Hochverfügbare, performante und skalierbare Web-Anwendungen

Cassandra ist eine hochverfügbare und skalierbare Datenbank, die aufgrund dieser architektonischen Eigenschaften gerne im Big-Data-Umfeld eingesetzt wird. Zusammen mit einem Applikationsframework können diese Eigenschaften auf unterschiedliche Anwendungen übertragen werden. Das Ergebnis sind leicht zu betreibende Anwendungen, wie sie in der Geschäftswelt des 21. Jahrhunderts immer häufiger vorkommen werden.

Einführung

Das Projekt geht zurück auf eine Forschungsarbeit zu hoch-performanten Shared-Nothing-Software-Architekturen. Diese wurde später als spezifische Software weiterentwickelt. Der innere, generische Kern dieser Software wurde anschließend als Framework extrahiert und fokussiert vor allem die Persistenz der Daten. Dieses Framework wurde in einer Software als Grundlage für ein strategisches Expertensystem eingesetzt. Dieses System soll einen zentralen, derzeit noch manuellen, Entscheidungsprozess deutlich vor allem inhaltlich verbessern aber auch organisatorisch beschleunigen.

Der Name des Frameworks ist R8. Es ist ein Framework für Kunden-Systeme, die besondere Anforderungen an Hochverfügbarkeit, Performance und Skalierbarkeit haben. Hochverfügbarkeit (HA) beschreibt die Eigenschaft, dass ein System in jedem Fall antwortet. Dabei können beliebige Knoten im Cluster ausfallen und deren Arbeit wird, nach außen nahtlos, von den verbleibenden Knoten übernommen.

Herausforderung in typischen JEE-Architekturen

In der Regel greifen die Benutzer eines Systems über einen Load Balancer auf einen Cluster von Applikationsservern zu; diese wiederum greifen über einen Cluster von Service-Layer-Knoten auf eine einzelne oder eine verteilte Datenbank zu. In diesem Fall bezeichnen „Cluster" jeweils jeden möglichen Fall – der „triviale Cluster" mit lediglich einem Knoten ist hierbei ebenfalls gemeint. Heutige Anwendungen bestehen damit im Allgemeinen aus vier Schichten: einer Client-, einer Applikationsserver-, einem optionalen Service- und Persistenz-Layer.[i] Von oben nach unten in absteigender technischer Wertschöpfungstiefe.

Der Service-Layer kann hierbei auch im technischen Deployment der Applikation enthalten sein. Wenn das nicht der Fall ist, der Service Layer also auf eigener Hardware läuft, entsteht bei jedem Abruf von Daten eine zusätzliche Netzwerklatenz. In Mainframe-Architekturen kann diese Latenz jedoch entfallen, da der Service-Layer hierbei etwa bei IBM aus Cobol-Programmen besteht. Diese Programme werden anstelle der reinen Datenbank angesprochen und erledigen dann stellvertretend den Zugriff auf die Datenbank. Da sie auf derselben Hardware laufen, auf der auch die Daten liegen, entsteht keine zusätzliche Netzwerklatenz.

In Java-Programmen kann der Service-Layer auch als Dependency im finalen Deployment enthalten sein. Das hat den Vorteil, dass jeder Entwickler einen geeigneten Service-Layer-Stand ansprechen kann ohne seine Kollegen zu beeinflussen. Ein eigenständiger Service-Layer erzeugt diese oben genannte zusätzliche Latenz – unabhängig davon ob er auf derselben oder einer anderen Hardware läuft. In ersterem Fall ist dieser Overhead allerdings deutlich geringer als bei einer separaten Maschine. Ein integrierter Service-Layer – ob in der Datenbank oder in der Applikation – tut das nur in kaum messbarer Weise. Der Nachteil an einem externen Service-Layer ist neben der erhöhten Latenz auch das Manifestieren eines bestimmten Softwarestandes für alle Entwickler. Ältere und neuere Client-seitige Softwarestände als der Service-Layer können im Allgemeinen nicht auf diesen Dienst zugreifen.

