geforscht. So findet Streaming in der Signalverwaltung, Signalverarbeitung und Signaltheorie eine Anwendung.

Während Streams auf einem Prozessorsystem verarbeitet werden können, muss eine hohe Kapazität von Daten auf einem oder mehreren Multiprozessorsystemen in einer geringen Latenz verteilt berechnet werden können. Tanenbaum und van Steen stellen die Grundlagen der Remote Procedure Call (RPC)-Verwendung in [TvS07, S. 150, K. 4.2.1] vor. Abstraktionen der Schnittstelle zur Transportebene, wie diese auf OSI-Modell Ebene 4 durch TCP angeboten werden, bilden dabei eine Vereinfachung um Funktionen mit übergebenen Parametern auf entfernten Rechnern aufzurufen. Nach der entfernten Berechnung wird das Ergebnis sofort an den Client zurückgeschickt. Dabei ist der Client bei einem synchronen Nachrichtenmodell blockiert bis der Server geantwortet hat. Sobald die Berechnung durchgeführt wurde, muss der Clientim asynchronen Nachrichtenmodell nicht warten und wird erst nach Abschluss der Berechnung am Server vom Server informiert. Währenddessen können weitere Anfragen durch den Client auf dem Server erfolgen.

Wie in [TvS07, S. 170, K. 4.3.2] vorgestellt wurde, wurde durch den Einsatz von Warteschlangensystemen ein zeitlich lose gekoppelter Nachrichtenaustausch zwischen Sender und Empfänger möglich. Der Empfänger entscheidet selbst wann und ob eine Nachricht eines Senders von der Warteschlange abgeholt wird. Zusätzlich entsteht die Möglichkeit des Warteschlangensystems Nachrichten zwischenzuspeichern. Im Gegensatz zu RPCs haben Nachrichten in Warteschlangensystemen eine Adresse und können beliebige Daten enthalten.

Mehrere Server in einem Verbund bilden ein Cluster. In einem Cluster übernehmen einzelne Rechner-Knoten die Berechnung. Außerhalb der Rechner-Knoten gibt es einen Master-Knoten mit dem die Rechenaufgaben auf die Rechner-Knoten verteilt werden. Dazu wird von Tanenbaum und van Steen in [TvS07, S. 35, A. 1.6] ein Cluster-Computersystem in einem Netzwerk gezeigt. Dieses Prinzip wird auch in den Streaming Frameworks eingesetzt. In dem Kapitel 4 werden die einzelnen Streaming Frameworks im Detail vorgestellt. Die Streaming Frameworks selbst bieten dabei ähnlich wie es bei den RPCs der Fall ist, eine Abstraktionsschicht um die Datenverarbeitung für den Entwickler zu vereinfachen. Dazu werden abstrakte Primitive und Operatoren für die Anwendung auf einem unterliegenden Cluster bereitgestellt.

Es wurden die Grundbegriffe eines Streaming Frameworks vorgestellt und eingeordnet. In Kapitel 2.2 wird ein Basis Streaming Framework als Referenz vorgestellt. Außerdem werden die Streaming Frameworks Apache Storm, Apache Kafka, Apache Flume und Apache S4 kurz zum Streaming Framework Referenzmodell in Zusammenhang gebracht. Die Technologie, die dazu zum Einsatz notwendig ist, wird in eigenen Unterkapiteln vorgestellt.

2.2 Technologie

Mit den gewonnen Grundbegriffen werden in diesem Kapitel ein Modell eines Basis Streaming Framework vorgestellt. Zuerst wird das Grundmodell und deren Komponenten gezeigt und beschrieben. Anschließend werden die Streaming Frameworks Apache Storm, Apache Kafka, Apache Flume und Apache S4 mit dem Modell des Streaming Frameworks in eine Beziehung gebracht. Das Unterkapitel 2.2 endet mit weiteren Komponenten für Streaming Frameworks und leitet in das Unterkapitel Zusammenfassung ein.

Ein Basis Modell für Streaming Frameworks soll durch eine Stream Processing Engine (SPE) Aurora/Borealis [ACc+03] veranschaulicht werden. Im weiteren Verlauf wird zur Vereinfachung das Schlagwort *Aurora* anstatt SPE Aurora/Borealis verwendet. So besteht ein Modell in [ACc+03, S. 2, Abb. 1 Aurora system model] aus ankommenden Daten, den *Input data streams*, aus ausgehenden Daten, dem *Output to applications* und aus wiederkehrenden Abfragen, den *Continous queries*. In Abbildung 2.1 wird ein Modell als Grundlage für weitere Betrachtungen zu Streaming Frameworks vorgestellt. Dabei wird ein azyklisch gerichteter Graph im Zentrum als Verarbeitungseinheit mit linker und rechter Datenflussüberführung gezeigt.

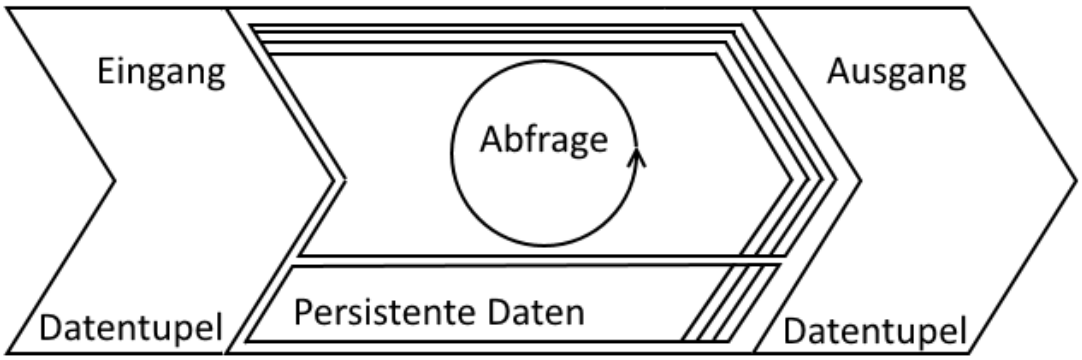

Abbildung 2.1: Exemplarische Darstellung eines Basis Modells für Streaming Frameworks

Dabei sind *Input data streams* eine Sammlung von Werten und werden von *Aurora* als eindeutiges Tupel mit einem Zeitstempel identifiziert. Innerhalb von *Aurora* können mehrere *Continous queries* gleichzeitig ausgeführt werden. Abbildung 2.1 stellt in der Mitte der Grafik zwischen Eingang und Ausgang der Datentupel mehrschichtige Ebenen als Repräsentation für mehrere *Continous queries* dar.

Ein *Continous query* besteht aus *boxes* und *arrows*. *Boxes* sind Operatoren um ankommende Datentupel in ausgehende Datentupel zu überführen. Durch die *Arrows* wird eine Beziehung zwischen den *Boxes* hergestellt. Ein komplexer Beziehungsgraph ist in eine Richtung gerichtet, es enthält keine Zyklen, hat mehrere Startknoten und einen Endknoten. Für die weitere Datenverarbeitung können in einem *Continous query* zusätzlich persistente Datenquellen in einer *Box* zur Transformation von Datentupeln hinzugefügt werden. Dazu wird in Abbildung 2.1 unterhalb der *Continous queries* ein mehrschichtiger separater Bereich für die persistenten Daten dargestellt. Im Endknoten des azyklisch gerichteten Graphen werden die transformierten Datentupel für weitere Anwendungen als Ausgabestrom von Datentupeln bereitgestellt. Die Abbildung 2.2 stellt beispielhaft einen azyklisch gerichteten Graphen dar. Der dargestellte Graph enthält zwei Eingangsdatenquellen. Die obere Datenquelle wird zeitlich kurz vor dem Endknoten mit der unteren Datenquelle kombiniert. Die untere Datenquelle wird nach der zweiten Transformation in zwei neue Datenquellen aufgespalten. Nach drei Transformationen wird die mittlere Datenquelle in die oberer Datenquelle zeitlich an der sechsten Transformation überführt. Die untere Datenquelle wird mit der oberen Datenquelle an der siebten Transformation verbunden. Die letzte Transformation bildet den Endknoten.

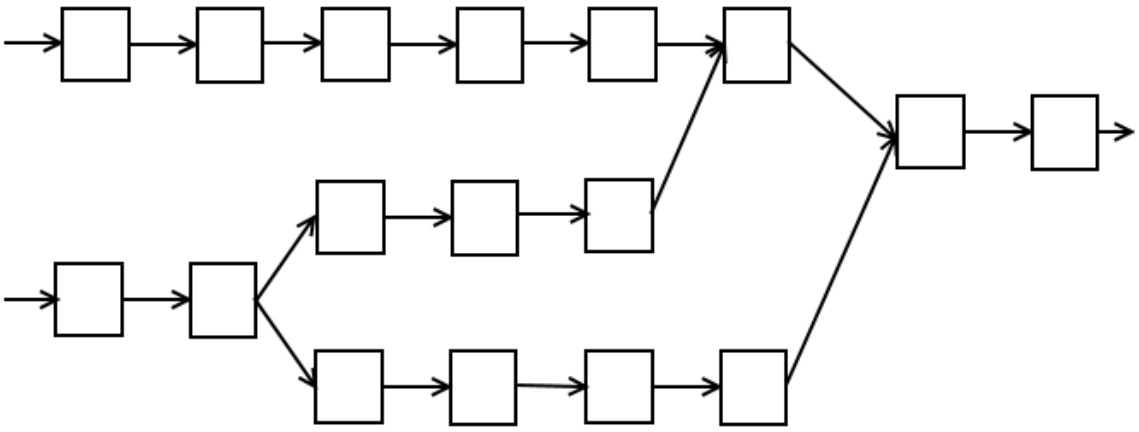

Abbildung 2.2: Darstellung eines azyklisch gerichteten Graphen

Das Datenmodell in *Aurora* besteht aus einem *Header*, dem Kopfbereich und Data, dem Datenteil als Tupel. Der Header in einem Basismodell besteht aus einem Zeitstempel. Mit dem Zeitstempel wird das Datenpaket eindeutig identifiziert und wird für das Monitoring in Quality of Service (QoS) als einen Dienst für die Güte eingesetzt. Im Gegensatz zu *Aurora* wird in der auf *Aurora* basierten Weiterentwicklung *Borealis* ein Vorhersagemodell für QoS zu jedem Zeitpunkt in einem Datenfluss möglich. Dazu wird jedem Datentupel ein Vector of Metrics (VM) hinzugefügt. Ein VM besteht aus weiteren Eigenschaften wie zum Beispiel Ankunftszeit oder Signifikanz. In [AAB+05, S. 3, Kap. 2.4 QoS Model] werden Vector of Metrics vorgestellt.

In *Borealis* gibt es statuslose Operatoren und Operatoren mit einem Status. Statuslose Operatoren sind *Filter*, *Map* und *Union*. Mit dem Filter kann eine Datenquelle nach bestimmten Bedingungen neue Datenquellen erzeugen. Der *Map*-Operator kann bestimmte Datentupel in einer Datenquelle transformieren wie zum Beispiel durch anreichern von Informationen. Mit

dem *Union*-Operator können mehrere Datenquellen in eine Datenquelle zusammengeführt werden. Dazu wird ein Zwischenspeicher in der Größe $n + 1$ benutzt. In [Tea06b, S. 9, Abb. 3.1 Sample outputs from stateless operators] wird eine Übersicht über die drei Operatoren *Filter*, *Map* und *Union* anhand eines konkreten Beispiels dargestellt. Operatoren mit einem Status wie *Join* und *Aggregate* werden in [Tea06b, S. 9, Kap. 3.2.2 Stateful Operators] als Berechnungen von speziellen Zeitfenstern, dem *window*, die mit der Zeit mitbewegen erläutert. In [Tea06b, S. 10, Abb. 3.2 Sample output from an aggregate operator] wird ein Schaubild zum Operator *Aggregate* mit der Funktion *group by*, *average* und *order* in einem *window* gezeigt. Dabei werden eingehende Datenquellen mit einem Schema nach Zeit, Ort und Temperatur in einem Zeitfenster von einer Stunde gruppiert nach Raum, gemittelt nach Temperatur und sortiert nach Zeit in eine ausgehende Datenquelle transformiert. In Abbildung 2.3 wird ein Abfrage-Diagramm in einer Stream Processing Engine dargestellt. Es werden zwei Sensoren S1 und S2 mit dem *Union*-Operator in einen *Stream* zusammengeführt. Der *Stream* wird von zwei *Aggregate*-Operatoren in einem Zeitfenster von 60 Sekunden getrennt und jeweils mit einem *Filter*-Operator reduziert. Durch den *Join*-Operator werden beide *Streams* in einem Zeitfenster von 60 Sekunden zu einem dritten *Stream* transformiert.

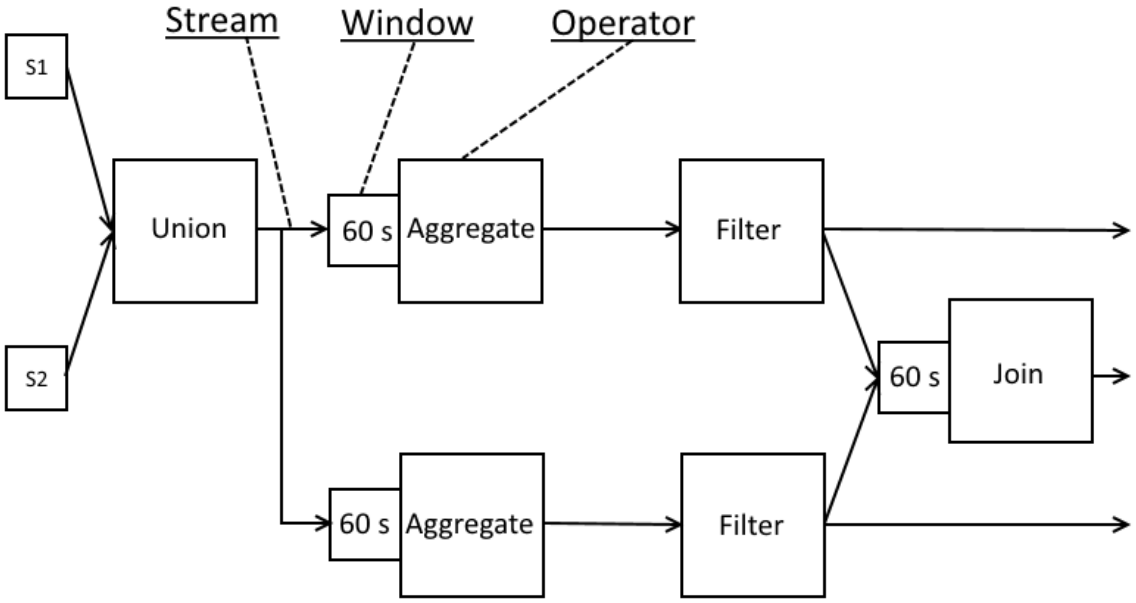

Abbildung 2.3: Stream Processing Engine

Datentupel in *Aurora* können aufgrund von technischen Fehlern, wie zum Beispiel Sensorausfall oder doppelter Parametrierung von mehreren Sensoren durch Hinzufügen, Löschen oder Aktualisieren verschiedene Versionen annehmen. Daher wurde das Datenmodell in *Borealis* im *Header* um einen Revisionstyp und einem Index erweitert. In separaten Speichern, den Connection Points (CPs), werden die Revisionen der Datentupel als Historie gehalten. Die CPs sind direkt an einer Datenquelle angeschlossen. Operatoren können auf die CPs durch die Identifikatoren im *Header* auf benötigte Datentupel in der Historie zugreifen.

In den Streaming Frameworks Storm, Kafka, Flume und S4 wird eine ähnliche Architektur wie sie im Referenzmodel von *Aurora* und *Borealis* vorgestellt wurde benutzt. Zwischen den Streaming Frameworks gibt es dennoch Unterschiede. Das Referenzmodell von *Aurora* und *Borealis* soll dem Verständnis bei der Vorstellung der Streaming Frameworks im Kapitel 4 dienen und die Unterschiede aufzeigen. Mit der Einführung der Grundbegriffe und eines Referenzmodells soll nun das Kapitel Grundlagen im Unterkapitel 2.3 zusammengefasst werden.

2.3 Zusammenfassung

Im Kapitel Grundlagen wurden die Streaming Frameworks in die Bereiche der Informationsverarbeitung in verteilten Systemen, der Signaltheorie und der wiederkehrenden Berechnung von Daten in Datenströmen eingeordnet. Dabei wurden aktuelle Forschungsbereiche aufgezeigt und ein Referenzmodell als Grundlage dargestellt. In der Beschreibung des Referenzmodells sind die komplexen Operatoren und die Primitive vorgestellt worden. Ein Abfrage-Diagramm konnte anhand eines azyklisch gerichteten Graphen gezeigt werden. Mögliche Fehlererkennung durch Qualitätssicherungsmaßnahmen wurden durch Vector of Metrics angesprochen. Fehlererkennungsmechanismen und Gütesicherung werden im Einzelnen im Kapitel 4 aufgezeigt. Bevor die Streaming Frameworks vorgestellt werden, wird in Kapitel 3 die Umgebung, der Markt, eine Studie, Kriterien für die Beurteilung der Streaming Frameworks vorgestellt und angewendet.

Kapitel 3

Vorstellung und Anwendung der Kriterien

In Kapitel 2 wurden für die weitere Betrachtung der Streaming Frameworks notwendige Grundbegriffe erläutert und ein Referenzmodell wie in Abbildung 2.1 gezeigt vorgestellt. Zunächst wird der Markt anhand der Studie [MLH+13] im Kontext von Big Data in dem Streaming Frameworks zum Einsatz kommt vorgestellt. Die Studie [MLH+13] wurde von Markl et al. im Auftrag des Bundesministeriums für Wirtschaft und Energie (BMWi) 2013 erstellt.

> Zentrales Ziel der vorliegenden Studie ist eine qualitative und quantitative Bewertung des Ist-Zustandes sowie der wirtschaftlichen Potenziale von neuen Technologien für das Management von Big Data. Daraus werden standortspezifische Chancen und Herausforderungen für Deutschland abgeleitet. In der Studie werden insbesondere auch rechtliche Rahmenbedingungen anhand von Einzelfällen betrachtet. Die Studie beinhaltet zudem konkrete Handlungsempfehlungen. [MLH+13, S. 3]

Big Data wurde im Artikel [Lan01] von Laney 2001 in drei Dimensionen *volume*, *velocity* und *variety* eingeordnet. Die Dimension *volume* beschreibt den Umgang mit dem rasanten Anstieg an Datentransaktionen. *Velocity* gibt die Geschwindigkeit an und *variety* gibt die steigende Vielfalt der Daten an. In der Abbildung 3.1 werden die drei Dimensionen in einem Würfel dem *Big Data Cube* dargestellt. Die Abbildung 3.1 wurde aus [Mei12, S. 1, Abb. 1] in einfacher Form übernommen. So beschreibt Meijer *volume* von klein *small* nach groß *big*, *velocity* von ziehen *pull* nach drücken *push* und *variety* von komplexen strukturierten Daten Fremd-/Primärschlüssel *fk/pk* nach einfachen Zeigern auf Daten und Daten *k/v*. Das herkömmliche relationale Datenbanksystem ist in der Abbildung 3.1 unter den Koordinaten (*small*, *pull*, *fk/pk*) zu finden. Unter den Koordinaten (*small*, *pull*, *k/v*) wären Anwendungen zu finden, die das Konzept *Object-relational mapping (ORM)* implementieren. Beim Konzept *ORM* werden Objekte in relationalen Datenbanken abgebildet [Mei12, S. 1]. Die Streaming Frameworks werden unter den Koordinaten (*big*, *push*, *fk/pk*) eingeordnet. Gegenüber den Streaming Frameworks unter den Koordinaten (*big*, *pull*, *fk/pk*) befinden sich die Batch Processing Engines, wie zum Beispiel Apache Hadoop [Fou14b]. In der unteren linken Ecke (big, pull, k/v) werden Lambda Ausdrücke eingeordnet. Lambda Ausdrücken werden anonyme Methoden möglich. Damit können einfache Abfragen auf Sammlungen formalisiert werden. Der Compiler erzeugt zur Laufzeit die Methoden im Hintergrund. In Java stehen die Lambda Ausdrücke erst ab Version 8 zur Verfügung [Cor14b].

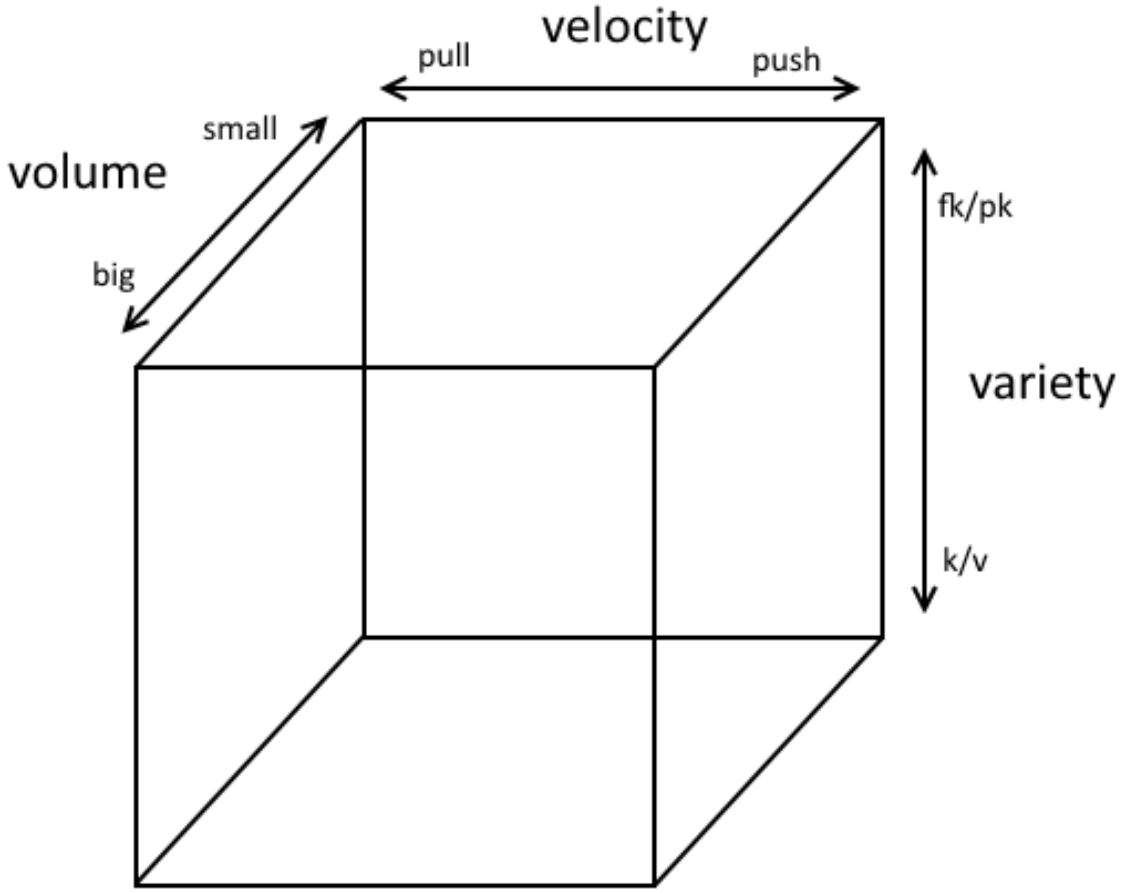

Abbildung 3.1: Darstellung Big Data Cube

Relationale Datenbanksysteme stoßen im Zusammenhang der horizontalen Skalierung in der zentralisierten Systemarchitektur auf Probleme, wenn die Datenmenge die Kapazität einer Maschine übersteigt und dadurch das Ergebnis in keiner akzeptablen Zeit zurückgegeben wird [EFHB11, S. 30, Kap. 2.2.1]. So zeigt Edlich et al. in dem Buch [EFHB11] einen alternativen Ansatz Daten zu halten. Dabei wird der Begriff *NoSQL* als nicht relationales Datenbanksystem eingeführt und definiert [EFHB11, S. 2, K 1.2]. In Verbindung mit horizontaler Skalierung, Replikation und niedriger Reaktionszeit wird in [EFHB11, S. 30, K. 2.2] das Consistancy Availability Partition Tolerance (CAP)-Theorem erklärt. Beim CAP-Theorem besteht der Konflikt in der Konsistenz *C*. Es gilt zu Entscheiden ob die Konsistenz gelockert wird oder nicht. Bei einer lockeren Konsistenz und damit einer hohen Verfügbarkeit und Ausfalltoleranz können in einem Verbindungsausfall alte Zustände zurückgegeben werden. Falls nicht gelockert wird kann der Umstand in Kraft treten, sehr lange Reaktionszeiten zu erhalten. Daher wurde das Konsistenzmodell Basically Available, Soft State, Eventually Consistant (BASE) eingeführt. Es basiert auf einem optimistischen Ansatz. Eine Transaktion nimmt den Status konsistent nicht unmittelbar ein. Erst nach einer gewissen Zeitspanne ist die Transaktion konsistent. Dieses Verhalten wird als *Eventually Consistancy* bezeichnet. Als Beispiel gibt Edlich et al. in [EFHB11, S. 33, K. 2.2.3] replizierende Knoten in einer Systemarchitektur an.

So wurden in der Studie [MLH+13] neben der Einführung in Big Data, Stärken, Schwächen, Chancen und Risiken für die Branchen Handel, Banken, Energie, Dienstleistungen, Öffentlicher Sektor, Industrie, Gesundheitssektor, Marktforschung, Mobilitätsleistungen, Energie und Versi-

cherungen als tabellarische Übersicht in [MLH+13, S. 105, Tab. 18] ausgegeben. In [MLH+13, S. 107, Abb. 54] werden Branchenschwerpunkte abgeleitet. Die genannte Abbildung wird im folgenden Zitat textuell erneut wiedergegeben:

> 1. Entwicklung neuartiger Technologien, um eine skalierbare Verarbeitung von komplexen Datenanalyseverfahren auf riesigen, heterogenen Datenmengen mit hoher Datenrate zu realisieren
> 2. Senkung der Zeit und Kosten der Datenanalyse durch automatische Parallelisierung und Optimierung von deklarativen Datenanalysespezifikationen
> 3. Schaffung von Technologieimpulsen, die zur erfolgreichen weltweiten Kommerzialisierung von in Deutschland entwickelten, skalierbaren Datenanalysesystemen führen
> 4. Ausbildung von Multiplikatoren im Bereich der Datenanalyse und der skalierbaren Datenverarbeitung, welche die Möglichkeiten von Big Data in Wissenschaft und Wirtschaft tragen werden
> 5. Technologietransfer an Pilotanwendungen in Wissenschaft und Wirtschaft
> 6. Schaffung eines Innovationsklimas durch Konzentration von kritischem Big Data Know-how, damit deutsche Unternehmen und Wissenschaft nicht im Schatten des Silicon Valleys stehen
> 7. Interaktive, iterative Informationsanalyse für Text und Weiterentwicklung geeigneter Geschäftsmodelle zur Schaffung von Marktplätzen für Daten, Datenqualität und Verwertung von Daten
> 8. Datenschutz und Datensicherheit
>
> [MLH+13, S. 107, Abb. 54]

Aus den abgeleiteten Schwerpunkten können mehrere Kriterien für die Betrachtung der Streaming Frameworks in Kapitel 4 und des Referenzmodells in Kapitel 2.2 herangezogen werden. Zunächst werden die gewonnen Kriterien in einer Liste aufgezählt. Anschließend werden die einzelnen Kriterien als Bewertungskriterien für die weitere Untersuchung der Streaming Frameworks definiert. Daraufhin werden die Bewertungskriterien auf das Referenzmodell angewendet.

Liste 3.1 - Bewertungskriterien:

- Architektur
- Prozesse und Threads
- Kommunikation
- Namenssystem
- Synchronisierung
- Pipelining und Materialisierung
- Konsistenz und Replikation
- Fehlertoleranz

- Sicherheit
- Erweiterung
- Qualität

Die ermittelten Bewertungskriterien aus Liste 3.1 unterstützen die Feststellung eines Streaming Frameworks und des Referenzmodells. Damit die einzelnen Bewertungskriterien bei der Anwendung eindeutig und klar sind, werden diese zunächst definiert und zusätzlich erläutert.

Architektur stellt den verwendeten Architekturstil vor und ordnet in eine Systemarchitektur ein.

Prozesse und Threads zeigen die Anwendung von blockierendem oder nicht blockierendem Zugriff, also einer Verbindung zwischen Client und einem Server. Während der Betrachtung wird der Einsatz der verteilten Verarbeitung in der eingesetzten Architektur geprüft.

Kommunikation gibt die Form des Nachrichtenaustauschs zwischen Client und Servern an. Zum Austausch der Nachrichten kommen Nachrichtenprotokolle zum Einsatz. Dabei wird auf die Protokollschicht *Middleware*-Protokoll eingegangen. Im OSI-Modell entspricht die Sitzungs- und Darstellungsschicht einer *Middleware*-Schicht [TvS07, S. 148, Abb. 4.3 angepasstes Referenzmodell]. Dabei werden unterschiedliche Strategien RPC, Warteschlangensysteme, Kontinuierliche Systeme und Multicast Systeme, die beim Nachrichtenaustauschs eingesetzt werden, innerhalb der *Middleware*-Protokolle eingeordnet. Außerdem wird das Verbindungsmodell, die Nachrichtenstruktur und der Einsatz einer Protokollversionierung vorgestellt. Weiterhin wird die Unterstützung von unterschiedlichen Nachrichtenkodierungen und Statusverwaltung betrachtet.

Namenssystem zeichnet den Ansatz eines Benennungssystems. Hierbei wird linear-, hierarchisch- oder attributbasiert klassifiziert.

Synchronisierung beschreibt die verwendeten Algorithmenarten.

Pipelining und Materialisierung gibt eine Technik an, ob komplexe Aggregate berechnet werden und innerhalb der Abfragen wieder benutzt werden können.

Konsistenz und Replikation zeigt die Skalierungstechnik auf und stellt die Verwaltung der Replikation vor.

Fehlertoleranz zeigt das verwendete Fehlermodell und stellt eine Strategie im Wiederherstellungsfall vor.

Sicherheit stellt das Konzept vor und beschreibt den Einsatz von sicheren Kanälen und der Zugriffssteuerung.

Erweiterung beschreibt Methoden weitere Systemarchitekturen anzuschließen.

Qualität zeigt das verwendete Modell für die Dienstgüte.

In der Liste 3.1 wurden Bewertungskriterien vorgestellt und definiert, die nun auf das Referenzmodell aus Kapitel 2.2 angewendet werden. In der Tabelle 3.1 wird eine Übersicht über die Bewertung zum Referenzmodell Aurora Borealis gegeben. Als Architektur wird strukturierte

Peer-to-Peer-Architektur angegeben. In einer dezentralisierten Architektur, wie es die strukturierte Peer-to-Peer-Architektur ist, werden Nachrichten zwischen den Rechnerknoten die auch Peers genannt werden, mit Hilfe von verteilten Hashtabellen ausgetauscht. Dabei übernimmt bei Aurora Borealis ein Knoten den Master bei dem Nachrichten von den Verarbeitungsknoten zurückkommen. So wird in Abbildung 3.2 eine Anwendung gezeigt in der ein Server 1 die Datenverarbeitung auf zwei Datenverarbeitungsservern 2 und 3 ausführen lässt und das Ergebnis aus der Daten- und Verarbeitungsebene an einen Rechner 4 mit der Benutzerschnittstelle zurückschickt. [TvS07, S. 64, Kap. 2.2.2]

Aus der strukturierten Architektur folgt die Frage wie Prozesse untereinander kommunizieren. Die Prozesse kommunizieren entweder lokal oder entfernt asynchron über RPC. Anfragen von entfernten Prozessen werden automatisch lokal übersetzt. Ein Rechnerknoten stellt damit eine vollständige Verarbeitungseinheit dar. Der Prozess muss also gleichzeitig als Client und als Server arbeiten und ist dadurch symmetrisch. [Tea06a, S. 6, Kap. 2]

Kriterium	**Bewertung**
Architektur	Strukturierte Peer-to-Peer-Architektur
Prozesse und Threads	Interaktion symmetrisch
Kommunikation	Transportunabhängiges RPC
Namenssystem	Attributbasierte Benennung
Synchronisierung	Zentralisierter Algorithmus
Pipelining und Materialisierung	In/Out Attribut
Konsistenz und Replikation	Push-basiertes Monitoring
Fehlertoleranz	Replikation
Sicherheit	Nur Verfügbarkeitsprüfung
Erweiterung	Nur Eigenentwicklung
Qualität	Monitor, Optimierer, Voruasberechnen, Lokal und Global

Tabelle 3.1: Bewertung Referenzmodell Aurora Borealis

In der Kommunikation wird ein transportunabhängiges RPC angegeben [Tea06a, S. 7, Abb. 2.1]. So findet der Nachrichtenaustausch zwischen zwei entfernten Rechnern asynchron statt. Die entfernten Rechner entsprechen den Verarbeitungseinheiten. Zwischen einem führenden Rechner und einem entfernten Rechner werden zwei Nachrichten verschickt. Die erste Nachricht führt eine Aktion auf dem entfernten Rechner aus und die zweite Nachricht ist das Ergebnis das vom entfernten Rechner dem führenden Rechner zurück gegeben wird. Hierbei beschreibt Tanenbaum et al. in [TvS07, S. 158, Kap. 4.2.3] den Nachrichtenaustausch als verzögerter synchroner RPC. Das kontinuierliche Verarbeiten von Abfragen wird dabei von einem führenden Rechner der *Middleware*-Schicht übernommen. Dem führenden Rechner dem *Global Load Manager* [Tea06a, S. 28, Kap. 5] wird beim Start des Systems eine *Topology* übergeben. Die *Topology* enthält einen Ausführungsplan mit komplexen Abfragen. Der *Global Load Manager* verwaltet die Auslastung der entfernten Rechner und gleicht hohe Last durch Umverteilung der Aufgaben auf andere Rechner aus. Jeder Verarbeitungseinheit besteht aus einem *Availabilty Monitor* [Tea06a, S. 38, Abb. 7.2] dem Verfügbarkeitsmonitor, einem *Load Manager* der mit dem *Global Load Manager* kommuniziert und einem *QueryProcessor* der die Abfrage ausführt [Tea06a, S. 10, Kap 3.2].

Bei der Anwendung einer Verarbeitung in Aurora Borealis wird eine Konfiguration in einer Extensible Markup Language (XML)-Datei für die Verteilung der Abfragen benötigt. Im Quelltext A.1 wird eine Konfiguration für Zwei Verarbeitungseinheiten formuliert. Einer Verarbeitungseinheit wird die Abfrage *mycount* und der anderen Verarbeitungseinheit die Abfrage *myfilter* zugeordnet. Beide Verarbeitungseinheiten abonnieren den Eingangsstrom *stream Aggregate* und veröffentlichen den *stream Packet* an die angegebene Verteilungseinheit. Für die Abfrage wird eine zustäzlich XML-Datei verwendet. In der Konfiguration A.2 werden die Abfragen *mycount* und *myfilter* für die Zwei Verarbeitungseinheiten definiert. Die Abfrage wird im XML-Tag *borealis* ausgezeichnet. Mit dem XML-Tag *schema* werden komplexe Aurora Borealis Datentypen definiert.

Die Benennung wird durch Attribute gekennzeichnet. Zum Beispiel hat das Schema *PacketTuple* ein Feld mit dem Namen *time* und den primitiven Datentypen *int* in C objektorientierte Programmiersprache (C++). Es werden Sechs Feldtypen (*int*, *long*, *single*, *double*, *string*, *timestamp*) unterstützt [Tea06b, S. 17, Tab. 4.2]. Borealis erzeugt durch die *Marshalling*-Anwendung eine C++-Struktur *struct* vom Typ *TupleHeader* [Tea06b, S. 37, Kap. 5.2.1]. Die *Marshalling*-Anwendung kapselt die komplexe auf *Borealis* spezialisierte Networking, Messaging, Servers, and Threading Library (NMSTL) für C++ [Tea06b, S. 35, Kap. 5.2]. Der Quelltext A.3 zeigt die Methoden der *Marshalling*-Anwendung für die beschriebenen Konfigurationen A.1 und A.2. In der Abfrage der ersten Verarbeitungseinheit wird im XML-Tag *parameter* mit die Aggregatsfunktion *count()* die Anzahl der Pakete jede Sekunde nach Zeit sortiert. Die Tabelle in [Tea06b, S. 23, Tab. 4.5] zeigt eine Übersicht über die möglichen Aggregat-Parameter. Die zweite Abfrage filtert den Ausgabestrom aus der ersten Abfrage nach geraden Zeitwerten und gibt den Ausgabestrom *Aggregate* zurück. Die Abbildung 3.2 zeigt die Ausführung der Kommunikation.

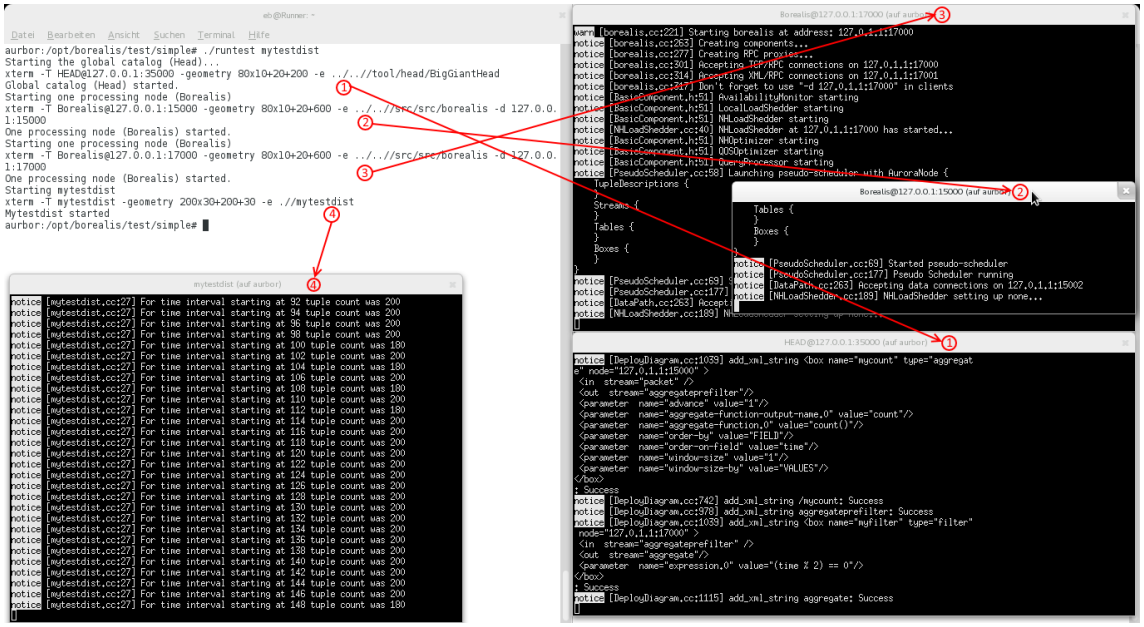

Abbildung 3.2: Aurora Borealis mit einem Master zwei Servern und einem Konsumenten

Pipelining wird durch den Einsatz von Eingangs- und Ausgangsstrom in Abfragen erreicht. In der Konfiguration A.2 wird von der ersten Einheit ein spezialisiertes Datentupel *AggregatePreFilter* erzeugt und die zweite Einheit bezieht das Ergebnis und verändert es. Zusätzlich können über eine *Map*-Funktion in der Abfrage mehrere Datenströme erzeugt und komplex verarbeitet

werden [Tea06b, S. 20, Kap. 4.9.1].

