



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 27. 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 Nachrichten durch unterschiedliche Quellen wie Sensoren (RFID) oder Nachrichtenquellen (RFD newsfeeds) wird es schwieriger Informationen beständig abzufragen. Um die Frage zu klären, welcher Rechner am häufigsten über TCP 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.

Abstract

Entwurf

Entwurf

Inhaltsverzeichnis

1	Einführung	1
2	Grundlagen	3
2.1	Grundbegriffe	3
2.2	Technologie	5
2.3	Zusammenfassung	8
3	Analyse	9
4	Vorstellung Streaming Frameworks	17
4.1	Apache Storm	17
4.2	Apache Kafka	22
4.3	Apache Flume	25
4.4	Apache S4	25
4.5	Zusammenfassung	25
5	Anwendungsfall und Prototyp	27
6	Auswertung	29
6.1	Benchmark Ergebnisse	29
6.2	Erkenntnis	29
7	Schlussbetrachtung	31
7.1	Zusammenfassung	31
7.2	Einschränkungen	31
7.3	Ausblick	31

8 Verzeichnisse	33
Literaturverzeichnis	36
Internetquellen	39
Abbildungsverzeichnis	41
Tabellenverzeichnis	43
Quellenverzeichnis	45
 A Zusätze	 47
A.1 Abkürzungen	47
A.2 Quelltext	48
A.3 Installationsanleitung Aurora/Borealis	55
A.4 Installationsanleitung Apache Storm	58
A.5 Installationsanleitung Apache Kafka	61

Entwurf

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 [SCZ05, 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. Architecture 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 [?] und Apache S4 [GJM⁺13] geben. 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 und einer Analyse mit 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 Analyse eines Referenzmodells statt. Dabei werden unter den gewonnen Grundlagen weitere Fachbegriffe eingeführt und Kriterien für eine Bewertung vorgestellt. 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 ein Anwendungsfall vorgestellt und die Streaming frameworks werden einem Belastungstest unterzogen. Das sechste Kapitel stellt das Bewertungsschema vor und stellt die Ergebnisse aus Kapitel Vier und Fünf in einer Übersicht dar. Anschließend werden die Streaming frameworks verglichen und Erkenntnisse werden gezogen. Das letzte Kapitel greift die Erkenntnisse aus Kapitel Sechs 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 Medium¹ gegenüber einem diskreten Medium², als zeitkritisch zwischen Signalen. Shannon beschreibt in [Sha48, S. 3 und S. 34] ein kontinuierliches System mit folgendem Zitat:

¹kontinuierliches Medium: Temperatursensor

²diskretes 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 Client im 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 zwischenspeichern. 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 *Continuous 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 *Continuous*

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 Referenzmodell 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 und der Markt analysiert.

Entwurf

Kapitel 3

Analyse

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 [Fou14a]. 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].

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



Abbildung 3.1: Darstellung Big Data Cube

[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 Consistency 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 Consistent (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 Consistency* 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 Versicherungen 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 Datenanalyse-Systemen 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 Nachrichtenaustausch 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, Vorausberechnen, 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 *Availability 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 zusätzlich 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 *Packet-Tuple* 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.