Drei-Schichten-Architektur

Die Grobarchitektur in R8 fußt auf einem integrierten Service-Layer. Sie besteht aus einer Drei-Schichten-Architektur. Da der Service-Layer im Deployment enthalten ist entsteht keine zusätzliche Latenz und die Entwickler können jeweils auf beliebig neuen und alten Software-Ständen arbeiten. Die Persistenz-Schicht in R8 greift daher direkt auf eine Cassandra-Datenbank zu. Sie wird auch „Cassandra-Ring" oder „Cassandra-Cluster" genannt. Die beiden letzteren Begriffe sind im Falle von Cassandra synonym für die Hardware-seitige Datenbank. Das geschieht über den DataStax-Binärtreiber mittels CQL (Cassandra Query Language; ein SQL-Dialekt für Cassandra). Cassandra und der dazugehörige Binärtreiber sind frei und kommerziell verfügbar. Beide unterstützen das Speichern der Daten in R8.

Der Grund für die Spezialisierung auf Cassandra war unter anderem die besonders hohe Schreibgeschwindigkeit.[ii] Cassandra bietet sich, durch ihre Fähigkeit sehr schnell Daten zu schreiben und zu ändern, sogar als externer Cache an. Die kommerzielle Version von Cassandra, DSE Edition (DataStax Enterprise), ist hierfür am besten geeignet, da sie auch reine In-Memory-Tabellen unterstützt. [iii] [iv] Für offene Cassandra-Distributionen ist die Nutzung als externer Cache ein Antipattern [v] [vi], da die Daten zwangsweise früher oder später auf der Festplatte persistiert werden müssen. Offene Versionen von Cassandra sind beispielsweise Apache Cassandra, auch Vanilla-Cassandra genannt, oder DSC Edition (DataStax Community).[vii] Die besondere Fähigkeit, schneller Schreiben als Lesen zu können, kommt allen Anwendungsfällen mit hoher Schreiblast entgegen. Zudem kann Cassandra vor allem durch Table Strategies, Consistency Levels und diverse Caches auch für das Lesen optimiert werden.

Die seltene Möglichkeit, Daten automatisch nach dem Schreiben zu verwerfen macht, Cassandra zudem zu einem präferierten Kandidaten für das Speichern von Websessions. Damit werden „heiße Sessions" im Cassandra-Ring vorgehalten und – sobald sie durch das Ablaufen der Session „erkalten" – automatisch gelöscht. Zudem ist Cassandra hochverfügbar und skalierbar: sowohl horizontal als auch vertikal. Das macht sie zu einer idealen Datengrundlage für ebenso hochverfügbare, skalierbare Anwendungen.

Das Ziel von R8 ist es diese Eigenschaften von der Cassandra-Datenbank in die Welt der Anwendungen zu bringen: einzelne Knoten können den Ring verlassen und beitreten ohne den Betrieb der Anwendung zu stören. Beim Ausfall eines Cassandra-Knotens übernehmen die verbleibenden Knoten im Ring seine Daten und Arbeitslast. Beim Beitritt eines neuen oder vorher ausgefallenen Knotens entlastet dieser seine Pendants.

Hochverfügbare Anwendungen

Die Notwendigkeit von HA-Software kommt einerseits aus Anwendungsfällen, in denen eine Anwendung 24/7 für Nutzer auf der ganzen Welt verfügbar sein soll. Ein Beispiel sind Logistikportale für weltweit tätige Konzerne mit ihren jeweils lokalen Zulieferern. Der zweite Grund, warum Anwendungen sicher gegen Ausfälle einzelner Knoten sein sollen, ist das Thema Continuous Delivery. Hierbei geht es darum Software in kleinen Schritten zu verfeinern und so oft wie möglich neu zur Verfügung zu stellen. Die Iterationsschleifen werden dadurch besonders kurz; dafür wird die Anwendung sehr oft neu zur Verfügung gestellt. Jedes dieser Deployments kann in einfachen Architekturen der Grund dafür sein, dass die Anwendung offline geht.

Das ist auch der Grund, warum Unternehmen lediglich quartalsweise neue Software ausrollen. Auf der anderen Seite können dadurch Fehler nur mit großem, zeitlichem Vorlauf beseitigt werden und Nutzer warten länger auf neue Features. Bei weltweit durchgehend genutzten Systemen gibt es keinen Zeitpunkt, an dem ein neues Deployment ohne Probleme möglich wäre. Daher werden nur sehr selten neue Stände ausgerollt: zu dem Zeitpunkt, an dem es die wenigsten Anwender stört. Allerdings ist diese Vorgehensweise das Gegenteil von Continuous Delivery. Der Grund für diese Vorgehensweise liegt in der Software-Architektur. Genauer gesagt vor allem im Session-Management – wie für Web-Anwendungen und beispielsweise Datenbanksessions.