Die Konsistenz in den Verarbeitungseinheiten wird mit dem *Consistancy Manager* erreicht. Durch die zusätzliche Komponente *Availability Monitor* werden Statusinformation zwischen den Einheiten ausgetauscht. Einzelne Verarbeitungseinheiten können repliziert werden. Die Konfiguration der Replikation wird in der Konfiguration A.1 hinzugefügt. Im Gegensatz zum XML-Tag *box* wird bei der Replikation *replica_set* verwendet. Die Abfrage wird ebenfalls dem *replica_set* zugeordnet. Innerhalb des *replica_set* werden einzelne *node*-Elemente mit Zieladresse hinterlegt. Durch die Replikation wird in Aurora Borealis Fehlertoleranz erreicht. [Tea06a, S. 34, Kap. 7]

In der Sicherheit werden keine Sicherheitsrichtlinien vorgestellt und angewendet. Die Kommunikation zwischen einzelnen Rechnern findet unverschlüsselt auf TCP-Ebene über RPC statt. Eine Authentifizierung und Autorisierung wird nicht durchgeführt. Eine leichtgewichtichtete Kontrolle kann durch den *Consistancy Manager* und dem *Availability Monitoring* als Protokollwerkzeuge angesehen werden. Für komplexe Kontrollen ist eine eigene Implementierung notwendig [Tea06a, S. 38, Kap. 7.2.2].

Erweiterungen können durch eigene Entwicklung in den bestehende Quelltext hinzugefügt werden. Methoden für weitreichende Abfragen in andere Umgebungen wie zum Beispiel Python sind nicht vorhanden. Eine umfangreiche Testabdeckung und eine gute Dokumentation für bestehende Methoden sind vorhanden. Das Erstellen von Aurora Borealis wurde bisher nur auf einer älteren Linux-Distribution durchgeführt. Der Quelltext in der letzten Version 2008 ist auf den Linux Compiler Version 3.1.1 angepasst und muss beim Einsatz der aktuellen Compiler-Version aktualisiert werden. Im Anhang A.6 wird eine ausführliche Anleitung zur Installation von Aurora Borealis in der aktuellen Version 2008 mit einer älteren Debian-Distribution vorgestellt. Der Quelltext von Aurora Borealis und die verwendete Debian-Version liegt im Verzeichnis *anhangSoftwareZusatz* bei. Eine lauffähige virtuelle Maschine steht ebenfalls im gleichen Verzeichnis bereit.

Die Qualität der Dienste wird in Aurora Borealis durch verschiedene Mechanismen erreicht. Lokal werden pro Rechnereinheit mit dem *Local Monitor* Statuswerte von Central Processing Unit (CPU), Festplatte, Bandbreite und Energieversorgung erfasst und an den globalen *End-point Monitor* übertragen. Der *End-point Monitor* wertet die Qualität des Dienstes aus und führt eine Statistik pro Erfassung. Optimiert wird lokal durch den *Local Monitor* mit dem *Local* und *Neighborhood Optimizer* und global durch den *Global Optimizer*. Probleme werden durch die Monitore erkannt. Da jedem Datentupel ein *Vector of Metrics* dazugeschaltet ist und es möglich ist zusätzlich einen *Vector of Weights* (Lifetime, Coverage, Throughput, Latency) dazuzuschalten, ist eine Berechnung der Ursache eines QoS-Problems möglich. [AAB+05, S. 7, Kap. 5]

In diesem Kapitel wurden die Vier Vs in *Big Data* vorgestellt und die Streaming Frameworks wurden darin in einem Vergleich zu relationalen Datenbanksystemen eingeordnet. Weiterhin wurden die Konsistenz und die Verfügbarkeit im Zusammenhang von CAP und BASE vorgestellt. Mit der Studie [MLH+13] und [TvS07] wurden Bewertungkriterien in der Liste 3.1 erarbeitet. Abschließend wurde ausgehend von den Bewertungskriterien das Referenzmodell Aurora Borealis ausgewertet. In Kapitel 4 werden nun die Streaming Frameworks vorgestellt und mit Bewertungskriterien in Liste 3.1 ausgewertet.

Kapitel 4

Vorstellung Streaming Frameworks

In Kapitel 2 und 3 wurden Grundlagen geschaffen, Bewertungskriterien wurden vorgestellt, eine Anwendung des Referenzmodells Aurora Borealis mit den Bewertungskriterien erläutert und für die Anwendung auf die Streaming Frameworks vorbereitet. In den folgenden Unterkapiteln werden die einzelnen Streaming Frameworks Apache Storm, Apache Kafka, Apache Flume und Apache S4 vorgestellt. Jedes Unterkapitel beginnt zuerst mit einer Übersicht über das Streaming Framework. Anschließend wird kurz auf die Entstehung des Streaming Frameworks bis zum Zeitpunkt der Erstellung dieser Thesis eingegangen. Nach der kurzen Übersicht, werden die Bewertungskriterien aus Liste 3.1 auf das Streaming Framework angewendet. Dabei wird wie in der Anwendung des Referenzmodells vorgegangen.

4.1 Apache Storm

Apache Storm wird vom Hauptentwickler Nathan Marz im Proposal als verteiltes, fehlertolerantes und hochperformantes Echtzeitberechnungssystem definiert. Ursprünglich wurde die Anwendung von der Firma Backtype in 2011 entwickelt. Im gleichen Jahr wurde die Firma Backtype von Twitter übernommen und der Quelltext auf Github [Inc14a] unter dem Repository *storm* [Mar14a] von Nathan Marz veröffentlicht. In 2013 wurde die Aufnahme von Storm in die Apache Software Foundation (ASF) geplant. Dazu wurde ein Storm Proposal von Nathan Marz eingereicht. [Mar13]

Seit 2013 befindet sich Storm im Apache Incubation-Prozess [Fou13]. Eine Überführungsversion 0.9.1-incubating wurde dafür eingerichtet. Der Quelltext und das Lizenzmodell wird in die ASF aufgenommen [Fou14c]. Der Verlauf des Überführungsprozesses zur ASF wird auf der Incubator-Statusseite [Fou14n] festgehalten. In der Tabelle 4.1 wird eine Kurzübersicht über Apache Storm gegeben. Darin wird ein aktiver Entwicklungsstatus angegeben. Die Aktivität wird aus dem GitHub *Contributors-Graph* bei 84 Projektteilnehmern bestimmt [Inc14b]. Zur Entwicklung von Apache Storm werden mehrere Sprachen Clojure, Java und Python verwendet. Nathan Marz gibt an Storm in der Programmiersprache Clojure [Hic14] zu entwickeln und mit Java [Cor14a] kompatibel zu sein, neben Java und Clojure findet die Github Sprachen-Suche [Mar14b] im Repository *storm* auch Python [Fou14q]. Ab Version 0.9.1-incubating wird eine verbesserte Plattformkompatibilität zum Betriebssystem Microsoft Windows angeboten und die Standardtransportschicht ZeroMq [iC14] durch Netty [Lee14] ersetzt [Con14].

Faktum	**Beschreibung**
Hauptentwickler	Nathan Marz
Stabile Version	0.9.1-incubating vom 22.02.2014
Entwicklungsstatus	Aktiv
Entwicklungsversion	0.9.2-incubating, 0.9.3-incubating
Sprache	Clojure, Java, Python
Betriebssystem	Platformübergreifend (Microsoft Windows mit Cygwin Umgebung)
Lizenz	Eclipse Public License 1.0 (Incubating Apache License version 2.0)
Webseite	[Mar14c]
Quelltext	[Mar14b]

Tabelle 4.1: Kurzübersicht Apache Storm

In Tabelle 4.2 werden die Bewertungskriterien aus Kapitel 3 in Apache Storm geprüft. Als Architektur wird die moderne Systemarchitektur Strukturierte Peer-To-Peer-Architektur, die eine horizontale Verteilung unterstützt, angegeben. Apache Storm besteht aus drei Komponenten: *Nimbus*, *Supervisor* und *UI*. Der *Nimbus* stellt die zentrale Stelle und übernimmt die Aufgabe des *Scheduler* - einem Arbeitsplaner. Der *Nimbus* ist klein gehalten und verteilt die Aufgaben zwischen den Arbeitsknoten. Die Arbeitsknoten werden in Apache Storm *Supervisor* genannt. Mehrere Supervisor-Instanzen sind in einem Apache Storm Cluster möglich. Die dritte Komponente *UI* visualisiert den momentanen Status der Apache Storm Komponenten *Nimbus* und *Supervisor*.

Bei der Verarbeitung von Informationen kann in Apache Storm pro Verarbeitungseinheit die Anzahl an benötigten Threads als Argument explizit übergeben werden. Die Konfiguration dazu findet im Quelltext statt. Um eine komplexe Verarbeitung durchzuführen, muss in Apache Storm eine *Topology* implementiert und veröffentlicht werden. Die *Topology* wird auf dem Apache Storm Cluster permanent ausgeführt und kann nicht dynamisch verändert werden. Die Kommunikation erfolgt zwischen den einzelnen Apache Storm Komponenten mit einem zusätzlichen Werkzeug, dem sogenannten Apache ZooKeeper [Fou14e]. Apache ZooKeeper wird als verteilte Synchronisation und Koordination der Aufgaben durch Nimbus auf tieferer Ebene verwendet. Auf der Transportebene kommunizieren Verarbeitungseinheiten durch das asynchrone Client-Server-Framework Netty [Lee14].

Eine komplexe Verarbeitung bzw. Abfrage in einer *Topology* besteht aus *Spouts* und *Bolts*. Die Kommunikation ist dabei einseitig. Ein Empfänger-*Bolt* kann keine Nachricht an einen Sender-*Bolt* zurück schicken. Mit einem *Spout* wird eine externe Datenquelle beschrieben und ein *Spout* liefert eine permanente Folge von ungebundenen Tupeln. Ein Tupel ist die Hauptdatenstruktur und kann unterschiedliche Datentypen (integers, longs, shorts, bytes, strings, doubles, floats, booleans und selbstentwickelte) enthalten. Die Folge von ungebundenen Tupeln wird in Apache Storm als *Stream* bezeichnet. Um einen *Spout* zu implementieren, reicht es die Schnittstelle *IRichSpout* zu implementieren oder die Klasse *BaseRichSpout* zu erweitern.

Bei einer Erweiterung von *BaseRichSpout* sind mindestens die Methodenamen *open*, *nextTuple* und *declareOutputFields* zu implementieren. In der Methode *open* kann zum Beispiel eine interne Liste über einen Java-Listener bei einem Dateneingang in einem Datenadapter gefüllt

werden. Die Methode *nextTuple* wird in jeder ersten Millisekunde ausgeführt und wenn Nachrichten in der Liste enthalten sind, kann der *SpoutOutputCollector* einen Stream mit einer eindeutigen StreamId und einem Tuple aussenden. Wenn die Nachricht nicht vollständig übertragen wurde, wird über eine *Callback-Methode* ein *ack* oder ein *fail* in der implementierten Klasse *Spout* zurückgegeben. Damit wird in Apache Storm sichergestellt, dass die Nachricht mindestens einmal vollständig verarbeitet wurde [Fou14f]. In der Methode *declareOutputFields* wird die Felddefinition der Ausgabe für die weitere Verarbeitung angegeben. [Fou14j]

Ein *Bolt* hingegen nimmt ein Tupel auf und gibt Tupel wieder aus. Innerhalb eines *Bolt* können Tupel verändert werden. Nachdem Start der *Topology* wird ein *Bolt* erst auf den *Supervisor* übertragen und deserialisiert. *Nimbus* ruft auf der Instanz anschließend die Methode *prepare* auf. In Java muss für die Implementierung eines *Bolt*, die Schnittstelle *IBolt* oder *IRichBolt* implementiert werden. Alternativ können auch Basisimplementierungen verwendet werden wie zum Beispiel *BaseBasicBolt* oder *BaseRichBolt*. Mit der Methode *execute* werden die Tupel angepasst und über den *OutputCollector* ausgesendet. Apache Storm erwartet beim Eingang eines Tupels in einem *Bolt* eine Bestätigung über die Methode *ack* oder *fail*. Andernfalls kann Apache Storm nicht feststellen, ob eine Nachricht vollständig verarbeitet wurde. [Fou14h]

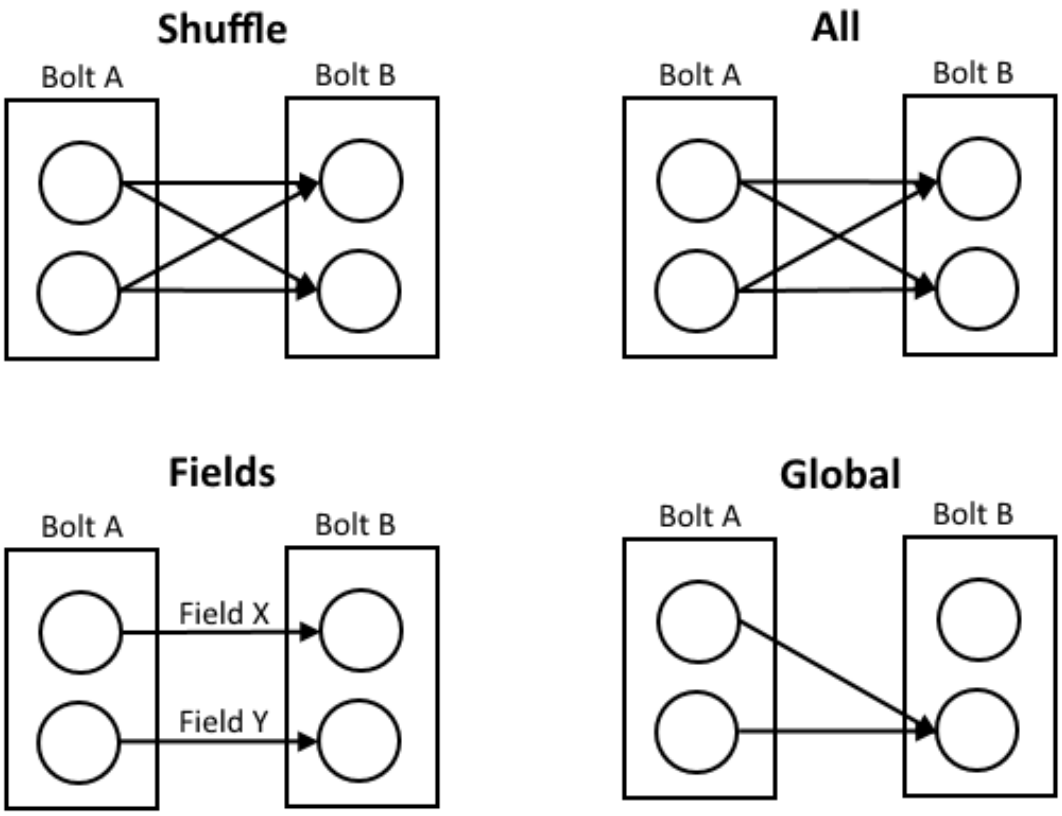

Abbildung 4.1: Apache Storm Gruppierungen

Durch den *TopologyBuilder* kann eine komplexe Abfrage aus *Spouts* und *Bolts* zusammengesetzt werden. Der *TopologyBuilder* stellt dazu *set*-Methoden für *Spouts* und *Bolts* bereit. Bei dem Setzen eines *Spout* oder eines *Bolt* muss immer eine Referenz-Identifikationsnummer angegeben werden. Durch die Referenz können *Bolt*- oder *Spout*-Komponenten untereinander verbunden werden. Weiterhin kann mit dem Argument *parallelism_hint* die Anzahl der *Tasks* eingestellt werden, die zur Ausführung benutzt werden. Jeder *Task* wird im Storm Cluster auf ei-

nem eigenen *Thread* ausgeführt. Sobald die *setBolt*-Methode aufgerufen wird, wird ein Objekt *InputDeclarer* erzeugt. Darin können verschiedene Gruppierungen (*Shuffle*, *Fields*, *All*, *Global*, *None*, *Direct*, *LocalOrShuffle* [Fou14i]) angegeben werden, um den Stream in definierte Teile zu trennen. In Abbildung 4.1 werden Vier Standardgruppierungen in Apache Storm gezeigt. Mit einem *ShuffleGrouping* werden Tupel eines *Stream* zufällig über die *Tasks* verteilt. Beim *FieldsGrouping* wird der *Stream* durch die Angabe eine Schlagworts getrennt. Tupel mit dem gleichen Schlagwort werden immer an den gleichen *Task* gesendet. Das *AllGrouping* wird auf allen Tasks des *Bolt* repliziert und beim *GlobalGrouping* der *Stream* zu einem *Task* gesendet. Weiterhin wird das *NoneGrouping* bei dem der *Stream* auf demselben *Task* ausgeführt wird und beim *DirectGrouping* durch den *Stream*-Erzeuger entschiedn, welchen Konsumenten-*Task* der *Stream* gesendet wird. Durch die Schnittstelle *CustomStreamGrouping* wird eine weitere konkrete Implementierung für eine *Grouping*-Strategie ermöglicht. [Fou14g]

In Apache Storm wird eine Abstraktion *Trident* für eine transaktionsorientierte Abfrage- und Datenverarbeitung bereitgestellt. Mit *Trident* ist es möglich eine Stapelverarbeitung mit Statusinformationen durchzuführen. Es gibt Fünf Ausführungstypen: *Partition-local*, *Repartitioning*, *Aggregation*, *GroupedStreams* und *Merges and Joins*. In [Fou14p] wird *Trident* näher erläutert. In den folgenden Absätzen wird kurz auf die möglichen Funktionen mit *Trident* aus [Fou14p] eingegangen.

Unter *Partition-local* werden Operatoren (*function*, *filter*, *partitionAggregate*, *partitionPersist*, *projection*) lokal in einem Stapelelement ausgeführt. Die Operatoren *function* und *filter* erben jeweils von der gleichen Basisklasse *BaseFunction*. Bei den Methoden *partitionAggregate* und *partitionPersists* können unterschiedliche Strategien entwickelt werden. Ein neues Aggregat kann mit der *Aggregator*-Schnittstelle oder den erweiterten Schnittstellen *CombinerAggregator* oder *ReducerAggregator* implementiert werden. Um nicht im bestehenden Arbeitsspeicher mit der *MemoryMapState.Factory()* Daten zu speichern, kann über eine konkrete Implementierung der Schnittstelle *IBackingMap* eine neue Strategie zur Datenablage erzeugt werden. Mit *Projection* können Teile der Felder aus einem *Stream* in einem neuen *Stream* unverändert abgebildet werden.

Beim *Repartitioning* kann die Stapelverarbeitung durch *Repartitioning*-Funktionen (*shuffle*, *broadcast*, *partitionBy*, *global*, *batchGlobal*, *partition*) geändert bzw. neu strukturiert werden. Zum Beispiel kann sich die Anzahl der *Tasks* zur parallelen Datenverarbeitung ändern. Mit dem *Repartitioning* ist es möglich die *Tasks* im Cluster neu zu verteilen. Die Methode *groupBy* nutzt *Repartitioning*, um den *Stream* mit *partitionBy* neu zu strukturieren. Gruppierte und aggregierte *Streams* können mit bestehenden Apache Storm Primitiven verkettet werden.

Streams können in *Trident* durch die spezielle *TridentTopology* zusammengeführt werden. Die *TridentTopology* bietet dazu die Methoden *merge* und *join* an. Beide Methoden erzeugen jeweils einen neu kombinierten *Stream*. Eine Zusammenführung in einem Zeitfenster, dem *Windows Join*, kann mit Hilfe von *partitionPersist* und *stateQuery* durchgeführt werden. Mit *partitionPersist* wird ein *Stream* nach der Identität, die im *join* referenziert wird, zerlegt und in einem Statusstapel mit der Methode *makeState* in der *TridentTopology* ein Status erzeugt. Der neue *Stream* steht dadurch permanent für eine Stapelverarbeitung über das *stateQuery* durch *lookup*-Abfragen auf die Identität bereit.

In der Archivdatei des Quelltextes von Apache Storm [Mar14b] ist im Unterverzeichnis *Example* eine Beispielanwendung *WordCountTopology.java* vollständig hinterlegt. Im Anhang A.2 wird dazu ein Beispiel-Quelltext A.4 zum Wortzählen ausgegeben. Der Quelltext stellt einen Auszug des Java-Projekts *storm-starter* aus dem gleichen Verzeichnis dar. In der Methode *main* wird

die *Topology* mit einem *RandomSentenceSpout* und Zwei *Bolts*, einem *SplitSentence* und einem *WordCount* erzeugt. Der *RandomSentenceSpout* bekommt Fünf *Tasks* zugeordnet und erhält die Identität „spout“. Der *Bolt SplitSentence* ist eine *ShellBolt*, in dem das Teilen des Satzes in einzelne Worte in der externen Programmiersprache Python über die Kommandozeile angestoßen wird. Beim ersten *Bolt* wird *SplitSentence* auf den *Stream* „spout“ mit Acht *Tasks*, einem *ShuffleGrouping* und der Identität „split“ angewendet. Der zweite *Bolt* nimmt den *Stream* „spout“ aus dem ersten *Bolt* auf und zählt die Worte im *Bolt WordCount*, mit Zwölf *Tasks*, einem *FieldGrouping* und der Identität „count“. Das *FieldGrouping* von *WordCount* wird nach dem Feld „word“ gruppiert und der Ausgabestrom enthält nach einer Verarbeitung die Anzahl eines Wortes als Schlüssel-Wert-Paar. Abschließend prüft eine Bedingung, ob die *Topology* im *local cluster mode* zum Debuggen, dem Fehlerlösen durch Haltepunkte, auf dem lokalen Rechner oder in einem Storm Cluster ausgeführt werden soll.

Eine *Topology* hat spezielle *acker-Tasks*, die die Verarbeitung der *Bolts* und *Spouts* überwachen. Wenn ein Abfragegraph vollständig abgearbeitet wurde, wird an das auslösende *Spout* vom *acker-Task* die Nachricht gesendet. In dem *Spout* wird die *Callback*-Methode *ack* oder *fail* anschließend verarbeitet. Zusätzlich wird in der *Storm UI* die Anzahl der Nachrichten zu *ack* und *fail* in der Summe dargestellt. Bei Ausfall eines Tasks bekommt der Wurzelknoten eines Abfragegraphens ein „timeout“ und der Abfragegraph wird „replayed“, also erneut abgearbeitet. Bei Absturz eines Spout Tasks muss das Warteschlangensystem dafür sorgen, sobald der Spout Task wieder verfügbar ist, die Nachrichten in der Quelle bereitzustellen. [Fou14m]

Kriterium	**Bewertung**
Architektur	Strukturierte Peer-to-Peer-Architektur
Prozesse und Threads	Client-Server Cluster
Kommunikation	TCP-basiert mit Apache Zookeeper
Namenssystem	Hierarchische Benennung
Synchronisierung	Zentralisierter Algorithmus
Pipelining und Materialisierung	Methodenverkettung
Konsistenz und Replikation	Reliability Algorithmus
Fehlertoleranz	Fail-Fast Strategie unter Supervision
Sicherheit	Nur eigene Maßnahmen
Erweiterung	Eigenentwicklung und Community-Beiträge
Qualität	Guaranteeing message processing

Tabelle 4.2: Bewertung Apache Storm

In Apache Storm können der *Worker*, die *Node*, der *Nimbus*- oder *Supervisor*-Dienst abstürzen. Der *Worker* wird vom *Supervisor* neugestartet falls der *Worker* einen Fehler hat. Der *Nimbus* leitet die Abfrage an eine andere Maschine, wenn der *Supervisor* den *Worker* nicht neugestartet bekommt. Sobald eine Node ausfällt, startet *Nimbus* die *Tasks* auf einer anderen Maschine. Da beide Dienste in einem Fehlerfall schnell ausfallen und keinen Statusmonitor für einen Ausfall anbieten, ist ein externes *Monitoring* (nagios [Gal14]) und eine *Supervision* (supervisord [McD14]) der Anwendungsdienste *Nimbus* und *Supervisor* notwendig. Nachdem der Neustart durch den *Nimbus*- oder dem *Supervisor*-Prozess erfolgte, können neue *Tasks* zu den laufenden *Tasks* auf den *Workern* hinzugefügt werden. [Fou14l]

Die Sicherheit unter Apache Storm wird bisher nur durch Einsatz zusätzlicher Administration des Anwendungssystems erreicht. Auf der Netzwerkebene müssen Internet Protocol (IP)-Verbindungen mit Internet Protocol Security (IPsec) gesichert werden. Eine Authentifizierung zwischen den Apache Storm Komponenten und dem Apache Zookeeper ist nicht vorhanden. Daher können auf der Anwendungsebene spezifische Richtlinien durch Access Control Lists (ACLs) umgesetzt werden. [Fou14o]

Da der Quelltext der Hauptanwendung öffentlich ist, können spezifische Implementierungen in einem separaten Repository hinzugefügt werden. Apache Storm gibt eine Garantie auf Nachrichtenverarbeitung, wie in [Fou14m] gezeigt. Ein QoS kann durch die Unterstützung des *Monitoring* iterativ entwickelt werden. Ein komplexes QoS-System ist in Apache Storm nicht vorhanden.

Apache Storm wurde zu Beginn kurz vorgestellt. Anschließend wurden die Kriterien aus Kapitel 3 eingeführt und die auf Apache Storm angewendet. Es wurden spezifische Begriffe in Apache Storm vorgestellt und in einer Anwendung WordCount gezeigt. Das Erstellen einer Anwendung ist in wenigen Klassen erledigt. Das Debuggen lässt sich in einem lokalen Clustermodus erledigen. Das Veröffentlichen einer Anwendung ist in einem Kommandozeilenaufruf ausgeführt. Die Sicherheit und QoS sind weniger bis nicht vorhanden. Trotzdem lässt sich ein Cluster mit wenig Konfigurationsaufwand aufbauen und vertikal durch Apache Zookeeper skalieren. Im folgenden Kapitel wird Apache Kafka eingeführt.

4.2 Apache Kafka

Nach der Vorstellung von Apache Storm wird in diesem Kapitel Apache Kafka näher beleuchtet. Zu Beginn erfolgt eine Kurzübersicht gegeben, um anschließend die Bewertungskriterien zu erläutern.

Apache Kafka wird von Rao in [Rao11] als verteiltes publish-subscribe Nachrichtensystem für die Verarbeitung hoher Mengen an fließenden Daten bezeichnet. Am 04.07.2011 wurde der Apache Incubation-Prozess aufgenommen und am 23.10.2012 wurde Apache Kafka qualifiziert [Fou12]. Ursprünglich wurde Apache Kafka von der Firma LinkedIn [Lin14] verwendet, um auf die eingehenden unterschiedlichen hohen Datenmengen der Webseiten von LinkedIn Zugang zu bekommen und zu verarbeiten [Rao11].

Die Architektur von Apache Kafka besteht aus einem Kafka Server, den *Producern* und den *Consumern*. Der Server stellt als *Broker* die Verbindungen zwischen einem *Producer* und einem *Consumer* her. In einem *Broker* werden *Topics* registriert. Ein spezifischer Nachrichtenstrom kann durch Angabe eines *Topic* bereitgestellt oder abgefragt werden. Der *Producer* hält eine Liste von Verbindungen zu *Brokern*. Nachrichten werden von *Producern* an *Broker*, wie in einem *Push*-System ‚gesendet. Bei Absturz eines *Brokers* wird ein bestehender *Broker* zum *Master* gewählt, der zuerst aus einer Anfrage auf Aktivität antwortet. Ein *Consumer* zieht Nachrichten von einem *Broker*, wie in einem *Pull*-System. Aktives Warten auf Dateneingang in einem „long poll“ und einem permanenten Abfragen eines *Brokers* durch den *Consumer*, kann durch Übergabe von Parametern in der *Consumer*-Abfrage blockiert werden. Der Nachrichtenstrom stellt in einem *Consumer* eine *Iterator*-Schnittstelle bereit. Sobald Nachrichten eintreffen, können weiterführende Operationen ausgeführt werden. Um eine Last zu verteilen, kann ein *Topic* in Partitionen aufgeteilt werden. Eine Nachricht besteht aus einem Schlüssel und einem Wert. Der Nachrichtenschlüssel kann je nach Strategie beispielsweise per Zufall

Faktum	Beschreibung
Hauptentwickler	Jay Kreps, Neha Narkhede, Jun Rao
Stabile Version	0.8.1.1 vom 29.04.2014
Entwicklungsstatus	Aktiv
Entwicklungsversion	0.8.2, 0.9.0
Sprache	Scala, Java, Python
Betriebssystem	Platformübergreifend (Microsoft Windows mit Cygwin Umgebung)
Lizenz	Apache License version 2.0
Webseite	[KNR14a]
Quelltext	[KNR14b]

Tabelle 4.3: Kurzübersicht Apache Kafka

auf bestimmte Partitionen zugeordnet werden. Partitionen sind auf Host-Maschinen in einem Kafka-Cluster verteilt. Ein *Topic* sollte in einem produktiven Einsatz die gleiche Anzahl an *Threads* wie Partitionen haben. Bei geringerer Anzahl von *Threads* als *Topics*, entstehen Wartezeiten für *Topics*. *Topics* können in diesem Fall die Arbeit nicht unmittelbar starten, sondern müssen auf einen *Thread*, der bereit wird, warten. Mit Kommandozeilen-Werkzeugen kann ein *Rebalance* der Partitionen von *Topics* in einem Kafka-*Cluster* angestoßen werden. [KNR11, S. 2, Kap. 3]

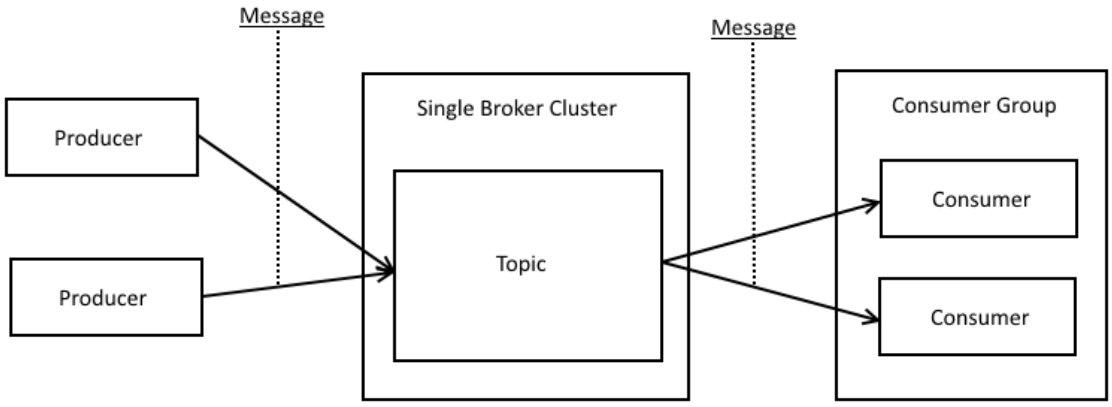

Abbildung 4.2: Apache Kafka Architektur - Single Broker Cluster

Abbildung 4.2 zeigt ein Beispiel mit einem *Single Broker Cluster* als Server. Das Kafka-Cluster kann auch aus mehreren *Brokern* bestehen. Die Installationsanleitung in Anhang A.8 zeigt eine Konfiguration für ein *Single Broker Cluster*. Für ein *Multi Broker Cluster* werden separate Konfigurationen der Kafka-Maschinen und ein Apache Zookeeper-Cluster benötigt. In der Abbildung 4.2 werden Nachrichten an das *Topic* von Zwei *Producern* gesendet. Aus der *Consumer Group* holen Zwei *Consumer* Nachrichten aus dem *Topic* ab. Für die Koordination der Nachrichten greifen der *Server*, die *Producer* und die *Consumer* im Hintergrund auf Apache Zookeeper zu. Für die horizontale Skalierung kann ein Apache Zookeeper Cluster genutzt werden. In einem Kafka-Cluster können maximal 255 Knoten existieren. Pro Konfiguration muss jeder Knoten eine eigene Identität unter der *Broker-Id* festgelegt bekommen. Mehrere Knoten

mit gleicher Identität führen zu einem unvorhersagbaren *Cluster*-Verhalten. [Gar13, S. 28]

Sobald Nachrichten von *Consumern* in einem Kafka-Cluster empfangen werden, werden diese lokal innerhalb einer Apache Zookeeper-Maschine im Dateisystem gespeichert. In einem *Consumer* werden die Nachrichten mit einem iterativen Zähler im *Offset* gespeichert. Durch die Angabe des *Offset* ist es möglich, Nachrichten von einer *Topic*-Partition ab einer bestimmten *Offset*-Position abzuholen. Beim Speichern der Nachrichten nutzt Apache Kafka die *zero-copy*-Optimierung [PN08]. Dabei wird die Nachricht in den Linux Page Cache einmalig geschrieben. Weitere Abfragen der Nachrichten werden vom Page Cache geliefert. Durch die zero-copy-Optimierung werden 4 context-switches pro Abfrage in einem Prozess reduziert. [Fou14k, Kap. 4.3]

In Kafka wird die Reihenfolge von Nachrichten, die von einer *Topic*-Partition abgeholt wird, garantiert. Die Reihenfolge von unterschiedlichen Partitionen kann allerdings abweichen und wird nicht garantiert. In Apache Kafka können Nachrichten mit der *Gzip*[1]-Anwendung oder der *Snappy*[2]-Bibliothek komprimiert werden. Nachrichten werden von Kafka nach einer Verarbeitung in einem *Consumer* nicht gelöscht. Durch das einstellbare Service Level Agreement (SLA) ist das Löschen von Nachrichten nach einer definierten Zeitspanne möglich. Durch diese Technik ist es in einem *Consumer*-Ausfall möglich, Nachrichten mit einem neuen bzw. bestehenden *Consumer* erneut abzufragen, also Nachrichten neu einzuspielen. Dennoch sind bei der Übernahme Nachrichten-Duplikate möglich. Eine Anwendung, die Kafka einsetzt und mit Duplikaten umgeht, muss eine Logik zur Erkennung von Duplikaten bereitstellen. So wird beim Einsatz von mehreren *Consumern* die Datenkapazität vervielfacht. Nachrichten gehen verloren, falls ein *Broker* mit nicht abgeholten Nachrichten ausfällt. [KNR11, S. 4, Kap. 3.3]

Aus dem Apache Kafka Archiv [KNR14b] wurde ein Beispiel für die Verwendung von *Producer* und *Consumer* im Anhang unter dem Quelltext A.5 und A.6 abgelegt. Beide Klassen erben von der Java *Thread*-Klasse. In einer externen Klasse können beide erweiterten *Threads* instanziiert und mit der *start*-Methode ausgeführt werden. Im *Producer*-Quelltext wird innerhalb der *run*-Methode in einer Schleife eine Nachricht dauerhaft an einen übergebenen *Topic* gesendet. Im *Consumer*-Quelltext wird ebenfalls in der *run*-Methode über ein *Mapping* der *KafkaStream* für das übergebene *Topic* geholt und über den *ConsumerIterator*, solange Nachrichten eintreffen, die Nachrichten in der Konsole ausgegeben. In beiden Implementierungen wird zuvor eine Konfiguration für das Kafka-Cluster gesetzt. Im Quelltext A.5 und A.6 liegt eine hierarchische Benennung vor. Die Klasse KafkaStream liegt z.B. im Namensraum „kafka.consumer.KafkaStream“. Die Hierarchie wird durch den Punkt abgetrennt. Spezifiziert wird von links nach rechts. Beim Synchronisieren zwischen Producer und Consumer werden über Apache Zookeeper Watcher Listener Konfigurationen aktualisiert.

Apache Kafka kann *Topic*-Partitionen replizieren. Mit einem *Replication*-Faktor kann die Anzahl der Replikate in der Konfiguration für einen *Topic* eingestellt werden. Das erste registrierte In-sync Replica (ISR) bekommt die führende Rolle. Weitere Replikate übernehmen den Status des *Follower*, dem Folgenden. Falls der führende ISR abstürzt, wird durch den Algorithmus *PacificA* [LYZZ08] aus den *Followern* der nächste führende ISR bestimmt. [Fou14k, Kap. 4.7]

Da Apache Kafka auf einem Publish-Subscribe-Verfahren aufbaut ist die Wiederbenutzung bzw. Weitergabe von Nachrichten nur unter Angabe eines weiteren *Topic* möglich. Dafür sind mindestens ein weiterer *Publisher* und *Consumer* zu implementieren. Aggregatoren und Operatoren, sowie es unter dem Referenzmodell Aurora/Borealis angeboten wird, kann unter Apache

[1] Kompressionswerkzeug: gzip – http://www.gzip.org/

[2] Kompressionsbibliothek: snappy – https://code.google.com/p/snappy/

Kriterium	Bewertung
Architektur	Strukturierte Peer-to-Peer-Architektur
Prozesse und Threads	Client-Server Cluster, Actice Push-Pull-Modell
Kommunikation	TCP-basiert mit Apache Zookeeper
Namenssystem	Hierarchische Benennung
Synchronisierung	Apache Zookeeper Watcher
Pipelining und Materialisierung	Publishing als Consumer
Konsistenz und Replikation	Replikation
Fehlertoleranz	Fail-Fast Strategie unter Supervision
Sicherheit	Nur eigene Maßnahmen
Erweiterung	Eigenentwicklung und Community-Beiträge
Qualität	At-least-once delivery, time-based SLA 7 Tage

Tabelle 4.4: Bewertung Apache Kafka

Kafka in der Hochsprache Scala oder Java in einem Producer entwickelt werden. Auch in der Sicherheit fehlen noch Anforderungen zur Authentifizierung und Verschlüsselung innerhalb des Kafka-Clusters [Kre14]. Erweiterungen für das Monitoring werden über Java Management Extensions (JMX) angeboten. Eine Integration in ein Monitoringsystem wie Nagios kann mit dem Java JMX Nagios Plugin[3] erfolgen.

In diesem Kapitel wurde die Installation und eine Beispielanwendung für den Austausch von Nachrichten gezeigt. Auf spezielle Eigenschaften wie das zero-copy [PN08], Parallelisierung und Replikation wurde eingegangen. Außerdem wurden die Bewertungskriterien aus Tabelle 4.4 für Apache Kafka erläutert. Im nächsten Kapitel wird nun Apache Flume vorgestellt.

4.3 Apache Flume

Nachdem Apache Storm und Apache Kafka bewertet wurden, wird als nächstes Apache Flume vorgestellt. Apache Flume wurde ursprünglich von Jonathan Hsieh und der Firma Cloudera im Jahr 2009 entwickelt und wird als ein verteiltes, zuverlässiges und verfügbares System für effizientes Sammeln, Aggregieren und Bewegen großer Datenmengen von Protokolldaten aus verschiedene Quellen zu einem Zentralen Datenspeicher beschrieben [Ver11]. Am 29 Juni 2010 wurde Apache Flume unter der Apache License Version 2.0 veröffentlicht und am 20 Juni 2012 in die Apache Software Foundation überführt [Flu12b]. Nachdem Apache Flume am 13 Juni 2013 in den Apache Incubations Prozess überführt wurde, wurde nach Version 0.9.5 in der neuen Fassung ab Version 1.0.0-incubating eine weitreichende Refaktorierung[4] durch Arvind Prabhakar, Prasad Mujumdar und Eric Sammer mit der Unterstützung von Jonathan Hsieh, Patrick Hunt und Henry Robinson durchgeführt [Sam12]. Die Abkürzung Next Generation

[3] Java JMX Nagios Plugin: check_jmx – http://exchange.nagios.org/directory/Plugins/Java-Applications-and-Servers/check_jmx/details

[4] Refaktorierung ist ein Prozess in der Software-Entwicklung, um die interne Struktur zu verbessern, während das äußere Verhalten unverändert bleibt [FBB+99, S. 9].

Faktum	**Beschreibung**
Hauptentwickler	Arvind Prabhakar, Prasad Mujumdar, Eric Sammer
	Jonathan Hsieh, Patrick Hunt, Henry Robinson
Stabile Version	1.5.0.1 vom 16.06.2014
Entwicklungsstatus	Aktiv
Entwicklungsversion	1.6.0
Sprache	Java
Betriebssystem	Linux/Unix konform, kein Support für Windows
Lizenz	Apache License version 2.0
Webseite	[PMS14a]
Quelltext	[PMS14b]

Tabelle 4.5: Kurzübersicht Apache Flume

(NG) in der neuen Version von Apache Flume steht für die Weiterentwicklung und der Refaktorierung [WRS12], [Sam11]. In dieser Arbeit wird ausschließlich die neue Fassung der Apache Foundation ab Version 1.0.0-incubating vorgestellt. In der Tabelle 4.5 wird eine Kurzübersicht über Apache Flume gezeigt. Dabei werden unter den Hauptentwicklern die ersten drei Entwickler der neuen Fassung Flume-NG und abschließend die drei Entwickler aus der ursprünglichen Fassung aufgelistet. Da die Online-Dokumentation von Apache Flume teilweise mit der Application Programming Interface (API) Version 1.5.0 nicht übereinstimmt, werden bei definierten Methoden auf die Dokumentation der API verwiesen.