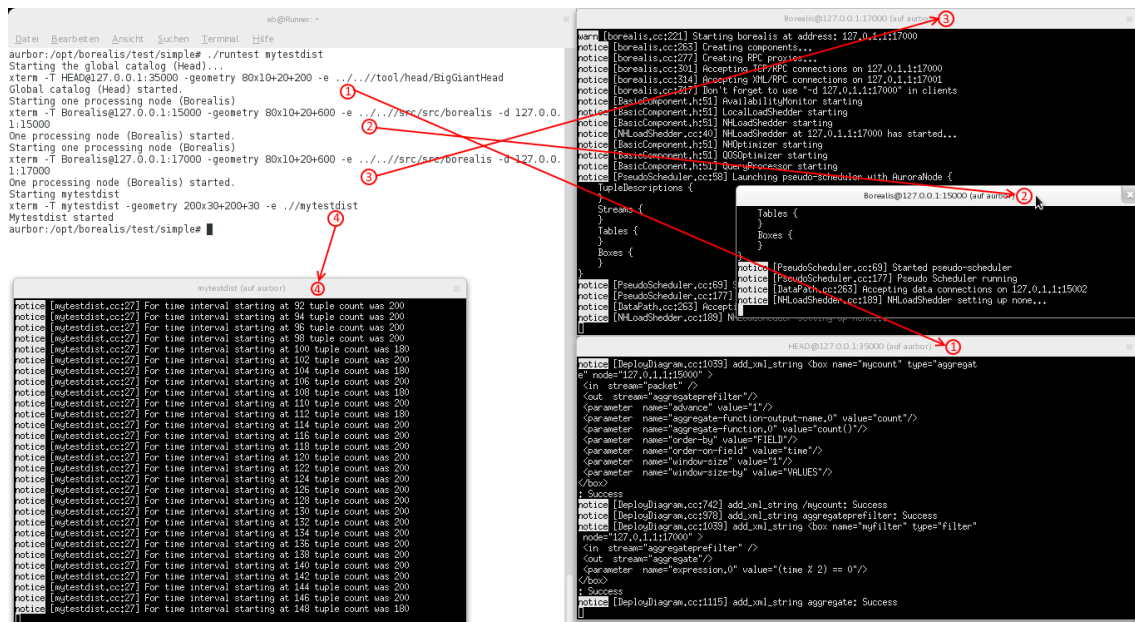


Abbildung 3.2: Aurora Borealis mit einem Master zwei Servern und einem Konsumenten

Pipelining wird durch den Einsatz von Eingangs- und Ausgangsström in Abfragen erreicht. In der Konfiguration A.2 wird von der ersten Einheit ein spezialisiertes Datentupel *AggregatePre-Filter* 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 *Consistency 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 leichtgewichtete Kontrolle kann durch den *Consistency 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.3 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 Rechneinheit 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. Optimierte 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) dazuschalten, ist eine Berechnung der Ursache eines QoS-Problems möglich. [AAB⁺05, S. 7, Kap. 5]

In der Analyse 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 Bewertungskriterien 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.

Entwurf

Kapitel 4

Vorstellung Streaming Frameworks

In Kapitel 2 und 3 wurden Grundlagen geschaffen, eine Analyse durchgeführt und ein Referenzmodell mit Bewertungskriterien vorgestellt und für die Anwendung auf die Streaming frameworks erläutert. In den folgenden Unterkapitel 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 Analyse des Referenzmodells vorgegangen. Eine kurzer Vergleich zwischen dem Referenzmodell wird am Ende des Unterkapitels eines Streaming frameworks durchgeführt.

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 [Fou14b]. Der Verlauf des Überführungsprozesses zur ASF wird auf der Incubator-Statusseite [Fou14l] 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 werden mehrere Sprachen Clojure, Java und Python angegeben. 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 [Fou14o]. Ab Version 0.9.1-incubating wird eine verbesserte Plattformkompatibilität zum Betriebssystem Microsoft Windows angeboten und die Standardtransportschicht ZeroMq [iC14] wurde durch Netty [Lee14] ersetzt [Con14].

In Tabelle 4.2 werden die Bewertungskriterien aus Kapitel 3 in Apache Storm geprüft. Als Ar-

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	Plattformübergreifend (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

chitektur 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: Apache ZooKeeper [Fou14c]. 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 jede erste 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 ein *Callback-Methode* *ack* oder *fail* in der implementierten Klasse *Spout* zurückgegeben. Damit wird in Apache Storm sichergestellt, dass die Nachricht mindestens einmal vollständig verarbeitet wurde [Fou14d]. In der Methode *declareOutputFields* wird die Felddefinition der Ausgabe für die weitere Verarbeitung angegeben. [Fou14h]

Ein *Bolt* 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* Bestätigung über die Methode *ack* oder *fail*. Andernfalls kann Apache Storm nicht feststellen, ob eine Nachricht vollständig verarbeitet wurde. [Fou14f]