Bei Continuous Delivery entscheidet das Business über die IT und nicht umgekehrt. Folglich dürfen technische Änderungen den Betrieb und das Testen von neuen Ständen nicht beeinträchtigen. Damit vergeht für den Anwender deutlich weniger Zeit bis er neue Funktionalitäten nutzen kann und Fehler beseitigt sind. Die Entwickler der Anwendung können in diesem Szenario Performance-Optimierungen, Parameter-Tuning und den Tausch von Hardware nur auf Test-Systemen vollziehen. Hilfreich wäre es allerdings, das auch im Produktivsystem erledigen zu können. Vor allem Verbesserungen von Laufzeiten und Parametern können einen großen Einfluss auf die Systemperformance haben. Diese Änderungen sind jedoch in der Regel nicht im laufenden Betrieb möglich.

Derzeit geschieht das Ausrollen einer neuen Version einer Anwendung normalerweise beinahe gleichzeitig für alle Applikationsserver. Jeder Server wird zu jeder Zeit mit derselben Version der Software betrieben. Wünschenswert wäre allerdings ein Rolling-Updates-Szenario wie bei Android-Updates: hier wird eine neue Software-Version zuerst auf einer kleinen Testmenge von Smartphones ausprobiert. Erst wenn sie weitestgehend fehlerfrei läuft und die größten Bugs und Anpassungen für jedes Gerät vollzogen sind wird die bewährte Version auf der großen Mehrheit der Geräte installiert.[viii]

Die Lösung ist eine Architektur, die so hochverfügbar ist, wie der Cassandra-Cluster selbst. Damit sind zentrale Anpassungen an einem Knoten möglich ohne das Gesamtsystem zu stören – selbst wenn dieser Knoten dabei ausfällt. Es muss jeder Knoten unabhängig von anderen ausfallen können ohne das gesamte System zu beeinträchtigen. Wenn nach außen Ausfälle im Ring verborgen werden sind auf allen Knoten rollende Updates möglich, wie bei Android. Zudem sind mit dieser Architektur Performance-Optimierungen im Produktivsystem möglich. Da jeder Knoten unabhängig ausfallen darf kann seine Java Virtual Machine zum Optimieren der Parameter neugestartet beziehungsweise auch der Ausfall dieses Teils des Systems riskiert werden. In herkömmlichen Architekturen ist genau das nicht möglich. Netflix optimiert bereits nach dieser Herangehensweise seine Produktivcluster: sie schießen gezielt Knoten ab („Chaos Monkey“) oder verlangsamen gezielt deren Antwort („Latency Monkey“).[ix]

Data-driven Business

R8 ist ein Framework für ein Daten-getriebenes Geschäft: es erzeugt im Hintergrund automatisch zu jeder Nutzerinteraktion Daten. Das hilft die Nutzer des Systems noch besser zu verstehen und das System gezielt zu verbessern. Vor allem interessant sind Metriken wie lange Service-Level-Agreements noch eingehalten werden können, wo es Performance-Probleme gibt, wo größere Caches sinnvoll wären und auch wo Speicher eingespart werden kann. Da R8 eine Cassandra-Datenbank nutzt hat dieser Cassandra-Ring keine Probleme mit der Schreiblast und kann bei Bedarf nahezu linear skalieren. Zudem sorgt der Column-basierte Speicher dafür, dass die Daten bestmöglich komprimiert werden.

Darüber hinaus werden gezielt Cassandra-Features, wie Batches und Asynchrone Requests, genutzt. Batch-Statements sichern die Cache-Statistiken, wie belegter Cache-Anteil, maximale Cache-Größe, Trefferquoten („hit ratios“) und „miss penalties“. Diese Werte werden zentral gesammelt und alle Statements in einem Batch geschrieben. Das gesammelte Schreiben der Daten ist effizienter und stellt sicher, dass alle Daten den exakt selben Zeitstempel tragen.