Apache Flume wurde als allgemeines Werkzeug eines Datenlieferanten für Apache Hadoop[5] entwickelt. Daher wird in den Bibliotheken von Apache Flume, eine Anbindung an das Apache Hadoop Dateisystem HDFS als *HDFS Sink* bereitgestellt. Dennoch sind weitere Sink-Implementierungen gegeben und möglich. In dieser Arbeit steht der Fokus in der kontinuierlichen Datenverarbeitung, weshalb die Schnittstelle zu Apache Hadoop nicht näher beleuchtet werden kann. [Hof13, S. 1]

Die Architektur von Apache Flume besteht aus mehreren einzelnen Maschinen, die als *Agents* bezeichnet werden. Jeder *Agent* wird über eine Konfigurationsdatei eingerichtet. Die Konfigurationsdatei kann während dem Produktivbetrieb automatisch oder manuell aktualisiert werden. Der *Agent* prüft die Laufzeitkonfiguration jede 30 Sekunden und aktualisiert diese, sobald eine Änderung in der Konfigurationsdatei stattfindet. Ein *Agent* besteht immer aus einer Quelle *Source*, einem Kanal *Channel* und einer Ausgabe *Sink*. Zwischen der *Source*, dem *Channel* und dem *Sink* werden Nachrichten *Flume events* ausgetauscht. Ein *Flume event* besteht aus dem Kopfbereich *Header* und einem Datenbereich *Body*. Der *Header* ist ein Schlüssel/Wert-Tupel, in dem während der Verarbeitung Metadaten angereichert werden können. Im Header werden die Metadaten im Klartext und im Body binär übertragen. In der Binärübertragung

[5] Apache Hadoop ist eine Bibliothek von Anwendungen für das verteilte Rechnen von großen Datenmengen in einem Cluster. Es besteht aus dem Dateisystem Hadoop Filesystem (HDFS), dem Algorithmus MapReduce und dem Aufgabenplaner Yarn. [Fou14b]

können die Daten mit Apache Avro[6] oder Apache Thrift[7] kodiert bzw. dekodiert übertragen werden. Daten werden von der *Source* in *Flume events* umgewandelt und an einen oder mehrere *Channels* geschrieben. Ein *Channel* ist der Bereich in dem *Events* gehalten und weiter an den *Sink* gereicht werden. Der *Sink* erhält ausschließlich *Flume events* von einem *Channel*. In einem *Agent* kann es mehrere *Sources*, *Channels* und *Sinks* geben. [Flu12a]

In Abbildung 4.3 wird ein Agent gezeigt. Die linke Spalte listet unterschiedliche *Sources* auf. Die *AvroSource* und *NetcatSource* sind mit dem *Channel MemoryChannel*, die *JmsSource* und *HttpSource* mit dem *FileChannel* und die *ExecSource* mit dem *JdbcChannel* verbunden. Der *LoggerSink* und *AvroSink* rufen Nachrichten von allen *Channels* ab. Im Anhang A.9.2 wird ein Beispiel aus einer *Source*, einem *Channel* und einem *Sink* gegeben. Die *Source* entspricht der *NetcatSource*. Die *NetcatSource* stellt einen Dienst bereit und wartet auf Nachrichteneingänge. Mit einem *Client* kann eine Netzwerkverbindung zum Dienst aufgebaut und Nachrichten zum Dienst gesendet werden. Sobald Nachrichten eingehen, werden Nachrichten in den *Channel MemoryChannel* übergeben. Die *Sink LoggerSink* holt die Nachricht durch Nachfragen vom *Channel* ab und schreibt den Inhalt in die Standardausgabe.

Die Benennung der einzelnen Bereiche findet in Apache Flume hierarchisch statt. So liegen die wesentlichen Teile *Source*, *Channel*, *Sink*, *Conf*, *Instrumentation* und *Serialization* in einem eigenen Ordner. Dennoch fällt Apache Thrift mit Implementierungen unterschiedlicher Aspekte in verschiedenen Ordnern auf.

Als *Source* können in Apache Flume unterschiedlich bestehende Implementierungen verwendet werden. Eine spezifische Implementierung muss die Abstrakte Klasse AbstractSource [Flu14b] erweitern und die Schnittstellen *Configurable* und *EventdrivenSource* implementieren. Folgende Liste stellt eine Übersicht über bestehende Implementierungen:

AvroSource verwendet *NettyTransceiver* für den Empfang von Nachrichten. Zur Übertragung kommt das Avro-Protokoll zum Einsatz. Die *AvroSource* kann mit der *AvroSink* oder mit einem spezifischen Avro-*Client* verbunden werden. Weiterhin kann eine Secure Sockets Layer (SSL)-Verschlüsselung und eine Kompression über Parameter eingestellt werden.[Flu14d]

ThriftSource setzt die Klasse *ThriftFlumeEvent* ein, um Nachrichten zu kodieren und zu dekodieren. Ein externer *Client* muss für die Kommunikation über Apache Thrift das *ThriftSourceProtocol* verwenden. [Flu14n]

ExecSource führt ein übergebenes Betriebssystemkommando aus. Weitere Parameter erlauben eine wiederkehrende Ausführung von Betriebssystemprozessen.[Flu14f]

NetcatSource stellt einen Dienst bereit der eine begrenzte Anzahl an Zeichen empfangen kann. Die *NetcatSource* wird vorwiegend für *Unittests*[8] oder *Debugging*[9] eingesetzt. [Flu14i]

HTTPSource empfängt *Flume Events* über HTTP POST. *HTTPSource* benötigt einen *HTTPSourceHandler* der die eingehenden Nachrichten in *Flume Events* konvertiert. Eine Verschlüsselung wird mit dem Parameter *enableSSL* eingeschaltet. [Flu14h]

---

[6] Apache Avro is ein Datenserializierungssystem [Fou14a]

[7] Apache Thrift is ein Software framework für die sprachenübergreifende Dienstentwicklung [Fou14d]

[8] JUnit Test [Bec06]

[9] Beim Debugging wird mit einem Werkzeug Debugger versucht Fehler in einer Anwendung zu finden.

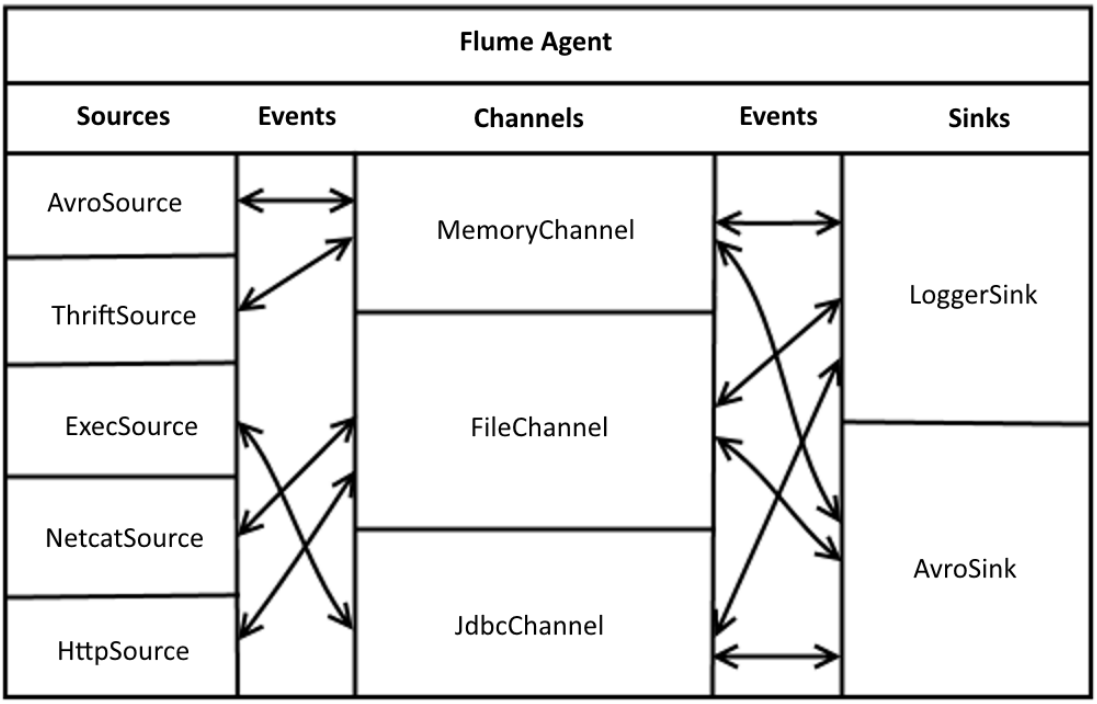

Abbildung 4.3: Apache Flume Agent - Ein Agent mit mehreren Sources, Channels und Sinks

Ein *Channel* stellt in Apache Flume einen Zwischenspeicher dar. Nachrichten werden in einem *Dataflow* von der *Source*, über den *Channel* an den *Sink* übertragen. In einem *Channel* können Nachrichten in einem flüchtigen Speicher dem *MemoryChannel* oder einem dauerhaften Speicher dem *FileChannel* abgelegt werden. Spezifische *Channel* müssen die Klasse *BasicChannelSemantics* [Flu14e] erweitern oder die Klasse *AbstractChannel* bei bestehenden Transaktionsmethoden erweitern. Der *JdbcChannel* stellt für die Datenpersistenz über die Konfigurationsdatei eine Verbindung mit einem SQL-Server her. Der *MemoryChannel* ist nicht transaktionssicher. Wenn der *Agent* ausfällt gehen Nachrichten im flüchtigen Speicher verloren. Der *FileChannel* schreibt im Gegensatz zum *MemoryChannel* die Nachrichten in ein definiertes Verzeichnis einer Festplatte. Um die Atomarität in Apache Flume beim Persistieren einzuhalten, wird das Prinzip von Write Ahead Log (WAL) eingesetzt. Mit dem WAL wird der Eingang und der Ausgang des *Channels* aufgezeichnet. Wenn der *Agent* neugestartet wird, kann das WAL wieder abgearbeitet werden, damit alle eingegangenen Nachrichten ausgeliefert werden. Einem *Channel* kann eine Kapazität als Parameter in der Konfiguration übergeben werden. Wenn die Kapazität für die Übertragung der Nachrichten erschöpft ist, werden keine weitere Nachrichten angenommen und ein Fehler *ChannelException* wird geworfen. [Hof13, S. 25, Kap. 3]

Die Ausgabe aus einem *Channel* erfolgt über einen *Sink*. Mit dem *LoggerSink* werden Protokolldaten abhängig von der Konfiguration auf die Standardausgabe oder in eine Datei ausgegeben. Der *AvroSink* [Flu14c] und der *ThriftSink* [Flu14m] erweitern die Klasse *AbstractRpcSink* [Flu14a] und ermöglichen mit deren Pendantm *AvroSource* und *ThriftSource Agents* miteinander zu verbinden. Ein Modell eines Datenfluss *data flow* unterstützt dabei die Übersicht. Beziehungen zwischen den *Agents* werden in den spezifischen Konfigurationsdateien abgelegt. In Abbildung 4.4 werden Vier *Agents* dargestellt. Es wird eine Trennung des Datenstroms in *Agent 2* und eine Zusammenführung des Datenstroms in *Agent 3* gezeigt. Der Datenfluss in

Agent 2 wird durch verwenden mehrerer *Channels* aufgeteilt. Der Ausgabestrom *Sink A* jeweils aus *Agent 1* und *Agent 2* wird in die *Source A* von *Agent 3* geleitet. Und der Ausgabestrom *Sink B* aus *Agent 2* wird in die *Source A* eines separaten *Agent 4* geleitet.

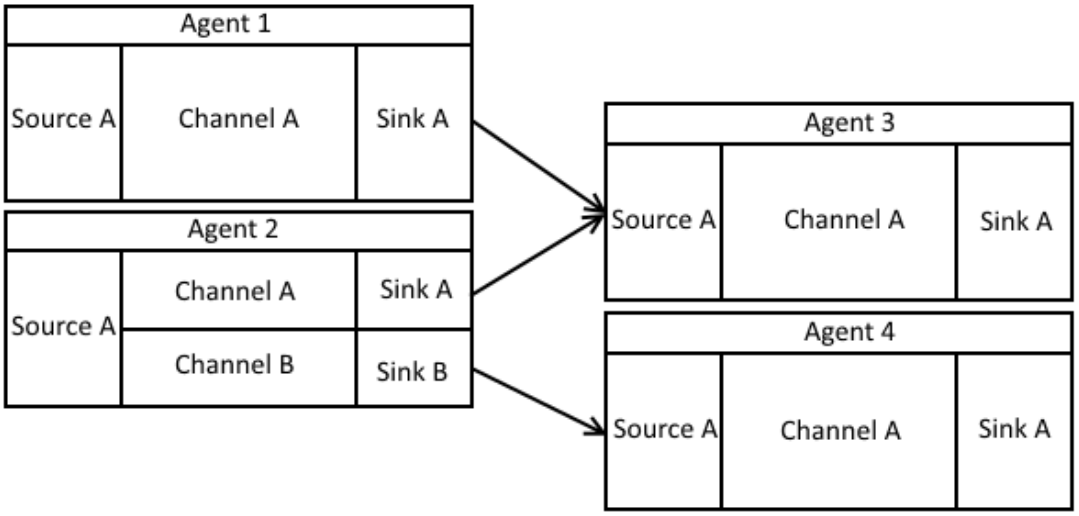

Abbildung 4.4: Apache Flume Agent Datenfluss

Bei hoher Last oder einer Ausfallsicherung ist es möglich *sinkgroups* einzusetzen. Eine *Sinkgroup* mit dem Prozessortyp *load_balance* kann eine Last je nach Parameter per *round_robin* oder *random* mehrere *Sinks* gleich verteilen. In der Ausfallsicherung muss der Prozessortyp *failover* verwendet werden. Den verwendeten *Sinks* wird über Parameter eine Priorität zugeordnet. Nach einem Ausfall eines *Agents* wird nach der Prioritätenliste der nächste verfügbare *Agent* eingesetzt. [Hof13, S. 43]

In Abbildung 4.4 schreibt die *Source* A vom *Agent* 2 abhängig vom *Flume Event* in *Channel* A oder *Channel* B. Über den Parameter *selector* ist es möglich *Flume Event* in bestimmte *Channels* zu leiten. Mit dem *selector*-Typ *multiplexing* und dem Parameter *mapping* können *Flume Events* abhängig vom *Header* zu einem *Channel* geleitet werden. Die Standardeinstellung im Vergleich zu *multiplexing* ist *replicating*. Durch den *selector*-Typ *replicating* werden *Flume Events* an alle *Channels* innerhalb eines *Agents* geleitet. [Hof13, S. 58]

Ein *Agent* kann als Sammler *collector* der *Flume Events* in einem Bulk von verschiedenen *Agents* aufnehmen und weiterleiten. Somit sind mit Apache Flume Schichten möglich. In der ersten Schicht können mehrere *Agents* Nachrichten von *Clients* aufnehmen, in *Flume Events* umwandeln und in bestimmten Zeitabständen an den Sammel-*Agent* weiter leiten. In der zweiten Schicht können die Daten zur weiteren Verarbeitung an Apache Hadoop und Apache Storm oder zur Datensicherung in ein sequentiellen Speicher weiter geleitet werden. [Hof13, S. 70]

Nachrichten können während der Laufzeit mit speziellen *Interceptors* verarbeitet werden. Spezifische *Interceptor* müssen die Schnittstellen *Interceptor* und *Interceptor.Builder* implementieren. Folgende Liste gibt einen Überblick über die bestehenden Typen:

TimestampInterceptor setzt den aktuellen Zeitstempel im Header aller abgefangenen *Flume Events* [Flu14o]

HostInterceptor fügt den Namen der Maschine oder die IP-Adresse in den *Header* aller abgefangenen *Flume Events* hinzu [Flu14g]

StaticInterceptor fügt ein definiertes Schlüssel/Wert-Paar in den *Header* aller abgefangenen *Flume Events* ein [Flu14l]

RegexFilteringInterceptor verwendet das Java-Paket *java.util.regex* mit einem definierten Ausdruck und prüft den *Body* eines *Flume Events*, ob der Ausdruck passt. Mit dem Parameter *excludeEvents* werden die *Flume Events* für die weitere Übertragung eingeschlossen oder ausgeschlossen. [Flu14k]

RegexExtractorInterceptor setzt ebenfalls das Java-Paket *java.util.regex* ein und sucht nach durch den regulären Ausdruck nach Mustern. Gefundene Muster werden als Schlüssel/Wert-Paar in den Header des *Flume Events* hinzugefügt. In den Parametern *serializers* muss Anzahl und der Bezeichner der Anzahl Platzhalter im regulären Ausdruck entsprechen. [Flu14j]

Apache Flume bietet Zwei verschiedene *Monitoring*-Typen *Ganglia*[10] und *Http* an. Der beim Start eines *Agents* verwendete Typ *http*, erzeugt eine Web-Server-Instanz und bei einer Anfrage wird eine Metrik als JavaScript Object Notation (JSON) zurückgegeben. Die Monitoring-Anwendung Nagios kann durch aktive Prüfungen das JSON filtern und das Ergebnis im *Service*-Bereich darstellen. Mit Ganglia-Integration können ähnlich wie in Nagios bestimmte Metriken ohne JSON zu filtern, direkt abgerufen werden.

Kriterium	**Bewertung**
Architektur	Strukturierte Peer-to-Peer-Architektur
Prozesse und Threads	Client-Server-Modell und RPC
Kommunikation	Streamorientierter synchroner Übertragungsmodus
Namenssystem	Hierarchische Benennung
Synchronisierung	Dezentraler Algorithmus
Pipelining und Materialisierung	Agent chaining
Konsistenz und Replikation	Replikation und Multiplexing
Fehlertoleranz	Load balancing und Failover
Sicherheit	Verschlüsselung und Datenkompression mit Apache Avro, HTTPSource unterstützt SSL, Monitoring mit JMX, Ganglia und Nagios
Erweiterung	Eigenentwicklung und Community-Beiträge
Qualität	FileChannel WAL

Tabelle 4.6: Bewertung Apache Flume

Eine weitere *Monitoring*-Möglichkeit besteht über JMX und der Java-Anwendung JConsole. Mit der JConsole können die die *MBean*-Attribute ausgelesen werden. Alternativ ähnlich wie

[10] Ganglia Monitoring System - http://ganglia.sourceforge.net/

in Apache Kafka kann über das JMX-Plugin in Nagios eine Abfrage auf die gegebenen Attribute erfolgen. In der Liste 4.1 wird ein Beispiel für das *Monitoring* über Hypertext Transfer Protocol (HTTP) gezeigt. Die gezeigten Argumente können beim Programmaufruf oder in der Konfigurationsdatei hinzugefügt werden. [Hof13, S. 77, Kap. 7]

Listing 4.1: Apache Flume Monitoring

```
1 -Dflume.monitoring.type=http
2 -Dflume.monitoring.port=55555
```

In diesem Kapitel und im Anhang wurde gezeigt wie Apache Flume installiert und für eine einfache Client/Server-Anwendung konfiguriert werden kann. Es wurden Techniken gezeigt, um Daten zu aggregieren, weiter zu leiten und zur Laufzeit zu bearbeiten. Verschiedene Einstiegspunkte im Quelltext von Apache Flume wurden gegeben, um spezifische Implementierungen zu entwickeln. Zuletzt wurden verschiedene Software-Werkzeuge für das *Monitoring* gezeigt. Im nächsten Kapitel wird Apache S4 vorgestellt.

4.4 Apache S4

Nach der Vorstellung von Apache Storm, Kafka und Flume wird in diesem Kapitel Apache S4 vorgestellt. Apache S4 ist eine Abkürzung und steht für Simple Scalable Streaming System und wird von Flavio Junqueira als allgemeine, verteilte, skalierbare, teilweise fehlertolerante und steckbare Plattform bezeichnet [Jun11]. Zunächst soll eine Kurzübersicht einen ersten Einblick in Apache S4 geben. Anschließend werden die Bewertungskriterien erläutert und vorgestellt.

Die Architektur von Apache S4 baut auf einem Apache Zookeeper Cluster auf und besteht aus mehreren Apache S4 *Cluster*, *Processing Nodes*, *Apps*, *Processing Elements* und dem *Communication Layer*. *Apps* sind Java Archive, die in einem Apache S4 *Cluster* bereitgestellt werden. Die Größe eines *Clusters* entspricht der Anzahl der *Tasks*. Pro *Task* muss jeweils eine *Processing Node* als selbständiger Prozess gestartet werden. Eine *Procssing Node* dient als Container für mehrere *Processing Elements*. *Apps* bestehen aus einem Graphen von *Processing Elements* und *Streams*. Ein *Processing Element* kommuniziert asynchron über unterschiedliche *Cluster* per *Streams*. Der Nachrichtenaustausch erfolgt über den *Communication Layer*.

Abbildung 4.5 zeigt Zwei Processing Nodes mit mehreren Processing Elements. Ein Raw Event wird von der äußeren Umgebung an die erste Processing Node in den Event Listener übergeben. Der Dispatcher erhält die verarbeitete Nachricht von einem Processing Element und leitet diese an den Emitter weiter. Der Emitter setzt die Nachricht in einen Stream. Der Stream wird vom Communication Layer an die zweite Processing Node vermittelt. Die Verarbeitung wird von der zweiten Processing Node durchgeführt. Das User Interface Model holt sich die Nachrichten vom neuen Stream ab und stellt die Information in der Benutzerschnittstelle dar.

Eine spezielle Implementierung einer Nachricht muss von der Klasse *Event* erben, aus einem Schlüssel/Wert-Tupel bestehen und an einen *Stream* weitergegeben werden können. Mit der Erweiterung der Basisklasse *AdapterApp* kann ein *Stream* erzeugt werden. Ein Beispiel wird im Anhang A.9 gezeigt. Dabei wird auf eine Netzwerkverbindung mit dem Anschluss 15000 gehört. Bei erfolgreicher Verbindung wird der Inhalt gelesen und in einen *Stream* gesetzt.

In einem *Processing Element* wird die Datenverarbeitung durchgeführt. In Apache S4 gibt es nur zwei Typen von *Processing Elements*, ein Schlüssel-loses und ein Schlüssel-behaftetes

Faktum	**Beschreibung**
Hauptentwickler	Matthieu Morel, Kishore Gopalakrishna, Flavio Junqueira Leo Neumeyer, Bruce Robbins, Daniel Gomez Ferro
Stabile Version	0.6.0 vom 03.06.2013
Entwicklungsstatus	Moderat
Entwicklungsversion	0.7.0
Sprache	Java
Betriebssystem	plattformunabhängig, benötigt die Java Virtual Machine und Apache Zookeeper
Lizenz	Apache License version 2.0
Webseite	[S413c]
Quelltext	[GJM+14]

Tabelle 4.7: Kurzübersicht Apache S4

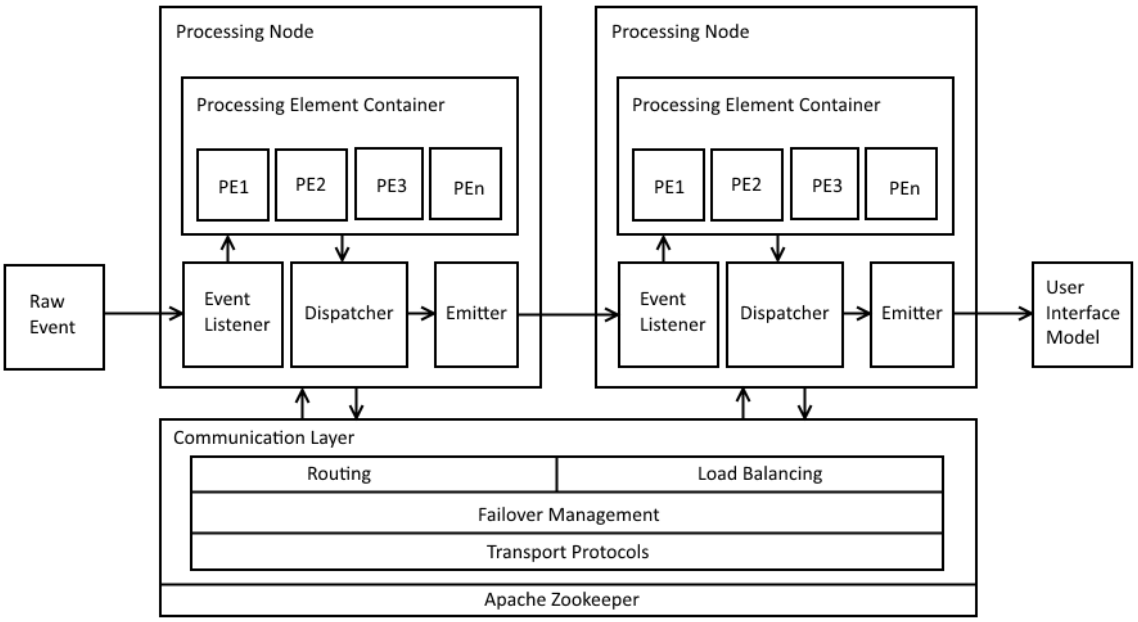

Abbildung 4.5: Apache S4 Processing Nodes

Processing Element. Der Schlüssel-lose Typ kann unterschiedliche Apache S4 Nachrichten empfangen. Der Schlüssel-behaftete Typ kann nur Apache S4 Nachrichten mit einem definierten Schlüssel in der Nachricht empfangen. Spezielle Aggregate und Operatoren von Nachrichten müssen in *Processing Elements*-Klassen explizit implementiert werden. Ein *Processing Element* besteht aus Zwei Teilen, dem *Prototype* und der *Instance*. Der *Prototype* erbt von der Basisklasse *ProcessingElement* und behandelt eingehende Nachrichten. Die *Instance* erbt von der Basisklasse *App* und behandelt den Anwendungsstart und Anwendungsstop. Im Anhang A.8 wird ein Beispiel für das Erzeugen eines *Streams* „names“ und die Weitergabe der Nachrichten an einen *Stream* gezeigt. [S413f]

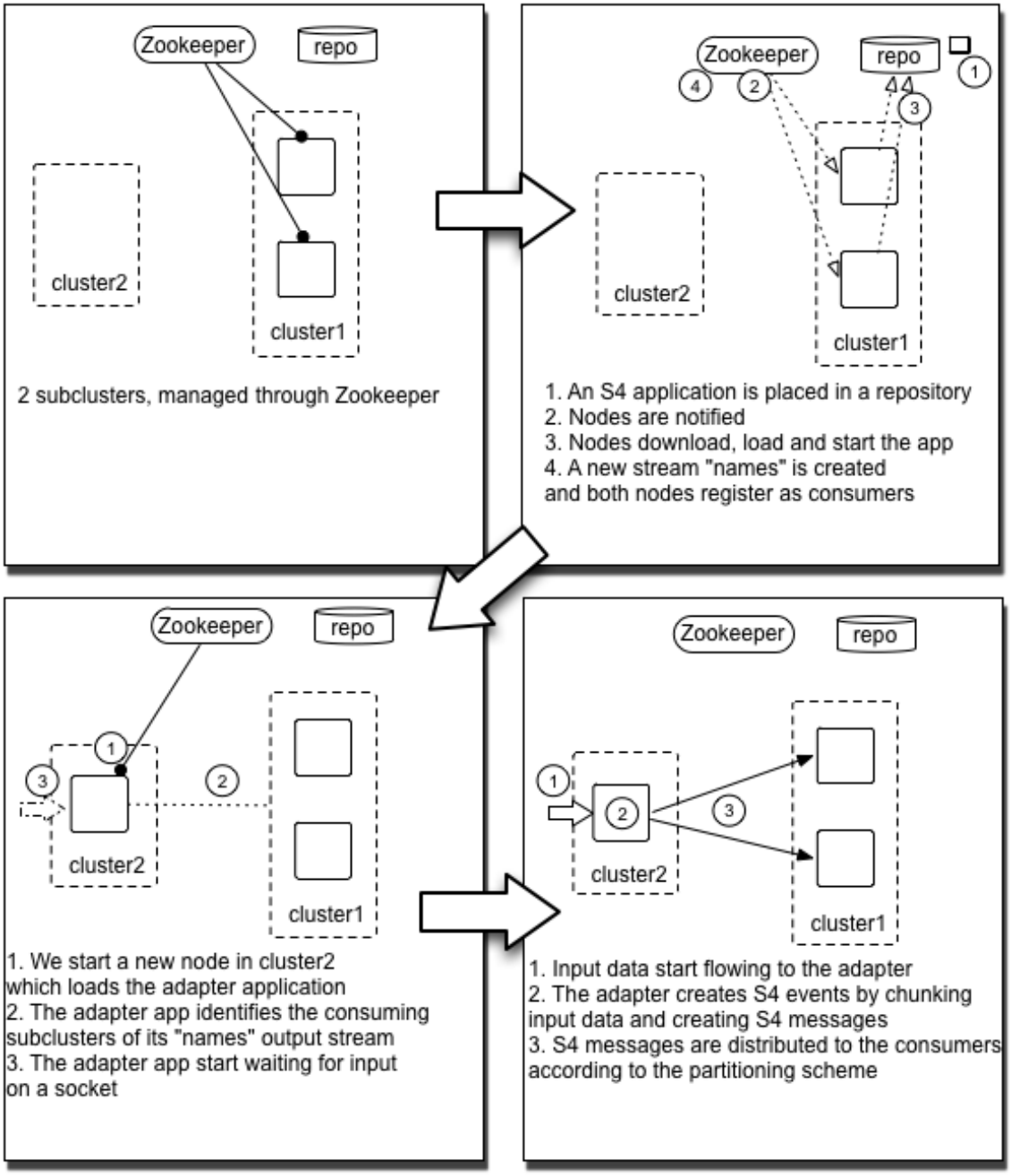

Abbildung 4.6: Apache S4 HelloApp Beispiel

In Abbildung 4.6 wird das Beispiel A.10.6 aus der Apache S4 Installation im Anhang A.10 gezeigt. Die Abbildung 4.6 wurde aus der Dokumentation [S413g] entnommen. Zuerst werden Zwei Cluster *cluster1* und *cluster2* in einem Apache Zookeeper *Cluster* bereitgestellt. Anschließend wird im *Repository* die S4 Anwendung *myApp* hinzugefügt. Die Apache S4 *Nodes* werden über Apache Zookeeper informiert und die Anwendung wird gestartet. Ein neuer *Stream* „names“ wird erstellt und beide *Nodes* werden als *Consumer* registriert. Anschließend wird im *Cluster cluster2* die *HelloInputAdapter*-Anwendung gestartet. Beim Bereitstellen der Anwendung im *Cluster cluster2* wird der *Stream* “names“ als Identität für den Ausgabestrom gesetzt. Die *HelloInputAdapter*-Anwendung ist nach dem Starten aktiv und wartet auf Dateneingang. Abschließend werden eintreffende Daten vom *Cluster cluster2* in Apache

S4-Nachrichten umgewandelt und zur weiteren Datenbehandlung an die *Consumer* verteilt.

Nachrichten werden in Apache S4 zwischen den *Nodes* durch den *Communication Layer* übertragen. Der *Communication Layer* nutzt für die Koordination der Nachrichten zwischen den Apache S4 *Nodes* Apache Zookeeper. Um Nachrichten an die *Nodes* im Apache S4 *Cluster* zu senden, können spezielle Bindungen in verschiedenen Programmiersprachen implementiert werden. Durch ein steckbares Design können unterschiedliche Nachrichtenprotokolle wie zum Beispiel Apache Avro oder Apache Thrift eingesetzt werden. Eine Implementierung für das User Datagram Protocol (UDP) [S413b] und dem TCP [S413a] wird von Apache S4 bereits unterstützt. [NRNK10, S. 4, Kap. II D]

Die Fehlertoleranz in Apache S4 wird durch die *Fail-Fast* Strategie von Apache Zookeeper übernommen. In einem Apache S4 Cluster mit Zwei *Tasks* und Vier gestarteten *Nodes*, sind Zwei *Nodes* Aktiv und Zwei *Nodes* im *Standby*-Betrieb. Wenn Apache Zookeeper einen *Session-Timeout* einer aktiven *Node* feststellt, wird sofort eine *Standby-Node* aktiviert. Die anderen *Nodes* werden durch den *Communication Layer* über neue aktive *Node* informiert und neue Nachrichten werden umgeleitet. Apache S4 *Nodes* nutzen für die Datenverarbeitung den lokalen Speicher von Apache Zookeeper. Im Fehlerfall geht der Zwischenspeicher einer Apache Zookeeper *Node* verloren. Damit die Daten nicht verloren gehen, kann mit dem *Checkpointing*-Mechanismus über die Konfiguration eine Datensicherung in einem externen Datenlager erfolgen. Beim Start der neuen *Nodes* aus dem *Standby*-Betrieb kann die Datensicherung in den lokalen Speicher zurückgeschrieben werden. In [Val12] untersucht Vallés verschiedene Ansätze von *Checkpointing* und zeigt eine geringe Performanz der Standardimplementierung gegenüber einer Implementierung mit Apache HBase[11]. [S413d]

Kriterium	**Bewertung**
Architektur	Strukturierte Peer-to-Peer-Architektur
Prozesse und Threads	Client-Server-Modell
Kommunikation	TCP-basiert und UDP-basiert via Apache Zookeeper
Namenssystem	Hierarchische Benennung
Synchronisierung	Communication Layer
Pipelining und Materialisierung	Chaining von Processing Elements
Konsistenz und Replikation	Consistent hashing, Checkpointing
Fehlertoleranz	Failover und Checkpointing
Sicherheit	Nur eigene Maßnahmen
	Monitoring mit JMX pro Node
Erweiterung	Modulare Eigenentwicklung
Qualität	Nur eigene Entwicklung für QoS

Tabelle 4.8: Bewertung Apache S4

Bei der Sicherheit sind eigene Maßnahmen notwendig. Nachrichten werden im Klartext oder binär übertragen. Die Verbindung zwischen den einzelnen *Processing Nodes* findet unautorisiert statt. Für die Autorisierung und Verschlüsselung ist eine spezielle Implementierung des

[11] Apache HBase ist eine Apache Hadoop Datenbank, ein verteilter, skalierbarer Big Data-Speicher [HBa14].

Communication Layer und der einzelnen *Processing Elements* notwendig. Weiterhin ist eine Verschlüsselung des lokalen Speichers von Apache Zookeeper in einem offenen Netz zu empfehlen. Über JMX können Metriken eines Apache S4 Clusters abgefragt werden [S413e]. QoS wird in [NRNK10, S. 3, Kap. II B] als anwendungsspezifisch eingestuft. In einer spezifischen Anwendung muss daher eine eigene Implementierung für das QoS in Apache S4, innerhalb von *Processing Elements* erfolgen.

Diese Kapitel stellt das Streaming Framework Apache S4 vor. Es wurde die Installation von Apache S4 beschrieben und im Anhang A.10 eine Anleitung abgelegt. Weiterhin erfolgte eine Erklärung der Architektur und der Informationsfluss in einem Apache S4 *Cluster* wurde erläutert. Es wurde die Fehlertoleranz in Zusammenhang von Apache Zookeeper und die Replikation mit *Checkpointing* beschrieben und zuletzt auf eine geringe Sicherheit, Monitoring und Qualität hingewiesen. Nach der Vorstellung der Streaming Frameworks werden im folgenden Kapitel die wesentlichen Kernelemente der einzelnen Streaming Frameworks zusammengefasst.

4.5 Zusammenfassung

In den Kapiteln zuvor wurden die Vier Streaming Framework Apache Storm, Apache Kafka, Apache Flume und Apache S4 in der Architktur, Installation und Entwicklung vorgestellt. Die in dieser Arbeit abgehandelten Streaming Frameworks sind in einer speziellen Umgebung aufgebaut und haben wenige Überschneidungen in der Verwendung gleicher Datenflussverarbeitung. Eine kurze Aufzählung der wesentlichen Kernelementen jeder Streaming Frameworks soll das Wissen auffrischen und für die folgenden Kapitel vorbereiten.

Ein Apache Zookeeper Cluster wird in allen Streaming Frameworks außer in Apache Flume für die Synchronisierung von verteilten Prozessen verwendet. Apache Storm benutzt einen *Task-Scheduler* zur Arbeitsverteilung und stellt mehrere Primitive, Operatoren und Funktionen für die direkte und mit Trident für die transaktionssichere Datenverarbeitung bereit. Bei Apache Kafka kommt das *Publisher-Subscriber* Nachrichtenmuster für die Verteilung der Nachrichten zum Einsatz. Dabei wird auf einem *Topic* jeweils die Nachricht gelegt und von einem *Consumer* abgeholt. Apache Flume hingegen kommuniziert über konfigurierte Direktverbindungen in einem Client-Server-Modell. Auch in Apache Flume werden, wie in Apache Kafka, Nachrichten über *Channels* ausgetauscht. Eine *Source* sendet in einen *Channel* und eine *Sink* wird über eine Nachricht informiert. Gegenüber Apache Kafka werden mehrere *Sources*-, *Channel*- und *Sink*-Typen bereitgestellt. Apache S4 benutzt zum Austausch der Nachrichten einen *Communication Layer* und kommt ohne einen Master aus. Es gibt einen Typen das Processing Element, das weiter spezifiziert werden kann. Für das Monitoring kann in allen Streaming Frameworks JMX benutzt werden. Die Entwicklung von eigenen Komponenten kann in der Programmiersprache Java erfolgen. Für den Vergleich der Streaming Frameworks werden die gewonnen Erkenntnisse aus den Kapiteln 2, 3 und 4 im nächsten Kapitel zur Entwicklung eines Prototypen herangezogen. Zunächst werden die einzelnen Implementierungen der Streaming Frameworks für die Performanzmessung vorgestellt und beschrieben.

Kapitel 5

Systemarchitektur

In den Kapiteln zuvor wurden die einzelnen Streaming Frameworks vorgestellt. Dieses Kapitel beschreibt eine Methode zur Messung der Performance. Es wird zunächst das Messverfahren gezeigt, die Messumgebung und die Anforderungen an die Prototypen beschrieben. Um einem Vergleich zwischen den Streaming Frameworks auf Entwicklungsebene näher zu kommen, ist es notwendig eine allgemeine Anwendung in einer homogenen Umgebung zu entwickeln und bereitzustellen. Dazu werden zuerst die funktionalen Anforderungen und anschließend die nichtfunktionalen Anforderungen in schriftlicher und darstellerischer Form beschrieben.

5.1 Funktionale Anforderung

Die funktionalen Anforderungen tragen dazu, bei die Anwendung zu implementieren. In den folgenden Listen werden Kriterien für die Prototypen der einzelnen Streaming Frameworks definiert. Das Use-Case-Diagramm zeigt die Muss-Kriterien für den Anwender in Abbildung 5.1.

