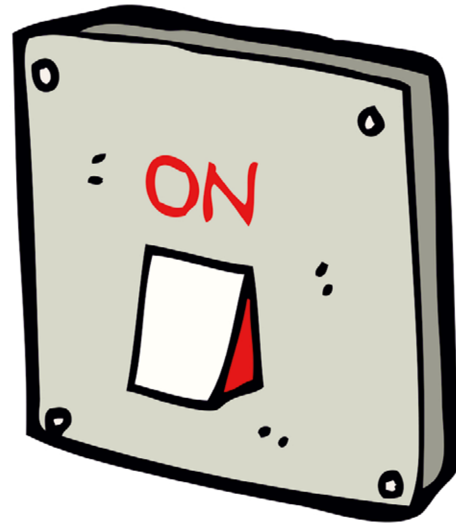
Quelloffene Leistungstrenner für alle

Hystrix – Wider den Totalausfall

Phillip Ghadir

Michael T. Nygard hat in seinem Buch „Release It!" sehr anschaulich die Folgen von sich fortpflanzenden Fehlern beschrieben. Seine klare Empfehlung lautete, Isolationsmuster wie zum Beispiel Leistungstrenner – engl. Circuit Breaker – in die Software zu integrieren, um Totalausfälle zu vermeiden. Mit Hystrix hat Netflix ein Framework bereitgestellt, das den Einbau von solchen Schaltern vereinfacht. Dieser Beitrag stellt das Framework vor.



Wozu Hystrix?

Hystrix ist ein von Netflix entwickeltes auf Github bereitgestelltes quelloffenes Framework zur Kapselung, Entkopplung und zum Überwachen von Abhängigkeiten. Hystrix erleichtert das Implementieren von zuverlässigen Komponenten in verteilten Umgebungen, in denen der Ausfall von einzelnen Bestandteilen nicht vermieden werden kann.

Hystrix verwendet das Command-Muster, um potenziell fehleranfällige Anfragen zu kapseln. Das Framework selbst bietet die Möglichkeit, eigene Commands synchron oder asynchron auszuführen und zur Laufzeit die Häufigkeit, Fehlerraten und Laufzeiten zu messen.

Jede Abhängigkeit wird dabei mit einem Kommando gekapselt und dann ausgeführt. Das Framework ist damit in der Lage, Aufrufe zu poolen, zu sammeln, Fehler zu protokollieren, Statistiken zu erstellen und bei Bedarf sogenannte Fallback-Methoden aufzurufen.

Wie schnell die Verfügbarkeit einer Komponente sinkt, wenn sie von mehreren anderen abhängt, zeigt Tabelle 1.

Hystrix' Design-Prinzipien

Das Hystrix Wiki [HystrixWiki] beschreibt die dem Entwurf von Hystrix zugrunde liegenden Prinzipien. In erster Linie darf keine Abhängigkeit alle verfügbaren Nutzer-Threads (des Containers) aufbrauchen können. Daher ist es besser, Requests zu verlieren und Fehler zu melden, als Anfragen zu queuen.

Wenn möglich, sollen Anwendern beziehungsweise Clients Fallbacks geliefert werden, anstatt eine Anfrage nicht beantworten zu können und zu scheitern. Hystrix verwendet Isolationsmechanismen (wie Bulkhead, Circuit Breaker und Timeouts), um die potenziellen Auswirkungen jeder einzelnen Abhängigkeit zu begrenzen. Quasi jeder Aspekt und jedes Verhalten von Hystrix kann durch Konfiguration angepasst werden.

Hystrix ermöglicht kurze Feedback-Zyklen zur Laufzeit. Durch kontinuierliche zeitnahe Messungen, Monitoring und Alarme kann das Laufzeitverhalten des Systems quasi live mitverfolgt werden. Durch das dynamische Konfigurationssystem werden Konfigurationsänderungen zur Laufzeit kaum verzögert propagiert. Dadurch lässt sich das System unmittelbar auf sich ändernde Lastszenarien anpassen und die benötigte Zeit zur Wiederherstellung der vollen Funktionsfähigkeit gering halten.

Metriken integriert

Für die Messung von Client-Aufrufen lassen sich Metriken konfigurieren. Hystrix misst zur Laufzeit verschiedenste Daten wie zum Beispiel Anzahl der erfolgreichen Aufrufe, der Fehler, der Cache-Zugriffe, der zusammengefassten Aufrufe und vieles mehr. Der Kasten „Kurzübersicht Metrics" liefert einen Überblick über die bereitgestellten Messinstrumente.