Asynchrone Requests wiederum werden für Log-Einträge genutzt. Logs in R8 erfolgen nur in die Datenbank und nicht in Logfiles. Per default werden diese Anfragen asynchron abgesetzt – also fire-and-forget. Das macht die Applikation nachweislich schneller, da sie nicht auf die Bestätigung von der Datenbank warten muss. Allein das Schreiben von Logs in die Datenbank mittels Asynchroner Requests trägt für den Nutzer deutlich spürbar positiv und messbar zur Performance-Optimierung bei. Die so erzielte Beschleunigung ist in unserem Produktivsystem um den Faktor drei besser als Synchrone Requests. Per default erfolgt das Schreiben dieser Log-Einträge auf dem Consistency Level ANY[x]: hierbei muss der Schreibvorgang nur auf einem aktiven oder passiven („drain“) Knoten erfolgen um erfolgreich zu sein. Ein sogenannter hinted handoff (Vermerk für das spätere Schreiben auf einem aktiven Knoten) genügt hierbei.[xi] Änderungen des Default-Verhaltens sind einfach konfigurierbar.

Vor allem aus diesen Gründen ist Cassandra bei datengetriebenen Unternehmen, wie Netflix und Spotify, so beliebt. Diese beiden Unternehmen sind führende Nutzer von Cassandra und werden im Folgenden weiter beleuchtet. R8 nutzt diese Eigenschaften gezielt um jegliche Nutzerinteraktion festzuhalten und die eigenen Systemzustände zu protokollieren. Cassandra bietet sich hierbei unter anderem durch ihre Integrationsfähigkeit in Apache Hadoop, als NoSQL-Speicher für Spark und der Analyse mittels R (statistische Programmiersprache) über das RCassandra-Projekt an. R8 bietet

damit die Möglichkeit das Optimum aus den bestehenden Daten herauszuholen – egal in welchem Umfang diese anfallen.

Das Framework kümmert sich des Weiteren im Hintergrund um die nötigen DDL-Statements (Data Definition Language) und legt damit Tabellen mit Spalten, Partition Keys, Clustering Columns und Secondary Indices, aber auch Prepared Statements an. Letztere werden für alle lesenden und schreibenden Datenbankzugriffe genutzt – können in besonderen Ausnahmen allerdings auch umgangen werden. Durch das Nutzen von Prepared Statements alleine wird die Laufzeit des Statements ungefähr verdoppelt und SQL-Injections werden deutlich erschwert.[xii] Durch das automatische Erzeugen der Datenbank-Strukturen („Keyspace“) eignet sich R8 auch für Entwickler, die sich nicht mit Cassandra-Interna auskennen. Sie definieren ein denormalisiertes, abstraktes Schema und R8 legt die optimalen Strukturen inklusive der Datenbank-seitigen Caches, Datenkomprimierungen und Table Strategies automatisch an.

Auf der anderen Seite kann R8 auch Full-Table-Scans erzwingen. Diese sind durch die verteilte Natur von Cassandra besonders teuer. Sie können aber eine willkommene Alternative zum Anlegen einer weiteren Tabelle sein, da Cassandra ausschließlich denormalisierte Tabellen unterstützt. Sollte also die Laufzeit für eine Query nicht wichtig sein, wie in einigen analytischen Statements, so kann R8 ein Statement für Cassandra erzeugen, dass zu einem Full-Table-Scan führt – mit dem Keyword „allow filtering“. Full-Table-Scans sind in vielen Datenbanken die teuerste Art des lesenden Datenzugriffs. Da sie in Cassandra besonders teuer sind führt sie diese Statements nur auf ausdrücklichen Wunsch aus.

In anderen Datenbanken bedarf es in der Regel keiner speziellen Parameter für das Erzwingen dieser Statements. Das hat den Nachteil, dass Full-Table-Scans teilweise ständig im Hintergrund laufen und das gesamte System ausbremsen.[xiii] Um das zu verhindern bieten sich regelmäßige Performance-Reviews in allen Datenbank-Systemen an.

Architektonische Verbesserungen

Für das gesamte System ist der Einsatz von Cassandra auch vorteilhaft, da die Architekten keine eigene Scale-Out-Strategie festlegen müssen. Eine Cassandra lässt sich in jedem Fall fast-linear erweitern. Zudem trägt sie entscheidend zu dem Ziel der Zero-Downtime bei. Durch sinkende Kosten für Rechenzeit in der Cloud und das Ausbreiten von Bot-Netzen gehören DoS- und DDoS-Attacken (Denial of Service) nicht nur für große Unternehmen zum Alltag.