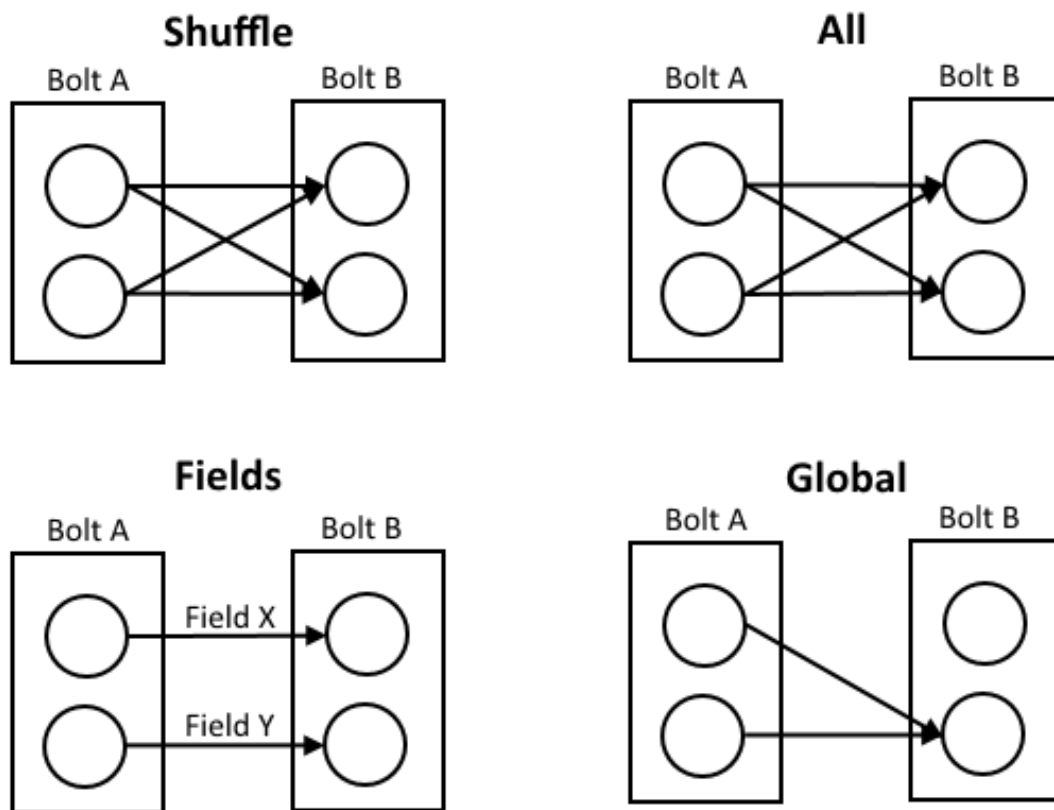


Abbildung 4.1: Apache Storm Gruppierungen

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 einem eigenen *Thread* ausgeführt. Wenn die *setBolt*-Methode aufgerufen wird, wird ein Objekt *InputDeclarer* erzeugt. Darin können verschiedene Gruppierungen (*Shuffle*, *Fields*, *All*, *Global*, *None*, *Direct*, *LocalOrShuffle* [Fou14g]) 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 eines 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* wird der *Stream* zu einem *Task* gesendet. Weiterhin gibt es noch das *NoneGrouping* bei dem der *Stream* auf dem gleichen *Task* ausgeführt wird und beim *DirectGrouping* entscheidet der *Stream*-Erzeuger auf welchen Konsumenten-*Task* der *Stream* gesendet wird. Durch die Schnittstelle *CustomStreamGrouping* ist eine weitere konkrete Implementierung für eine *Grouping*-Strategie möglich. [Fou14e]

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 [Fou14n] wird *Trident* näher erläutert. In den folgenden Absätzen wird kurz auf die möglichen Funktionen mit *Trident* aus [Fou14n] eingegangen.

Unter *Partition-local* werden Operatoren (*function*, *filter*, *partitionAggregate*, *partitionPersist*, *projection*) lokal in einem Stapелеlement 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 werden. 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, dann 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 einen „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. [Fou14k]

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 ein *Superversion* (supervisord [McD14]) der Anwendungsdienste *Nimbus* und *Supervisor* notwendig. Nachdem der *Nimbus*- oder *Supervisor*-Prozess durch den Supervisiondienst neugestartet wurde, können neue *Tasks* zu den laufenden *Tasks* auf den *Workern* hinzugefügt werden. [Fou14j]

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 zwischen Apache Zookeeper ist nicht vorhanden. Daher können auf der Anwendungsebene spezifische Richtlinien durch Access Control Lists (ACLs) umgesetzt werden. [Fou14m]

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 [Fou14k] 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 und anschließend durch die Bewertungskriterien genauer erläutert. 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 Kommandozeilenaufwurf 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 gebracht. Zu Beginn wird 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. 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] um auf die eingehenden unterschiedlichen hohen Datenmengen der Webseiten von LinkedIn Zugang zu bekommen und zu verarbeiten [Rao11].