Damit Hystrix weiß, welche Messwerte in unserem System gesammelt werden sollen, muss Hystrix entsprechend konfiguriert werden. Dazu verwendet Hystrix ein hierarchisches Namensschema für Konfigurationsparameter: Das Präfix für die Konfiguration eines Kommandos ist stets „hystrix.command.<NAME DES KOMMANDOS>". Um den Default für alle Kommandos zu setzen, kann man für „NAME DES KOMMANDOS" auch „default" einsetzen. Um Metriken für ein Kommando zu konfigurieren, fügt man an das Präfix den entsprechenden Konfigurationsparameter an.

	Anzahl an Abhängigkeiten														
Verfügbarkeit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
99,99%	99,99%	99,98%	99,97%	99,96%	99,95%	99,94%	99,93%	99,92%	99,91%	99,90%	99,89%	99,88%	99,87%	99,86%	99,85%
99,90%	99,90%	99,80%	99,70%	99,60%	99,50%	99,40%	99,30%	99,20%	99,10%	99,00%	98,91%	98,81%	98,71%	98,61%	98,51%
99%	99,00%	98,01%	97,03%	96,06%	95,10%	94,15%	93,21%	92,27%	91,35%	90,44%	89,53%	88,64%	87,75%	86,87%	86,01%
98%	98,00%	96,04%	94,12%	92,24%	90,39%	88,58%	86,81%	85,08%	83,37%	81,71%	80,07%	78,47%	76,90%	75,36%	73,86%
97%	97,00%	94,09%	91,27%	88,53%	85,87%	83,30%	80,80%	78,37%	76,02%	73,74%	71,53%	69,38%	67,30%	65,28%	63,33%
96%	96,00%	92,16%	88,47%	84,93%	81,54%	78,28%	75,14%	72,14%	69,25%	66,48%	63,82%	61,27%	58,82%	56,47%	54,21%
95%	95,00%	90,25%	85,74%	81,45%	77,38%	73,51%	69,83%	66,34%	63,02%	59,87%	56,88%	54,04%	51,33%	48,77%	46,33%
90%	90,00%	81,00%	72,90%	65,61%	59,05%	53,14%	47,83%	43,05%	38,74%	34,87%	31,38%	28,24%	25,42%	22,88%	20,59%

Tabelle 1: Übersicht über die theoretische Verfügbarkeit bei einer oder mehreren Abhängigkeiten zu Komponenten mit bezeichneter Verfügbarkeit

Kurzübersicht Metrics

Metrics bietet eine Registry, über die verschiedene Messinstrumente registriert werden können, um von unterschiedlichen Programmteilen einfach verwendet werden zu können. Zum Messen von Daten bietet Metrics

- Gauges – Messinstrumente, die das aktuelle Messergebnis wiedergeben,
- Counters – Zähler, also Messinstrumente für ganzzahlige Werte,
- Meters – messen die Quote von Ereignissen pro Sekunde, pro Minute, pro 5-Minuten, pro 15-Minuten,
- Histograms – Histogramme, liefern die statistische Wert-Verteilung, Minimum, Maximum, Durchschnitt, Mittelwert, 75%-, 90%-, 95%-, 98%, 99%- und 99,9%-Perzentil,
- Timers – messen die Häufigkeit und Dauer von Aufrufen.

Die erfassten Messwerte können über verschiedene Back-Ends dann aufbereitet und auch visualisiert werden. In Hystrix integriert sind derzeit Implementierungen für Yammer und Servo.

Möchten wir beispielsweise Perzentile messen, können wir über den **ConfigurationManager** die Anzahl der Buckets für den Konfigurationsparameter **hystrix.command.default.metrics.rollingPercentile.numBuckets** setzen. Dies geschieht zum Beispiel in einer Initialisierungsroutine so

```
final AbstractConfiguration cfg =
  ConfigurationManager.getConfigInstance();
cfg.setProperty(
  "hystrix.command.default.metrics.rollingPercentile.numBuckets", 60);
```

Die erfassten Messwerte können in einem Hystrix-eigenen Dashboard beobachtet werden.

Das Szenario

In der vergangenen Ausgabe haben wir ein fiktives Rating-System [GhaSch14] skizziert. So ein Bewertungssystem fragt innerhalb eines Rating-Vorgangs auch andere Auskunftsdienste ab, um ein Gesamtergebnis zu berechnen. Abbildung 1 zeigt das System Rating-Core mit seinen Um-Systemen – einschließlich der beiden Abhängigkeiten zu dem Schufa-Connector und dem Document-Archive. Konzentrieren wir uns erst einmal auf das Document-Archive.