Studien zufolge wird jedes zweite Unternehmen monatlich angegriffen. Diese Unternehmen zahlen jedes Jahr zwischen sechs und zwölf Millionen Euro [xiv] [xv] [xvi] [xvii] [xviii] [xix] alleine für das Beseitigen der direkten Probleme aus solchen Attacken. Investitionen zur Prävention fallen zusätzlich an.[xx] (Verteilte) DoS-Attacken sind allerdings auch auf dem Vormarsch in kleinen und mittelständischen Unternehmen. Da Zulieferer häufig weniger gut abgesichert sind bieten sich Angriffe hier besonders an.[xxi] Für Banken und informationszentrierte Unternehmen können (verteilte) DoS-Attacken auch deutlich teurer als einstellige Millionenbeträge ausfallen. Einige der Hauptprobleme sind der Personaleinsatz, die Imageschäden und möglicherweise das Erliegen des täglichen Geschäfts.[xxii] Mit einer hochverfügbaren Architektur ist es für Angreifer deutlich schwieriger einzelne Knoten geschweige denn das gesamte System zu schädigen.

Wenn beliebte Webdienste, wie Soziale Netzwerke, ausfallen macht diese Neuigkeit auf den verbleibenden Kanälen sofort die Runde, da das Internet als 24/7 verfügbar angesehen wird.[xxiii] Diese Imageschäden[xxiv] durch Berichte in der Presse und Sozialen Netzen führen dazu, dass Hochverfügbarkeit ein strategisch wichtiges Ziel in vielen Unternehmen wird. R8 bietet hierfür eine solide Grundlage, um Enterprise-Anwendungen, Webseiten und Service-Layer zu erstellen. Das gesamte System ist dadurch deutlich robuster gegen erhöhte Last und Angriffe. Die genutzte Web-Engine in R8 ist Apache Wicket. Wenn Entwickler lieber JSP/JSF oder ein anderes UI-Framework nutzen kann R8 auch als reines Backend fungieren.

Verhaltensbasierte Firewall

Darüber hinaus sichert sich R8 gegen (verteilte) DoS-Angriffe mittels einer eigenen Firewall ab. Diese Firewall, bezeichnet als Fort Knox[xxv], ist Teil des Deployments und erweitert statische Hardware-seitige Firewalls und andere, wie Apache ModSecurity.

Fort Knox ist eine verhaltensbasierte Firewall, die Nutzer zeitweise blockiert, wenn diese bösartige Nutzungsmuster zeigen. Diese Muster werden aufgrund der Zugriffe in typischerweise sehr kurzen – circa sekundenweisen – Intervallen mit historischen Nutzungen abgeglichen. Die Intervalle für einen solchen Abgleich sind einfach konfigurierbar. Dabei werden kurzzeitig höhere Lasten, wie durch noch leere Caches beziehungsweise gerade invalidierte Caches, berücksichtigt. In beiden Fällen sieht Fort Knox den sprunghaften Anstieg der Requests als ungefährlich an.

Werden hingegen Angriffe gestartet, die nicht kurzer Natur sind, so blockiert die Firewall unverzüglich das Verarbeiten von weiteren Requests von diesem Computer, um die Applikation gegen Angriffe zu schützen.

Session-Management

Es existieren im Allgemeinen drei wichtige Arten von Session-Management: einfache Sessions „nach dem Lehrbuch“, Sticky Sessions und Session-Replikation. Sie haben entscheidenden Einfluss auf die Hochverfügbarkeit des Systems.

Die einfachste Möglichkeit, nach dem Lehrbuch, ist nicht verteilt und damit auch nicht hochverfügbar, da jeder Webserver seine eigene Session aufbaut. Die Sessions werden im Speicher des jeweiligen Servers gehalten und im Zweifel auf die lokale Festplatte ausgelagert.

Sticky Sessions werden oftmals in Verbindung mit verteilten Systemen genutzt. Hier arbeitet jeder Nutzer immer nur auf einem einzelnen Server. Der Load Balancer erkennt anhand der Session-ID welcher das ist und routet alle Requests immer auf diesen Server. Einen Serverausfall verzeiht dieses System jedoch nicht, da die Nutzer ihre Session verlieren, dessen Server offline geht. Das Problem ist zwar eingegrenzt aber noch immer nicht hochverfügbar. Sticky Sessions sind eine gute Möglichkeit eine Applikation horizontal zu skalieren. Sie sind aus Sicht der Performance-Optimierung ein guter Lösungsansatz, da sie die Server ungefähr gleichverteilt belasten. Sticky Sessions sind allerdings weder notwendig noch hinreichend für den Bau einer HA-Architektur. Sie sind allerdings eine Best-Practice für verteilte Architekturen. Auch hier werden die Sessions im Speicher des jeweiligen Servers gehalten und im Zweifel auf die lokale Festplatte ausgelagert.