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	Plattformübergreifend (Microsoft Windows mit Cygwin Umgebung)
Lizenz	Apache License version 2.0
Webseite	[KNR14a]
Quelltext	[KNR14b]

Tabelle 4.3: Kurzübersicht Apache Kafka

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“, 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 z.B. per Zufall 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]

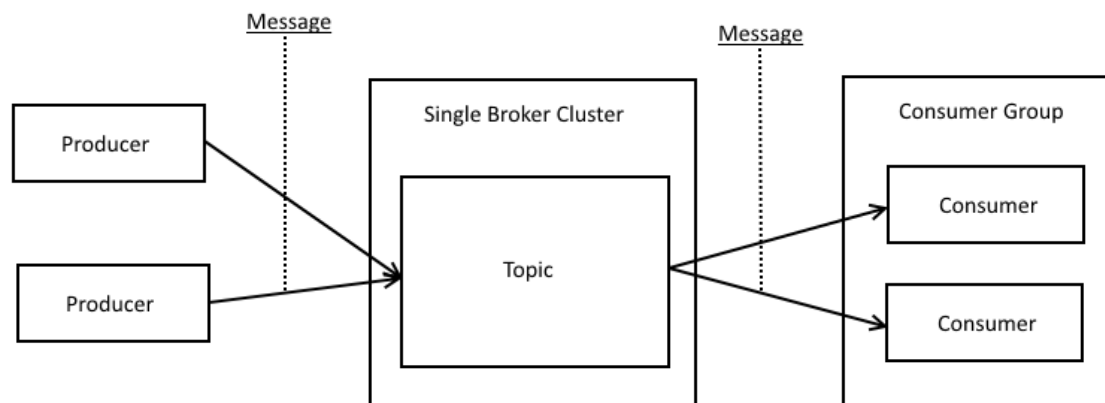


Abbildung 4.2: Apache Kafka Architektur - Single Broker Cluster

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.5 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 [PN11]. 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. [Fou14i, 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* abschließend nicht gelöscht. Durch das einstellbare Service Level Agreement (SLA) ist es möglich Nachrichten erst nach einer definierten Zeitspanne zu löschen. In einem *Consumer*-Ausfall ist es durch diese Technik 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 umgehen muss, muss eine Logik zur Erkennung von Duplikaten bereitstellen. So wird beim Einsatz von mehreren *Consumern* die Datenkapazität vervielfacht. Nachrichten sind 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.

Kriterium	Bewertung
Architektur	Strukturierte Peer-to-Peer-Architektur
Prozesse und Threads	Client-Server Cluster, Consumer 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

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

¹Kompressionswerkzeug: gzip – <http://www.gzip.org/>

²Kompressionsbibliothek: snappy – <https://code.google.com/p/snappy/>

PacificA [LYZZ08] aus den *Followern* der nächste führende ISR bestimmt. [Fou14i, 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 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 Plugin³ erfolgen.

In diesem Kapitel wurde die Installation und ein Beispielanwendung für den Austausch von Nachrichten gezeigt. Auf spezielle Eigenschaften wie das zero-copy [PN11], 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. Am Anfang wird in einer Tabelle eine Kurzübersicht gegeben. Anschließend werden die Bewertungskriterien zu Apache Flume erläutert.

Faktum	Beschreibung
Hauptentwickler	Prasad Mujumdar, Arvind Prabhakar und Eric Sammer
Stabile Version	1.5.0.1 vom 16.06.2014
Entwicklungsstatus	Aktiv
Entwicklungsversion	1.6.0
Sprache	Java
Betriebssystem	Plattformübergreifend (Microsoft Windows mit Cygwin Umgebung)
Lizenz	Apache License version 2.0
Webseite	[PMS14]
Quelltext	[PMS]

Tabelle 4.5: Kurzübersicht Apache Flume

4.4 Apache S4

4.5 Zusammenfassung

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

Entwurf

Kapitel 5

Anwendungsfall und Prototyp

Entwurf

Entwurf

Kapitel 6

Auswertung

6.1 Benchmark Ergebnisse

6.2 Erkenntnis

Entwurf

Entwurf

Kapitel 7

Schlussbetrachtung

7.1 Zusammenfassung

7.2 Einschränkungen

7.3 Ausblick

Entwurf

Entwurf

Kapitel 8

Verzeichnisse

Entwurf

Entwurf

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.
- [Cap14] CAPITAL, KPMG: *Going beyond 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, 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.
- [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.
- [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.
- [PN11] PALANIAPPAN, SATHISH K. und PRAMOD B. NAGARAJA: *Efficient data transfer through zero copy*, September 2011.
- [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.
-