Muss-Kriterien:

M1 Paketierung der Implementierungen mit Apache Maven

M2 Ausführung der Implementierungen unter Java und dem Betriebssystem Linux

M3 Ausführung einer Implementierung auf einem Single-Node-Cluster

M4 Ausführung einer Implementierung von konstanter Größe von 100 Byte-Daten (statischer Payload)

M5 Ausführung einer Implementierung von variablen Größen von Daten (dynamischer Payload)

M6 Aufnahme von Daten: aktueller Nachrichten pro Sekunde und CPU-Belastung während der Ausführung einer Implementierung in separaten Dateien

M7 Während der Ausführung, anzeigen der Daten pro Streaming Framework auf einer Webseite - Übersichtsseite

Soll-Kriterien:

S1 Ausführung einer Implementierung auf verschiedenen Rechnersystemen (Professional Workstation, Notebook, Virtuelle Maschine)

S2 die dynamische Implementierungen sollen Wörter aus einem offen und frei zugänglichen großen Datensatz zählen und die Anzahl der höchsten 5 Wörter sowie das Wort selbst pro Sekunde automatisch anzeigen

S3 Die Inhalte auf der Webseite sollen über Javascript-Trigger aktualisiert werden

Kann-Kriterien:

K1 Ausführung der Implementierungen auf Multi-Node-Cluster

K2 Ausführung auf proprietären Betriebssystemen

Abgrenzungskriterien:

A1 Für die prototypische Entwicklung werden erst in einem weiterführenden Konzept Unit- und Verhaltens-Tests eingesetzt

A2 Die Darstellung von Informationen auf einer Webseite benötigt beim Prototypen keine Serverseitige Absicherung

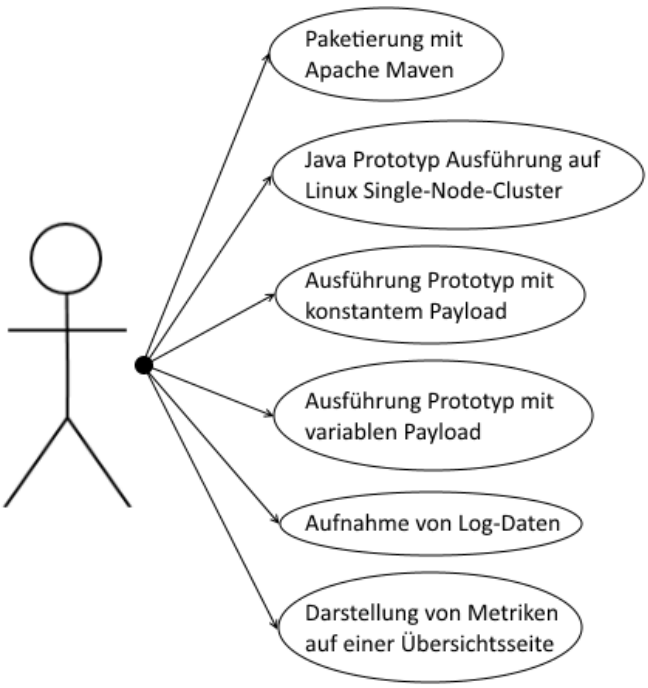

Abbildung 5.1: Muss-Kriterien Use-Case-Diagramm

Nachdem die Kriterien vorgestellt wurden wird als nächstes der Einsatzbereich und die Umgebung gezeigt.

5.2 Nichtfunktionale Kriterien

Die Implementierungen der Prototypen werden hauptsächlich in einem wissenschaftlichen Studium eingesetzt. Als Zielgruppe können Wissenschaftler in der Informationstechnologie, Datenanalysten und Entwickler aus dem Bereich der Datenverarbeitung von zeitkritischen Daten und der Daten einer großen Menge, ein Interesse finden. Die Prototypen werden auf einem Notebook, in einer virtuellen Instanz und auf einer professionellen Workstation ausgeführt. Alle Rechnereinheiten werden über ein Switch verkabelt. Dabei wird ein homogenes Netz gebildet.

Um Störgrößen zu vermeiden, ist eine feste Verdrahtung essentiell. Eine Verbindung in das Internet und Intranet ist nicht vorgesehen. Kabellose Verbindungen werden aufgrund größerer Störempfindlichkeit wie zum Beispiel bei einer Kanalauslöschung nicht unterstützt. Das System wird mit kostenloser Open-Source Software entwickelt, dabei wird auf eine gute Wartbarkeit und Performance geachtet.

Angeschlossene Fremdrechner benötigen für die Darstellung der Übersichtsseite einen aktuellen Webbrowser[1]. Die Nachrichten werden innerhalb der Streaming Frameworks, binär und nach außen, zur Übersichtsseite als JSON übertragen. Für die Eingabedaten wird ein freier, offener und großer Shakespeare-Datensatz [Sha94] benutzt. Bevor die Prototypen dokumentiert werden, erfolgt zunächst eine Vorstellung der möglichen Probleme und Lösungsansätze.

5.3 Probleme und Lösungsansätze

Für die Entwicklung geeigneter Prototypen, sind bezüglich der Anforderung zunächst mögliche Konflikte vor der Implementierung zu lösen. Das Kapitel 4 hat die Streaming Frameworks vorgestellt und konnte unterschiedliche Implementierungssprachen zeigen. So werden von den Streaming Frameworks unterschiedliche Programmierschnittstellen bereitgestellt. Aufgrund der prototypischen Entwicklung steigt die Komplexität und Übersicht, die Implementierungen für die Streaming Frameworks in ein Anwendungsprojekt zu packen. Da alle Streaming Frameworks die Java Plattform unterstützen, wird für die Entwicklung der Prototypen in einzelnen Java-Projekten entschieden. Durch Trennung der Verarbeitungslogik und der Ein- und Ausgabeformatierung in den Prototypen können Erweiterungen an geziehlten Stellen im Quelltext effizient erfolgen. Da die Implementierungen nacheinander erfolgen, können demnach gewonnene Erkenntnisse in die bestehenden Implementierungen direkt übernommen werden.

Durch die unterschiedlichen Streaming Frameworks werden ebenfalls unterschiedliche Bereitstellungsmechanismen eingesetzt. In Apache Storm wird zum Beispiel über ein Kommando in der Shell, die Anwendung im Cluster bereitgestellt. Apache Flume hingegen benötigt zunächst eine Konfiguration, die anschließend über die Shell gestartet wird. Um die Komplexität weiterhin in der Entwicklung und Bereitstellung der Prototypen klein zu halten, ist ein gemeinsames Konzept für die Java-Paketierung ein wichtiges Instrument. Mit Hilfe von Apache Maven kann durch die Deklaration der Bereitstellung in einer Konfigurationsdatei[2] sowie die Erstellung der einzelnen Prototypen von der Entwicklung getrennt und einheitlich erzeugt werden.

Mit den unterschiedlichen Streaming Frameworks werden teilweise unterschiedliche Logging-Frameworks eingesetzt. Damit bei der Abnahme des IST-Zustands und bei der Laufzeitdiagnose

[1] Webbrowser: https://www.mozilla.org/de/firefox/desktop/

[2] Die Konfigurationsdatei von Apache Maven heißt pom.xml und liegt in der ersten Ebene des Projektarchivs. Die Datenstruktur der Konfiguration besteht aus XML-Auszeichnungen.

ein einheitliches Konzept verwendet wird, die Einarbeitungszeit in das Logging-Framework klein bleibt und um das Logging effizient zu halten, soll das Logging Framework Apache Log4j[3] eingesetzt werden. Dabei soll die Weitergabe der Logging-Daten an die Übersichtsseite über das WebSocket-Protokoll erfolgen. Durch den Einsatz von WebSocket und Javascript EventHandler ist ein *PageRefresh*[4] der Übersichtsseite nicht nötig. Die IST-Werte werden somit, sobald Werte eintreffen, auf der Übersicht durch Javascript Trigger aktualisiert.

Eine zeitgleiche Ausführung aller Streaming Frameworks auf einer Maschine kann zu unvorhergesehenen Ergebnissen bzw. Problemen führen. Auch für den Vergleich muss die Voraussetzung für die Messung gleich sein. Um die Messung möglichst gleich und Störeinflüsse gering zu halten, sollen alle Prototypen nacheinander auf je einer Maschine ausgeführt werden.

Für die Stream Data Management System (SDMS) wurde der Linear Road Benchmark (LRB) von Arasu et al. [ACG+04, S. 488, Kap. 3.3] entwickelt, um einen großen Umfang von Datenströmen wie auch historische Daten zu verarbeiten. Das Ziel beim LRB ist die Ermittlung des *L*-Rating eines SDMS. In vier Schritten wird der LRB ausgeführt. Zuerst werden mit dem *Historical data generator* historische Daten in einer Datei erzeugt. Anschließend werden mit dem *Traffic simulator* und dem Data driver die *L*-Daten in eine Datei geschrieben. Mit dem Start des LRB werden Ausgabedaten mit Zeitstempel geschrieben. Abschließend werden mit dem *Validationtool* die Antwortzeiten und die Genauigkeit der Ausgabedaten geprüft und das *L*-Rating bestimmt.

Da die Implementierung des LRB in der Programmiersprache C++ besteht, muss eine umfangreiche Portierung des Quelltextes in Java erfolgen. Für die Überführung des Quelltextes sind geeignete Qualitätssicherungen notwendig. An dieser Stelle wird nur ein Teilaspekt des LRB in Java entwickelt. Für den Benchmark in Java wird in den Prototypen eine Methode für das Verarbeiten von konstanten 100 Byte Daten 240 Sekunden lang implementiert und in der zweiten Methode werden Wörter aus Texten mit variable Wortlänge gezählt. Dabei werden die Top-K-Werte pro Sekunde 240 Sekunden lang bestimmt.

In diesem Kapitel wurden mögliche Probleme in der Betriebnahme oder während der Entwicklung erläutert sowie Lösungsansätze vorgestellt. Im nächsten Kapitel wird der Systementwurf für die Prototypen vorgestellt.

5.4 Systementwurf

Bezugnehmend auf die Anforderungen aus Kapitel 5.1 und 5.2 wird in diesem Kapitel ein Überblick über den Systementwurf gegeben. Dabei werden weiterführende Details der einzelnen Prototypen im Kapitel 6 vorgestellt. Zuerst werden die eingesetzten Komponenten in einem Schaubild dargestellt und die Zusammenhänge erläutert.

In Abbildung 5.2 werden in drei Schichten verschiedene Arbeitsbereiche eingeteilt. Die obere Schicht wird als *Backend* bezeichnet. Im *Backend* werden die Cluster der einzelnen Streaming Frameworks in separaten Server-Maschinen betrieben. Auch die Prototypen der Streaming Frameworks werden im *Backend* bereitgestellt. Die mittlere Schicht stellt die Anwendungsschicht dar und wird als *Visualization* bezeichnet. In der *Visualization* wird ein Apache Http Web

[3] Apache Log4j: http://logging.apache.org/log4j/2.x/

[4] Ein *PageRefresh* entspricht einem Neuladen der Webseite.

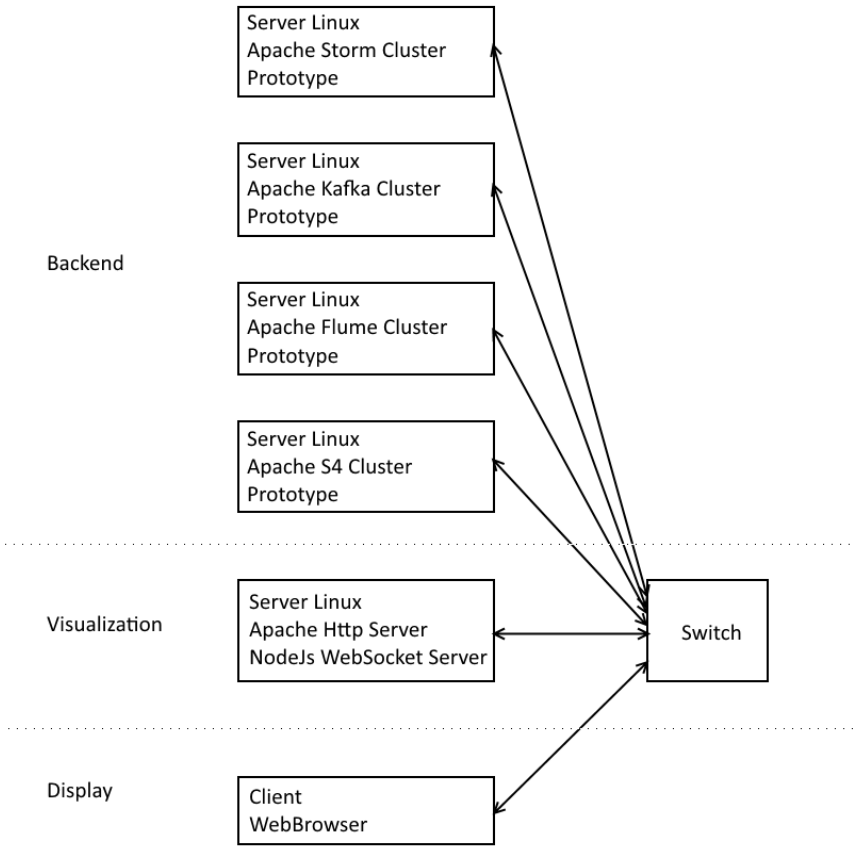

Abbildung 5.2: Systementwurf

Server[5] für die Bereitstellung der Webseite zur Übersicht und ein NodeJs[6] WebSocket[7] Server für die Verteilung der Log-Daten eingesetzt. Die untere Schicht stellt den Anwender dar und wird als *Display* bezeichnet. Im *Display* wird ein Webbrowser aufgerufen, um die *Visualization* aus der mittleren Schicht anzuzeigen. Alle drei Schichten werden über TCP/IP durch den *Switch* verbunden. Damit die Maschinen untereinander Daten austauschen können, wird jeder Maschine eine eindeutige IP-Adresse zugeordnet. Das folgende Kapitel 5.5 stellt anhand des Systementwurfs die Systemspezifikation vor.

5.5 Systemspezifikation

Für die Ausführung der Prototypen in einem Single-Node-Cluster kommen Zwei Systeme zum Einsatz. Eine High-End-Workstation und eine Low-End-Workstation. Die High-End-Workstation ist mit einer starken CPU und einer hohen Anzahl an Random Access Memory (RAM) ausgestattet. In der Tabelle 5.1 werden die Maschineneigenschaften der High-End Maschine aufgestellt. Bei Low-End-Workstation hingegen, kommt eine schwache CPU mit einer kleinen

[5] Apache Httpd Web Server: http://httpd.apache.org/

[6] NodeJs ist eine Javascript Platform für die Entwicklung schneller und skalierter Netzwerkanwendungen. http://nodejs.org/

[7] NodeJs WebSocket module für die Client-Server Entwicklung: https://www.npmjs.org/package/nodejs-websocket/

Anzahl an RAM zum Einsatz. Die Tabelle 5.2 zeigt die Werte der Low-End-Maschine. Beide Maschinen sind mit einem Switch über die Netzwerkkarten[8] miteinander verbunden.

Maschineneigenschaft	
CPU Bezeichnung	AMD FX-8350
CPU Taktfrequenz	4 GHz
CPU Anzahl Kerne	8
CPU Einführung	Quartal 4 in 2012
RAM	16 GByte
Netzwerkkarte	1 GBit

Tabelle 5.1: High End Workstation Eigenschaften

Maschineneigenschaft	
CPU Bezeichnung	Intel i5-480M
CPU Taktfrequenz	2,66 GHz
CPU Anzahl Kerne	2
CPU Einführung	Quartal 1 in 2011
RAM	4 GByte
Netzwerkkarte	1 GBit

Tabelle 5.2: Low End Workstation Eigenschaften

Die Maschinen werden mit einer Festplatte über External Serial Advanced Technology Attachment (eSATA)[9] angeschlossen. Für den Vergleich wird nur eine Festplatte mit dem eingesetzten Betriebssystem und den Prototypen betrieben. Mit eSATA ist es problemlos die Maschinen durch das Wechseln des Anschlusses zu tauschen. Die Tabelle 5.3 zeigt die Geräteeigenschaften der Festplatte. In Anhang A.3 Abbildung A.1 wird ein Leistungstest der Festplatte aus Tabelle 5.3 angehängt. Darin wird das Ergebnis eines Lese- und Schreib-Vergleichstest gezeigt. Die Mittelwerte aus dem Leistungstest wurden aus dem Ausschnitt Abbildung A.1 in die Tabelle 5.3 übernommen.

Als Betriebssystem wird die Linux[10] Distribution Debian[11] in der Version 7.6.0 Stable (Wheezy) eingesetzt. Zusätzlich zum Betriebssystem wird die freie Implementierung der Java Plattform Open Java Developer Kit (OpenJDK)[12] in der Version 7 bereitgestellt und eingesetzt. Zusätzlich zum Betrieb eines Single-Node-Cluster wird für die Streaming Frameworks Apache Storm, Apache Kafka und Apache S4 bezüglich der Koordination der Nachrichten in einem verteil-

[8] Die Netzwerkkarten haben eine Bandbreite von 1 GBit/s

[9] eSATA Spezifikation: https://www.sata-io.org/esata

[10] Linux Kernel https://www.kernel.org/

[11] Linux Distribution Debian: http://www.debian.org/

[12] OpenJDK http://openjdk.java.net/

Geräteeigenschaft	
Festplattenbezeichnung	Hitachi HDS721616PLA380
Anschlusstyp	eSATA
Größe	165 GByte
Spindelgeschwindigkeit	7200 rpm
Mittlere Lesegeschwindigkeit	62,1 MByte/s
Mittlere Schreibgeschwindigkeit	52,1 MByte/s
Mittlere Zugriffszeit	13,6 s

Tabelle 5.3: Eigenschaften Festplatte

ten System Apache Zookeeper[13] mit den Standardeinstellungen als Dienst bereitgestellt. Die Installation von OpenJDK und Apache Zookeeper erfolgt über die Paketverwaltung *apt* in Debian. Weiterführende Installationsschritte für die Bereitstellung eines Single-Node-Cluster sind im Anhang für die einzelnen Streaming Frameworks in den Kapiteln A.7, A.8, A.9 und A.10 beschrieben. Spezielle Optimierungen für Netzwerkkarten oder Anpassungen der zu ladenden Linux-Kernel-Module wurden nicht vorgenommen und die Standardeinstellungen werden eingesetzt. Mit der Vorstellung der Systemspezifikation wird anschließend ein Algorithmus zur Messung der einzelnen Streaming Frameworks gezeigt.

5.6 Algorithmus

In Kapitel 5.3 wurden bereits mögliche Probleme während der Entwicklung eines Prototypen angesprochen. Weiterhin wurde das einzusetzende System in Kapitel 5.4 und 5.5 mit Anforderungen und Gerätebeschreibungen näher erläutert. In diesem Kapitel wird durch Sequenzdiagramme der Ablauf für die Erzeugung und das Abholen von Nachrichten dargestellt.

Wie in den Kapitel 4 gezeigt weisen die Streaming Frameworks unterschiedliche Architekturen auf und besitzen eine unterschiedliche Java-API. Dennoch wird aufgrund des Vergleichs zwischen den Streaming Frameworks versucht, ein allgemeines Konzept zur Beschreibung des Algorithmus zur Messung der Prototypen vorzustellen. Die vorgestellten Sequenzdiagramme in Abbildung 5.3 und 5.4 dienen bei der Implementierung der Prototypen als Vorgabe. Aufgrund der Spezialisierung der Prototypen, kann es in der Implementierung Abweichungen in der Erzeugung der Nachrichten zur Übergabe an die *Consumer* geben. Detailierter werden die Prototypen im Kapitel 6 beschrieben.

In Abbildung 5.3 wird allgemein für die Prototypen ein Erzeuger von Werten in einer endlosen Schleife dargestellt. Da die Prototypen als eigenständige Anwendungen entweder in einem eigenem Systemprozess oder innerhalb des Streaming Frameworks ausgeführt werden, wird das Beenden des Prozesses durch den Anwender ausgelöst. Beim Anwendungsstart wird in der Schleife eine Methode *generateValue()* für die Erstellung der Nachricht aufgerufen. An dieser Stelle entstehen bei der Implementierung eines Prototypen zwei Variationen. Die erste Variation implementiert die Erzeugung eines Wertes mit konstanter Dimension und die zweite

[13] Apache Zookeeper http://zookeeper.apache.org/

Variante implementiert die Erzeugung von Werten mit variabler Dimension. Dabei wird in der zweiten Variation Text aus dem Datensatz Shakespeare [Sha94] ausgelesen und in einzelne Worte getrennt. Der Datensatz wird dauerhaft durchlaufen und Nachrichten werden pro Wort durch den *Producer* gesendet.

Die Abbildung 5.4 zeigt allgemein ein Sequenzdiagramm eines Prototypen für das Abholen von Nachrichten, die von einem Producer ursprünglich gesendet wurden.

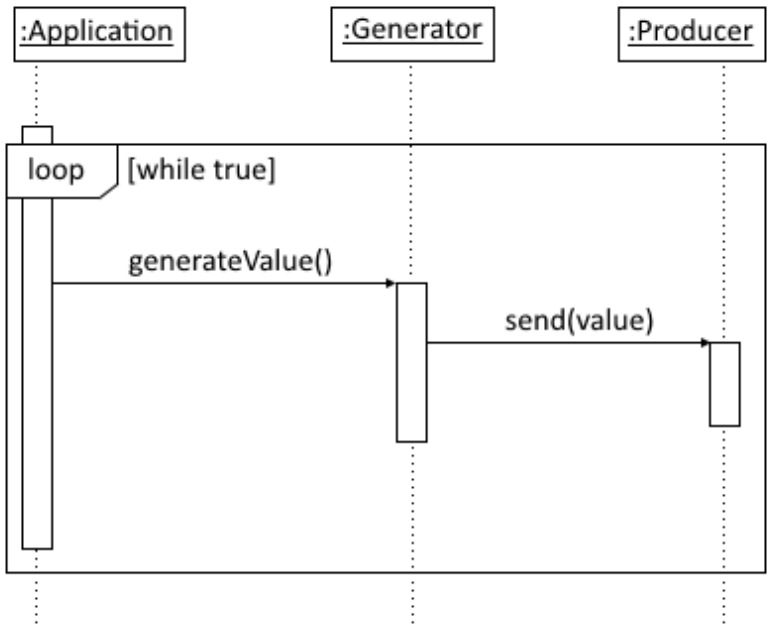

Abbildung 5.3: Sequenzdiagramm Protoyp Producer

Das folgende Kapitel beschreibt als nächstes ausgehend von den Anforderungen, den Lösungsansätzen und dem Systementwurf ein Vorgehen für die Implementierung der Prototypen und der notwendigen Zusatzanwendungen. Mit dem Anwendungsstart wird in einer Schleife dauerhaft auf Nachrichten geprüft. Sobald eine Nachricht vorhanden ist, wird die Nachricht vom *Consumer* geholt und im Prototypen konvertiert. An dieser Stelle entstehen in einem Prototypen zwei Variationen. In der ersten Variation wird die empfangene Nachricht aufgenommen, es wird die Anzahl der Nachrichten pro Sekunde gezählt und nach jeder Sekunde wird die Anzahl der Nachricht zur nächsten Sekunde an den *Logger* weitergegeben. Zu der ersten Variation kommt in der zweiten Variation zusätzlich zur Übergabe der Log-Information an den *Logger* die Übergabe der Log-Information an die Übersichtsseite über *WebSocket*. Dazu wird die zusätzliche Implementierung der *WebSocket*-Anwendung eingesetzt. Auch der *Consumer* läuft entweder im Cluster eines Streaming Framework oder in einem Systemprozess. Daher wird der *Consumer*-Prozess durch den Anwender gestoppt. Nachdem der Algorithmus allgemein für den *Producer* und *Consumer* beschrieben wurde, wird im nächsten Kapitel das Vorgehen zur Entwicklung eines Prototypen gezeigt. Dabei wird auf Werkzeuge und Verfahren, die zur einfachen und sicheren Prototyp-Entwicklung eingesetzt werden, eingegangen.

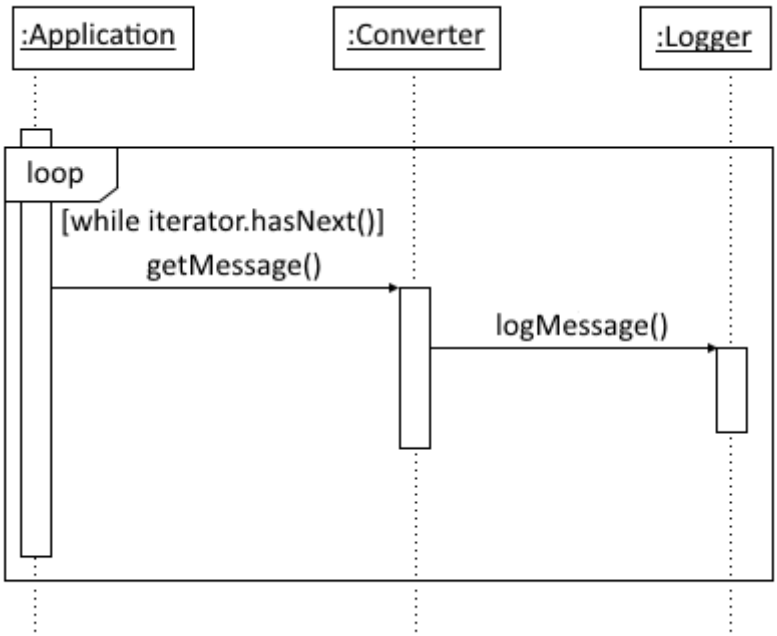

Abbildung 5.4: Sequenzdiagramm Protoyp Consumer

5.7 Vorgehen

Bevor mit der Implementierung begonnen werden kann, ist es notwendig zu Beginn den allgemeinen Vorgang der Entwicklung der Prototypen nachvollziehen zu können. Vor der Implementierung der Prototypen ist die Bedingungen einer Voraussetzung für ein Streaming Framework dem Grundgerüst bereitzustellen. Allgemein entspricht dem Grundgerüst eine lauffähige Fassung eines Streaming Frameworks in einem Single-Node-Cluster. Um den Prototypen bereitzustellen, zu starten und zu stoppen müssen Werkzeuge in Form von Bash-Scripts[14] entwickelt werden. Da jedes Streaming Framework unterschiedliche Optionen für die Anwendung der Prototypen besitzt, werden die Bash-Scripte während dem Implementierungsprozess innerhalb eines Prototypen entwickelt.

Allgemein wird für die Entwicklung eines Prototypen zuerst mit Apache Maven eine Projekt Struktur angelegt und die Konfigurationsdatei für das Deployement eingerichtet. Anschließend wird mit Apache Maven ein Projekt für die Entwicklungsumgebung *Eclipse*[15] angelegt. Da mit der Versionsverwaltung *git*[16] gearbeitet wird, wird nach der Erzeugung des *Eclipse*-Projekts und damit der Einrichtung des Prototypen ein initaler *Commit*[17] in die Versionsverwaltung erstellt. Ein *Commit* wird immer in einem funktionierendem Zustand des Projekts erstellt. Da *git* in einem *Commit* nur lokale Änderungen hinzufügt, müssen die *Commits* mit dem Kommando *push* an den zentralen Server übermittelt werden. In der Versionsverwaltung stehen

[14] Scripte sind Anwendungen, die in einer Shell ausgeführt werden können. Bash steht für die GNU Bourne Again SHell: http://tiswww.case.edu/php/chet/bash/bashtop.html

[15] Java Integrated Development Environment (IDE) Eclipse: https://www.eclipse.org/

[16] Git Versionsverwaltung: http://git-scm.com/

[17] Mit einem *Commit* werden die Änderungen unter einem eindeutigen Referenzwert in die Versionsverwaltung aufgenommen.

im *Repository*[18] *streaming-frameworks* für die Erfassung der Änderungen zwei Zweige, der Entwicklungszweig *thesis* und der Hauptzweig *master* bereit. Zu Beginn jeder Woche werden die neuen Entwicklungen aus dem Entwicklungszweig in den Hauptzweig überführt und gekennzeichnet. Entwickelt wird nur auf dem Entwicklungszweig.

Die Entwicklung eines Prototypen teilt sich in zwei Teile. Der erste Teil entspricht der Implementierung einer Methode für die Prüfung von konstanten Werten. Das Logging-Framework bekommt so schnell wie möglich konstant und kontinuierlich 100-Byte große Datenpakete. Diese werden pro Sekunde gezählt und vom Logging-Framework ausgegeben. Der zweite Teil implementiert eine Methode für die Prüfung von variablen Werten. Dabei wird Text eingelesen, nach Worten getrennt, diese pro Sekunde gezählt und an das Logging-Framework weitergegeben.

Bei Fertigstellung eines Prototypen wird abschließend ein Java-Paket erzeugt und in die Versionsverwaltung hinzugefügt. Für eine binäre Datenhaltung ist die Versionsverwaltung *git* nicht geeignet. Unterschiede zwischen den Versionen von binären Dateien können von einer Person in einem *git repository* nicht nachvollzogen werden. Um den Aufwand klein zu halten und aufgrund der prototypischen Entwicklung, wird dennoch in *git* unter dem Verzeichnis *target*[19] innerhalb eines *Eclipse*-Projekts die aktuelle Version hinzugefügt.

Für das Logging wird einmalig ein Werkzeug benötigt, um den Datentransport zwischen dem Prototypen und der Übersichtsseite herzustellen. Dazu wird bevor der erste Prototyp implementiert wird ein virtueller Server mit dem Apache WebServer und einer Übersichtsseite bereitgestellt. Dabei wird mit Twitter Bootstrap[20] ein Einseitenlayout angelegt und mit Javascript werden die Hypertext Markup Language (HTML)-Inhalte über Javascript-Handler aktualisiert. Die Visualisierung von Diagrammen erfolgt über zusätzliche Javascript-Bibliotheken[21]. Zusätzlich wird auf der gleichen virtuellen Instanz ein NodeJs WebSocket Server bereitgestellt. Die Logging-Daten werden von den Prototypen gesendet, vom NodeJs WebSocket Server empfangen und an die Übersichtsseite per WebSocket verteilt.

Damit aus den Prototypen heraus per *WebSocket* Nachrichten an den *NodeJs* Server verschickt werden können, wird über eine zusätzliche Java-Bibliothek *Tyrus*[22] und einem *Eclipse*-Projekt ein einfaches Werkzeug implementiert. Dazu werden Java-Schnittstellen für die Konfigurationsklasse der *WebSocket* Server benutzt. Die implementierten Konfigurationsklassen werden in dem Prototyp-Projekt bereitgestellt und über *resource*-Dateien werden die Schnittstellen den konkreten Klassen zugeordnet. Dabei soll keine zusätzliche Bibliothek benutzt werden, sondern es wird der existierende *ServiceLoader*[23] für das Zusammenführen zwischen der Schnittstelle und der konkreten Implementierung aus dem *Namespace java.util* benutzt.

Das Vorgehen wurde detaillierter beschrieben und auf spezielle Aspekte, wie den zusätzlichen Werkzeugen wurde eingegangen. Im nächsten Kapitel wird eine Zusammenfassung zum Kapitel

[18] Ein *git repository* bildet einen Datencontainer mit allen Änderungen, die zu einem Projekt hinzugefügt wurden.

[19] In einem produktiven Betrieb muss eine andere Build- und Deployment-Strategie für die zentrale Bereitstellung von versionierten Artefakten herangezogen werden. JetBrains TeamCity https://www.jetbrains.com/teamcity/ oder Jenkins http://jenkins-ci.org/ ermöglichen eine kontinuierliche Integration von geändertem Quelltext.

[20] Twitter Bootstrap: http://getbootstrap.com/

[21] Javascript library Data Driven Documents (D3): http://d3js.org/

[22] Java WebSocket library Tyrus: https://tyrus.java.net/

[23] Java ServiceLoader: http://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html

5 gegeben.

5.8 Zusammenfassung

Im Kapitel 5 wurde die Entwicklung eines Prototypen beschrieben. Zu Beginn wurden Anforderungen definiert und mögliche Probleme betrachtet sowie Lösungsansätze vorgestellt. Speziell wurde dabei auf den LRB eingegangen und in einem Teil in der Entwicklung des Algorithmus berücksichtigt. Weiterhin wurde ein Systementwurf vorgestellt und in einer kleinen Grafik der Zusammenhang zwischen den einzelnen Komponenten aus dem Bereich *Backend*, *Visualization* und *Display* gezeigt und erläutert. Anknüpfend darauf wurden die Geräte vorgestellt, die bei der Messung und bei der Entwicklung zum Einsatz kommen. Um die Performance des langsamsten Elements herauszufinden wurde eine Messung der Hitachi-Festplatte durchgeführt und im Anhang bereitgestellt. Aufbauend auf den Definitionen aus den Kapiteln zuvor, wurde ein Vorgehen für das Entwickeln eines Prototypen dargelegt. Dabei wurde versucht das Vorgehen der Implementierung in den einzelnen Prototypen gleich zu halten. Im Kapitel 6 werden als nächstes die einzelnen Prototypen dokumentiert.

Kapitel 6

Prototypdokumentation

Das vorgehende Kapitel 5 hat den Systementwurf, die Anforderungen an die Prototypen und das Vorgehen erläutert. Mögliche Probleme und Lösungsansätze wurden aufgegriffen und erläutert. In diesem Kapitel werden die einzelnen Prototypen dokumentiert und die wesentlichen Methoden bezogen auf den Algorithmus aus Kapitel 5.6 erläutert.

In der Dokumentation der Prototypen wird der vollständige Quelltext im Rahmen dieser Arbeit nicht erklärt. Dennoch werden technische Details und wesentliche Methoden speziell erläutert. Um den Schwerpunkt auf den Prototypen zu legen, wird zunächst der Aufbau und die Struktur der einzelnen Java Projekte vorgestellt. Allgemeine Arbeitsschritte und die zusätzlich entwickelten Anwendungen werden zuerst beschrieben. Anschließend werden die Prototypen nacheinander dokumentiert.

6.1 Aufbau und Struktur

In diesem Kapitel wird der allgemeine Aufbau und die Struktur für die einzelnen Java-Projekte beschrieben. Die implementierten Prototypen bauen auf dieser Struktur auf und haben damit die gleiche Projekt-Basis, sind jedoch unabhängig. Alle Projekte in dieser Arbeit wurden mit Apache Maven erstellt und für die Entwicklungsumgebung *Eclipse* vorbereitet. Da für die Entwicklung der Anwendungen die Versionsverwaltung *git* eingesetzt wird, werden alle implementierten Anwendungen im *git repository streaming-frameworks* unter dem Verzeichnis *prototype* hinzugefügt. Ausgehend von dem Verzeichnis *prototype* werden verschiedene Verzeichnisse für das *Backend* und der *Visualization* bereitgestellt. Im Unterverzeichnis *prototypes* liegt das *Backend*. Darin sind alle Prototype-Projekte enthalten. Das Unterverzeichnis *helper* besteht aus dem *websocketclient* und den *scriptAndConfigs* Verzeichnis. Der *Websocketclient* unterstützt die Prototypes über den *Broker* in der Weiterleitung der Log-Informationen an die Übersichtsseite. Die erzeugte Bibliothek wird in dem Verzeichnis *protoMavenRepository* hinzugefügt. Die einzelnen Protoypes benötigen dazu einen Konfigurationseintrag unter dem Bereich *dependency* in der Datei *pom.xml*. Das Verzeichnis *scriptAndConfigs* enthält ausführbare Scripte für das Bereitstellen und den Start der Prototypen auf Shell[1]-Ebene. Der *Broker* vermittelt die WebSocket Nachrichten als Dienst zwischen dem *Backend* und der *Visualization*. Das Unterverzeichnis *webApp* entspricht zuletzt der *Visualization* und enthält die

[1] Unter Shell wird an dieser Stelle die Kommandoebene unter dem Betriebssystem Linux verstanden.

Übersichtsseite für den Apache WebServer zur Darstellung auf dem WebClient. In der Abbildung 6.1 werden die einzelnen Prototypen wie der Broker, der WebServer und der WebBrowser dargestellt. Der Broker wird als einzelner Javascript WebSocket-Server implementiert. Damit die Streaming Framework Prototypen nicht über Broker-Systemprozess arbeiten und um den Aufwand gering zu halten, wird jedem Streaming Framework ein Broker in einem eigenen Systemprozess bereitgestellt.

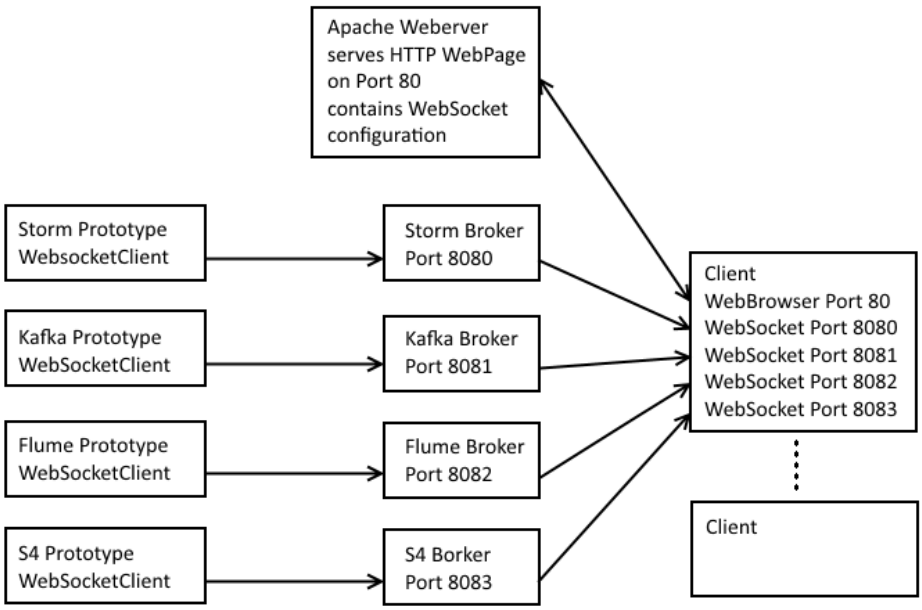

Abbildung 6.1: Systemübersicht

Die Übersichtsseite enthält die Verbindungsinformationen für die Broker. Dadurch können mehrere WebBrowser die Übersichtsseite öffnen und den aktuellen Zustand der Broker beziehen und darstellen. Bei dem erstem Zugriff auf die Übersichtsseite werden alle vier Diagramme der Prototypen noch leer angezeigt. In Abbildung 6.2 wird die Übersichtsseite beim ersten Start dargestellt. Im unteren Bereich der Übersichtsseite werden die Informationen und Fehler zu den Verbindungsaufbau zu den Brokern und der Verarbeitung aufgelistet.

Exemplarisch ist der Storm Broker im Listing A.10 dokumentiert. Für die anderen Prototypen muss die Variable `wsSocketPort` in Zeile 3 im Listing A.10 entsprechend der Portnummer angepasst werden. Für die Ausführung wurden entsprechend in den Unterverzeichnissen lauffähige Broker bereitgestellt.

Die Broker-Systemprozesse werden nach einander in der Shell im Verzeichnis `prototype/broker` mit folgendem Kommando gestartet:

```
>node storm/server.js
>node kafka/server.js
>node flume/server.js
>node s4/server.js
```

In der Übersichtsseite sind alle Javascripte für die WebSocket Verbindungen und die Darstellung der Log-Information mit der D3-Bibliothek angegeben. Im Listing A.11 wird der Quelltext für die D3 Grafik bereitgestellt. Sobald über die Javascript Funktion `webSocketManager`

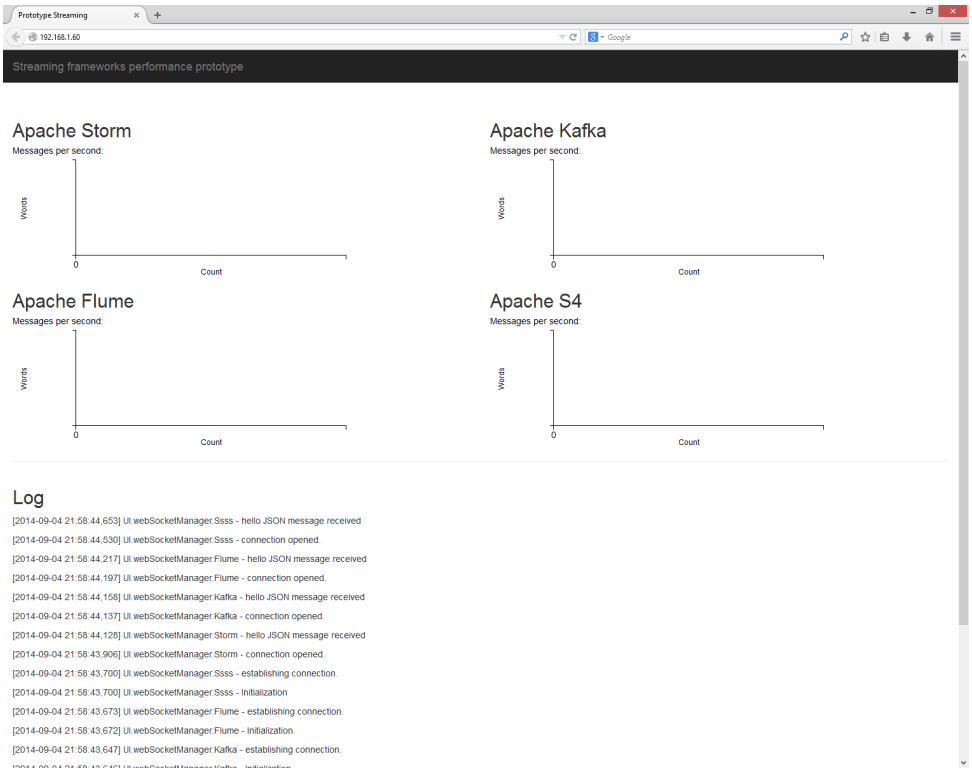

Abbildung 6.2: Systemübersicht bei erstem Start

Nachrichten über den *Trigger* `messgeIncome` eingehen, kann über den *Javascript Handler* im Listing A.12 in Zeile 8 die Methode `update` aus der D3 Grafik Listing A.11 in Zeile 127 aufgerufen werden. Da die Nachrichten vom Prototypen jede Sekunde gesendet werden, wird die D3 Grafik pro Sekunde aktualisiert.