Session-Replikation beschreibt ein Szenario, in dem Server ihre Websessions sowohl im Speicher halten als auch in eine zentrale Datenbank persistieren. Aus dieser Datenbank können alle Webserver Sessions laden, die sie selbst nicht (mehr) im Speicher haben – beispielsweise bei einem Ausfall des ursprünglichen Servers. Der einspringende Server würde in diesem Fall die bereits erstellte Session anhand der Session-ID aus der zentralen Datenbank laden und im Speicher des neuen Webservers ablegen. Hier ist das Auslagern auf die lokale Festplatte lediglich empfehlenswert.

Es empfiehlt sich auch R8-basierte Applikationen hinter einem Load Balancer mit Sticky Sessions zu betreiben. Denn bei einem Round-Robin-Wechsel vom Client auf immer neue Server je Request führen Lookups in der Regel zu einem Zugriff auf die Datenbank und machen damit Caches weniger effizient.

R8 hat eine eingebaute und besonders effiziente Session-Replikation. Dieses Feature kann auch ohne R8 genutzt werden, um eine hochverfügbare Architektur zu bauen.

Voraussetzung dafür ist, dass der Webserver Session-Replikation unterstützt. Das ist jedoch bei modernen Webservern der Regelfall:

- Tomcat & Jetty (beide jeweils als „Session Clustering“),
- JBoss AS & Wildfly (als „Session Failover“),
- Glassfish (als „Session Persistence“ / „HADB“ in eine Datenbank oder als „In-Memory Replication“ zwischen zwei Instanzen),
- Weblogic (als „Session Replication“),
- WebSphere (als „Distributed Sessions“ in eine Datenbank oder als „Memory-To-Memory“-Lösung),
- TomEE (als „Session Replication“) und
- Resin (als „Distributed Caching“)

Für den Netty wird ein externer Cache wie Infinispan oder Hazelcast benötigt.

CAP-Theorem

Das CAP- beziehungsweise Brewer-Theorem besagt, dass verteilte Systeme nach ihrer Architektur nur zwei von drei möglichen Eigenschaften mit Sicherheit haben können. [xxvi] [xxvii] Diese Eigenschaften sind Hochverfügbarkeit[xxviii] (A für availability), Konsistenz[xxix] (C für consistency) und Partitionssicherheit[xxx] (P für partition tolerance). Sowohl Cassandra als auch R8-Applikationen sind nach dem Brewer-Theorem AP – also hochverfügbar und partitionssicher.[xxxi] Damit ist R8 nach dem CAP-Theorem vergleichbar mit Domain-Name-Servern und das aus folgenden systematischen Gründen:

- Hochverfügbarkeit beschreibt die Eigenschaft, dass ein System stets antwortet und einzelne Knotenausfälle im Hintergrund kaschiert werden.
- Partitionssicherheit bezeichnet die Verteilung der Daten beispielsweise in einer Datenbank. In Cassandra werden verschiedene Zeilen aus einer Tabelle je nach ihrem Partition Key (Hash aus vorderem Teil ihres Primärschlüssels oder Hash aus ganzem Primärschlüssel) auf diverse Knoten verteilt. Somit kann bei einem Knotenausfall ermittelt werden wo die jetzt verlorenen Daten in Kopie vorliegen und wo eine weitere, neue Kopie erstellt wird.
- Die Eigenschaft, die sowohl Cassandra als auch R8 fehlt, ist die strenge Konsistenz der Daten. Bei Cassandra spricht man von „tunable consistency“.[xxxii] Diese Eigenschaft erlaubt Geschwindigkeit und Hochverfügbarkeit gegen Konsistenz eintauschen zu können. Früher sprach man von „eventual consistency“. Davon ist man in letzter Zeit abgerückt, da aus dieser Formulierung teilweise der Schluss gezogen wurde, dass man selbst nicht wüsste, ob die Daten nun konsistent seien oder nicht. Um klarzustellen, dass die Anwender volle Kontrolle über die Konsistenz ihrer Daten haben spricht man daher von „tunable consistency“.[xxxiii] Trotz dieses irreführenden Wordings sind Banken ein dankbarer Nutzer von Cassandra – vor allem wegen ihrer Hochverfügbarkeit und der Möglichkeit nahezu linear zu skalieren. In dieser Branche wird Cassandra in der Regel auf strenge Konsistenz konfiguriert. Für R8 bedeutet Inkonsistenz lediglich, dass verschiedene Versionen einer Anwendung auf den Knoten in einem Cluster betrieben werden können. Das eignet sich vor allem für rollende Updates.