Internetquellen

- [Con14] CONTRIBUTOR, 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.
- [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 Hadoop* URL: <http://hadoop.apache.org/>, Juni 2014. Abgerufen am 05.06.2014.
- [Fou14b] FOUNDATION, APACHE SOFTWARE: *Apache Software Foundation* URL: <http://www.apache.org/foundation/>, Juni 2014. Abgerufen am 22.06.2014.
- [Fou14c] FOUNDATION, APACHE SOFTWARE: *Apache ZooKeeper* URL: <http://zookeeper.apache.org/>, Juni 2014. Abgerufen am 22.06.2014.
- [Fou14d] FOUNDATION, APACHE SOFTWARE: *Class SpoutOutputCollector* URL: <https://storm.incubator.apache.org/apidocs/backtype/storm/spout/SpoutOutputCollector.html>, Juni 2014. Abgerufen am 22.06.2014.
- [Fou14e] FOUNDATION, APACHE SOFTWARE: *Class TopologyBuilder* URL: <https://storm.incubator.apache.org/apidocs/backtype/storm/topology/TopologyBuilder.html>, Juni 2014. Abgerufen am 22.06.2014.
- [Fou14f] FOUNDATION, APACHE SOFTWARE: *Interface IBolt* URL: <https://storm.incubator.apache.org/apidocs/backtype/storm/task/IBolt.html>, Juni 2014. Abgerufen am 22.06.2014.
-

- [Fou14g] 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.
- [Fou14h] FOUNDATION, APACHE SOFTWARE: *Interface ISpout URL: <https://storm.incubator.apache.org/apidocs/backtype/storm/spout/ISpout.html>*, Juni 2014. Abgerufen am 22.06.2014.
- [Fou14i] FOUNDATION, APACHE SOFTWARE: *Kafka 0.8.1 Documentation URL: <https://kafka.apache.org/documentation.html>*, Juli 2014. Abgerufen am 07.07.2014.
- [Fou14j] FOUNDATION, APACHE SOFTWARE: *Storm Fault Tolerance URL: <https://storm.incubator.apache.org/documentation/Fault-tolerance.html>*, Juni 2014. Abgerufen am 29.06.2014.
- [Fou14k] FOUNDATION, APACHE SOFTWARE: *Storm Fully Processed URL: <https://storm.incubator.apache.org/documentation/Guaranteeing-message-processing.html>*, Juni 2014. Abgerufen am 29.06.2014.
- [Fou14l] FOUNDATION, APACHE SOFTWARE: *Storm Project Incubation Status Page URL: <http://incubator.apache.org/projects/storm.html>*, Juni 2014. Abgerufen am 22.06.2014.
- [Fou14m] FOUNDATION, APACHE SOFTWARE: *Storm Security URL: <https://github.com/apache/incubator-storm/blob/master/SECURITY.md>*, Juni 2014. Abgerufen am 29.06.2014.
- [Fou14n] FOUNDATION, APACHE SOFTWARE: *Storm Trident URL: <https://storm.incubator.apache.org/documentation/Trident-API-Overview.html>*, Juni 2014. Abgerufen am 29.06.2014.
- [Fou14o] 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⁺13] GOPALAKRISHNA, KISHORE, FLAVIO JUNQUEIRA, MATTHIEU MOREL, LEO NEUMEYER, BRUCE ROBBINS und DANIEL GOMEZ FERRO: *S4 distributed stream computing platform. URL: <http://incubator.apache.org/s4/>*, Juni 2013. Abgerufen am 30.06.2013.
- [Goo14] GOOGLE: *Google Search URL: <https://www.google.de/>*, Mai 2014. Abgerufen am 12.05.2014.
- [Hic14] HICKEY, RICH: *Clojure Rationale URL: <http://clojure.org/rationale>*, Juni 2014. Abgerufen am 22.06.2014.
- [iC14] CORPORATION IMATIX: *Zero MQ URL: <http://zgguide.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.
- [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.
- [PMS] PRABHAKAR, ARVIND, PRASAD MUJUMDAR und ERIC SAMMER.
- [PMS14] PRABHAKAR, ARVIND, PRASAD MUJUMDAR und ERIC SAMMER: *Apache Flume distributed, reliable, and available service for efficiently operating large amounts of log data*. URL: <http://flume.apache.org/>, Juli 2014. Abgerufen am 14.07.2014.
- [Rao11] RAO, JUN: *Kafka Proposal* URL: <https://wiki.apache.org/incubator/KafkaProposal>, Juni 2011. Abgerufen am 30.06.2014.
-