Für die Verbindung zwischen den Prototypen und dem Broker wird im Verzeichnis `prototype /helper/websocketclient` Java ein Projekt `websocketclient` bereitgestellt. In den Prototypen wird das Java Archiv referenziert und die Instanz für den WebsocketClient über die Factory Klasse mit der richtigen Konfiguration zurückgegeben. In Listing A.13 wird über die Methode `getInstance()` in Zeile 10 die Rückgabe der Instanz für den WebSocketClient gezeigt.

Allgemein werden die Prototypen im Verzeichnis `target` mit Apache Maven mit folgendem Kommando erzeugt:

```
>mvn package
```

Allerdings müssen für den Prototypen Apache S4 jeweils eigene Java Archive für die einzelnen Variationen erstellt werden. Die folgende Kommandos erzeugen die einzelnen Variationen:

```
>mvn package -P intProducer
>mvn package -P intConsumer
>mvn package -P wordCountProducer
>mvn package -P wordCountConsumer
```

Nachdem die allgemeinen Werkzeuge vorgestellt wurden, werden als nächstes die Prototypen dokumentiert.

6.2 Prototype Apache Storm

Im Kapitel zuvor wurden notwendige Werkzeuge für die Entwicklung der Prototypen vorgestellt. In diesem Kapitel wird der Prototyp Storm beschrieben.

Im Listing 6.1 wird ein Auszug des Storm Prototypen dargestellt. Eine Instanz vom `Topology Builder` wird erzeugt, mit spezifischen Storm-Primitiven konfiguriert und mit der Methode `submitTopology` in Zeile 15 in das Storm Cluster bereitgestellt. Das Einlesen und teilen des Textes wird in der Klasse `WordGeneratorSpout` dauerhaft ausgeführt und über `Tupel`, einem Storm-Datentyp weitergegeben. In der Klasse `CountPerSecondBolt` werden die Worte aufgenommen, gezählt und pro Sekunde in einem `Tupel` formatiert weitergegeben. In den Klassen `PrintBolt` und `WebSocketBolt` werden jeweils die Log-Information an das Logging-Framework *log4j* und an den Storm-*Broker* weitergegeben. Verbunden werden der `Spout` und die `Bolts` über die eindeutigen Namen beim zuweisen der Spouts und Bolts in der Topology, wie zum Beispiel in Zeile 2 *wordGenerator*. Weiterhin wird bei der Konfiguration des `Bolt` `CountPerSecondBolt` ein `fieldsGrouping` auf den Schlüssel *word* gruppiert. Bei den anderen `Bolts` und dem `Spout` wird standardmäßig ein `shuffleGrouping` eingesetzt. Dabei werden die eintreffenden `Tupel` gleichmäßig über die `Tasks` im Storm Cluster verteilt verarbeitet.

Listing 6.1: Java Prototype Storm App Word Counter

```
1  TopologyBuilder builder = new TopologyBuilder();
2  builder.setSpout("wordGenerator", new WordGeneratorSpout(1000), 8);
3  builder.setBolt("countPerSecond", new CountPerSecondBolt(), 1).
       fieldsGrouping("wordGenerator", new Fields("word"));
4  builder.setBolt("printWord", new PrintBolt(), 1).shuffleGrouping("
       countPerSecond");
5  builder.setBolt("sendCountWords", new WebSocketBolt()).shuffleGrouping("
       printWord");
6
7  Config conf = new Config();
8  conf.setDebug(false);
9
10 [...]
11
12 conf.setNumWorkers(1);
13 conf.put("fileToRead", "shaks12.txt");
14 LocalCluster cluster = new LocalCluster();
15 cluster.submitTopology("wordCounter", conf, builder.createTopology() );
```

Der *Counter* für die konstante Dimension von Werten ist in der Klasse `AppIntegerCounter` implementiert. Die Implementierung gleicht der Implementierung der variablen Dimension von Werten mit der Klasse `AppWordCounter`. Anstelle vom Einlesen einer Textdatei werden `Bytes` von konstanter Größe mit der Dimension 100 in der Klasse `IntegerGeneratorSpout` eingelesen. Weiterhin wird die Log-Information nur an das Logging-Framework ausgegeben. In Listing 6.2 wird ein Auszug der Klasse `AppIntegerCounter` dargestellt.

Listing 6.2: Java Prototype Storm App mit konstanter Dimension

```
1  TopologyBuilder builder = new TopologyBuilder();
```

```
builder.setSpout("integerGenerator", new IntegerGeneratorSpout(), 8);
builder.setBolt("countPerSecond", new CountPerSecondBolt(), 1).
    shuffleGrouping("integerGenerator");
builder.setBolt("printWord", new PrintBolt(), 1).shuffleGrouping("
    countPerSecond");
```

Die Topologien werden in Storm mit den Scripten `deployInteger.sh` und `deployWord.sh` bereitgestellt. Das nächste Kapitel dokumentiert den Prototyp zu Apache Kafka.

6.3 Prototype Apache Kafka

In Apache Kafka wird allgemein ein `KafkaProducer` instanziiert, ein `Producer` geholt, Daten in einem `KeyedMessage` verpackt und mit der Methode `send` aus `Producer`-Instanz verschickt. In Listing 6.3 wird in Zeile 4 ein `Producer` aus dem `KafkaProducer` geholt. In Zeile 31 wird ein Wort in einem `KeyedMessage` verpackt und in Zeile 33 mit der Methode `send` verschickt.

Listing 6.3: Java Prototype Kafka App mit variabler Dimension

```
1  public void start() throws IOException {
2    List<String> lines = transpose.readAllLines("/" + fileToRead);
3    KafkaProducer kafkaProducer = new KafkaProducer();
4    Producer<Integer, byte[]> producer = kafkaProducer.GetProducer();
5    int currentRetry = 1;
6    int currentIndex = 0;
7    while (currentRetry <= retries) {
8      while (currentIndex < lines.size()) {
9        splitLineAndSendWord(producer, lines.get(currentIndex));
10       currentIndex++;
11     }
12     currentIndex = 0;
13     currentRetry++;
14   }
15   producer.close();
16 }
17
18 private void splitLineAndSendWord(Producer<Integer, byte[]> producer,
       String line) {
19   for (String word: line.split(" ")) {
20     String preparedWord = convert.prepare(word);
21     if (!preparedWord.equals("")) {
22       sendWord(producer, preparedWord);
23     }
24   }
25 }
26
27 private void sendWord(Producer<Integer, byte[]> producer, String word) {
28   Map<String, byte[]> hashMap = new HashMap<String, byte[]>();
29   hashMap.put("word", word.getBytes());
30   try {
31     KeyedMessage<Integer, byte[]> keyedMessage =
32     new KeyedMessage<Integer, byte[]>(this.topicName, convert.toByteFrom
         (hashMap));
33     producer.send(keyedMessage);
34   } catch (IOException e) {
35     e.printStackTrace();
36   }
```

```
37 }
```

Auf der Consumer-Seite läuft es genau umgekehrt. Zuerst wird die Klasse KafkaConsumer instanziiert und aus der Instanz mit der Methode GetFirstKafkaStream(topics, topicName) aus der Liste von topics ein bestimmter topicName ermittelt und ein KafkaStream zurückgegeben. Mit dem Iterator aus dem KafakStream können anschließend in einer Schleife die einzelnen Nachrichten geholt und ausgegeben werden.

In der Klasse KafkaWordCountProducer werden die Texte eingelesen, in Worte getrennt und in eine Liste geschrieben. Die Klasse AppIntegerProducer überspringt diesen Schritt und es wird eine konstante Größe von Worten mit der Dimension 100 als Byte vom Producer gesendet.

Die Klasse KafkaWordCountConsumer empfängt die Nachrichten, zählt diese in der Methode printMessagePerSecond() und gibt die Log-Information an das Logging-Framework und den Kafka-*Broker* weiter. Der KafkaIntegerConsumer hingegen gibt die Log-Information nur an das Logging-Framework weiter.

Vor dem Start eines Producers und Consumers muss ein Kafka-Server gestartet werden und ein *Topic* existieren und anschließend unter dem die Producer und Consumer Nachrichten austauschen. Die folgenden zwei Kommandos starten einen Server und erstellen zwei *Topics* integerTopic und wordCountTopic:

```
>./bin/kafka-server-start.sh config/server.properties
>./bin/kafka-topics.sh --create --zookeeper localhost:2181 //
  --replication-factor 1 --partition 1 --topic integerTopic
>./bin/kafka-topics.sh --create --zookeeper localhost:2181 //
  --replication-factor 1 --partition 1 --topic wordCountTopic
```

Der Start eines Producers und Consumers wird mit den folgenden Kommandos ausgeführt:

```
>./bin/startIntProducer.sh
>./bin/startIntConsumer.sh
>./bin/startWordProducer.sh
>./bin/startWordConsumer.sh
```

Die wesentlichen Methoden des Prototyps zu Apache Kafka und die Ausführung mit den Scripten wurden dokumentiert und erläutert. In dem nächsten Kapitel wird der Prototype Apache Flume dokumentiert.

6.4 Prototype Apache Flume

Das Listing 6.4 zeigt einen Auszug der *WordCount*-Implementierung. Die Klasse WordCount Source erweitert dabei die abstrakte Flume-Klasse AbstractSource und implementiert die beiden Flume-Schnittstellen Configurable und PollableSource. Innerhalb der überschriebenen Methode process() wird sobald der Text vollständig in Worte getrennt und die Schleife vollständig durchlaufen wurde, der Status READY zurückgegeben und die Transaktion wird

bestätigt. Wenn ein Fehler eintritt wird der Status `BACKOFF` übergeben und die Transaktion wird rückgängig gemacht. Im Listing 6.4 wird in Zeile 25 eine Nachricht über den `EventBuilder` zu einem `Event` verpackt und durch die Methode `getChannelProcessor()` der `Channel` aus der `AbstractSource` geholt. Abschließend wird das `Event` mit der Methode `processEvent(event)` an den `Channel` verschickt.

Listing 6.4: Java Prototype Flume App mit variabler Dimension

```
1  @Override
2  public Status process() throws EventDeliveryException {
3    Status status = Status.READY;
4    int currentIndex = 0;
5    while (currentIndex < lines.size()) {
6      splitLineAndSendWord(lines.get(currentIndex));
7      currentIndex++;
8    }
9    return status;
10 }
11
12 private void splitLineAndSendWord(String line) {
13   for (String word: line.split(" ")) {
14     String preparedWord = convert.prepare(word);
15     if (!preparedWord.equals("")) {
16       sendWord(preparedWord);
17     }
18   }
19 }
20
21 private void sendWord(String word) {
22   Map<String, byte[]> hashMap = new HashMap<String, byte[]>();
23   hashMap.put("word", word.getBytes());
24   try {
25     Event event = EventBuilder.withBody(convert.toByteFrom(hashMap));
26     getChannelProcessor().processEvent(event);
27   } catch (IOException e) {
28     e.printStackTrace();
29   }
30 }
```

Auf der Gegenseite wird ähnlich wie in der Klasse `WordCountSource` die Klasse `WordCountSink` implementiert. Dabei wird die abstrakte Flume-Klasse `AbstractSink` erweitert und die Schnittstelle `Configurable` implementiert. In der überschriebenen Methode `process()` wird der `Channel` aus der abstrakten Klasse `AbstraktSink` mit der Methode `getChannel()` geholt. Der `Channel` stellt eine `Transaction` bereit. Zuerst wird die Transaktion mit der Method `begin()` gestartet. Mit der Methode `take()` wird aus dem `Channel` die Nachricht als `Event` wieder geholt. Wenn es aber kein `Event` gibt, wird der Status `BACKOFF` zurück gegeben und die Transaktion wird zurückgeführt. Andernfalls wird die Nachricht aus dem Event umgewandelt und an das Logging-Framework und den Flume-*Broker* weitergegeben. Die Transaktion wird abschließend bestätigt und der Status `READY` wird zurückgegeben.

Die Implementierungen für die konstanten Wortlängen verhalten sich genauso wie in der Klasse `WordCountSource` und der Klasse `WordCountSink`. Der Unterschied ist dabei das Wegfallen des Einlesens des Textes in der Klasse `ConsoleSource` und der Ausgabe an den Flume-*Broker* in der Klasse `ConsoleSink`.

Bei der Ausführung des Flume Prototypen muss die Java Archiv-Datei in Flume Verzeichnis lib liegen. Zusätzlich müssen über eine Konfigurationsdatei Flume für die Sink- und Source-Dateien vorbereitet werden. Das Listing 6.5 zeigt für die Klasse `WordCountSource` und die Klasse `WordCountSink` eine Konfiguration. Für die Klasse `ConsoleSource` und der Klasse `ConsoleSink` liegt im Flume-Script-Verzeichnis die Konfigurationsdatei `flume-conf-WordCount.properties` bereit.

Listing 6.5: Java Prototype Flume App mit variabler Dimension Konfiguration

```
1 agent.sources = r1
2 agent.channels = memoryChannel
3 agent.sinks = k1
4
5 agent.sources.r1.type = lan.s40907.protopubFlume.WordCountSource
6 agent.sources.r1.channels = memoryChannel
7
8 agent.sinks.k1.type = lan.s40907.protopubFlume.WordCountSink
9 agent.sinks.k1.channel = memoryChannel
10
11 agent.channels.memoryChannel.type = memory
12 agent.channels.memoryChannel.capacity = 10000
13 agent.channels.memoryChannel.transactionCapacity = 10000
14 agent.channels.memoryChannel.byteCapacityBufferPercentage = 20
```

Gestartet werden die einzelnen Variationen der Prototypen Apache Flume in der Shell mit den folgenden Scripten:

```
>./startAgentInteger.sh
>./startAgentWord.sh
```

Der Prototype zu Apache Flume wurde erläutert und der wesentliche Quelltext dokumentiert. Zusätzlich wurde auf die Ausführung und die Konfiguration eingegangen. Im nächsten Kapitel wird abschließend der Prototyp zu Apache S4 dokumentiert.

6.5 Prototype Apache S4

Nach der Dokumentation des Prototypen Apache Flume wird in diesem Kapitel der Prototyp von Apache S4 dokumentiert. Der Prototyp besteht aus drei Klassen: der `App`, dem `Adapter` und dem `ProcessingElement`. Die Klasse `WordCountApp` erweitert die abstrakte S4-Klasse `App`. In der überschriebenen Methode `onInit()` wird die Verarbeitungseinheitsklasse `WordCountProcessingElement` durch die Methode `createPE()` im Singleton-Modus instanziiert. Abschließend wird zu einem eindeutigen Schlüssel mit der Methode `createInputStream` mit dem `ProcessingElement` ein Datenstrom erzeugt. Die Klasse `WordCountInputAdapter` erweitert die Klasse `AdapterApp`, liest den Text ein und trennt nach Worten. Mit der Klasse `Event` wird eine neue Nachricht erstellt. Die Methode `getRemoteStream().put(event)` schreibt die Nachricht mit dem eindeutigen Schlüssel in den `InputStream`. Dazu zeigt das Listing 6.6 das Verpacken eines Wortes in einem `Event` und das Verschicken mit der Methode `getRemoteStream()`. Die Klasse `WordCountProcessingElement` wird von der abstrakten Klasse `ProcessingElement` erweitert, in der Methode `onEvent(event)` wird die Nachricht entpackt sowie dem Logging-Framework und dem S4-*Broker* weitergegeben.

Listing 6.6: Java Prototype S4 App mit variabler Dimension

```
1 private void sendWord(String word) {
2   Event event = new Event();
3   event.put("word", String.class, word);
4   if (getRemoteStream() != null) {
5     getRemoteStream().put(event);
6   } else
7   {
8     logger.error("Cannot get remotestream.");
9   }
10 }
```

Die Implementierung für die konstanten Wortlängen `IntegerInputAdapter`, `IntegerProcessingElement` und `IntegerApp` erfolgt ebenso wie bei den variablen Wortlängen, `WordCountInputAdapter`, `WordCountProcessingElement` und `WordCountApp`. Der Unterschied zu den variablen Wortlängen besteht bei der Erzeugung der Wörter in konstanter Wortlänge anstelle dem Einlesen des Textes und bei der ausschließlichen Ausgabe der Log-Informationen an das Logging-Framework.

Beim Ausführen der Varianten des S4-Prototypen muss zuvor ein S4-Cluster existieren. Mit folgendem Befehl aus S4-Script-Verzeichnis wird ein S4-Cluster in der Shell erzeugt:

```
>./deployNewCluster.sh
```

Und mit den folgenden Befehle aus dem S4-Script-Verzeichnis werden die einzelnen Varianten des S4-Prototypen gestartet:

```
>startIntConsumer.sh
>startIntProducer.sh
>startWordConsumer.sh
>startWordProducer.sh
```

Nach der Dokumentation des S4-Prototypen werden abschließend in der Zusammenfassung Erkenntnisse aus dem Kapitel der Prototype Dokumention zusammen getragen.

6.6 Zusammenfassung

In Kapitel 6 wurden zuerst die notwendigen Werkzeuge für die Entwicklung der Prototypen vorgestellt und erläutert. Dabei wurde das Vorgehen aus Kapitel 5.7 berücksichtigt. Weiterhin wurde die Projektstruktur und der Aufbau gezeigt. Anschließend erfolgte eine Dokumentation der einzelnen Prototypen. Dabei wurde auf den wesentlichen Quelltext eingegangen und der spezielle Quelltext gezeigt und erläutert. Zu jedem Prototyp wurde die Konfiguration und die Ausführung per Kommmando in einer Shell gezeigt und erklärt. Im nächsten Kapitel werden Messungen beschrieben, ausgeführt und die Messergebnisse gezeigt.

Kapitel 7

Evaluierung

In den Kapitel zuvor wurden die Streaming Frameworks Apache Storm, Apache Kafka, Apache Flume und Apache S4 vorgestellt, eine Systemarchitektur für die einzelnen Prototypen definiert und die Implementierungen der Prototypen dokumentiert. In diesem Kapitel wird die Messung durchgeführt. Zuerst wird die Messumgebung beschrieben, anschließend wird jeweils zu den Prototypen eine Messung durchgeführt und das Messergebnis dargestellt. Weiterhin werden die Messergebnisse diskutiert und bewertet. In Kapitel 7.3 wird eine Erkenntnis aus der Bewertung gezogen und im letzten Kapitel eine Zusammenfassung gegeben.

7.1 Messumgebung

Die Messumgebung besteht wie im Kapitel 5 Abbildung 5.1 gezeigt aus zwei Maschinen für die Prototypen, einer virtuellen Maschine für die Verteilung der Log-Informationen und der Darstellung der Metriken. Verbunden sind die einzelnen Maschinen mit einem Switch. Die Messung selbst wird in mehreren Schritten durchgeführt. Zuerst werden schrittweise die Prototypen auf der High-End-Maschine 5.1 ausgeführt. Auf der Low-End-Maschine 5.2 wird die virtuelle Maschine gestartet. Abschließend werden die Maschinen getauscht und auf der High-End-Maschine wird die virtuelle Maschine gestartet und auf der Low-End-Maschine werden die Prototypen schrittweise ausgeführt. Die Ausführung der Prototypen wurden im Kapitel 6 bereits vorgestellt und die notwendige Vorbereitung der Maschine beschrieben. In der folgenden Liste werden die Kommandos für eine klare Übersicht schrittweise aufgezählt.

Ausführung der Dienste auf der virtuellen Maschine

```
>node storm/server.js
>node kafka/server.js
>node flume/server.js
>node s4/server.js
```

Ausführung des Storm-Prototypen

```
>deployInteger.sh
>deployWord.sh
```

Anschließend muss über die Storm-Web-Oberfläche der Prototyp gestartet bzw. gestoppt werden.

Ausführung des Kafka-Prototypen

```
>./bin/kafka-server-start.sh config/server.properties
>./bin/startIntProducer.sh
>./bin/startIntConsumer.sh
>./bin/startWordProducer.sh
>./bin/startWordConsumer.sh
```

Die Ausführung des Kafka-Prototypen kann mit dem gleichzeitigen drücken der Tastenkombination `Steuerung (STRG)` und `C` beendet werden.

Ausführung des Flume-Prototypen

```
>./startAgentInteger.sh
>./startAgentWord.sh
```

Der Flume-Prototype kann ebenfalls mit der Tastenkombination `STRG` und `C` beendet werden.

Ausführung des S4-Prototypen

```
>startIntConsumer.sh
>startIntProducer.sh
>startWordConsumer.sh
>startWordProducer.sh
```

Im S4-Prototype wird die Anwendung ebenfalls über die Tastenkombination `STRG` und `C` beendet.

Nach der Vorstellung des Messaufbaus und der schrittweisen Ausführung der Kommandos der Prototypen, werden im nächsten Kapitel die Messergebnisse nach der Ausführung vorgestellt und beschrieben.

7.2 Benchmark Ergebnisse

Im Kapitel zuvor wurden in einer kurzen Übersicht die notwendigen Kommandos für die schrittweise Ausführung der Prototypen aufgezählt. Dieses Kapitel zeigt die Messergebnisse in einem Diagramm. Das Diagramm zeigt auf der Y-Achse die Anzahl der Nachrichten und auf der X-Achse die Zeit in Sekunden. Für eine gleichmäßige Darstellung der Diagramme wurde die Skala auf der Y-Achse so angepasst, damit die eingetragenen Messergebnisse für den Betrachter unmittelbar sichtbar sind. In einer Legende werden die Messergebnissen aus den einzelnen Messungen eines Prototypen farblich getrennt aufgelistet. Für die Darstellung in der Legende wurden Abkürzungen der einzelnen Messungen eingeführt. Die Farben *Rot* und *Blau* beziehen sich auf die Abkürzung `Hi` und bezeichnen die High-End-Maschine. Die Farben *Orange* und *Grün* beziehen sich auf die Abkürzung `Lo` und bezeichnen die Low-End-Maschine. Die Abkürzungen `Sta` und `Dyn` bedeuten für `Sta` statisch und `Dyn` dynamisch. Dabei steht `Sta` für die Messung mit konstanter Wortlänge und die Abkürzung `Dyn` für die Messung mit variabler Wortlänge. Neben den Nachrichten pro Sekunde wird zeitgleich die Prozessorbelastung in Prozent gemessen. So wird pro Prototyp zusätzlich ein Diagramm mit der Prozessorbelastung vorgestellt. Das Diagramm zeigt auf der Y-Achse die Prozessorbelastung in Prozent und auf der X-Achse die Zeit in Sekunden. Farblich werden die Messungen genauso wie in dem Diagramm für die Messung der Nachrichtenanzahl gekennzeichnet. Die aufgezeichneten Messergebnisse liegen in einzelnen Dateien im Anhang bereit und sind in den folgenden Diagrammen als Messpunkte überführt.

Um einen möglichen Flaschenhals in der Datenverarbeitung auszuschließen, wird die virtuelle Maschine neben den Prototypen ebenfalls auf die Performance untersucht. Im Anhang wird ein Diagramm in Abbildung A.2 für die Messung von Nachrichten pro Sekunde gezeigt. Dabei kann ein Mittelwert von etwa 4000 Nachrichten pro Sekunde abgeleitet werden. Da in der Übersichtsseite[1] der Prototypen nur eine Nachricht pro Sekunde angezeigt wird und die darunterliegenden Broker weniger als 1000 Nachrichten pro Sekunde verarbeiten müssen, kann damit ein Flaschenhals für die beschriebene Messung aus Kapitel 7.1 in der virtuellen Maschine ausgeschlossen werden.

Die Abbildungen 7.1 und 7.2 zeigen die Messungen zum Apache Storm Prototypen. Im Vergleich zwischen der High- und Low-End-Maschine werden während der Berechnung der Worte bei der variablen Wortlänge mehr Nachrichten pro Sekunde verarbeitet. Die High-End-Maschine verarbeitet im Durchschnitt $0{,}81\times10^6$ variable Wortlängen pro Sekunde im Vergleich zur konstanten Wortlänge mit $0{,}73\times10^6$ Nachrichten pro Sekunde. Die Prozessorbelastung ist bei der variablen Wortlänge 10 Prozent höher und liegt bei 53 Prozent. Bei der Low-End-Maschine ist ein ähnliches Verhalten festzustellen. In der variablen Wortlänge werden durchschnittlich $0{,}55\times10^6$ Nachrichten pro Sekunde im Vergleich zur konstanten Wortlänge mit $0{,}51\times10^6$ Nachrichten pro Sekunde übertragen.

Im Unterschied zur High-End-Maschine liegt die Prozessorbelastung bei der konstanten Wortlänge bei 80 Prozent und ist 16 Prozent höher als bei der variablen Wortlänge. Weiterhin gibt es Unterschiede bei der Prozessorbelastung der Low-End-Maschine mit Lastspitzen in der 4ten, 66ten und 151ten Sekunde. Im Unterschied dazu ist die Prozessorbelastung bei der High-End-Maschine nach dem Start der Messung gleichmäßig. Der signifikante Unterschied wird auf die unterschiedlichen Prozessoren bezogen und wird in dieser Arbeit nicht näher betrachtet.

[1] Systemübersicht der Prototypen Abbildung 6.2

In der Storm-Dokumentation[2] wird bei einem Test mit einem spezifischen Prozessor $1{,}0{\times}10^6$ Nachrichten pro Sekunde erreicht. Nähere Angaben zu einem Test werden nicht angegeben und die gemessenen Messewerte unter den beschriebenen Messbedingungen in dieser Arbeit weiter betrachtet. Weiterführende Betrachtungen werden hierzu im Kapitel 7.3 gegeben.

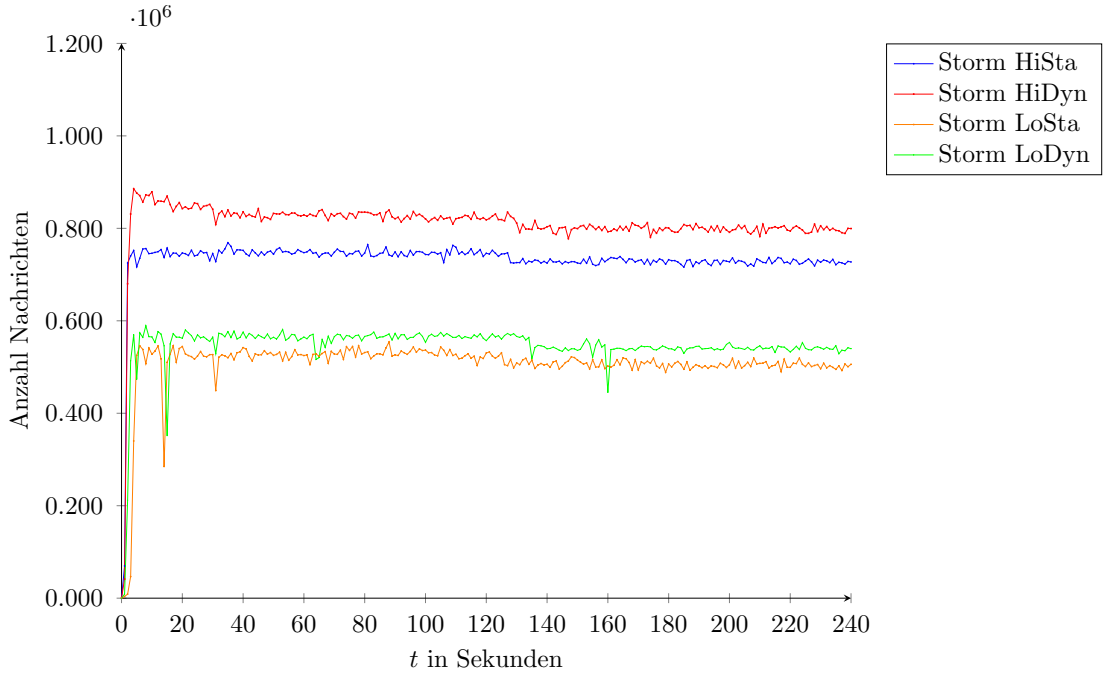

Abbildung 7.1: Messung Apache Storm Nachrichtendurchsatz

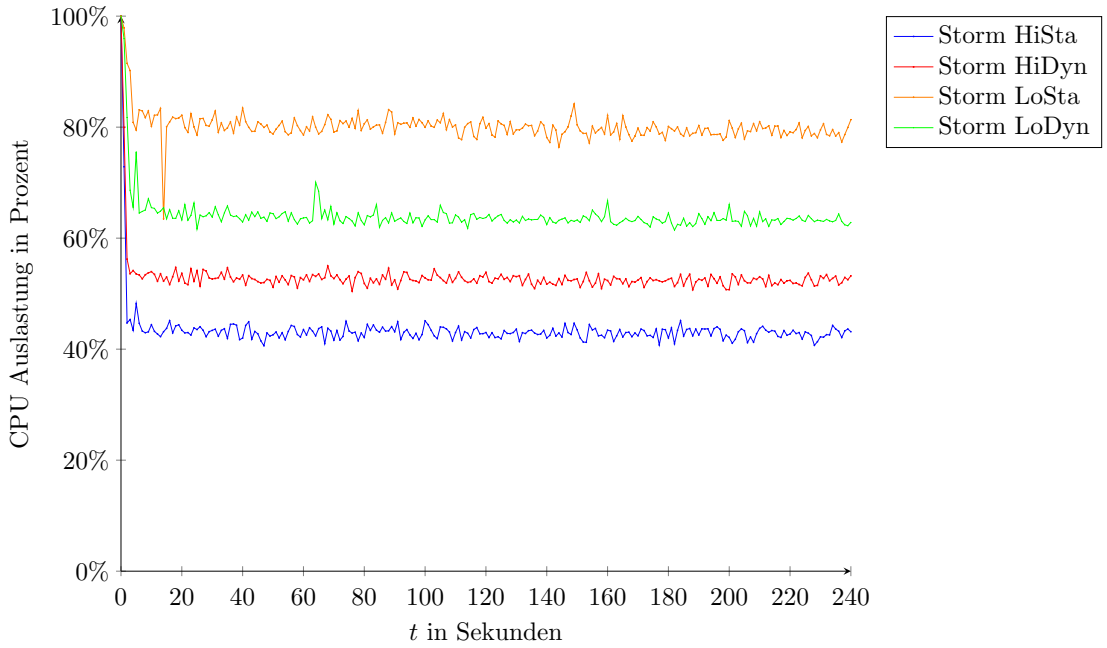

Abbildung 7.2: Messung Apache Storm CPU Auslastung

Die Abbildungen 7.3 und 7.4 zeigen die Messergebnisse zum Apache Kafka Prototypen. Mit

[2] Storm Dokumentation zum Benchmark *one million message per second per node*: https://storm.incubator.apache.org/about/scalable.html

der High-End-Maschine werden im Durchschnitt $1{,}05\times10^5$ Nachrichten pro Sekunde in der Messung mit konstanter Wortlänge im Vergleich zur Messung der variablen Wortlänge im Durchschnitt mit $1{,}0\times10^5$ Nachrichten pro Sekunde übertragen. Die Prozessorbelastung ist bei der Messung mit konstanter Wortlänge im Durchschnitt 2 Prozent niedriger als bei der Messung mit variabler Wortlänge im Durchschnitt mit 41 Prozent.

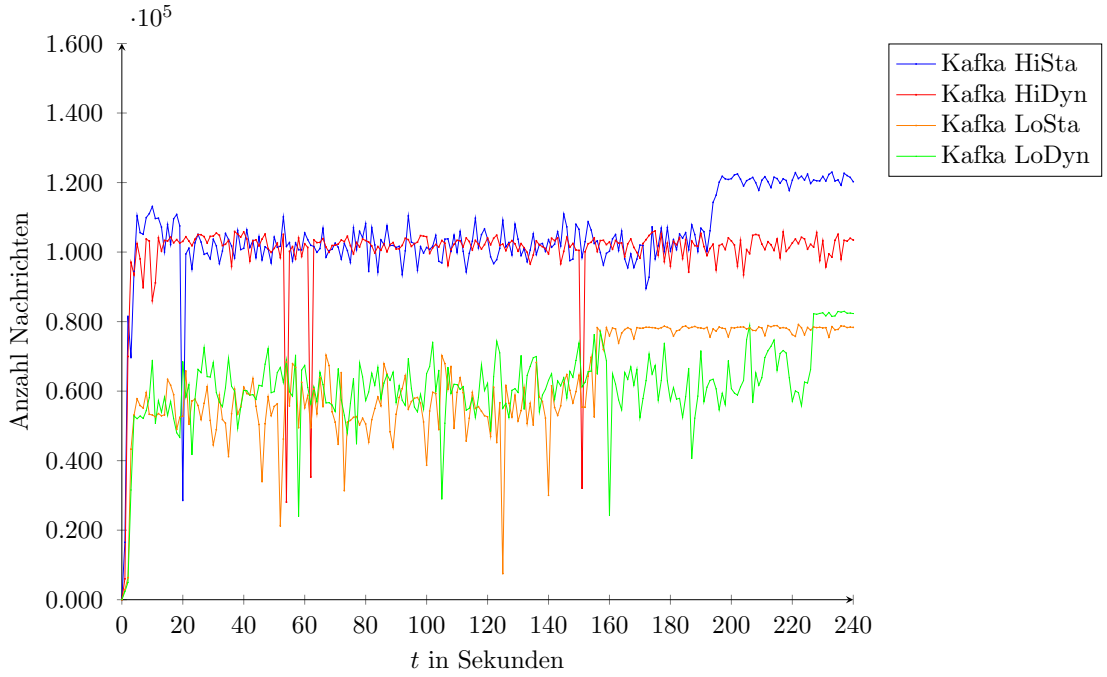

Abbildung 7.3: Messung Apache Kafka Nachrichtendurchsatz

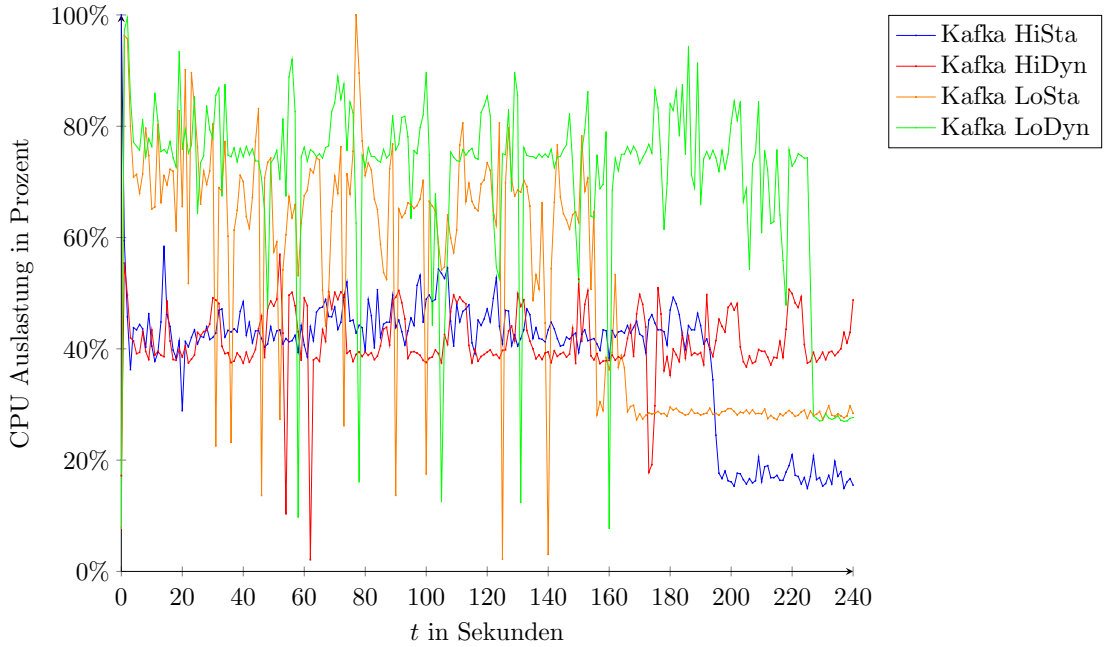

Abbildung 7.4: Messung Apache Kafka CPU Auslastung

Im Vergleich zur Low-End-Maschine verhält sich die Berechnung ähnlich. Es werden bei der Messung mit konstanter Wortlänge im Durchschnitt $0{,}62\times10^5$ Nachrichten pro Sekunde im Vergleich zur Messung der variablen Wortlänge mit $0{,}61\times10^5$ Nachrichten pro Sekunde im Durch-

schnitt übertragen. Der Unterschied der Prozessorbelastung ist in der Low-End-Maschine bei der Messung mit konstanter Wortlänge 19 Prozent niedriger als bei der Messung der variablen Wortlänge mit 71 Prozent. Weiterhin tauchen regelmäßige Lastspitzen in der Prozessorauslastung auf. Die höhere Last erklärt sich durch das Persistieren der Daten auf die Festplatte. Am Ende einer Messung in den variablen Wortlängen Variationen des Apache Kafka Prototypen wird der Prozessor massiv entlastet, da keine weiteren Daten aus einer Datei eingelesen werden. Der Apache Kafka *Channel* arbeitet somit den Datenpuffer ab.