Netflix‘ Inkonsistenz-Experiment

Netflix-Experimente mit zwei großen Cassandra-Ringen haben die tatsächliche Konsistenz auf einem theoretisch inkonsistent-konfigurierten Ring untersucht. In zwei verbundene Cluster wurden oft und schnell Daten eingefügt und verändert und im jeweils anderen Cluster wieder gelesen. Die internen Prozesse einer Cassandra sorgen zwar dafür, dass sich die verbundenen Ringe selbst synchronisieren. Allerdings garantiert Cassandra Konsistenz nur, wenn sie als konsistent konfiguriert wird. Erwartet hatten die Tester, dass durch das Schreiben und Lesen auf dem zweit-geringsten Konsistenzniveau ONE in zwei sich oft und schnell ändernden Cassandra-Clustern viele inkonsistente Daten auftreten würden. Gemessen wurde kein einziger inkonsistenter Datensatz.[xxxiv]

Das heißt, dass Cassandras interne Prozesse so gut arbeiten, dass man kaum einen praktischen Unterschied feststellen kann. Für das Arbeiten mit diesem theoretisch inkonsistenten Stand ergaben sich allerdings ein paar praktische Vorteile: die Latenz ist um den Faktor drei zurückgegangen und der Ring ist weniger anfällig gegen Knoten-Ausfälle. Die sinkende Latenz ist Ergebnis der Nutzung eines Token-basierten Load Balancern[xxxv] im Client und der durch das geringe Konsistenzniveau entfallenden sogenannten read repairs. Daher betreiben Netflix[xxxvi] und Spotify[xxxvii] ihre Cassandra-Cluster auf Consistency Level ONE.[xxxviii]

Diese Eigenschaft, eine Cassandra-Datenbank zu tunen, und das durch die Hochverfügbarkeit auch im Produktivsystem tun zu können, ist ein wichtiger Grund Cassandra zu nutzen. Außerdem kann Cassandra, im Gegensatz zu vielen anderen Datenbanken, schneller Schreiben als Lesen. Durch ihre Ring-Architektur ist sie auch besonders einfach zu betreiben und überlebt Ausfälle ganzer Rechenzentren. Der Client-seitige Binärtreiber erlaubt zudem einen sehr effizienten Zugriff und eine gleichverteilte Last auf den Kopien der Daten im Ring. Zudem kann eine Cassandra eine sehr schnelle und hochverfügbare Analyse-Plattform im Zusammenspiel mit Apache Spark sein. Die oben genannten Features beziehen sich auf die frei einsetzbare Cassandra-Distribution.

Die kommerzielle Version DSE Edition (DataStax Enterprise) hat darüber hinaus technischen Support, zusätzliche Qualitätssicherungsmaßnahmen für viele Plattformen, Management-Tools für Administratoren, eine Cluster-weite Suchfunktion und eine verbesserte Hadoop-Integration. Dazu kommen auch In-Memory-Tabellen, die

Cassandra in einen externen Cache verwandeln, und Verschlüsselung von gespeicherten, ruhenden Daten – zusätzlich zur Transportverschlüsselung in allen Versionen.

Zusammenfassung

R8 ist ein sehr spezifisches Framework – es wurde nicht dazu gebaut Backend- oder Frontend-Frameworks abzulösen. Es eignet sich vor allem für Systeme und Anwendungen mit besonders hohen Anforderungen an Hochverfügbarkeit, Performance und Skalierbarkeit. R8 erfordert spezielle Erfahrungen von Entwicklern: beispielsweise wird kein standardisiertes Persistenz-Framework genutzt, jedoch eine in der Welt der Anwendungssoftware seltene Datenbank und ein spezielles UI-Framework wie Wicket. Darüber hinaus stellt R8 spezifische Anforderungen an das IT-System, wie eine Cassandra-Datenbank als Speicher.