Entwurf

Abbildungsverzeichnis

2.1	Exemplarische Darstellung eines Basis Modells für Streaming frameworks . . .	5
2.2	Darstellung eines azyklisch gerichteten Graphen	6
2.3	Stream Processing Engine	7
3.1	Darstellung Big Data Cube	10
3.2	Aurora Borealis mit einem Master zwei Servern und einem Konsumenten . . .	14
4.1	Apache Storm Gruppierungen	19
4.2	Apache Kafka Architektur - Single Broker Cluster	23

Entwurf

Tabellenverzeichnis

3.1	Bewertung Referenzmodell Aurora Borealis	13
4.1	Kurzübersicht Apache Storm	18
4.2	Bewertung Apache Storm	21
4.3	Kurzübersicht Apache Kafka	22
4.4	Bewertung Apache Kafka	24
4.5	Kurzübersicht Apache Flume	25

Entwurf

Entwurf

Listings

A.1	Aurora Borealis Konfiguration für die Verteilung	48
A.2	Aurora Borealis Konfiguration für die Abfragen	48
A.3	Aurora Borealis Testanwendung	49
A.4	Apache Storm WordCount Demo	51
A.5	Apache Kafka Producer Beispiel	53
A.6	Apache Kafka Consumer Beispiel	54

Entwurf

Entwurf

Anhang A

Zusätze

A.1 Abkürzungen

ACL	Access Control List. 21
ANTLR	ANother Tool for Language Recognition. 58
ASF	Apache Software Foundation. 17
BASE	Basically Available, Soft State, Eventually Consistent. 10
C++	C objektorientierte Programmiersprache. 14
CAP	Consistency Availability Partition Tolerance. 10
CP	Connection Point. 7
CPU	Central Processing Unit. 15
ESP	Event Stream Processing. 4
fk	foreign key. 9
IP	Internet Protocol. 21
IPsec	Internet Protocol Security. 21
ISO	International Organization for Standardization. 57
ISR	In-sync Replica. 24, 25
JMX	Java Management Extensions. 25
k	key. 9

MPEG	Motion Pictures Expert Group. 4
NMSTL	Networking, Messaging, Servers, and Threading Library. 14, 58
ORM	Object-relational mapping. 9
OSI	Open Systems Interconnection Model. 3, 4, 12
pk	primary key. 9
QoS	Quality of Service. 6, 15, 22
RPC	Remote Procedure Call. 4, 5, 12, 13
SBT	Scala Build Tool. 61
SLA	Service Level Agreement. 24
SPE	Stream Processing Engine. 5, 7, 41
TCP	Transmission Control Protocol. 3, 4
v	value. 9
VM	Vector of Metrics. 6, 8
XML	Extensible Markup Language. 13, 14

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" />

    <parameter name="window-size-by"        value="VALUES" />
    <parameter name="window-size"          value="1" />
    <parameter name="advance"               value="1" />
    <parameter name="order-by"              value="FIELD" />
    <parameter name="order-on-field"        value="time" />
  </box>

  <box name="myfilter" type="filter">
    <in stream="AggregatePreFilter" />
    <out stream="Aggregate" />
    <parameter name="expression.0" value="(time % 2) == 0"/>
  </box>

</borealis>

```

Listing A.3: Aurora Borealis Testanwendung

```

#include "args.h"
#include "MytestdistMarshal.h"

using namespace Borealis;

const uint32 SLEEP_TIME = 100;           // Delay between injections.
const uint32 BATCH_SIZE = 20;           // Number of input tuples per
    batch.
const uint32 PROTOCOL_SIZE = 4;         // Number of elements in
    PROTOCOL.

const string PROTOCOL[] = { string( "dns" ), string( "smtp" ),
                           string( "http" ), string( "ssh" )
                           };

const Time time0 = Time::now() - Time::msecs( 100 );

////////////////////////////////////
//
// Print the content of received tuples.

```

```

//
void MytestdistMarshal::receivedAggregate( AggregateTuple *tuple )
{
//.....