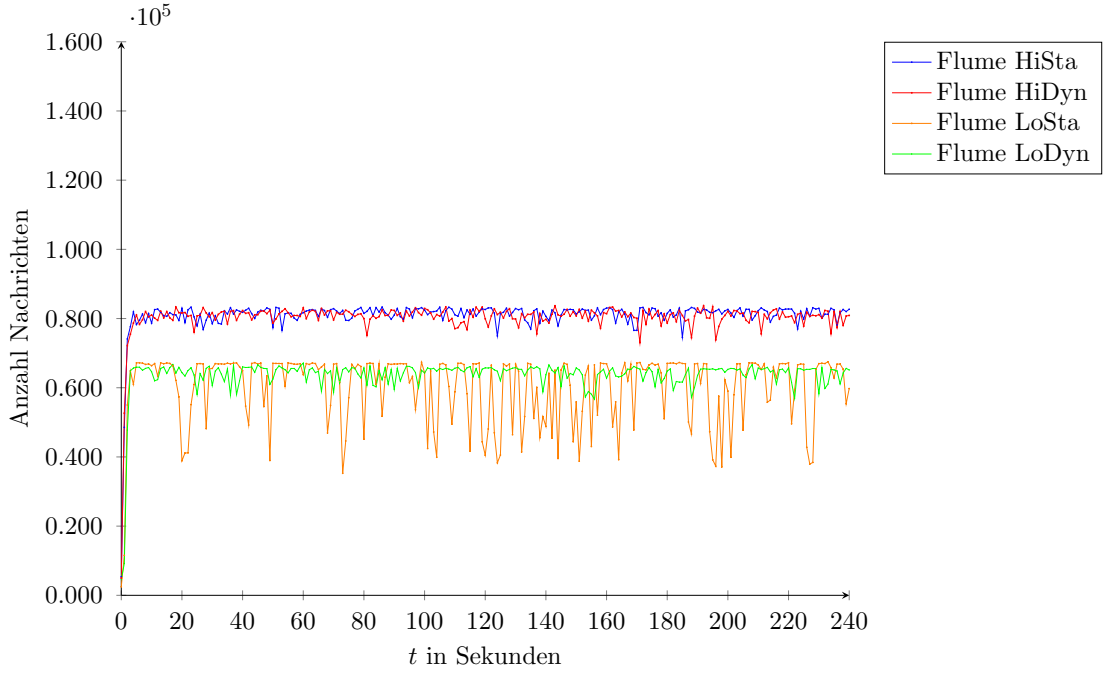

Abbildung 7.5: Messung Apache Flume Nachrichtendurchsatz

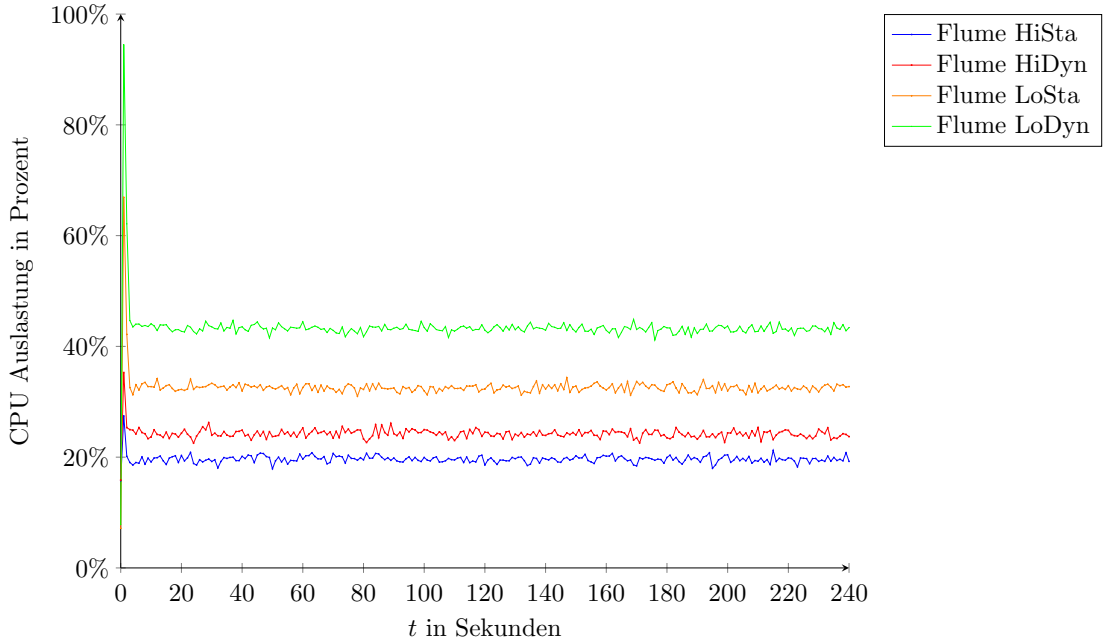

Abbildung 7.6: Messung Apache Flume CPU Auslastung

In Abbildung 7.5 und 7.6 werden die Messungen zu dem Apache Flume Prototypen gezeigt.

Die High-End-Maschine liefert bei der Messung mit konstanter Wortlänge im Durchschnitt $0{,}81\times10^5$ Nachrichten pro Sekunde und bei der Messung mit variabler Wortlänge im Durchschnitt $0{,}80\times10^5$ Nachrichten pro Sekunde.

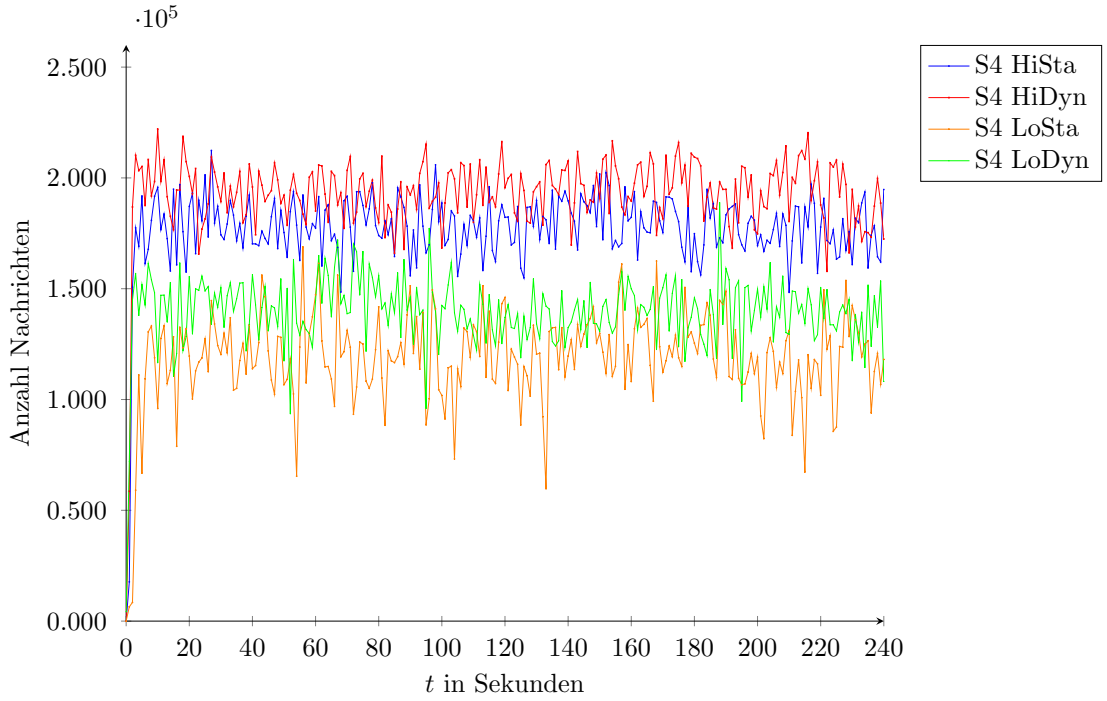

Abbildung 7.7: Messung Apache S4 Nachrichtendurchsatz

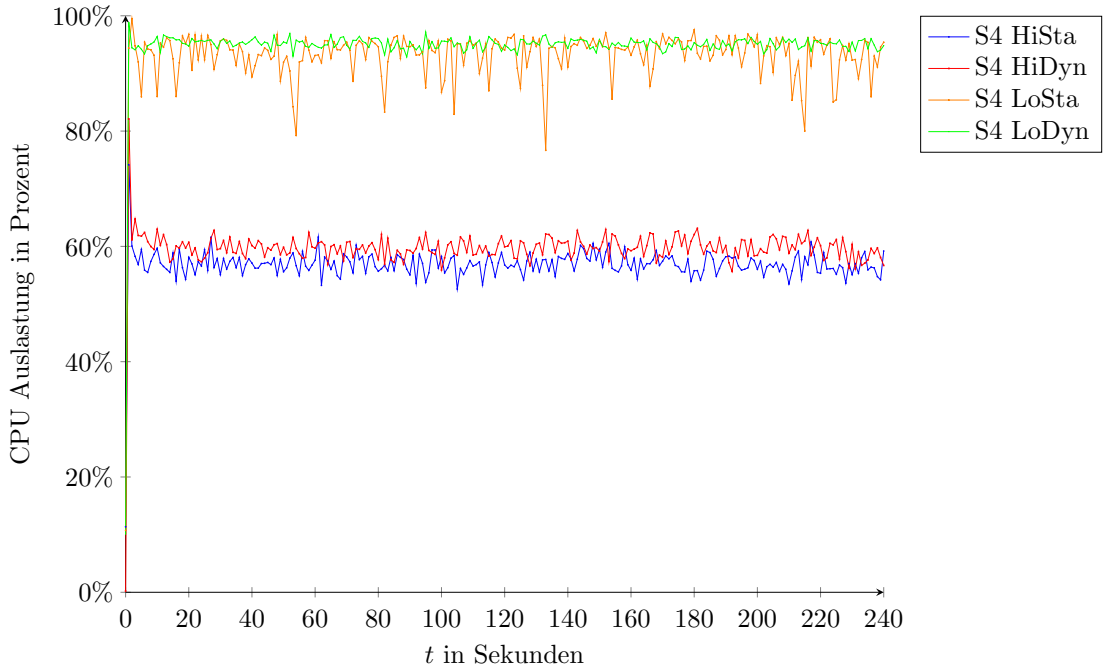

Abbildung 7.8: Messung Apache S4 CPU Auslastung

Dabei wird von der Messung mit konstanter Wortlänge im Durchschnitt 4 Prozent weniger Prozessorlast erzeugt als bei der Messung mit variabler Wortlänge und der Prozessorlast von 24 Prozent. Bei nahezu gleicher Anzahl pro Sekunde wird von der Messung mit dynamischer

Wortlänge mehr Prozessorlast benötigt. Die Low-End-Maschine verarbeitet in der Messung mit variabler Wortlänge im Durchschnitt $0{,}63\times10^5$ und mit konstanter Wortlänge im Durchschnitt $0{,}60\times10^5$ Nachrichten pro Sekunde. Die Prozessorlast liegt bei der Messung mit konstanter Wortlänge im Durchschnitt 10 Prozent niedriger und ist bei der Messung von variabler Wortlänge durchschnittlich 43 Prozent. Im Vergleich zur High-End-Maschine wird bei der Low-End-Maschine mit der Messung der variablen Wortlänge im Durchschnitt mehr Nachrichten pro Sekunde bei erhöhter Prozessorbelastung übertragen.

Die Abbildungen 7.7 und 7.8 zeigen die Messung des Apache S4 Prototypen. Mit den Messungen auf der High-End-Maschine werden bei der Messung mit konstanter Wortlänge im Durchschnitt $0{,}18\times10^5$ und bei der Messung mit variabler Wortlänge im Durchschnitt $0{,}19\times10^5$ Nachrichten pro Sekunde übertragen. Die Prozessorbelastung ist bei der Messung mit konstanter Wortlänge 57 Prozent und ist gegenüber der Messung mit variable Wortlänge im Durchschnitt 3 Prozent niedriger. Bei der Low-End-Maschine werden in der Messung mit konstanter Wortlänge im Durchschnitt $0{,}12\times10^5$ und mit variabler Wortlänge $0{,}14\times10^5$ Nachrichten übertragen. Die Prozessorbelastung liegt fast bei 100 Prozent. Die Messung mir konstanter Wortlänge beansprucht 93 Prozent und liegt 2 Prozent niedriger als die Messung mit variabler Wortlänge. Bei allen Varianten des Apache S4 Prototypen fallen zahlreiche Wechsel zwischen vielen und wenigen Nachrichten pro Sekunden in kleinen Zeitspannen gleichmäßig auf.

Nach der Beschreibung der Messergebnisse wird kurz auf die Darstellung der Übersichtsseite eingegangen. Mit der letzten Messung mit dem Prototypen Apache S4 wird die leere Übersichtsseite aus Abbildung 6.2 durch den Broker mit Log-Informationen gefüllt. In der Abbildung 7.9 werden die Prototypen mit den Werten Nachrichten pro Sekunde und die ersten Fünf häufigsten Worten mit deren Anzahl gezeigt. Dabei wird farblich zwischen Gelb und Grün getrennt. Eine geringe Anzahl an Worten bekommt ein kräftiges Grün. Eine große Anzahl an Worten wird mit einem kräftigen Gelb in einem Balkendiagramm ausgegeben.

Abbildung 7.9: Systemübersicht nach vollständiger Messreihe

Nachdem in diesem Kapitel der Messaufbau vorgestellt, mehrere Messungen durchgeführt, die Ergebnisse dargestellt und erläutert wurden, werden im nächsten Kapitel die Ergebnisse vorgestellt, diskutiert und bewertet.

7.3 Diskussion und Bewertung

Im Kapitel 7.1 und 7.2 wurden der Aufbau und die Struktur der Messung vorgestellt. Weiterhin wurden die Messungen durchgeführt, die Messergebnisse dargestellt und erläutert. In diesem Kapitel werden die gewonnenen Ergebnisse zusammengetragen, diskutiert und bewertet.

In den Tabellen 7.1 und 7.2 werden die Nachrichten pro Sekunde und die Prozessorbelastung in Prozent als Durchschnitte aus den Messergebnissen des Kapitels 7.2 aufgelistet. Aus den Durchschnitten kann ein Gesamtquotient aus den Messungen der konstanten und variablen Wortlängen und zwischen den High- und Low-End-Maschinen gezogen werden. Im Gesamtdurchschnitt wird der Prozessor der Low-End-Maschine etwa 24 Prozent mehr belastet und es werden etwa 31 Prozent weniger Nachrichten pro Sekunde als mit der High-End-Maschine übertragen.

Streaming Framework	Nachrichtendurchsatz pro S	CPU Auslastung in %
Apache Storm Hi	732003	43
Apache Storm Lo	506703	80
Apache Kafka Hi	104571	39
Apache Kafka Lo	62423	52
Apache Flume Hi	80953	20
Apache Flume Lo	60095	33
Apache S4 Hi	177036	57
Apache S4 Lo	118507	93

Tabelle 7.1: Übersicht Durchschnitte Streaming Frameworks CPU Auslastung und Nachrichtendurchsatz bei konstanter Wortlänge

Streaming Framework	Nachrichtendurchsatz pro S	CPU Auslastung in %
Apache Storm Hi	810005	53
Apache Storm Lo	546328	64
Apache Kafka Hi	100009	41
Apache Kafka Lo	61345	71
Apache Flume Hi	80311	24
Apache Flume Lo	63659	43
Apache S4 Hi	192865	60
Apache S4 Lo	139642	95

Tabelle 7.2: Übersicht Durchschnitte Streaming Frameworks CPU Auslastung und Nachrichtendurchsatz bei variabler Wortlänge

Weiterhin können die Messergebnisse aus den Tabellen 7.1 und 7.2 innerhalb der Maschinen zwischen der Messung mit konstanter Wortlänge und variabler Wortlänge verglichen werden. Zuerst wird die High-End-Maschine betrachtet. Bei den Prototypen Apache Storm und Apa-

che S4 werden in der Messung mit konstanter Wortlänge weniger Nachrichten pro Sekunde übertragen und es wird weniger Prozessorlast erzeugt als bei der Messung mit variabler Wortlänge. Im Unterschied dazu werden bei den Prototypen Apache Kafka und Apache Flume bei der Messung mit konstanter Wortlänge mehr Nachrichten pro Sekunde übertragen und weniger Prozessorlast erzeugt als bei der Messung mit variabler Wortlänge. Als nächstes wird die Low-End-Maschine betrachtet. Die Prototypen Apache Kafka und Apache S4 haben das gleiche Verhalten wie unter der High-End-Maschine. Die Prototypen Apache Storm und Apache Flume zeigen ein unterschiedliches Verhalten gegenüber der High-End-Maschine. Der Prototype Apache Storm überträgt in der Messung mit konstanter Wortlänge weniger Nachrichten pro Sekunde und belastet den Prozessor stärker als in der Messung mit variablen Wortlänge. Im Unterschied zur High-End-Maschine wurde der Prozessor weniger belastet. Beim Apache Flume Prototype werden in der Messung mit konstanter Wortlänge weniger Nachrichten pro Sekunde übertragen und weniger Prozessorlast erzeugt als in der Messung mit variabler Wortlänge. Daraus kann ein relativ konstantes Verhalten der Prototypen Apache Kafka und Apache S4 über den Vergleich der High- und Low-End-Maschine abgeleitet werden.

Bei der Betrachtung der Prozessorbelastung zwischen der High- und Low-End-Maschine wird der Prozessor bei der Low-End-Maschine von Apache Storm und Apache S4 stärker und von Apache Kafka und Apache Flume schwächer belastet. Die Belastung des Prozessors liegt im Durchschnitt bei 60 Prozent. Bei der High-End-Maschine wird ein ähnliches Verhalten bei der Prozessorbelastung festgestellt. Während die Prototypen Apache Storm und Apache S4 den Prozessor stärker belasten, ist die Belastung des Prozessors durch die Prototypen Apache Kafka und Apache Flume deutlich geringer. Die Belastung des Prozessors liegt hier im Durchschnitt bei 50 Prozent. Daraus kann für die Prototypen Apache Storm und Apache S4 eine stärkere Nutzung des Prozessors gegenüber den Prototypen Apache Kafka und Apache Flume gefolgert werden.

In der Betrachtung der Prozessorbelastung zwischen den Prototypen wird die Anzahl der Nachrichten pro Sekunde verglichen. Betrachtet werden zuerst die Prototypen, die den Prozessor stärker beanspruchen. Der Prototype Apache Storm überträgt die höchste Anzahl von Nachrichten pro Sekunde mit einem Spitzenwert von $8{,}85\times10^5$ Nachrichten pro Sekunde. Der Prototype Apache S4 überträgt in einem Spitzenwert von $2{,}22\times10^5$ Nachrichten pro Sekunde. Im Durchschnitt verarbeitet Apache Storm dennoch etwa 76 Prozent mehr Nachrichten pro Sekunde als Apache S4. Als nächstes werden die Prototypen mit schwacher Belastung des Prozessors betrachtet. Der Prototype Apache Kafka überträgt im Spitzenwert $0{,}11\times10^5$ Nachrichten pro Sekunde gegenüber Apache Flume mit dem Spitzenwert $0{,}08\times10^5$ Nachrichten pro Sekunde. Im Durchschnitt überträgt Apache Kafka 13 Prozent Nachrichten pro Sekunde mehr als Apache Flume. Insgesamt überträgt Apache Storm signifikant mehr Nachrichten pro Sekunde im Durchschnitt als die Prototypen Apache Kafka, Apache Flume und Apache S4. Bei den Prototypen Apache Kafka und Apache Flume gibt es kleine signifikante Unterschiede. In der Low-End-Maschine werden von Apache Flume weniger Nachrichten in der Messung mit konstanter Wortlänge übertragen als von der High-End-Maschine. Apache S4 unterscheidet sich zu Apache Kafka und Apache Flume in der höheren Übertragung der Nachrichten pro Sekunde und der stärkeren Belastung des Prozessors. Aus der Erkenntnis ergibt sich eine Rangfolge, in der zuerst Apache Storm mit sehr gutem Ergebnis abschneidet. Anschließend kommen Apache Kafka mit einem guten sowie Apache Flume und Apache S4 mit einem befriedigendem Ergebnis. Die Rangfolge der Streaming Frameworks in dieser Arbeit beginnt somit mit Apache Storm, setzt sich weiter fort mit Apache Kafka und Apache Flume und endet mit Apache S4.

Nach der Diskussion der Messergebnisse und der Bewertung der Apache Prototypen basierend auf den Ergebnissen aus Kapitel 7.2, werden zusätzlich die Kriterien aus Kapitel 5.1, 5.2 und

5.3 auf Einhaltung überprüft. Zuerst werden die funktionalen Anforderungen aus Kapitel 5.1 betrachtet.

Der Einsatz von Apache Maven wie in **M1** gefordert erweist sich in Apache Storm, Apache Kafka und Apache Flume als problemlos integrierbar. Apache S4 benutzt zur eigenen Entwicklung nicht Apache Maven und hat damit den Umstand eine Paketierung in Apache Maven nachzubilden. Alle Prototypen sind für die Paketierung mit Apache Maven vorbereitet. Alle Implementierungen der Prototypen erfolgten in der Entwicklungssprache Java. Die Ausführungen der Prototypen wurden unter dem Betriebssystem Linux durchgeführt. Damit ist **M2** erfüllt. Für die Ausführung der Prototypen wurde eine Umgebung mit einem Single-Node-Cluster bereitgestellt. In Kapitel 6 werden der *Start* und der *Stop* jedes Single-Node-Clusters der einzelnen Streaming Frameworks vorgestellt. Eine Installation und Bereitstellung eines Single-Node-Clusters wird im Anhang A.7, A.8, A.9 und A.10 schrittweise erklärt. **M3** wird damit erfüllt. Die Implementierung von zwei Varianten eines Prototypen, einem mit konstanter Wortlänge **M4** und einem mit variabler Wortlänge **M5** werden im Kapitel 6 beschrieben und somit erfüllt. Die Implementierungen der Prototypen Apache Storm, Apache Kafka und Apache Flume erfolgte aufgrund einer guten Dokumentation reibungslos. Dennoch wird eine Quelltext-Dokumentation des Streaming Frameworks während der Entwicklung benötigt. Da Apache S4 eine stark veraltete Dokumentation hat und der Quelltext sehr schwach dokumentiert ist, bleibt nur ein schrittweises Vorgehen der Implementierung, unter Einsatz der Streaming Framework Quelltextes aus der Apache-Versionsverwaltung[3] übrig. Die Aufnahme der Log-Informationen **M6** und Darstellung auf einer Übersichtsseite **M7** werden ebenfalls erfüllt und werden in diesem Kapitel 7.2 gezeigt und diskutiert. Die erfassten Messergebnisse stehen als Datei im Anhang A.11 bereit. Damit sind alle Muss-Kriterien erfüllt.

Das Soll-Kriterium **S1** erwartet eine Ausführung der Prototypen auf verschiedenen Rechnersystemen. In Kapitel 5.5 werden zwei verschiedene Rechnersysteme vorgestellt und die Ergebnisse in Kapitel 7.2 in einem Diagramm dargestellt und erläutert. Demnach ist **S1** erfüllt. In **S2** sollen die ersten Fünf Wörter mit der höchsten Anzahl aus einem offenen und frei zugänglichen Datensatz berechnet und automatisch angezeigt werden. Im Kapitel 6.1 wird bezogen auf den Systementwurf aus Kapitel 5.4 der Weg der gemessenen Werte über den Broker an die Übersichtsseite gezeigt und beschrieben. Außerdem wird auf **S3** eingegangen und der Transport der Nachricht über Javascript-Trigger im Kapitel 6.1 erklärt. Damit werden **S2** und **S3** erfüllt. Allerdings werden die Kann-Kriterien **K1**, die Ausführung der Prototypen auf einem Multi-Node-Cluster und **K2**, die Ausführung der Prototypen auf einem proprietären Betriebssystem, nicht erreicht. Die Abgrenzungskriterien **A1** prototypische Entwicklung und **A2** Serverseitige Absicherung wird wie in der Dokumentation der Implementierung der Prototypen in Kapitel 6 gezeigt, vollständig eingehalten.

Die Nichtfunktionalen Kriterien werden, wie in dem Kapitel 5.4 Systementwurf und in der Implementierung der Prototypen mit der Variante der variablen Wortlänge in Kapitel 6.2, 6.3, 6.4 und 6.5 gezeigt, erfüllt. Mögliche Probleme und Lösungsansätze wurden in Kapitel 5.3 vorgestellt und bei der Beschreibung des Algorithmus in Kapitel 5.6, des Vorgehens in Kapitel 5.7 und in der Implementierung in Kapitel 6 berücksichtigt. Beim Messverfahren in Kapitel 6.1 und im Benchmark in Kapitel 7.2 werden die Ergebnisse sichtbar. Damit werden die Lösungsansätze aus Kapitel 5.3 von den Prototypen unterstützt. In diesem Kapitel wurden die Ergebnisse diskutiert und bewertet. Weiterhin wurden Anforderungen an die Prototypen auf Einhaltung geprüft und die Umsetzung erläutert. Abschließend erfolgt im nächsten Kapitel eine Zusammenfassung zu diesem Kapitel gegeben.

[3] Apache S4 Mirror auf github: https://github.com/apache/incubator-s4

7.4 Zusammenfassung

Im Kapitel 7 wurde eine Evaluierung der Streaming Frameworks Prototypen Apache Storm, Apache Kafka, Apache Flume und Apache S4 durchgeführt. Zuerst wurde der Aufbau der Messumgebung vorgestellt und die Ausführung der Prototypen in kurzen Schritten aufgelistet. Dabei wurde auf das Starten und Stoppen der Prototypen eingegangen. Anschließend wurden die Prototypen ausgeführt und die Ergebnisse in Diagrammen dargestellt. Auf die Struktur der Diagramme wurde kurz eingegangen und die Messwerte in einen Bezug gebracht. Auf spezifische Eigenschaften während der Messung und Ausprägungen in den Diagrammen, wie zum Beispiel Lastspitzen wurde eingegangen und in einen Zusammenhang gebracht. Zusätzlich wird kurz die Übersichtsseite mit den empfangenen Messwerten gezeigt und erklärt. Abschließend wurden die Messwerte im Kapitel 7.3 diskutiert und bewertet. Zunächst erfolgte mittels einer tabellarischen Übersicht ein Überblick über die Messwerte. Daraus konnten verschiedene Betrachtungen zwischen den eingesetzten Maschinen und den unterschiedlichen Prototype-Varianten vorgestellt und gegenübergestellt werden. Die Diskussionen bauen aufeinander auf und führen die Betrachtungen zu einem Ergebnis. Es wird das Ergebnis in Form einer Rangfolge vorgestellt. Das Ergebnis der Rangfolge, der in dieser Arbeit verglichenen Streaming Frameworks ist: Apache Storm, Apache Kafka, Apache Flume, Apache S4. Weiterhin wurden funktionelle und nichtfunktionelle Anforderungen aus Kapitel 5 auf Einhaltung geprüft und deren Umsetzung erläutert. Das nächste Kapitel schließt diese Arbeit mit einer Schlussbetrachtung ab und gibt einen Ausblick.

Kapitel 8

Schlussbetrachtung

Im Kapitel zuvor wurde die Evaluierung der Prototypen Apache Storm, Apache Kafka, Apache Flume und Apache S4 durchgeführt. In diesem Kapitel werden alle Kapitel zusammengefasst und ein Ausblick gegeben. Zuerst wird eine differenzierte Zusammenfassung aus den vorhergehenden Kapitel gegeben. Abschließend werden im Ausblick mögliche Erweiterungen und Optimierungsmöglichkeiten aufgezeigt.

8.1 Zusammenfassung

In dieser Arbeit werden die Streaming Frameworks Apache Storm, Apache Kafka, Apache Flume und Apache S4 verglichen. Dazu wird zuerst in das Thema eingeführt und die Grundlagen geschaffen. Um Gemeinsamkeiten zwischen den Streaming Frameworks festzustellen werden Kriterien im Kapitel 3 vorgestellt und an einem Referenzmodell angewendet. Anschließend werden die einzelnen Streaming Frameworks vorgestellt und durch die vorgestellten Kriterien aus dem Kapitel zuvor angewendet. In den folgenden Kapiteln wird aufbauend auf der Vorstellung der Streaming Frameworks die Systemarchitektur vorgestellt. Mit der Deklaration der Systemarchitektur werden die Prototypen zu den einzelnen Streaming Frameworks erstellt und im Kapitel Prototypdokumentation dokumentiert. Nach der Implementierung der Prototypen wird im Kapitel Evaluierung der Messaufbau vorgestellt, eine Messung durchgeführt und die Messergebnisse vorgestellt. Weiterhin werden die Messergebnisse diskutiert und bewertet. Als nächstes werden die einzelnen Kapitel zusammengefasst.

In Kapitel 1 wird eine Einführung in das Thema anhand verschiedener Anwendungsfälle wie zum Beispiel dem Verlust einer Kreditkarte gegeben. Nach der Vorstellung der Beispiele wird der Aufbau dieser Arbeit erklärt und in das Kapitel Grundlagen eingeleitet. Im Kapitel 2 werden für den Vergleich der Streaming Frameworks Grundbegriffe eingeführt und in einen Bezug gebracht. Dabei wird der allgemeine Nachrichtenaustausch aufgezeigt, der Begriff *Streaming* abgegrenzt und der Begriff *Frameworks* erklärt. Weiterhin wird eine Berechnung in einem Streaming Framework anschaulich beschrieben. Anhand eines Graphen werden Abfragen in einem Streaming Framework illustriert und erläutert.

Das Kapitel 3 stellt den Big Data Cube und ordnet Streaming im Vergleich zu relationalen Datenbanksystemen ein. Zudem werden Kriterien vorgestellt und an dem Referenzmodell Aurora Borealis als Anwendung erläutert. Anschließend werden in Kapitel 4 die einzelnen Streaming

Frameworks vorgestellt. Dabei wird zu Beginn eine Kurzübersicht gegeben und anhand der Anwendung der Kriterien aus Kapitel 3 die ersten Gemeinsamkeiten, für die weitere Betrachtung in Kapitel 5 festgestellt. In Kapitel 4 werden alle notwendigen Methoden und Konzepte für die später zu erstellenden Prototypen erklärt und ausprobiert. Im Anhang dazu werden Beispiele für die Installation A.7, A.8, A.9, A.10 und für eine Demo-Anwendung A.2 hinterlegt. Weiterhin wird bei jedem Streaming Framework die Client-Server-Kommunikation betrachtet und erklärt.

In Kapitel 5 werden aufbauend auf den Erkenntnissen aus Kapitel 4 Anforderungen definiert und daraus Lösungsansätze für erkannte Probleme vorgestellt. Dazu wird ein Systementwurf vorgestellt, indem in drei Ebenen das Backend, die Visualization und der Display eingeordnet werden. Weiterhin wird die Systemspezifikation, der Algorithmus zur Messung vorgestellt und das Vorgehen während der Implementierung der einzelnen Prototypen vorgestellt.

Das folgende Kapitel 6 beschreibt den Aufbau und Struktur der Prototyp-Projekte. Dabei wird ebenso die Systemübersicht bestehend aus den einzelnen Streaming Framework Prototypen, Brokern, der WebServer und dem WebClient dargestellt und erläutert. Zudem wird auf die Implementierung der einzelnen Prototypen eingegangen, besondere Quelltextstellen vorgestellt und erklärt.

In Kapitel 7 wird die Messumgebung beschrieben, die Messung durchgeführt und die Messergebnisse in Diagrammen vorgestellt und erläutert. Dabei wurden die Messergebnisse diskutiert und bewertet. Die unterschiedlichen Betrachtungen in der Bewertung ergeben eine Rangfolge als Ergebnis mit Apache Storm an erster Stelle, Apache Kafka an zweiter Stelle, Apache Flume an dritter Stelle und zuletzt Apache S4. Abschließend werden die Anforderungen aus Kapitel 5 auf Einhaltung geprüft.

Der Prototyp Apache Storm erreicht die höchsten Werte während der Datenverarbeitung und setzt sich gegenüber den anderen Prototypen signifikant ab. Dadurch nimmt Apache Storm den ersten Platz in der Rangfolge ein. Apache Kafka verarbeitet mehr Daten, als Apache Flume bei etwa gleicher Prozessorauslastung. Damit ist Apache Kafka auf dem zweiten Platz. Apache S4 verarbeitet zwar mehr Daten als Apache Kafka hat aber im Vergleich die Prozessorauslastung wie bei Apache Storm. Im Verhältnis zwischen verarbeiteter Nachrichten pro Sekunde zur Prozessorauslastung fällt Apache S4 auf den vierten Platz. Damit ist Apache Flume auf dem dritten Platz.

Während der Entwicklung der Prototypen zeichnen sich die Streaming Frameworks Apache Storm, Apache Kafka und Apache Flume mit einer guten Dokumentation und einer aktiven Entwickler-Community aus. Bei Apache S4 ist die Entwickler-Community im Vergleich sehrklein und besteht aus den Hauptentwicklern. Zudem hat das Apache S4 Projekt auf der Apache Incubationseite den Status *retired*. Ein retired kann aus mehreren gründen entstehen. Ein möglicher Grund könnte lange Zeitraum des letzten Commits in die Versionsverwatlung sein. Der aktuelle S4 Quellcode wird auf der Hauptseite bereitgestellt. Das Herunterladen wird mit einer Fehlermeldung beendet. Der Quelltext kann nur aus der Versionsverwaltung heruntergeladen werden. Im Gegensatz zu den anderen Streaming Frameworks befindet sich Apache S4 seit 2011 im Apache-Incubationsprozess.

Beim Debugging fällt Apache Storm aus der Reihe. Apache Storm unterstützt als einziges Streaming Framework ein lokales Debugging in der Entwicklungsumgebung. Alle anderen Prototypen müssen im Fehlerfall über das Logging-Framework Fehler aufzeichnen. Ein externes Monitoring muss daher die Aufzeichnungen überwachen.

In allen Prototypen konnte das Erstell- und Paketierungswerkzeug Apache Maven eingesetzt werden. Obwohl Apache S4 ein anderes Erstellungswerkzeug einsatz konnte eine geeignete Konfiguration in Apache Maven bereitgestellt werden. Dadurch kann eine Änderung im Quelltext der Prototypen mit einem Kommando bereitgestellt werden. Da Fehler nur über das Logging auffallen, müssen unter Umständen häufig neue Versionen bereitgestellt werden. Durch die konfigurierte Bereitstellung werden zudem mögliche Anwenderfehler während der Bereitstellung vermieden.

Apache Storm eignet sich für die dauerhafte Verarbeitung hoher Anzahl an Informationen. Speziell liefert die Methode der Topology eine schnelle Übersicht über die eingesetzten Verarbeitungseinheiten. Wenn eine Topology in einem Cluster, muss bei einer Änderung die ganze Topology gestoppt, entfernt und neu bereitgestellt werden. Damit Kafka-Clients mit Kafka-Server kommunizieren können, müssen die Maschinen-Namen im lokalen Domain Name System aufgelöst werden können.

Apache Kafka eignet sich für das Speichern von großen Datenmengen auf rotierenden Festplatten. Dabei werden optimierte Methoden aus dem Linux-Kern-Betriebssystem eingesetzt. Apache Kafka schreibt Informationen hintereinander weg. Daher eignet sich Apache Kafka als Logging-Persistenz.

In Apache Flume kann die Konfiguration zur Laufzeit verändert werden, wodurch eine hohe Flexibilität entsteht. Dennoch müssen für spezifische Anwendungen wie dem Apache Flume Prototypen eigene Implementierungen entwickelt werden.

In Apache S4 müssen Anwendungen im Cluster bereitgestellt werden. Um eine Fehlerdiagnose durchzuführen, muss der einkompilierte Standard-Logger, durch Anpassung des Levels und erneutes Kompilieren verändert werden. Die Logger Information aus der Resource-Datei im Prototypen wird von Apache S4 nicht erkannt. Apache S4 wirkt durch das Patchen des Streaming Frameworks, um zum Beispiel Debug Log-Informationen auszugeben, noch prototypisch und nicht als ein Produktivsystem.

8.2 Ausblick

In diesem Kapitel werden mögliche Erweiterungen und Optimierungen vorgeschlagen. In der folgenden Liste werden dazu Maßnahmen vorgestellt:

LRB Damit ein Vergleich mit dem Referenzmodell Aurora Borealis möglich wird, wird eine Implementierung des LRB in Java benötigt. Dazu müssen anschließend die Prototypen entsprechend angepasst und erweitert werden.

Unit-Test und Refactoring Mit guten Unit-Tests kann die Logik der einzelnen Prototypen abgesichert werden. Um den Quelltext wartbarer zu gestalten, können bestimmte Quelltextpassagen in separate Java-Archive ausgelagert werden.

Continous-Integration Für ein einfaches Bereitstellen der Prototypen in einem Testsystem oder Livesystem, ist es notwendig die Prototypen automatisch vorher zu testen und anschließend erst bereitzustellen. Bei Fehlern in der Erstellung soll der fehlerhafte Quelltext nicht weiter bereitgestellt werden. Die Erstellung soll auf einer Übersichtsseite die gelaufene Erstellung entsprechend mit einer Fehlermeldung signalisieren und den Fehlerstapel ausgeben können.

Optimierung In Apache Storm wird ein interner Transport von $1{,}00\times10^6$ Nachrichten pro Sekunde angegeben. Der Apache Storm Prototyp hat in der angegebenen Messung einen Höchstwert von $0{,}88\times10^6$ erreicht. Eine Optimierung des Quelltexts könnte eine Steigerung der Anzahl pro Sekunde in dem Prototypen liefern.

Multi-Node-Cluster Die Messungen in dieser Arbeit sind auf ein Single-Node-Cluster bezogen. Die Ausführung auf einem Multi-Node-Cluster und die Berechnung mit dem LRB in Kombination, ist für die Bestimmung des *L*-Index aus dem LRB interessant.

Durch die Erkenntnisse wird gezeigt, dass die Streaming Frameworks Apache Storm, Apache Kafka und Apache Flume permanent weiterentwickelt werden. Mit einer Weiterentwicklung der entsprechenden Prototypen in Hinsicht auf Stabilität und Performance durch Anwendung der vorgestellten Maßnahmen, kann eine Güte-Sicherung erreicht werden. Die Streaming Frameworks Apache Storm, Apache Kafka, Apache Flume sowie weitere Streaming Frameworks, können durch die Weiterentwicklung der Prototypen zu einer Benchmark Plattform und durch Nutzen der Plattform-Dienste insgesamt einen Vorteil erhalten.

Kapitel 9

Verzeichnisse

Literaturverzeichnis

[AAB+05] Abadi, Daniel J, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina et al.: *The Design of the Borealis Stream Processing Engine.* In: *CIDR*, Band 5, Seiten 277–289, 2005.

[ACc+03] Abadi, Daniel J., Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul und Stan Zdonik: *Aurora: A New Model and Architecture for Data Stream Management*. The VLDB Journal, 12(2):120–139, August 2003.

[ACG+04] Arasu, Arvind, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker und Richard Tibbetts: *Linear Road: A Stream Data Management Benchmark*. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, Seiten 480–491. VLDB Endowment, 2004.

[Cap14] Capital, KPMG: *Going beyound the data: Achieving actionable insight with data and analytics*, Januar 2014.

[CD97] Chaudhuri, Surajit und Umeshwar Dayal: *An overview of data warehousing and OLAP technology*. SIGMOD Rec., 26(1):65–74, März 1997.

[Dig14] Digital, EMC: *Digital universe around the world*, April 2014.

[EFHB11] Edlich, Stefan, Achim Friedland, Jens Hampe und Benjamin Brauer: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser Fachbuchverlag, 2 Auflage, 2011.

[FBB+99] Fowler, Martin, Kent Beck, John Brant, William Opdyke und Don Roberts: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1., Aufl. Auflage, Juli 1999.

[Gar13] Garg, Nishant: *Apache Kafka*. Packt Publishing, 2013.

[GP84] Goldberg, Allen und Robert Paige: *Stream processing*. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, Seiten 53–62, New York, NY, USA, 1984. ACM.

[Hof13] Hoffman, Steve: *Apache Flume: distributed log collection for Hadoop*. Packt Publishing Ltd., Birmingham, Juli 2013.

[KNR11] Kreps, Jay, Neha Narkhede und Jun Rao: *Kafka: A distributed messaging system for log processing*. In: *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[Lan01] Laney, Doug: *3-D Data Management: Controlling Data Volume, Velocity and Variety*. Application Delivery Strategies, 949:4, Februar 2001.

[LYZZ08] Lin, Wei, Mao Yang, Lintao Zhang und Lidong Zhou: *PacificA: Replication in Log-Based Distributed Storage Systems*. Technischer Bericht MSR-TR-2008-25, Microsoft Research, Februar 2008.

[Mei12] Meijer, Erik: *Your mouse is a database*. Queue, 10(3):20, 2012.

[MLH+13] Markl, Volker, Alexander Löser, Thomas Hoeren, Helmut Krcmar, Holger Hemsen, Michael Schermann, Matthias Gottlieb, Christoph Buchmüller, Philip Uecker und Till Bitter: *Innovationspotenzialanalyse für die neuen Technologien für das Verwalten und Analysieren von großen Datenmengen (Big Data Management)*, 2013.

[MMO+13] McCreadie, Richard, Craig Macdonald, Iadh Ounis, Miles Osborne und Sasa Petrovic: *Scalable distributed event detection for Twitter*. In: *2013 IEEE International Conference on Big Data*, Seiten 543–549. IEEE, Oktober 2013.

[Mut10] Muthukrishnan, S.: *Massive data streams research: Where to go*. Technischer Bericht, Rutgers University, März 2010.

[NRNK10] Neumeyer, Leonardo, Bruce Robbins, Anish Nair und Anand Kesari: *S4: Distributed Stream Computing Platform*. In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, Seiten 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[SÇZ05] Stonebraker, Michael, Uğur Çetintemel und Stan Zdonik: *The 8 requirements of real-time stream processing*. ACM SIGMOD Record, 34(4):42–47, 2005.