Das Framework nimmt dem Nutzer viele Kleinigkeiten ab: zum Beispiel automatisches Anlegen von Datenstrukturen, das Sammeln von Request-Daten und das Optimieren von Datenbank-Statements. Weiterhin ist R8 out-of-the-box hochverfügbar und seine eigene Firewall sichert es zusätzlich gegen (verteilte) Denial-of-Service-Angriffe ab.

Quellen

[i] en.wikipedia.org/wiki/Multilayered_architecture#Common_layers
[ii] planetcassandra.org/nosql-performance-benchmarks
[iii] datastax.com/documentation/datastax_enterprise/4.0/datastax_enterprise/inMemory.html
[iv] datastax.com/wp-content/uploads/2014/02/WP-DataStax-Enterprise-In-Memory.pdf
[v] Cassandra Design Patterns, von Sanjay Sharma, erschienen bei Packt Publishing, vom 24. Januar 2014, ISBN 978-1783288809
[vi] de.slideshare.net/AxelLiljencrantz/cassandra-summit-2013-how-not-to-use-cassandra
[vii] datastax.com/download/dse-vs-dsc
[viii] androidauthority.com/how-android-updates-roll-out-force-clear-gcf-318744
[ix] techblog.netflix.com/2011/07/netflix-simian-army.html
[x] datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html
[xi] wiki.apache.org/cassandra/HintedHandoff
[xii] techblog.netflix.com/2013/12/astyanax-update.html
[xiii] dba-oracle.com/art_dbazine_expert_tuning.htm
[xiv] infosecurity-magazine.com/news/a-ddos-attack-could-cost-1-million-before
[xv] ddosattacks.biz/impact/ddos-can-cost-1m-mitigation-even-starts
[xvi] securityweek.com/ddos-attacks-cost-40000-hour-incapsula
[xvii] scmagazine.com/incapsula-found-the-of-ddos-attacks-to-be-substantial/article/383179
[xviii] incapsula.com/blog/ddos-impact-cost-of-ddos-attack.html
[xix] lp.incapsula.com/rs/incapsulainc/images/eBook%20-%20DDoS%20Impact%20Survey.pdf
[xx] neustar.biz/resources/whitepapers/ddos-protection/2014-annual-ddos-attacks-and-impact-report.pdf
[xxi] kaspersky.com/de/about_kaspersky/news/virus/2013/Jahrestrend_2013_Gezielte_Attacken_auf_Unternehmen_Partner_und_Zulieferer
[xxii] security.radware.com/uploadedFiles/Resources_and_Content/Attack_Tools/CyberSecurityontheOffense.pdf
[xxiii] mercurynews.com/news/ci_27399971/great-instagram-and-facebook-outage-january-2015-social
[xxiv] blog.radware.com/security/2013/05/how-much-can-a-ddos-attack-cost-your-business
[xxv] Securing Web Applications with a Behaviour-Based Firewall, Lawrie Brown et al., Seminarunterlagen, Belvine University
[xxvi] royans.net/wp/2010/02/14/brewers-cap-theorem-on-distributed-systems
[xxvii] julianbrowne.com/article/viewer/brewers-cap-theorem
[xxviii] en.wikipedia.org/wiki/High_availability#System_design_for_high_availability
[xxix] en.wikipedia.org/wiki/Consistency_(database_systems)#As_a_CAP_trade-off
[xxx] en.wikipedia.org/wiki/Network_partition#As_a_CAP_trade-off
[xxxi] radar.oreilly.com/2013/03/returning-transactions-to-distributed-data-stores.html
[xxxii] labs.spotify.com/2015/01/09/personalization-at-spotify-using-cassandra
[xxxiii] youtube.com/watch?v=A6qzx_HE3EU
[xxxiv] planetcassandra.org/blog/a-netflix-experiment-eventual-consistency-hopeful-consistency-by-christos-kalantzis
[xxxv] 2014.nosql-matters.org/cgn/wp-content/uploads/2014/04/GoingNativeWithApacheCassandra.pdf
[xxxvi] techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html
[xxxvii] sics.se/sites/default/files/pub/sics.se/scaling_storage_to_millions_of_users_-_presentation_at_sics_ws.pdf
[xxxviii] youtube.com/watch?v=v5stV9Knpw0