    NOTICE << "For time interval starting at "
              << tuple->time << " tuple count was " << tuple->count;

    return;
}

/////////////////////////////////////////////////////////////////
//
// Return here after sending a packet and a delay.
//
void MytestdistMarshal::sentPacket()
{
    int32      random_index;
    int32      timestamp;
    Time       current_time;
    //.....

    current_time = Time::now();

    timestamp = (int32)( current_time - time0 ).to_secs();
    if ( timestamp < 0 ) timestamp = 0;
    //DEBUG << "timestamp = " << timestamp << " current_time = " <<
        current_time;

    for ( uint32 i = 0; i < BATCH_SIZE; i++ )
    {
        random_index = rand() % PROTOCOL_SIZE;

        // This has to be in the loop scope so the constructor is rerun.
        Packet tuple;

        tuple._data.time = timestamp;
        setStringField( PROTOCOL[ random_index ], tuple._data.protocol,
            4 );

        // DEBUG << "time=" << tuple._data.time << " proto=" << tuple.
            _data.protocol;
        batchPacket( &tuple );
    }

    // Send a batch of tuples and delay.
    //
    //DEBUG << "call sendPacket...";
    sendPacket( SLEEP_TIME );

    return;
}

/////////////////////////////////////////////////////////////////
//
int main( int argc, const char *argv[] )

```



```

{
    MytestdistMarshal    marshal;           // Client and I/O stream state.
    //
    // Maximum size of buffer with data awaiting transmission to Borealis
    //.....

    // Run the front-end, open a client, subscribe to outputs and inputs
    //
    // In this edited version, open will print a message and quit if an
    // error occurs.
    marshal.open();

    DEBUG    << "time0 = " << time0;

    // Send the first batch of tuples. Queue up the next round with a
    // delay.
    marshal.sendPacket();

    DEBUG    << "run the client event loop...";
    // Run the client event loop. Return only on an exception.
    marshal.runClient();
}

```

Listing A.4: Apache Storm WordCount Demo

```

/**
 * 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 storm.starter;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.task.ShellBolt;
import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.IRichBolt;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import storm.starter.spout.RandomSentenceSpout;

import java.util.HashMap;

```

```

import java.util.Map;

/**
 * This topology demonstrates Storm's stream groupings and multilang
 * capabilities.
 */
public class WordCountTopology {
    public static class SplitSentence extends ShellBolt implements
        IRichBolt {

        public SplitSentence() {
            super("python", "splitsentence.py");
        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }

        @Override
        public Map<String, Object> getComponentConfiguration() {
            return null;
        }
    }

    public static class WordCount extends BaseBasicBolt {
        Map<String, Integer> counts = new HashMap<String, Integer>();

        @Override
        public void execute(Tuple tuple, BasicOutputCollector collector) {
            String word = tuple.getString(0);
            Integer count = counts.get(word);
            if (count == null)
                count = 0;
            count++;
            counts.put(word, count);
            collector.emit(new Values(word, count));
        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word", "count"));
        }
    }

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("spout", new RandomSentenceSpout(), 5);

        builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("
            spout");
        builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split"
            , new Fields("word"));

        Config conf = new Config();
        conf.setDebug(true);

        if (args != null && args.length > 0) {
            conf.setNumWorkers(3);
        }
    }
}

```

```

        StormSubmitter.submitTopologyWithProgressBar(args[0], conf,
            builder.createTopology());
    }
    else {
        conf.setMaxTaskParallelism(3);

        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("word-count", conf, builder.createTopology
            ());

        Thread.sleep(10000);

        cluster.shutdown();
    }
}
}
}

```

Listing A.5: Apache Kafka Producer 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 kafka.examples;

import java.util.Properties;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class Producer extends Thread
{
    private final kafka.javaapi.producer.Producer<Integer, String>
        producer;
    private final String topic;
    private final Properties props = new Properties();

    public Producer(String topic)
    {
        props.put("serializer.class", "kafka.serializer.StringEncoder");
        props.put("metadata.broker.list", "localhost:9092");
        // Use random partitioner. Don't need the key type. Just set it to
        Integer.
        // The message is of type String.
        producer = new kafka.javaapi.producer.Producer<Integer, String>(new
            ProducerConfig(props));
        this.topic = topic;
    }
}