[Sha48] Shannon, Claude E.: *A Mathematical Theory of Communication*. The Bell System Technical Journal, 27:379–423, 623–656, Juli, Oktober 1948.

[Tea06a] Team, Borealis: *Borealis application developer's guide*. Technischer Bericht, Brown University, Mai 2006.

[Tea06b] Team, Borealis: *Borealis application programmer's guide*. Technischer Bericht, Brown University, Mai 2006.

[TvS07] Tanenbaum, Andrew S. und Maarten van Steen: *Verteilte Systeme*. PEARSON STUDIUM, 2., Aufl. Auflage, 2007.

[Uni94] Union, International Telecommunication: *Information technology – Open Systems Interconnection – Basic Reference Model: The basic model*, 1994.

[Val12] Vallés, Mariano: *An Analysis of a Checkpointing Mechanism for a Distributed Stream Processing System*. Diplomarbeit, Universitat Politécnica de Catalunya, Katalonien, Juli 2012.

[WRS12] Wang, Chengwei, Infantdani Abel Rayan und Karsten Schwan: *Faster, larger, easier: reining real-time big data processing in cloud*. In: *Proceedings of the Posters and Demo Track*, Middleware '12, Seiten 4:1–4:2, New York, NY, USA, 2012. ACM. Poster to apache:flume:conf/middleware/WangRESTWH12.

Internetquellen

[Bec06] Beck, Kent: *Infected: Programmers Love Writing Tests URL: http://junit.sourceforge.net/doc/testinfected/testing.htm*, Februar 2006. Abgerufen am 28.07.2014.

[Con14] Contributer, Apache Storm: *Storm Changelog URL: https://github.com/apache/incubator-storm/blob/master/CHANGELOG.md*, Juni 2014. Abgerufen am 22.06.2014.

[Cor14a] Corporation, Oracle: *Erfahren Sie mehr über die Java-Technologie URL: https://www.java.com/de/about/*, Juni 2014. Abgerufen am 22.06.2014.

[Cor14b] Corporation, Oracle: *Lambda Expressions URL: http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html*, Juni 2014. Abgerufen am 05.06.2014.

[Dat14] Data, Twitter: *State of The Union address 2014 URL: http://twitter.github.io/interactive/sotu2014/#p1*, Mai 2014. Abgerufen am 12.05.2014.

[Fac14] Facebook: *Facebook URL: https://www.facebook.com/*, Mai 2014. Abgerufen am 12.05.2014.

[Flu12a] Flume, Apache Software Foundation: *Flume 1.5.0 User Guide URL: https://flume.apache.org/FlumeUserGuide.html*, Juni 2012. Abgerufen am 21.07.2014.

[Flu12b] Flume, Apache Software Foundation: *Flume Project Incubation Status URL: http://incubator.apache.org/projects/flume.html*, Juni 2012. Abgerufen am 21.07.2014.

[Flu14a] Flume, Apache Software Foundation: *Class AbstractRpcSink URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/sink/AbstractRpcSink.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14b] Flume, Apache Software Foundation: *Class AbstractSource URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/source/AbstractSource.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14c] Flume, Apache Software Foundation: *Class AvroSink URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/sink/AvroSink.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14d] Flume, Apache Software Foundation: *Class AvroSource URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/source/AvroSource.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14e] Flume, Apache Software Foundation: *Class BasicChannelSemantics URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/channel/BasicChannelSemantics.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14f] Flume, Apache Software Foundation: *Class ExecSource URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/source/ExecSource.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14g] Flume, Apache Software Foundation: *Class HostInterceptor URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/interceptor/HostInterceptor.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14h] Flume, Apache Software Foundation: *Class HTTPSource URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/source/http/HTTPSource.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14i] Flume, Apache Software Foundation: *Class NetcatSource URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/source/NetcatSource.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14j] Flume, Apache Software Foundation: *Class RegexExtractorInterceptor URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/interceptor/RegexExtractorInterceptor.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14k] Flume, Apache Software Foundation: *Class RegexFilteringInterceptor URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/interceptor/RegexFilteringInterceptor.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14l] Flume, Apache Software Foundation: *Class StaticInterceptor URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/interceptor/StaticInterceptor.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14m] Flume, Apache Software Foundation: *Class ThriftSink URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/sink/ThriftSink.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14n] Flume, Apache Software Foundation: *Class ThriftSource URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/source/ThriftSource.html*, Juli 2014. Abgerufen am 28.07.2014.

[Flu14o] Flume, Apache Software Foundation: *Class TimestampInterceptor URL: https://flume.apache.org/releases/content/1.5.0/apidocs/org/apache/flume/interceptor/TimestampInterceptor.html*, Juli 2014. Abgerufen am 28.07.2014.

[Fou12] Foundation, Apache Software: *Kafka Project Incubation Status URL: http://incubator.apache.org/projects/kafka.html*, Oktober 2012. Abgerufen am 30.06.2014.

[Fou13] Foundation, Apache Software: *Storm Project Incubation Status URL: http://incubator.apache.org/projects/storm.html*, September 2013. Abgerufen am 16.06.2014.

[Fou14a] Foundation, Apache Software: *Apache Avro a data serialization system URL: http://avro.apache.org/*, Juli 2014. Abgerufen am 28.07.2014.

[Fou14b] Foundation, Apache Software: *Apache Hadoop URL: http://hadoop.apache.org/*, Juni 2014. Abgerufen am 21.07.2014.

[Fou14c] Foundation, Apache Software: *Apache Software Foundation URL: http://www.apache.org/foundation/*, Juni 2014. Abgerufen am 22.06.2014.

[Fou14d] Foundation, Apache Software: *Apache Thrift software framework, for scalable cross-language services development URL: http://thrift.apache.org/*, Juli 2014. Abgerufen am 28.07.2014.

[Fou14e] Foundation, Apache Software: *Apache ZooKeeper URL: http://zookeeper.apache.org/*, Juni 2014. Abgerufen am 22.06.2014.

[Fou14f] Foundation, Apache Software: *Class SpoutOutputCollector URL: https://storm.incubator.apache.org/apidocs/backtype/storm/spout/SpoutOutputCollector.html*, Juni 2014. Abgerufen am 22.06.2014.

[Fou14g] Foundation, Apache Software: *Class TopologyBuilder URL: https://storm.incubator.apache.org/apidocs/backtype/storm/topology/TopologyBuilder.html*, Juni 2014. Abgerufen am 22.06.2014.

[Fou14h] Foundation, Apache Software: *Interface IBolt URL: https://storm.incubator.apache.org/apidocs/backtype/storm/task/IBolt.html*, Juni 2014. Abgerufen am 22.06.2014.

[Fou14i] Foundation, Apache Software: *Interface InputDeclarer<T extends InputDeclarer> URL: https://storm.incubator.apache.org/apidocs/backtype/storm/topology/InputDeclarer.html*, Juni 2014. Abgerufen am 23.06.2014.

[Fou14j] Foundation, Apache Software: *Interface ISpout URL: https://storm.incubator.apache.org/apidocs/backtype/storm/spout/ISpout.html*, Juni 2014. Abgerufen am 22.06.2014.

[Fou14k] Foundation, Apache Software: *Kafka 0.8.1 Documentation URL: https://kafka.apache.org/documentation.html*, Juli 2014. Abgerufen am 07.07.2014.

[Fou14l] Foundation, Apache Software: *Storm Fault Tolerance URL: https://storm.incubator.apache.org/documentation/Fault-tolerance.html*, Juni 2014. Abgerufen am 29.06.2014.

[Fou14m] Foundation, Apache Software: *Storm Fully Processed URL: https://storm.incubator.apache.org/documentation/Guaranteeing-message-processing.html*, Juni 2014. Abgerufen am 29.06.2014.

[Fou14n] Foundation, Apache Software: *Storm Project Incubation Status Page URL: http://incubator.apache.org/projects/storm.html*, Juni 2014. Abgerufen am 22.06.2014.

[Fou14o] Foundation, Apache Software: *Storm Security URL: https://github.com/apache/incubator-storm/blob/master/SECURITY.md*, Juni 2014. Abgerufen am 29.06.2014.

[Fou14p] Foundation, Apache Software: *Storm Trident URL: https://storm.incubator.apache.org/documentation/Trident-API-Overview.html*, Juni 2014. Abgerufen am 29.06.2014.

[Fou14q] Foundation, Python Software: *Python About URL: https://www.python.org/about/*, Juni 2014. Abgerufen am 22.06.2014.

[Gal14] Galstad, Ethan: *Nagios - The Industry Standard in IT Infrastructure Monitoring URL: http://www.nagios.org/*, Juni 2014. Abgerufen am 29.06.2014.

[GJM+14] Gopalakrishna, Kishore, Flavio Junqueira, Matthieu Morel, Leo Neumeyer, Bruce Robbins und Daniel Gomez Ferro: *S4 GitHub Repository Apache Git Mirror URL: https://github.com/apache/incubator-s4*, August 2014. Abgerufen am 04.08.2014.

[Goo14] Google: *Google Search URL: https://www.google.de/*, Mai 2014. Abgerufen am 12.05.2014.

[HBa14] HBase, Apache Software Foundation: *Apache HBase is the Hadoop database, a distributed, scalable, big data store. URL: http://hbase.apache.org/*, August 2014. Abgerufen am 18.08.2014.

[Hic14] Hickey, Rich: *Clojure Rationale URL: http://clojure.org/rationale*, Juni 2014. Abgerufen am 22.06.2014.

[iC14] Corporation iMatix: *Zero MQ URL: http://zguide.zeromq.org/page:all*, Juni 2014. Abgerufen am 22.06.2014.

[Inc14a] Inc., GitHub: *GitHub Powerful collaboration, code review, and code management for open source and private projects. URL: https://github.com/features*, Juni 2014. Abgerufen am 22.06.2014.

[Inc14b] Inc., GitHub: *Storm Project Contributors URL: https://github.com/apache/incubator-storm/graphs/contributors*, Juni 2014. Abgerufen am 22.06.2014.

[Jam14a] James, Josh: *Data Never Sleeps 1.0 URL: http://www.domo.com/blog/wp-content/uploads/2012/06/DatainOneMinute.jpg*, Mai 2014. Abgerufen am 12.05.2014.

[Jam14b] James, Josh: *Data Never Sleeps 2.0 URL: http://www.domo.com/blog/wp-content/uploads/2014/04/DataNeverSleeps_2.0_v2.jpg*, Mai 2014. Abgerufen am 05.05.2014.

[Jun11] Junqueira, Flavio: *S4 Proposal URL: http://wiki.apache.org/incubator/S4Proposal*, September 2011. Abgerufen am 04.08.2014.

[KNR14a] Kreps, Jay, Neha Narkhede und Jun Rao: *Apache Kafka is publish-subscribe messaging rethought as a distributed commit log. URL: http://kafka.apache.org/*, Juni 2014. Abgerufen am 30.06.2014.

[KNR14b] Kreps, Jay, Neha Narkhede und Jun Rao: *Kafka GitHub Repository Apache Git Mirror URL: https://github.com/apache/kafka*, Juni 2014. Abgerufen am 30.06.2014.

[Kre14] Kreps, Jay: *Security URL: https://cwiki.apache.org/confluence/x/rhIHAg*, Juli 2014. Abgerufen am 14.07.2014.

[Lee14] Lee, Trustin: *The Netty project URL: http://netty.io/index.html*, Juni 2014. Abgerufen am 22.06.2014.

[Lin14] LinkedIn: *Das weltweit größte berufliche Netzwerk URL: https://www.linkedin.com/*, Juni 2014. Abgerufen am 30.06.2014.

[Mar13] Marz, Nathan: *Storm Proposal URL: http://wiki.apache.org/incubator/StormProposal*, September 2013. Abgerufen am 16.06.2014.

[Mar14a] Marz, Nathan: *Obsolete Storm GitHub Repository URL: https://github.com/nathanmarz/storm*, Juni 2014. Abgerufen am 22.06.2014.

[Mar14b] Marz, Nathan: *Storm GitHub Repository Apache Git Mirror URL: https://github.com/apache/incubator-storm*, Juni 2014. Abgerufen am 22.06.2014.

[Mar14c] Marz, Nathan: *Storm is a distributed realtime computation system. URL: http://storm.incubator.apache.org/*, June 2014. Abgerufen am 22.06.2014.

[McD14] McDonough, Chris: *Supervisor: A Process Control System URL: http://supervisord.org/index.html*, Juni 2014. Abgerufen am 29.06.2014.

[PMS14a] Prabhakar, Arvind, Prasad Mujumdar und Eric Sammer: *Apache Flume distributed, reliable, and available service for efficientliy operating large amounts of log data. URL: http://flume.apache.org/*, Juli 2014. Abgerufen am 14.07.2014.

[PMS14b] Prabhakar, Arvind, Prasad Mujumdar und Eric Sammer: *Flume GitHub Repository Apache Git Mirror URL: https://github.com/apache/flume*, Juli 2014. Abgerufen am 14.07.2014.

[PN08] Palaniappan, Sathish K. und Pramod B. Nagaraja: *Efficient data transfer through zero copy URL: https://www.ibm.com/developerworks/linux/library/j-zerocopy/*, September 2008. Abgerufen am 21.07.2014.

[Rao11] Rao, Jun: *Kafka Proposal URL: https://wiki.apache.org/incubator/KafkaProposal*, Juni 2011. Abgerufen am 30.06.2014.

[S413a] S4, Apache Software Foundation: *Class TCPEmitter URL: http://people.apache.org/~mmorel/apache-s4-0.6.0-incubating-doc/javadoc/org/apache/s4/comm/tcp/TCPEmitter.html*, Juni 2013. Abgerufen am 11.08.2014.

[S413b] S4, Apache Software Foundation: *Class UDPEmitter URL: http://people.apache.org/~mmorel/apache-s4-0.6.0-incubating-doc/javadoc/org/apache/s4/comm/udp/UDPEmitter.html*, Juni 2013. Abgerufen am 11.08.2014.

[S413c] S4, Apache Software Foundation: *S4 distributed stream computing platform. URL: http://incubator.apache.org/s4/*, Juni 2013. Abgerufen am 04.08.2014.

[S413d] S4, Apache Software Foundation: *S4 Fault Tolerance URL: http://incubator.apache.org/s4/doc/0.6.0/fault_tolerance/*, Juni 2013. Abgerufen am 11.08.2014.

[S413e] S4, Apache Software Foundation: *S4 Metrics URL: http://incubator.apache.org/s4/doc/0.6.0/metrics/*, Juni 2013. Abgerufen am 18.08.2014.

[S413f] S4, Apache Software Foundation: *S4 Overview URL: http://incubator.apache.org/s4/doc/0.6.0/overview/*, Juni 2013. Abgerufen am 11.08.2014.

[S413g] S4, Apache Software Foundation: *S4 Walkthrough URL: http://incubator.apache.org/s4/doc/0.6.0/walkthrough/*, Juni 2013. Abgerufen am 18.08.2014.

[Sam11] Sammer, Eric: *Flume NG refactoring URL: https://issues.apache.org/jira/browse/FLUME-728*, August 2011. Abgerufen am 21.07.2014.

[Sam12] Sammer, Eric: *Flume NG URL: https://cwiki.apache.org/confluence/x/_46oAQ*, Juni 2012. Abgerufen am 21.07.2014.

[Sha94] Shakespeare, William: *The Complete Works of William Shakespeare URL: http://www.gutenberg.org/files/100/old/shaks12.txt*, Band 100 der Reihe *ser-PROJECT-GUTENBERG*. pub-PROJECT-GUTENBERG, pub-PROJECT-GUTENBERG:adr, 1994. Abgerufen am 08.09.2014.

[Ver11] Verbeck, Nicholas: *Flume Proposal URL: https://wiki.apache.org/incubator/FlumeProposal*, Juni 2011. Abgerufen am 21.07.2014.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Anhang A

Zusätze

A.1 Abkürzungen

ACL	Access Control List. 24
ANTLR	ANother Tool for Language Recognition. 116
API	Application Programming Interface. 28, 45
ASF	Apache Software Foundation. 19
BASE	Basically Available, Soft State, Eventually Consistant. 12
C++	C objektorientierte Programmiersprache. 16, 42
CAP	Consistancy Availability Partition Tolerance. 12
CP	Connection Point. 9
CPU	Central Processing Unit. 17, 43, 44
eSATA	External Serial Advanced Technology Attachment. 44, 45
ESP	Event Stream Processing. 6
fk	foreign key. 11
HDFS	Hadoop Filesystem. 28
HTML	Hypertext Markup Language. 48
HTTP	Hypertext Transfer Protocol. 33
IDE	Integrated Development Environment. 47
IP	Internet Protocol. 24, 43

IPsec	Internet Protocol Security. 24
ISO	International Organization for Standardization. 115
ISR	In-sync Replica. 26
JMX	Java Management Extensions. 27, 32, 33, 37
JSON	JavaScript Object Notation. 32, 41
k	key. 11
LRB	Linear Road Benchmark. 42, 49, 75, 76
MPEG	Motion Pictures Expert Group. 6
NG	Next Generation. 27
NMSTL	Networking, Messaging, Servers, and Threading Library. 16, 116
OpenJDK	Open Java Developer Kit. 44, 45
ORM	Object-relational mapping. 11
OSI	Open Systems Interconnection Model. 5, 6, 14
pk	primary key. 11
QoS	Quality of Service. 8, 17, 24, 36, 37
RAM	Random Access Memory. 43, 44
RPC	Remote Procedure Call. 6, 7, 14, 15, 32
SBT	Scala Build Tool. 119
SDMS	Stream Data Management System. 42
SLA	Service Level Agreement. 26, 27
SPE	Stream Processing Engine. 7, 9, 89
SSL	Secure Sockets Layer. 29
STRG	Steuerung. 62
TCP	Transmission Control Protocol. 5, 6, 36, 43

UDP User Datagram Protocol. 36

v value. 11

VM Vector of Metrics. 8, 10

WAL Write Ahead Log. 30, 32

XML Extensible Markup Language. 16, 17, 41

A.2 Quelltext

Listing A.1: Aurora Borealis Konfiguration für die Verteilung

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "../../src/src/borealis.dtd">

<deploy>
    <publish    stream="Packet"/>
    <subscribe  stream="Aggregate" endpoint="127.0.0.1:25000"/>

    <node   endpoint="127.0.0.1:15000"  query="mycount" />
    <node   endpoint="127.0.0.1:17000"  query="myfilter" />
</deploy>
```

Listing A.2: Aurora Borealis Konfiguration für die Abfragen

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "../../src/src/borealis.dtd">

<!-- Borealis query diagram for:  mytestdist.cc -->

<borealis>
    <input   stream="Packet"     schema="PacketTuple"     />
    <output  stream="Aggregate"  schema="AggregateTuple"  />

    <schema name="PacketTuple">
        <field  name="time"         type="int" />
        <field  name="protocol"     type="string"  size="4" />
    </schema>

    <schema name="AggregateTuple">
        <field  name="time"         type="int" />
        <field  name="count"        type="int" />
    </schema>

    <box name="mycount"   type="aggregate" >
        <in   stream="Packet"    />
        <out  stream="AggregatePreFilter" />

        <parameter   name="aggregate-function.0"  value="count()"  />
        <parameter   name="aggregate-function-output-name.0"
                    value="count"  />

```

```
28          <parameter   name="window-size-by"          value="VALUES"   />
29          <parameter   name="window-size"             value="1"        />
30          <parameter   name="advance"                 value="1"        />
31          <parameter   name="order-by"                value="FIELD"    />
32          <parameter   name="order-on-field"          value="time"     />
33      </box>
34
35      <box name="myfilter" type="filter">
36          <in stream="AggregatePreFilter" />
37          <out stream="Aggregate" />
38          <parameter name="expression.0" value="(time % 2) == 0"/>
39      </box>
40
41  </borealis>
```

Listing A.3: Aurora Borealis Testanwendung

```
1  #include "args.h"
2  #include "MytestdistMarshal.h"
3
4  using namespace Borealis;
5
6  const uint32  SLEEP_TIME = 100;          // Delay between injections.
7  const uint32  BATCH_SIZE = 20;           // Number of input tuples per
       batch.
8  const uint32  PROTOCOL_SIZE = 4;         // Number of elements in
       PROTOCOL.
9
10 const string  PROTOCOL[] = { string( "dns" ),  string( "smtp" ),
11                               string( "http" ), string( "ssh" )
12                             };
13
14
15 const Time  time0 = Time::now() - Time::msecs( 100 );
16
17
18 ////////////////////////////////////////////////////////////////////////
19 //
20 // Print the content of received tuples.
21 //
22 void  MytestdistMarshal::receivedAggregate( AggregateTuple  *tuple )
23 {
24 //.....................................................................
25
26
27      NOTICE << "For time interval starting at "
28             << tuple->time << " tuple count was " << tuple->count;
29
30      return;
31 }
32
33
34
35 ////////////////////////////////////////////////////////////////////////
36 //
37 // Return here after sending a packet and a delay.
38 //
39 void  MytestdistMarshal::sentPacket()
40 {
41      int32        random_index;
```

```
42      int32         timestamp;
43      Time          current_time;
44      //.....................................................................
45
46
47      current_time = Time::now();
48
49      timestamp = (int32)( current_time - time0 ).to_secs();
50      if ( timestamp < 0 ) timestamp = 0;
51      //DEBUG  << "timestamp = " << timestamp << "  current_time = " <<
            current_time;
52
53      for ( uint32  i = 0; i < BATCH_SIZE; i++ )
54      {
55          random_index = rand() % PROTOCOL_SIZE;
56
57          // This has to be in the loop scope so the constructor is rerun.
58          Packet  tuple;
59
60          tuple._data.time = timestamp;
61          setStringField( PROTOCOL[ random_index ], tuple._data.protocol,
                4 );
62
63          // DEBUG  << "time=" << tuple._data.time << "  proto=" << tuple.
                _data.protocol;
64          batchPacket( &tuple );
65      }
66
67      // Send a batch of tuples and delay.
68      //
69      //DEBUG << "call sendPacket...";
70      sendPacket( SLEEP_TIME );
71
72      return;
73  }
74
75
76
77
78  ////////////////////////////////////////////////////////////////////////
79  //
80  int  main( int  argc, const char  *argv[] )
81  {
82      MytestdistMarshal   marshal;        // Client and I/O stream state.
83  //
84  // Maximum size of buffer with data awaiting transmission to Borealis
85  //.....................................................................
86
87
88      // Run the front-end, open a client, subscribe to outputs and inputs
            .
89      // In this edited version, open will print a message and quit if an
            error occurs.
90      marshal.open();
91
92      DEBUG  << "time0 = " << time0;
93
94      // Send the first batch of tuples.  Queue up the next round with a
            delay.
95      marshal.sentPacket();
96
```

```
97      DEBUG  << "run the client event loop...";
98      // Run the client event loop.  Return only on an exception.
99      marshal.runClient();
100
101 }
```

Listing A.4: Apache Storm WordCount Demo

```
1  /**
2  * Licensed to the Apache Software Foundation (ASF) under one
3  * or more contributor license agreements. See the NOTICE file
4  * distributed with this work for additional information
5  * regarding copyright ownership. The ASF licenses this file
6  * to you under the Apache License, Version 2.0 (the
7  * "License"); you may not use this file except in compliance
8  * with the License. You may obtain a copy of the License at
9  *
10 * http://www.apache.org/licenses/LICENSE-2.0
11 *
12 * Unless required by applicable law or agreed to in writing, software
13 * distributed under the License is distributed on an "AS IS" BASIS,
14 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
15 * See the License for the specific language governing permissions and
16 * limitations under the License.
17 */
18 package storm.starter;
19
20 import backtype.storm.Config;
21 import backtype.storm.LocalCluster;
22 import backtype.storm.StormSubmitter;
23 import backtype.storm.task.ShellBolt;
24 import backtype.storm.topology.BasicOutputCollector;
25 import backtype.storm.topology.IRichBolt;
26 import backtype.storm.topology.OutputFieldsDeclarer;
27 import backtype.storm.topology.TopologyBuilder;
28 import backtype.storm.topology.base.BaseBasicBolt;
29 import backtype.storm.tuple.Fields;
30 import backtype.storm.tuple.Tuple;
31 import backtype.storm.tuple.Values;
32 import storm.starter.spout.RandomSentenceSpout;
33
34 import java.util.HashMap;
35 import java.util.Map;
36
37 /**
38 * This topology demonstrates Storm's stream groupings and multilang
       capabilities.
39 */
40 public class WordCountTopology {
41   public static class SplitSentence extends ShellBolt implements
         IRichBolt {
42
43     public SplitSentence() {
44       super("python", "splitsentence.py");
45     }
46
47     @Override
48     public void declareOutputFields(OutputFieldsDeclarer declarer) {
49       declarer.declare(new Fields("word"));
```

```
50      }
51
52      @Override
53      public Map<String, Object> getComponentConfiguration() {
54        return null;
55      }
56    }
57
58    public static class WordCount extends BaseBasicBolt {
59      Map<String, Integer> counts = new HashMap<String, Integer>();
60
61      @Override
62      public void execute(Tuple tuple, BasicOutputCollector collector) {
63        String word = tuple.getString(0);
64        Integer count = counts.get(word);
65        if (count == null)
66          count = 0;
67        count++;
68        counts.put(word, count);
69        collector.emit(new Values(word, count));
70      }
71
72      @Override
73      public void declareOutputFields(OutputFieldsDeclarer declarer) {
74        declarer.declare(new Fields("word", "count"));
75      }
76    }
77
78    public static void main(String[] args) throws Exception {
79
80      TopologyBuilder builder = new TopologyBuilder();
81
82      builder.setSpout("spout", new RandomSentenceSpout(), 5);
83
84      builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("
            spout");
85      builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split"
            , new Fields("word"));
86
87      Config conf = new Config();
88      conf.setDebug(true);
89
90
91      if (args != null && args.length > 0) {
92        conf.setNumWorkers(3);
93
94        StormSubmitter.submitTopologyWithProgressBar(args[0], conf,
              builder.createTopology());
95      }
96      else {
97        conf.setMaxTaskParallelism(3);
98
99        LocalCluster cluster = new LocalCluster();
100       cluster.submitTopology("word-count", conf, builder.createTopology
              ());
101
102       Thread.sleep(10000);
103
104       cluster.shutdown();
105     }
106   }
```

```
107 }
```

Listing A.5: Apache Kafka Producer Beispiel

```
1  /**
2  * Licensed to the Apache Software Foundation (ASF) under one or more
3  * contributor license agreements. See the NOTICE file distributed with
4  * this work for additional information regarding copyright ownership.
5  * The ASF licenses this file to You under the Apache License, Version
       2.0
6  * (the "License"); you may not use this file except in compliance with
7  * the License. You may obtain a copy of the License at
8  *
9  * http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17 package kafka.examples;
18
19
20 import java.util.Properties;
21 import kafka.producer.KeyedMessage;
22 import kafka.producer.ProducerConfig;
23
24 public class Producer extends Thread
25 {
26   private final kafka.javaapi.producer.Producer<Integer, String>
         producer;
27   private final String topic;
28   private final Properties props = new Properties();
29
30   public Producer(String topic)
31   {
32     props.put("serializer.class", "kafka.serializer.StringEncoder");
33     props.put("metadata.broker.list", "localhost:9092");
34     // Use random partitioner. Don't need the key type. Just set it to
         Integer.
35     // The message is of type String.
36     producer = new kafka.javaapi.producer.Producer<Integer, String>(new
         ProducerConfig(props));
37     this.topic = topic;
38   }
39
40   public void run() {
41     int messageNo = 1;
42     while(true)
43     {
44       String messageStr = new String("Message_" + messageNo);
45       producer.send(new KeyedMessage<Integer, String>(topic, messageStr)
           );
46       messageNo++;
47     }
48   }
49
50 }
```

Listing A.6: Apache Kafka Consumer Beispiel

```
1  /**
2  * Licensed to the Apache Software Foundation (ASF) under one or more
3  * contributor license agreements. See the NOTICE file distributed with
4  * this work for additional information regarding copyright ownership.
5  * The ASF licenses this file to You under the Apache License, Version
       2.0
6  * (the "License"); you may not use this file except in compliance with
7  * the License. You may obtain a copy of the License at
8  *
9  * http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17 package kafka.examples;
18
19
20 import java.util.HashMap;
21 import java.util.List;
22 import java.util.Map;
23 import java.util.Properties;
24 import kafka.consumer.ConsumerConfig;
25 import kafka.consumer.ConsumerIterator;
26 import kafka.consumer.KafkaStream;
27 import kafka.javaapi.consumer.ConsumerConnector;
28
29
30 public class Consumer extends Thread
31 {
32   private final ConsumerConnector consumer;
33   private final String topic;
34
35   public Consumer(String topic)
36   {
37     consumer = kafka.consumer.Consumer.createJavaConsumerConnector(
38             createConsumerConfig());
39     this.topic = topic;
40   }
41
42   private static ConsumerConfig createConsumerConfig()
43   {
44     Properties props = new Properties();
45     props.put("zookeeper.connect", KafkaProperties.zkConnect);
46     props.put("group.id", KafkaProperties.groupId);
47     props.put("zookeeper.session.timeout.ms", "400");
48     props.put("zookeeper.sync.time.ms", "200");
49     props.put("auto.commit.interval.ms", "1000");
50
51     return new ConsumerConfig(props);
52
53   }
54
55   public void run() {
56     Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
57     topicCountMap.put(topic, new Integer(1));
```

```
58      Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
            consumer.createMessageStreams(topicCountMap);
59      KafkaStream<byte[], byte[]> stream = consumerMap.get(topic).get(0);
60      ConsumerIterator<byte[], byte[]> it = stream.iterator();
61      while(it.hasNext())
62        System.out.println(new String(it.next().message()));
63    }
64 }
```

Listing A.7: Apache S4 Processing Element Beispiel

```
1  /**
2   * Licensed to the Apache Software Foundation (ASF) under one
3   * or more contributor license agreements.  See the NOTICE file
4   * distributed with this work for additional information
5   * regarding copyright ownership.  The ASF licenses this file
6   * to you under the Apache License, Version 2.0 (the
7   * "License"); you may not use this file except in compliance
8   * with the License.  You may obtain a copy of the License at
9   *
10  *     http://www.apache.org/licenses/LICENSE-2.0
11  *
12  * Unless required by applicable law or agreed to in writing, software
13  * distributed under the License is distributed on an "AS IS" BASIS,
14  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
        implied.
15  * See the License for the specific language governing permissions and
16  * limitations under the License.
17  */
18
19 package hello;
20
21 import org.apache.s4.base.Event;
22 import org.apache.s4.core.ProcessingElement;
23
24 public class HelloPE extends ProcessingElement {
25
26     // you should define downstream streams here and inject them in the
            app definition
27
28     boolean seen = false;
29
30     /**
31      * This method is called upon a new Event on an incoming stream
32      */
33     public void onEvent(Event event) {
34         // in this example, we use the default generic Event type, by
                you can also define your own type
35         System.out.println("Hello " + (seen ? "again " : "") + event.get
                ("name") + "!");
36         seen = true;
37     }
38
39     @Override
40     protected void onCreate() {
41     }
42
43     @Override
44     protected void onRemove() {
45     }
```

```
46
47 }
```

Listing A.8: Apache S4 Processing Element Instanz Beispiel

```
1  /**
2   * Licensed to the Apache Software Foundation (ASF) under one
3   * or more contributor license agreements.  See the NOTICE file
4   * distributed with this work for additional information
5   * regarding copyright ownership.  The ASF licenses this file
6   * to you under the Apache License, Version 2.0 (the
7   * "License"); you may not use this file except in compliance
8   * with the License.  You may obtain a copy of the License at
9   *
10  *     http://www.apache.org/licenses/LICENSE-2.0
11  *
12  * Unless required by applicable law or agreed to in writing, software
13  * distributed under the License is distributed on an "AS IS" BASIS,
14  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
        implied.
15  * See the License for the specific language governing permissions and
16  * limitations under the License.
17  */
18
19 package hello;
20
21 import java.util.Arrays;
22 import java.util.List;
23
24 import org.apache.s4.base.Event;
25 import org.apache.s4.base.KeyFinder;
26 import org.apache.s4.core.App;
27
28 public class HelloApp extends App {
29
30     @Override
31     protected void onStart() {
32     }
33
34     @Override
35     protected void onInit() {
36         // create a prototype
37         HelloPE helloPE = createPE(HelloPE.class);
38         // Create a stream that listens to the "names" stream
39         // and passes events to the helloPE instance.
40         createInputStream("names", new KeyFinder<Event>() {
41
42             @Override
43             public List<String> get(Event event) {
44                 return Arrays.asList(new String[] { event.get("name") })
                        ;
45             }
46         }, helloPE);
47     }
48
49     @Override
50     protected void onClose() {
51     }
52
53 }
```

Listing A.9: Apache S4 HelloInputAdapter Beispiel

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package hello;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

import org.apache.s4.base.Event;
import org.apache.s4.core.adapter.AdapterApp;

public class HelloInputAdapter extends AdapterApp {

    @Override
    protected void onStart() {
        new Thread(new Runnable() {
            @Override
            public void run() {

                ServerSocket serverSocket = null;
                Socket connectedSocket;
                BufferedReader in = null;
                try {
                    serverSocket = new ServerSocket(15000);
                    while (true) {
                        connectedSocket = serverSocket.accept();
                        in = new BufferedReader(new InputStreamReader(
                            connectedSocket.getInputStream()));

                        String line = in.readLine();
                        System.out.println("read: " + line);
                        Event event = new Event();
                        event.put("name", String.class, line);
                        getRemoteStream().put(event);
                        connectedSocket.close();
                    }

                } catch (IOException e) {
                    e.printStackTrace();
                    // System.exit(-1);
```

```
58                  } finally {
59                      if (in != null) {
60                          try {
61                              in.close();
62                          } catch (IOException e) {
63                              throw new RuntimeException(e);
64                          }
65                      }
66                      if (serverSocket != null) {
67                          try {
68                              serverSocket.close();
69                          } catch (IOException e) {
70                              throw new RuntimeException(e);
71                          }
72                      }
73                  }
74              }
75          }).start();
76
77      }
78  }
```

A.3 Zusatz Systemarchitektur

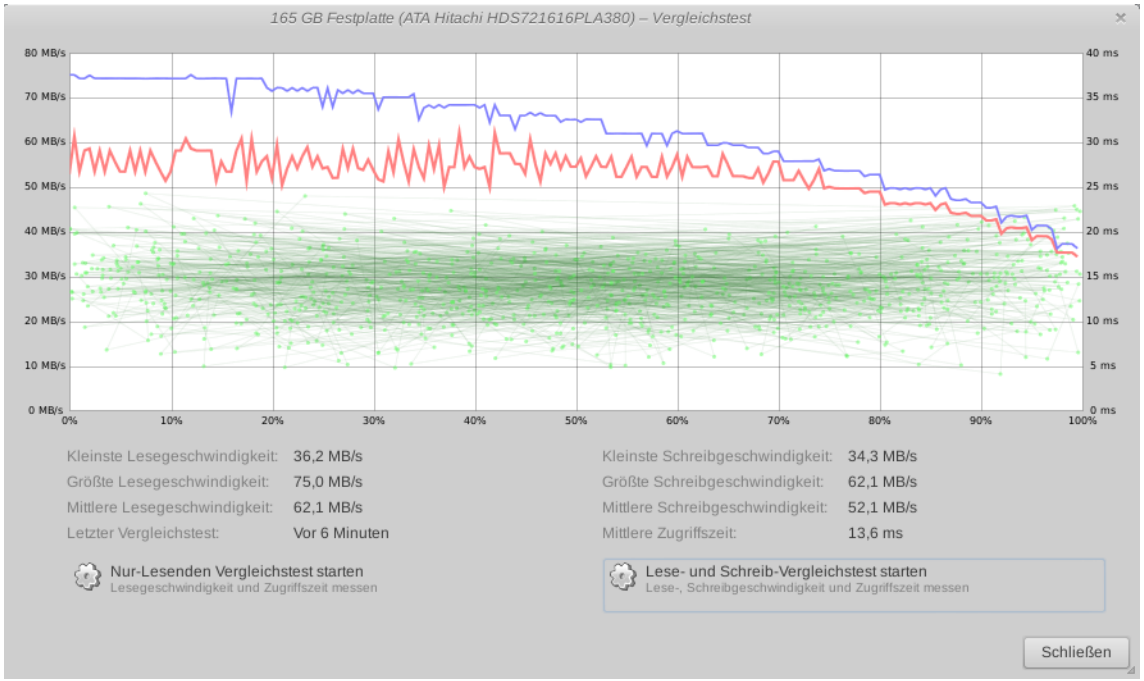

Abbildung A.1: Leistungstest Festplatte Hitachi

A.4 Zusatz Evaluierung

A.5 Zusatz Prototyp Dokumentation

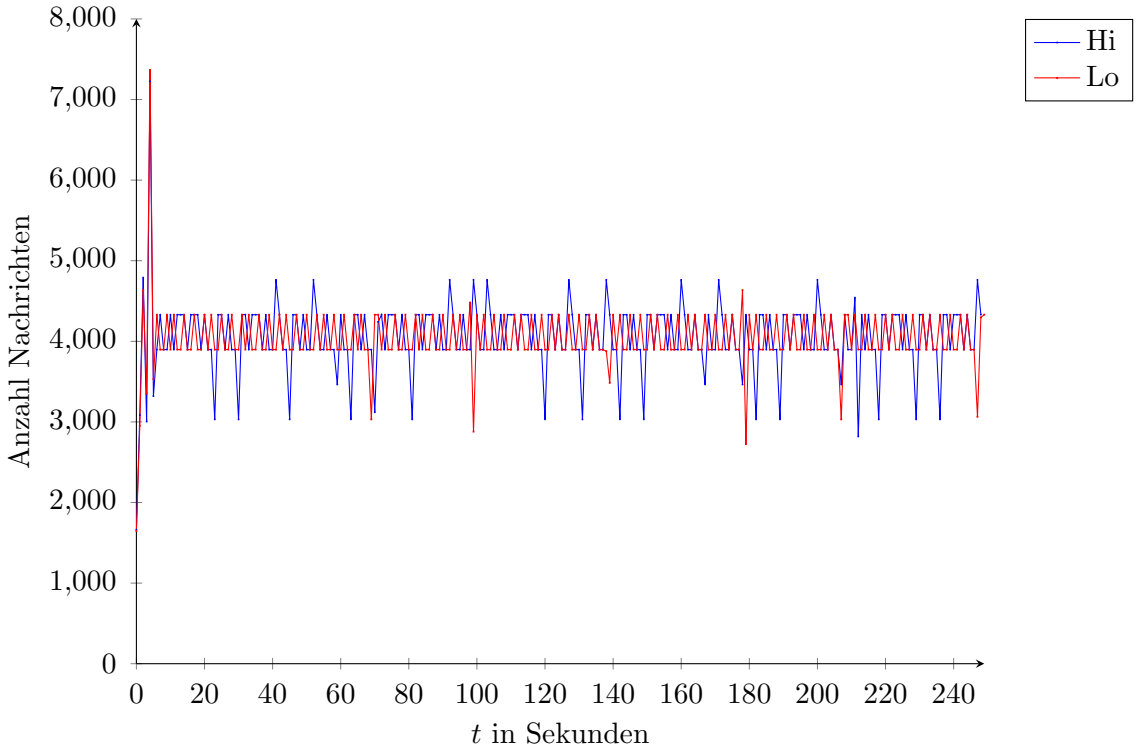

Abbildung A.2: Messung Nachrichtendurchsatz in der virtuellen Maschine

Listing A.10: Broker für Apache Storm

```
"use strict";

var wsSocketPort = 8080;
process.title = 'protod' + wsSocketPort;
var wsServer = require('websocket').server;
var http = require('http');
var clients = [];
var server = http.createServer(function (request, response) {
 response.writeHead(404);
 response.end();
});

server.listen(wsSocketPort, function () {
 console.log((new Date()) + ' listening on port ' + wsSocketPort);
});

var ws = new wsServer({
 httpServer: server,
 autoAcceptConnections: false
});

function originIsAllowed (origin) {
 return true;
};

ws.on('connect', function (request) { });

ws.on('request', function (request) {
 if (!originIsAllowed(request.origin)) {
```