```

```

public void run() {
    int messageNo = 1;
    while(true)
    {
        String messageStr = new String("Message_" + messageNo);
        producer.send(new KeyedMessage<Integer, String>(topic, messageStr)
            );
        messageNo++;
    }
}
}

```

Listing A.6: Apache Kafka Consumer 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 kafka.examples;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

public class Consumer extends Thread
{
    private final ConsumerConnector consumer;
    private final String topic;

    public Consumer(String topic)
    {
        consumer = kafka.consumer.Consumer.createJavaConsumerConnector(
            createConsumerConfig());
        this.topic = topic;
    }

    private static ConsumerConfig createConsumerConfig()
    {
        Properties props = new Properties();
    }

```

```
props.put("zookeeper.connect", KafkaProperties.zkConnect);
props.put("group.id", KafkaProperties.groupId);
props.put("zookeeper.session.timeout.ms", "400");
props.put("zookeeper.sync.time.ms", "200");
props.put("auto.commit.interval.ms", "1000");

return new ConsumerConfig(props);
}

public void run() {
    Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
    topicCountMap.put(topic, new Integer(1));
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
        consumer.createMessageStreams(topicCountMap);
    KafkaStream<byte[], byte[]> stream = consumerMap.get(topic).get(0);
    ConsumerIterator<byte[], byte[]> it = stream.iterator();
    while(it.hasNext())
        System.out.println(new String(it.next().message()));
}
}
```

A.3 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 Quelltextfragmenten und Kommandozeilenausschnitten schrittweise die Eingabe und Ausgabe gezeigt.

A.3.1 Voraussetzungen am Betriebssystem

Die Installation von Aurora Borealis benötigt ein auf linuxbasiertes Betriebssystem. 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.3.2 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.3.3 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.3.4 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 INSTALLANTLR='/usr'
export INSTALL_XERCESE='/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.3.5 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.3.6 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.3.7 Erzeugen von Borealis

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

```
bbb
bbbt
retool
make install
```

A.3.8 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.4 Installationsanleitung Apache Storm

In diesem Kapitel wird die Installation von Apache Storm in kleinen Unterkapiteln vorgestellt. Die Anleitung setzt ein Wissen in der Verwendung, Administration und Erzeugen von Anwendungen unter dem Betriebssystem 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.4.1 Voraussetzungen am Betriebssystem

Als Betriebssystem wird in dieser Anleitung Linux mit der Distribution Ubuntu 13.10 verwendet. Apple Mac OS X und Microsoft Windows mit einer Cygwin-Umgebung werden in dieser Anleitung nicht betrachtet. Unter Ubuntu wird für die Installation von Paketen das Kommandozeilen-Werkzeug *aptitude* eingesetzt.

A.4.2 Paketabhängigkeiten

Einige Pakete benötigt Apache Storm zur Laufzeit bzw. werden zum Erzeugen gebraucht. Die folgende Liste stellt die notwendigen Pakete dar:

- zookeeper
- libtool
- autoconf
- automake
- openjdk-7-jdk
- zeromq (evt. nicht im Linux Repository)
- jzmq (evt. nicht im Linux Repository)

Falls die Pakete zu ZeroMQ in den Paketquellen fehlen, müssen beide manuell erzeugt und dem Betriebssystem hinzugefügt werden. Folgende Kommandozeilen zeigen das Herunterladen, Erzeugen und Bereitstellen von ZeroMQ schrittweise:

```
wget download.zeromq.org/zeromq-3.2.4.tar.gz
tar xvfz zeromq-3.2.4.tar.gz
./configure
make
make install
ldconfig
```

Apache Storm benutzt zur Kommunikation mit ZeroMQ Java-Bindings. Folgende Kommandozeilenaufrufe müssen zur Installation schrittweise ausgeführt werden:

```
wget github.com/zeromq/jzmq/archive/master.zip
./autogen.sh
./configure
make
```

A.4.3 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-src.zip  
unzip apache-storm-0.9.1-incubating-src.zip  
ln -s storm apache-storm-0.9.1-incubating-src.zip
```

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

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

A.4.4 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.4.5 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.5 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, Mac OS X und Microsoft Windows wird in dieser Arbeit nicht behandelt. Bevor die Installation beginnen kann, werden zuerst Voraussetzungen 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.5.1 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 Maven¹ 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.

¹Maven: Java Build Werkzeug <http://maven.apache.org/>

```
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.

A.5.2 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.5.3 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
```