```
30   request.reject();
31   console.log((new Date()) + ' Rejected: ' + request.origin);
32  }
33
34  var connection = request.accept('echo-protocol', request.origin);
35  var mainClient = {
36   key: request.key,
37   connection: connection,
38   remoteAddress : request.remoteAddress,
39   remoteType: ''
40  };
41  clients.push(mainClient);
42
43  connection.on('message', function (message) {
44   if (message.type === 'utf8') {
45    var json = JSON.parse(message.utf8Data);
46
47    clients.forEach(function (client) {
48     if (client.key === request.key) {
49      console.log(client.key + ' ' + request.key + ' ' + client.
            remoteType + ' ' + json.remoteType);
50      client.remoteType = json.remoteType;
51     }
52    });
53    var remoteTypeUiClients = clients.filter(function (client) {
54     return client.remoteType === 'ui';
55    });
56    remoteTypeUiClients.forEach(function (client) {
57     console.log('sending message to ' + client.remoteAddress + ' ' +
           message.utf8Data);
58     client.connection.send(message.utf8Data);
59    });
60   } else if (message.type === 'binary') {
61    console.log('Received binary length: ' + message.binaryData.length);
62   }
63  });
64
65  connection.on('close', function (connection) {
66   var client = {
67    key: request.key,
68    connection: connection
69   };
70   var index = clients.indexOf(client);
71   for (var i in clients) {
72    if (clients[i].key === request.key) {
73     index = i;
74    }
75   };
76   console.log((new Date()) + ' Disconnected client: ' + client.key + '
        at index: ' + index);
77   clients.splice(index, 1);
78   console.log('Clients amount after disconnect: ' + clients.length);
79  });
80 });
```

Listing A.11: D3 Graph

```
1 var Graph = function (config, chartIdentity) {
2  var logger = log4javascript.getLogger("UI.graph");
3  var chart= '';
```

```
4  var rect = '';
5  var data = new Array();
6  var width = 0;
7  var height = 0;
8  var widthScale = 0;
9  var heightScale = 0;
10 var color = 0;
11 var yAxis = 0;
12 var xAxis = 0;
13 var currentMeasureDatum = new Date();
14 var currentMessagesPerSecond = 0;
15
16 this.start = function () {
17  init();
18  updateEnvironment();
19  create();
20 };
21
22 this.update = function (newData, countPerSecond) {
23  update(newData, countPerSecond);
24 }
25
26 function getCurrentDatum() {
27  return new Date().getTime() / 1000;
28 }
29
30 function init () {
31  currentMeasureDatum = getCurrentDatum();
32  width = config.width - config.margin.left - config.margin.right;
33  height = config.height - config.margin.top - config.margin.bottom;
34  chart = d3.select(chartIdentity)
35   .append('svg')
36    .attr('width', width + config.margin.left + config.margin.right)
37    .attr('height', height + config.margin.top + config.margin.bottom)
38   .append('g')
39    .attr('transform', 'translate(' + config.margin.left+ ', ' + config.
        margin.top + ')');
40 }
41
42 function updateEnvironment() {
43  var maxDataX = d3.max(data, function (d) { return d.count; });
44  color = d3.scale.linear()
45   .domain([0, maxDataX])
46   .range(config.colorRange);
47
48  widthScale = d3.scale.linear()
49   .domain([0, maxDataX])
50   .range([0, width]);
51
52  heightScale = d3.scale.ordinal()
53   .domain(d3.range(data.length))
54   .rangeBands([0, height], 0.1);
55
56  xAxis = d3.svg.axis().scale(widthScale)
57   .orient('bottom')
58   .ticks(5);
59
60  yAxis = d3.svg.axis().scale(heightScale)
61   .orient('left')
62   .tickFormat(function (d,i) { return data[i].word; });
63 }
```

```
64
65 function create() {
66  var bars = '';
67  bars = chart.selectAll('.bar')
68   .data(data)
69   .enter().append('g')
70   .attr('class', 'bar')
71   .attr("transform", function(d, i) { return 'translate(' + 0 + ',' +
        heightScale(i) + ')'; });
72
73  bars.append('rect')
74   .attr('fill', function(d) { return color(d.count); })
75   .attr('width', function(d) { return widthScale(d.count); })
76   .attr('height', heightScale.rangeBand());
77
78  bars.append('text')
79   .attr('x', function(d) { return widthScale(d.count); })
80   .attr('y', 0 + heightScale.rangeBand() / 2)
81   .attr('dx', -18)
82   .attr('dy', '.35em')
83   .attr('text-anchor', 'end')
84   .text(function(d) { return d.count; });
85
86  chart.append('g')
87   .attr('class', 'x axis')
88   .attr('transform', 'translate(0,' + height + ')')
89   .call(xAxis);
90
91  chart.append('g')
92   .attr('class', 'y axis')
93   .call(yAxis);
94
95  chart.append('text')
96   .attr('x', width / 2)
97   .attr('y', height + config.margin.bottom)
98   .style('text-anchor', 'middle')
99   .style('fill', 'black')
100  .style('font-family', 'arial')
101  .style('font-size', '12px')
102  .text('Count');
103
104  chart.append('text')
105  .attr('transform', 'rotate(-90)')
106  .attr('y', 10 - config.margin.left)
107  .attr('x', 0 - (height / 2))
108  .attr('dy', '1em')
109  .style('text-anchor', 'middle')
110  .style('fill', 'black')
111  .style('font-family', 'arial')
112  .style('font-size', '12px')
113  .text('Words');
114
115  chart.append('text')
116  .attr('x', 0 - config.margin.left)
117  .attr('y', 10 - config.margin.top)
118  .text('Messages per second: ');
119
120  chart.append('text')
121  .attr('class', 'mps')
122  .attr('x', 80)
123  .attr('y', 10 - config.margin.top)
```

```
124   .text('0');
125 }
126
127  function update(newData, countPerSecond) {
128   data = newData;
129   d3.selectAll(chartIdentity + ' > *').remove();
130   init();
131   updateEnvironment();
132   create();
133
134   chart.select('.mps')
135    .transition()
136     .text(countPerSecond);
137
138   var rect = chart.selectAll(".bar rect").data(data);
139   var text = chart.selectAll(".bar text").data(data);
140     rect.transition()
141       .attr("width", function(d) { return widthScale(d.count); })
142       .attr("fill", function(d) { return color(d.count); });
143     text.transition()
144       .attr("x", function(d) { return widthScale(d.count); })
145       .text(function(d) { return d.count; });
146   chart.selectAll(".y.axis")
147         .call(yAxis)
148   chart.selectAll(".x.axis")
149         .call(xAxis);
150  }
151 }
```

Listing A.12: Auszug Main Javascript

```
1  // ... AUSZUG ...
2
3  function startApacheStorm () {
4   var stormWebSocketManager = new WebSocketManager(
        webSocketConfiguration.storm);
5   var stormGraph = new Graph(graphConfiguration.uiSetup,
        graphConfiguration.storm.chartIdentity);
6
7   stormWebSocketManager.start();
8   stormGraph.start();
9
10  $('#eventNode').on('messageIncomeStorm', function (event) {
11   var json = JSON.parse($(this).data('eventsData'));
12   stormGraph.update(json.wordCount, json.countPerSecond);
13  });
14 };
```

Listing A.13: Java Factory WebSocketClient

```
1 package lan.s40907.websocketclient;
2
3 import java.util.Iterator;
4 import java.util.ServiceLoader;
5
6
7 public class WebSocketClientFactory {
8   private static IWebSocketClient webSocketClient;
9
```

```
10   public static IWebSocketClient getInstance() throws Exception {
11     if (webSocketClient == null) {
12       IWebSocketConfiguration webSocketConfiguration = getService(
             IWebSocketConfiguration.class);
13       webSocketClient = getService(IWebSocketClient.class);
14       webSocketClient.setup(webSocketConfiguration);
15       webSocketClient.connect();
16     }
17     return webSocketClient;
18   }
19
20   private static <T> T getService(Class<T> classToLoad) throws
         InstantiationException, IllegalAccessException,
         ClassNotFoundException {
21     ServiceLoader<T> serviceLoader = (ServiceLoader<T>) ServiceLoader.
           load(classToLoad);
22     try {
23       return getSingle(serviceLoader.iterator());
24     } catch (IllegalAccessException e) {
25       return null;
26     }
27   }
28
29   private static <T> T getSingle(Iterator<T> iterator) throws
         IllegalAccessException, ClassNotFoundException {
30     T first = null;
31     try {
32       first = iterator.next();
33     } catch (Exception e) {
34       throw new ClassNotFoundException("Check resources META-INF", e);
35     }
36
37
38
39     if (first != null && !iterator.hasNext()) {
40       return first;
41     } else {
42       throw new IllegalAccessException("Collection has no or more
             element(s). A single expects one element.");
43     }
44   }
45 }
```

A.6 Installationsanleitung Aurora/Borealis

In dieser Anleitung wird die Installation von Aurora Borealis in kleinen Unterkapiteln vorgestellt. Diese Anleitung setzt ein Vorwissen in der Verwendung und Administration von Linux voraus. Zudem wird Erfahrung von Erzeugen von Anwendungen aus Quelltext benötigt. Zum Beispiel können Konflikte auftreten, wenn neue Versionen von Bibliotheken benötigt werden. Dabei müssen die Abhängigkeiten beachtet und abhängige Konflikte aufgelöst werden. Bevor das Erstellen der Anwendung beginnt werden zuerst die Voraussetzungen bestimmt und erläutert. Anschließend wird mit Quelltextfragementen und Kommandozeilenausschnitten schrittweise die Eingabe und Ausgabe gezeigt.

A.6.1 Voraussetzungen am Betriebssystem

Die Installation von Aurora Borealis benötigt ein auf linuxbasiertes Betriebsystem. Auf dem Betriebssystem Microsoft Windows wurde eine Erstellung des Quelltextes von Aurora Borealis bisher nicht durchgeführt. Zum Zeitpunkt der Erstellung dieser Anleitung wird versucht ein Ist-Zustand der Anwendung aufzunehmen. Für das Verteilte System Aurora Borealis wird die Linuxdistribution Debian benutzt.

A.6.2 Voraussetzung Erstellsystem

Einige Pakete bzw. Bibliotheken werden für das Erstellen von Aurora Borealis benötigt. Als Paketverwaltung wird unter Debian *apt* benutzt. Mit dem Befehl *apt-get* können Pakete dem Betriebssystem aus dem Standard Debian Paket-Repository hinzugefügt werden. Folgende Liste zeigt benötigte Pakete für das Erstellen von Aurora Borealis:

- build-essentials (gcc, g++, configure, make)
- ccache
- antlr
- libxerces-c3.1 (Xerces-c: Used by Borealis to parse XML)
- libtool
- autoconf
- automake
- libdb5.1 (Berkeley-Db)
- glpk (GNU Linear Programming Kit)
- gsl (GNU Scientific Library - collection of routines for numerical analysis: used for predictive queries)
- opencv (open source computer vision: used for array processing)
- doxygen (serves to generate documentation)
- openjdk-7-jdk (java 7)

Da die letzte Version von Aurora Borealis aus dem Jahr 2008 ist, gibt es beim Erstellen mit neueren Versionen von *gcc* und *g++* Fehler. Damit die neue Version von *gcc* benutzt werden kann muss der Quelltext der Version aus 2008 angepasst werden. Eine erste Anpassung wurde versucht durchzuführen. Zum Beispiel sind bestimmte Standardmethoden direkt angegeben worden. Trotzdem wurden weiterführende Fehler gefunden. Als Fehler wird *missing #include* gemeldet. Im *borealis* Verzeichnis sind 239 *Makefiles* vorhanden. Alle *Makefiles* und die darin verbundenen Quelltext-Dateien müssen auf die neue Version geprüft werden. Eine Stabile Version mit den neuen Anpassungen kann nur durch effektive Testläufe gewährleistet werden. Um den Quelltext von Aurora Borealis nicht anzupassen kommt die ältere Version 4.0 von Debian zum Einsatz. In diesem Fall muss die Liste von benötigten Paketen angepasst werden:

- build-essentials (gcc, g++, configure, make)
- ccache
- antlr
- libxerces27 (Xerces-c: Used by Borealis to parse XML)
- libtool
- autoconf
- automake
- libdb (Berkeley-Db)
- glpk (GNU Linear Programming Kit)
- gsl (GNU Scientific Library - collection of routines for numerical analysis: used for predictive queries)
- opencv (open source computer vision: used for array processing)
- doxygen (serves to generate documentation)
- sun-java5-jdk (Java 1.5 von SUN)
- libexpat1-dev
- ibreadline5-dev

A.6.3 Quelltext von Aurora Borealis herunterladen

Die Datei liegt nicht in einer öffentlichen Versionsverwaltung, sondern kann als Archive von der Brown University unter folgendem Link heruntergeladen werden: http://www.cs.brown.edu/research/borealis/public/download/borealis_summer_2008.tar.gz
Alternativ liegt der Quelltext und das Installationsmedium Debian 4.0 als International Organization for Standardization (ISO) 9660 Abbild im Ordner *anhangSoftwareZusatz*. Nachdem die Anwendung im Verzeichnis */opt* liegt kann sie im gleichen Verzeichnis entpackt werden. Im Verzeichnis liegen anschließend zwei Unterordner *borealis* und *nmstl*.

A.6.4 Kommandozeile Umgebungsvariablen festlegen

Unter Debian wird für die Kommandozeile die Shell *Bash* eingesetzt. Wenn eine Shell eröffnet wird, wird die Datei *.bashrc* im Benutzerverzeichnis aufgerufen. Darin werden Benutzerabhängige Konfigurationen abgelegt. Die Umgebungsvariablen für Aurora Borealis werden im folgenden Abschnitt gezeigt:

```
alias debug='export LOG_LEVEL=2'
alias debug0='export LOG_LEVEL=0'
export PATH=${PATH}:/opt/nmstl/bin:/${HOME}/bin
export CLASSPATH='.:/usr/share/java/antlr.jar:$CLASSPATH'
```

```
export JAVA_HOME=$(readlink -f /usr/bin/javac | sed "s:/bin/javac::")
export PATH=${JAVA_HOME}/bin:${PATH}
export CXX='ccache g++'
export CVS_SANDBOX='/opt'
export INSTALL_BDB='/usr'
export INSTALL_GLPK='/usr'
export INSTALL_GSL='/usr'
export INSTALL_ANTLR='/usr'
export INSTALL_XERCESC='/usr'
export INSTALL_NMSTL='/usr/local'
export LD_LIBRARY_PATH='/usr/lib'
export ANTLR_JAR_FILE='/usr/share/java/antlr.jar'

mkdir -p bin
alias bbb='/opt/borealis/utility/unix/build.borealis.sh'
alias bbbt='/opt/borealis/utility/unix/build.borealis.sh -tool.head
 -tool.marshal'
alias retool='/bin/cp -f ${CVS_SANDBOX}/borealis/tool/head/BigGiantHead
 ${HOME}/bin; /bin/cp -f ${CVS_SANDBOX}/borealis/tool/marshal/marshal
 ${HOME}/bin'
```

A.6.5 Notwendige Quelltext Anpassung

Beim Erzeugen wird unter anderen Paketen auch ANother Tool for Language Recognition (ANTLR) benutzt. Während der Erstellens findet ein Fehler auf. Bei der Konfiguration für das *Makefile* wurde ein Pfad fest einprogrammiert. Dieser feste Pfad wird nun durch einen Variable in die Umgebungsvariable *ANTLR_JAR_FILE* ausgelagert. Dazu muss in der Datei */opt/borealis/src/configure.ac* der Inhalt an der Stelle ANTLR_JAR_FILE= mit *$ANTLR_JAR_FILE* nach dem Gleichzeichen ausgetauscht werden.

A.6.6 Erzeugen von NMSTL

Borealis benutzt eine angepasste Version von NMSTL. Im Verzeichnis */opt/nmstl* werden nun folgende Befehle nacheinander ausgeführt:

```
autoconf
./configure
make
make install
```

A.6.7 Erzeugen von Borealis

In das Verzeichnis */opt/borealis* zurückspringen und folgende Befehle nacheinander ausführen:

```
bbb
```

```
bbbt
retool
make install
```

A.6.8 Zusätzliche Informationen

Der Support für Aurora Borealis ist seit 2008 eingestellt. Weitere Informationen stehen unter folgendem Link: http://cs.brown.edu/research/borealis/public/install/install.borealis.html

A.7 Installationsanleitung Apache Storm

In diesem Kapitel wird die Installation von Apache Storm in kleinen Unterkapiteln vorgestellt. Die Anleitung setzt ein Wissen in der Verwendung und Administration des Betriebssystems Linux voraus. Zuerst werden Voraussetzungen bestimmt und erläutert. Anschließend wird der Start eines Clusters gezeigt. Zuletzt wird eine Beispiel-Anwendung *WordCount* im *local cluster mode* ausgeführt.

A.7.1 Voraussetzungen am Betriebssystem

Als Betriebssystem wird in dieser Anleitung Linux mit der Distribution Debian in Version 7 verwendet. Apple Mac OS X und Microsoft Windows mit einer Cygwin-Umgebung werden in dieser Anleitung nicht betrachtet. Unter Debian wird für die Installation von Paketen das Kommandozeilen-Werkzeug *aptitude* eingesetzt. Folgende Linux-Pakete werden in Debian für die Ausführung von Apache Storm benötigt:

- zookeeper
- openjdk-7-jdk

A.7.2 Storm Konfiguration

Apache Storm in das Verzeichnis /opt herunterladen, entpacken und einen Link *storm* erstellen.

```
>wget http://apache.openmirror.de/incubator/storm/ //
apache-storm-0.9.1-incubating/apache-storm-0.9.1-incubating.tar.gz
>tar xvfz apache-storm-0.9.1-incubating.tar.gz
>ln -s storm apache-storm-0.9.1-incubating
```

Die Konfigurationsdatei /opt/storm/conf/storm.yaml öffnen und folgenden Eintrag hinzufügen:

```
storm.local.dir: "/opt/storm"
```

A.7.3 Cluster starten

Zookeeper muss bereits im Hintergrund als Dienst laufen, damit das Storm Cluster starten kann.

Mit folgendem Befehl kann der Zookeeper Dienst auf Aktivität geprüft werden.

```
>./zkCli.sh -server 127.0.0.1:2181
```

Die folgenden Schritte zeigen nacheinander den Start der Storm Komponenten:

```
/opt/storm/bin/storm nimbus
/opt/storm/bin/storm supervisor
/opt/storm/bin/storm ui
```

A.7.4 WordCount Demo im local cluster mode

Mit git wird zuerst die Beispiel Anwendung WordCount in ein lokales Verzeichnis dublizieren:

```
>git clone git://github.com/apache/incubator-storm.git
```

Die Anwendung WordCountTopology wird mit Apache Maven im storm cluster bereitgestellt und ausgeführt:

```
>mvn -f m2-pom.xml compile exec:java -Dexec.classpathScope=compile //
-Dexec.mainClass=storm.starter.WordCountTopology
```

Da keine Argumente bei der Ausführung übergeben werden, wird als *cluster* der *LocalCluster* benutzt. In der Java Klasse *WordCountTopology* wird in der *main*-Methode entschieden, ob der LocalCluster benutzt wird. Als Ausgabe werden während der Verarbeitung Log Informationen ausgegeben. Eine Erfolgsmeldung wird ausgegeben, falls das Erstellen und Ausführen auf dem Cluster erfolgreich durchgeführt wurde.

A.8 Installationsanleitung Apache Kafka

Dieses Kapitel beschreibt schrittweise die Installation von Apache Kafka. Die Installation erfordert Kenntnisse in der Installation und Administration von Anwendungen unter dem Betriebssystem Linux. Eine Installation unter dem Betriebssystem Unix, Max OS X und Microsoft Windows wird in dieser Arbeit nicht behandelt. Bevor die Installation beginnen kann, werden zuerst Voraussetzung an das Betriebssystem aufgezählt. Im Kapitel Installation wird der Quelltext kompiliert und ein erster Start von Kafka wird gezeigt. Abschließend wird eine Konfiguration für ein Kafka Single Broker Cluster vorgestellt und Beispiel zwischen einem Producer, dem Nachrichtenerzeuger und einem Consumer, dem Nachrichtenempfänger gezeigt.

A.8.1 Voraussetzungen in Linux

Als Linux Distribution wird Debian 7 verwendet. Da Apache Kafka auf Scala basiert, werden folgende Linux-Pakete benötigt:

- zookeeper
- scala
- openjdk-7-jdk

Das Hinzufügen der Linux-Pakete kann mit der Paketverwaltung *apt-get* oder mit *aptitude* erfolgen. Weitere Paket-Abhängigkeiten werden automatisch vorgeschlagen. Die Installation der Paket-Abhängigkeiten ist zwingend und Konflikte müssen aufgelöst werden.

Um die notwendige Abhängigkeiten, wie unter Java mit Maven[1] unter Scala aufzulösen, wird das Werkzeug Scala Build Tool (SBT) von der Webseite http://www.scala-sbt.org/download.html benötigt. Zunächst wird in das Verzeichnis *opt* gewechselt und das Archiv *sbt* heruntergeladen. Anschließend wird das Archiv entpackt und Drei *sbt*-Dateien werden im System global bereitgestellt:

```
cd /opt
wget http://dl.bintray.com/sbt/native-packages/sbt/0.13.5/sbt-0.13.5.tgz
tar xvfz sbt-0.13.5.tgz
cd sbt
mv sbt /usr/local/bin
mv sbt-launch-lib.bash /usr/local/bin
mv sbt-launch.jar /usr/local/bin
```

Mit *sbt console* kann eine Scala-Shell eröffnet werden. Das *Build*-Werkzeug ist korrekt bereitgestellt, wenn nach dem Aufruf eine Scala-Eingabezeile erscheint. Mit dem Befehl *:quit* kann die Scala-Eingabeumgebung wieder geschlossen werden. Als nächstes wird der Quelltext von Apache Kafka heruntergeladen, entpackt, Apache Kafka kompiliert und der Packet-Cache aktualisiert. Beim ersten Aufruf von *sbt* werden benötigte Scala-Bibliotheken automatisch heruntergeladen.

```
cd /root
wget http://apache.openmirror.de/kafka/0.8.1.1/kafka-0.8.1.1-src.tgz
cd kafka-0.8.1.1
gradlew jar
sbt update
sbt package
sbt sbt-dependency
```

Falls Zookeeper unter Debian in der Standardkonfiguration noch nicht ausgeführt wird, kann mit „*/etc/init.d/zookeeper start*“ der Dienst gestartet werden.

[1] Maven: Java Build Werkzeug http://maven.apache.org/

A.8.2 Start Single Broker Cluster

Im Kafka-Verzeichnis *config* liegen die Konfigurationen für den Betrieb eines Clusters. In der Datei *server.properties* wird eine eindeutige Nummer für den *Broker*, eine Log-Datei und eine Referenz zum *Zookeeper-Server* angegeben.

```
Broker.id=0
log.dir=/tmp/kafka.log
zookeeper.connect=localhost:2181
```

Folgender Befehl startet den Kafka-Server in den Standardeinstellungen:

```
bin/kafka-server-start.sh config/server.properties
```

A.8.3 Producer und Consumer ausführen

Sobald der Single Broker Cluster läuft kann mit den Shell-Skripten in separaten Shells ein Konsolen-Producer und ein Konsolen-Consumer gestartet werden.

Starten einer Topics:

```
bin/kafka-Console-topic.sh --replica 2 --zookeeper localhost:2181
 --topic contop
```

Starten einer Producers. Nach dem Start kann sofort in der Eingabezeile etwas eingetippt werden.

```
bin/kafka-Console-producer.sh --broker-list localhost:9092 --sync
 --topic contop
```

Starten einer Consumers. Sobald der Consumer gestartet wurde, werden die Eingaben vom Producer auf der Kommandozeile ausgegeben.

```
bin/kafka-Console-consumer.sh --zookeeper localhost:2181 --from-beginning
 --topic contop
```

A.9 Installationsanleitung Apache Flume

In diesem Kapitel wird die Installation von Apache Flume schrittweise gezeigt. Für die Installation von Apache Flume wird die aktuelle Version 1.5.0.1 verwendet. Als Betriebssystem wird Linux mit der Distribution Debian 7 eingesetzt. Eine Kompilation der Anwendung Apache Flume wird nicht durchgeführtm, daher werden einfache Kenntnisse in der Administration und Pflege von Linux-Betriebssystemen in der Shell Bash vorausgesetzt. Nach der Installation wird eine kleine Anwendung eines Apache Flume Agenten vorgestellt.

Für den Einsatz von Apache Flume ist eine Java-Umgebung-Laufzeitumngebung notwendig. Die Installation der Debian Java7-Pakets kann mit Werkzeug *aptitude* oder *apt-get* durchgeführt werden.

```
aptitude install openjdk-7-jdk
```

Anschließend wird die vorkompilierte Apache Flume Anwenundung als Archiv vom Apache Server in das Verzeichnis */opt* heruntergeladen und entpackt. Abschließend wird das entpackte Verzeichnis in den Namen *flume* umbenannt:

```
cd /opt
wget http://www.apache.org/dist/flume/1.5.0.1/apache-flume-1.5.0.1-bin.tar.gz
tar xvfz apache-flume-1.5.0.1-bin.tar.gz
mv apache-flume-1.5.0.1-bin.tar.gz flume
```

A.9.1 Apache Flume Konfiguration

Damit Apache Flume ordentlich ausgeführt wird, muss die Konfiguration von Apache Flume für die Java-Umgebung und die JAVA_HOME-Variable gesetzt sein. Im Unterverzeichnis conf befinden sich template-Konfigurationsdateien. Zuerst werden die Template-Dateien in Konfigurationsdateien kopiert und anschließend bearbeitet:

```
cp flume-env.sh.template flume-env.sh
cp flume-conf.properties.template flume-conf.properties
```

Mit einem Texteditor wie zum Beispiel vim, emacs oder nano können Dateien bearbeitet werden:

```
vim flume-env.sh
```

Die folgende Konfigurationsdatei *flume-env.sh* stellt ein Beispiel für Apache Flume dar. Zum einen wird die JAVA_HOME-Variable mit dem richtigen Pfad zum Java-Installationsort gesetzt und zum Anderen wird das Moninoring über JMX mit Optionen für den Java Heap bereitgestellt.

Listing A.14: Apache Flume Konfiguration

```
1  # Licensed to the Apache Software Foundation (ASF) under one
2  # or more contributor license agreements.  See the NOTICE file
3  # distributed with this work for additional information
4  # regarding copyright ownership.  The ASF licenses this file
5  # to you under the Apache License, Version 2.0 (the
6  # "License"); you may not use this file except in compliance
7  # with the License.  You may obtain a copy of the License at
8  #
9  #     http://www.apache.org/licenses/LICENSE-2.0
10 #
11 # Unless required by applicable law or agreed to in writing, software
12 # distributed under the License is distributed on an "AS IS" BASIS,
```

```
13 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
14 # See the License for the specific language governing permissions and
15 # limitations under the License.
16
17 # If this file is placed at FLUME_CONF_DIR/flume-env.sh, it will be
       sourced
18 # during Flume startup.
19
20 # Enviroment variables can be set here.
21
22 JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64
23
24 # Give Flume more memory and pre-allocate, enable remote monitoring via
       JMX
25 JAVA_OPTS="-Xms100m -Xmx200m -Dcom.sun.management.jmxremote"
26
27 # Note that the Flume conf directory is always included in the classpath
       .
28 #FLUME_CLASSPATH=""
```

Die Konfigurationsdatei *flume-conf.properties* kann verändert werden, wenn das Standardprotokollverhalten unerwünscht ist. In der Standardeinstellung wird über einen eigenen Agenten mit Log4j im Verzeichnis *logs* und der Datei *flume.log* protokolliert. Weitere Einstellungen können in der Datei *log4j.properties* vorgenommen werden.

Folgender Aufruf zeigt Optionen für einen Apache Flume-Aufruf aus dem *flume*-Verzeichnis:

```
bin/flume-ng --help
```

A.9.2 Apache Flume Single-Node Beispiel

Folgendes Beispiel wird an dieser Stelle von der Apache Flume 1.5.0 User Guide [Flu12a] wieder gegeben und wird im Verzeichnis *conf* als *example.conf* abgepeichert.

Listing A.15: Apache Flume Beispiel

```
1 # example.conf: A single-node Flume configuration
2
3 # Name the components on this agent
4 a1.sources = r1
5 a1.sinks = k1
6 a1.channels = c1
7
8 # Describe/configure the source
9 a1.sources.r1.type = netcat
10 a1.sources.r1.bind = localhost
11 a1.sources.r1.port = 44444
12
13 # Describe the sink
14 a1.sinks.k1.type = logger
15
16 # Use a channel which buffers events in memory
17 a1.channels.c1.type = memory
18 a1.channels.c1.capacity = 1000
19 a1.channels.c1.transactionCapacity = 100
```

```
20
21 # Bind the source and sink to the channel
22 a1.sources.r1.channels = c1
23 a1.sinks.k1.channel = c1
```

Im Beispiel Listing A.15 wird ein Agent *a1* definiert. In der Quelle *source r1* wird der Linux-Befehl *netcat* auf der internen Netzwerkkarte unter dem Port 44444 als Server-Dienst aktiviert. Damit können beliebige Daten über einen Client wie zum Beispiel *telnet* an den Apache Flume-Agenten gesendet werden. Die Quelle *source r1* leitet die Nachrichten an den Kanal *channel c1* im Zwischenspeicher weiter. In der Sänke *sink c1* wird die Protokollanwendung *log4j* mit dem Bezeichner *logger* aus dem Unterkapitel A.9.1 gesetzt. Die Ausgabe erfolgt abschließend durch *log4j-Appender* (Konsole oder Datei).

Der Apache Flume Agent wird in einer eigenen Shell mit der Beispiel Konfiguration gestartet:

```
bin/flume-ng agent --conf conf --conf-file conf/example.conf --name a1
```

In einer separaten Shell kann eine erfolgreiche Verbindung über *telnet* mit dem TCP-Endpunkt *localhost* Port *44444* aufgebaut, Nachrichten eingegeben und mit dem Bestätigen der Eingabetaste an den Agenten übermittelt werden:

```
telnet localhost 44444
```

Bei einer Eingabe von *Ich bin s40907.* über Telnet auf den Port 44444, wird vom Agenten folgende Nachricht an den LoggerSink ausgegeben:

Listing A.16: Apache Flume Ausgabe LoggerSink

```
1 21 Jul 2014 20:59:19,820 INFO  [SinkRunner-PollingRunner-
     DefaultSinkProcessor] (org.apache.flume.sink.LoggerSink.process:70)
      - Event: { headers:{} body: 49 63 68 20 62 69 6E 20 73 34 30 39 30
     37 0D    Ich bin s40907. }
```

A.10 Installationsanleitung Apache S4

Diese Kapitel zeigt die Installation von Apache S4 mit einem einfachen Beispiel. Für die Installation wird ein Basiswissen in der Verwendung von Linux-Betriebssystemen und Kenntnisse in der Java-Programmierung vorausgesetzt. Zuerst werden die Voraussetzungen für das Betriebssystem vorgestellt.

A.10.1 Voraussetzungen am Betriebssystem

Als Betriebssystem wird die die Distribution Debian Version 7 verwendet. Mit der Paketverwaltung *aptitude* werden folgende Pakete und deren Paketabhängigkeiten benötigt. Die Paketabhängigkeiten werden von *aptitude* automatisch vorgeschlagen. Mögliche Konflikte in den Paketabhängigkeiten müssen zuvor manuell aufgelöst werden.

- zookeeper
- zookeeperd
- openjdk-7-jdk
- unzip

Sobald die Pakete im System bereitgestellt wurden, wird für das Erstellen von Apache S4 das Build-Management-Werkzeug *Gradle* benötigt. Beim Erzeugen von Apache S4 in der aktuellen Version *0.6.0-incubating* bekommt *Gradle* ab Version 2.0 Fehler. Daher wird *Gradle* in der Version 1.12 eingesetzt.

A.10.2 Installation Build-Managment Gradle

Mit *Gradle* kann Apache S4 erzeugt und gepflegt werden. Weiterhin kann mit *Gradle* eine Beispiel-Anwendung *HelloApp* bereitgestellt werden.

```
wget https://services.gradle.org/distributions/gradle-1.12-bin.zip
unzip gradle-1.12-bin.zip
```

Umgebungsvariablen setzen:

```
export GRADLE_HOME=/opt/gradle
export PATH=$PATH:$GRADLE_HOME/bin
```

Gradle erzeugen:

```
/opt/gradle/gradle
```

A.10.3 Bereitstellen Apache S4

Da bestimmte Befehle eine mehrere Optionen benötigen, kann es in der Darstellung zu unleserlichen Umbrüchen kommen. An den markanten Stellen wird daher explizit umgebrochen und mit doppelten Schrägstrichen gekennzeichnet. In der Konsole werden keine Umbrüche benötigt. Als nächstes wird Apache S4 in das Verzeichnis „/opt“ heruntergeladen, entpackt und ein Verweis „s4“ wird erzeugt.

```
wget http://www.apache.org/dist/incubator/s4/s4-0.6.0-incubating //
 /apache-s4-0.6.0-incubating-src.zip
unzip apache-s4-0.6.0-incubating-src.zip
ln -s /opt/apache-s4-0.6.0-incubating-src s4
```

Für das erfolgreiche Ausführen sind verschiedene Umgebungsvariablen in der Konsole notwendig. Die folgende Einträge werden in der Datei „.bashrc“ der Konsolen-Konfiguration des Benutzerprofils hinterlegt. Damit die neuen Einträge in der Umgebung bekannt sind, ist eine erneute Anmeldung in der Konsole notwendig.

```
export S4_HOME=/opt/s4
export PATH=$PATH:$S4_HOME
```

A.10.4 Installation S4

Da bestimmte Werkzeuge in Apache S4 den *Wrapper gradlew* Version 1.4 benötigen, wird zuerst in das Verzeichnis gewechselt und *gradlew* bereitgestellt.

```
cd /opt/s4
gradle wrapper
```

Als nächstes wird Apache S4 im gleichen Verzeichnis installiert.

```
gradle install
gradle s4-tools::installApp
```

A.10.5 Cluster erzeugen

Für die Verarbeitung von Informationen wird in Apache S4 ein Cluster benötigt. Das Cluster benutzt im Hintergrund Apache Zookeeper. Der Dienst Apache Zookeeper muss in dieser Anwendung auf dem Standard-Port 2181 bereitstehen. Folgende Befehl startet im Verzeichnis „/opt/s4“ ein Cluster mit dem Namen cluster1, Zwei Verarbeitungseinheiten und dem Port 12000.

```
./s4 newCluster -c=cluster1 -nbTasks=2 -flp=12000
```

In Zwei weiteren Konsolen wird jeweils ein Prozess im Cluster cluster1 gestartet.

```
./s4 node -c=cluster1
```

A.10.6 Beispiel HelloApp

Die Beispiel-Anwendung „HelloApp“ kann mit *Gradle* in einem separaten Verzeichnis „/opt /myApp“ mit folgenden Befehlen in der Konsole aus dem Verzeichnis „/opt/s4“ bereitgestellt werden.

```
./s4 newApp myApp -parentDir=/opt
```

Anschließend muss die Beispiel-Anwendung im neuen Verzeichnis „/opt/myApp“ erstellt werden.

```
cd /opt/myApp
./s4 s4r -a=hello.HelloApp -b=/opt/myApp/build.gradle myApp
```

Zuletzt wird die Beispiel-Anwendung mit dem Namen „myApp“ im Cluster „cluster1“ bereitgestellt und gestartet.

```
./s4 deploy
 -s4r=/opt/myApp/build/libs/myApp.s4r //
 -c=cluster1 //
 -appName=myApp
./s4 node -c=cluster1
```

Bei erfolgreichem Start zeigt der *S4 platform loader* einen aktiven Eingangsstrom „names“.

```
21:25:31.399 [S4 platform loader] DEBUG o.a.s4.comm.topology.ClustersFromZK
- Adding input stream [names] in cluster [cluster1]
21:25:31.430 [S4 platform loader] INFO  org.apache.s4.core.App
- Init prototype [hello.HelloPE].
```

Damit das Cluster cluster1 Daten verarbeiten kann, sind Eingangsdaten notwendig. Aus der Beispiel-Anwendung HelloApp wird dazu die Klasse HelloInputAdapter verwendet und unter dem Namen adapter und einem neuen Cluster cluster2 bereitgestellt.

```
./s4 newCluster -c=cluster2 -nbTasks=1 -flp=13000
./s4 deploy
 -appClass=hello.HelloInputAdapter //
 -p=s4.adapter.output.stream=names //
 -c=cluster2 //
 -appName=adapter
```

Der folgende Befehl startet den Eingangsdatenverarbeitung im *HelloInputAdapter* aus dem „Cluster2“ unter dem Namen „adpater“ und leitet den Eingangsstrom weiter auf den *S4 stream* „names“.

```
./s4 adapter -c=cluster2
```

Mit dem Linux-Werkzeug *netcat (nc)* können Nachrichten an den *HelloInputAdapter* gesendet werden.

```
echo "s40907" | nc localhost 15000
```

In der S4 Anwendung „adapter“ wird die folgende Nachricht ausgegeben.

```
read: s40907
```

Abschließend wird je nach Auslastung die Nachricht in einem der *S4 nodes* aus Cluster „cluster1“ ausgegeben.

```
Hello s40907!
```

Ein Status der *S4 platform* kann mit folgendem Befehl ausgegeben werden.

```
./s4 status
```

Die Ausgabe des vorhergehenden Befehls wird abschließend gezeigt.

```
App Status
-------------------------------------------------------------------------
   Name      Cluster                      URI
-------------------------------------------------------------------------
 adapter     cluster2      null
  myApp      cluster1      file:/opt/myApp/build/libs/myApp.s4r
-------------------------------------------------------------------------



Cluster Status
-------------------------------------------------------------------------
                                        Active nodes
   Name      App     Tasks   ----------------------------------------
                              Number     Task id     Host     Port
-------------------------------------------------------------------------
 cluster2  adapter   1          1          Task-0    s4.lan    13000
 cluster1   myApp    2          1          Task-0    s4.lan    12000
-------------------------------------------------------------------------



Stream Status
-------------------------------------------------------------------------
   Name                  Producers                Consumers
-------------------------------------------------------------------------
  names              cluster2(adapter)         cluster1(myApp)
-------------------------------------------------------------------------
```

A.11 Inhalt der beigelegten DVD

Der Master Thesis beiliegenden DVD hat folgenden Inhalt:

/git Im Verzeichnis *git* liegt versioniert die vollständige Entwicklung des Dokuments der Master Thesis und der Prototypen.

/git/thesis/anhangMessung Im Verzeichnis */git/thesis/anhangMessung* liegen alle Messdaten der Prototypen.

/git/prototype/helper Im Verzeichnis */git/prototype/helper* liegen die Scripte und Konfigurationen für die Ausführung der Prototypen sowie der WebSocketClient als Eclipse-Projekt.

/git/prototype/prototypes Im Verzeichnis */git/prototype/prototypes* liegen die Eclipse Projekte der einzelnen Prototypen.

/MasterThesisEduardBergen.pdf Die Datei MasterThesisEduardBergen.pdf ist das Dokument zur Master Thesis Vergleich von Streamingframeworks: STORM, KAFKA, FLUME, S4 von Eduard Bergen in digitaler Form im Adobe Acrobat Dateiformat.

/prototypeExecutedSave.tar Die Datei *prototypeExecutedSave.tar* ist ein Tar-Archiv und enthält eine Sicherung des kompletten Verzeichnisses, in dem die Prototypen und die Single-Node-Cluster der Streaming Frameworks ausgeführt wurden.

/prototypeVirtualMachine Das Verzeichnis *prototypeVirtualMachine* enthält die virtuelle Maschine. Die virtuelle Maschine kann mit VirtualBox gestartet werden. Darin wird der WebServer gestratet und über die Shell können die Broker gestartet werden. Der Loginname lautet `root` und das Passwort lautet `prototype`.

/streamingFrameworks Das Verzeichnis streamingFrameworks enthält die verwendeten Streaming Frameworks im Tar-Gunzip- und Zip-Archiv.