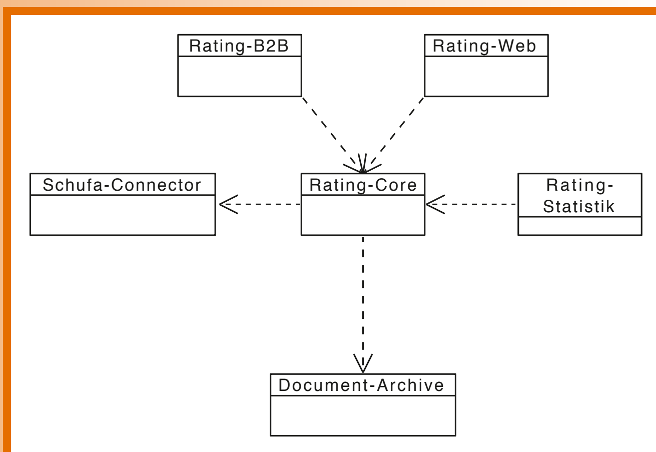


Abb. 1: Darstellung des Rating-Systems

Das Document-Archive aufrufen

Die Implementierung eines Aufrufs des Dokumentenarchivs ist einfach. Über ein einfaches GET kann ein Dokument geladen werden. Über POST können Dokumente abgelegt werden. Jede dieser Interaktionen kapselt man in Hystrix mit einer Implementierung des Kommando-Musters, das wir in seiner Grundform aus [GoF] kennen.

Wir müssen also ein Kommando zum Laden eines Dokuments (FetchDocumentCommand) und eines zum Ablegen eines Dokuments (StoreDocumentCommand) implementieren. Nehmen wir das Beispiel „Laden eines Dokuments".

Anstatt die Abfrage des Archivsystems im Rating-Core so direkt zu implementieren

```
Document doc = get( DOCUMENT_ARCHIVE_URI, credentials, documentId );
```

kapseln wir den Aufruf in einer speziellen von Hystrix-Command abgeleiteten Implementierung. Die für den Aufruf benötigten Parameter werden in Attributen unseres Kommandos zwischengespeichert und vom Konstruktor initialisiert.

```
public class FetchDocumentCommand extends HystrixCommand<Document> {
  // Attribute
  private String uriTemplate;
  private Credentials credentials;
  private String documentId;

  // Konstruktor
  public FetchDocumentCommand(
      String uriTemplate, Credentials credentials,
      String documentId) {
    super(HystrixCommandGroupKey.Factory.asKey("DocArchive"));

    this.uriTemplate = uriTemplate;
    this.credentials = credentials;
    this.documentId = documentId;
  }

  // Methoden
  @Override
  protected Document run() throws Exception {
    return get( uriTemplate, credentials, documentId );
  }

  // ...
}
```

Listing 1: FetchDocumentCommand kapselt den Aufruf des Dokumentenarchivs

Die Methode **run()** in Listing 1 enthält den eigentlichen Client-Code. Diese Methode rufen wir aber nicht selbst auf, sondern überlassen das dem Hystrix-Framework.

Hystrix-Kommandos aufrufen

Damit ein Kommando ausgeführt werden kann, müssen wir es Hystrix zur Ausführung geben. Dazu stehen drei verschiedene Methoden zur Verfügung:

- **execute()** für den synchronen Aufruf,
- **queue()** für den asynchronen und
- **observe()**, worauf wir später noch eingehen.

Das gewünschte Verhalten der obigen Zeile können wir nun mit dem Kommando einfach schreiben

```
Document doc = new FetchDocumentCommand(
DOCUMENT_ARCHIVE_URI, credentials, documentId).execute();
```

Der `execute()`-Aufruf lässt nun Hystrix zaubern.

Sicherungsschotts und Leistungstrenner

In [Nyg07] hat Michael Nygard ein Muster für die Stabilität des Systems beschrieben: das Bulkhead-Muster. Er schreibt dazu „*Protecting critical clients by giving them their own pool to call.*“ Abhängig von dem Abstraktionsgrad kann mit dem Pool ein Rechenzentrum, eine Server-Farm, (virtuelle) Server, Applikationen, Thread-Pools oder auch Connection-Pools gemeint sein.

Das im Zusammenhang mit seinem Buch deutlich häufiger beschriebene Muster ist das Circuit Breaker-Muster. Es dient dazu, eine Aufrufkette zu unterbrechen, sobald ein Fehler festgestellt wird. So wird verhindert, dass Aufrufe eines defekten Systemteils den aufrufenden Client in Mitleidenschaft ziehen und so nach und nach das Gesamtsystem lahmlegen.

Abbildung 2 skizziert den Zustandsautomaten, der dem Circuit Breaker-Muster zugrunde liegt. Er zeigt, wie die Aufrufkette unterbrochen wird, wenn die Fehlerhäufigkeit bei den Aufrufen einen definierten Schwellwert erreicht.

Solange der Leistungstrenner geschlossen ist, werden die Aufrufe durchgelassen. Jeder Fehler wird gezählt, bis der Schwellenwert erreicht ist und der Leistungstrenner die Aufrufkette unterbricht. Einmal unterbrochen werden Aufrufe erst nach einer gewissen Zeit (dem Timeout) wieder durchgelassen. Ein erfolgreicher Aufruf setzt den Fehlerzähler zurück und alles ist wieder gut.

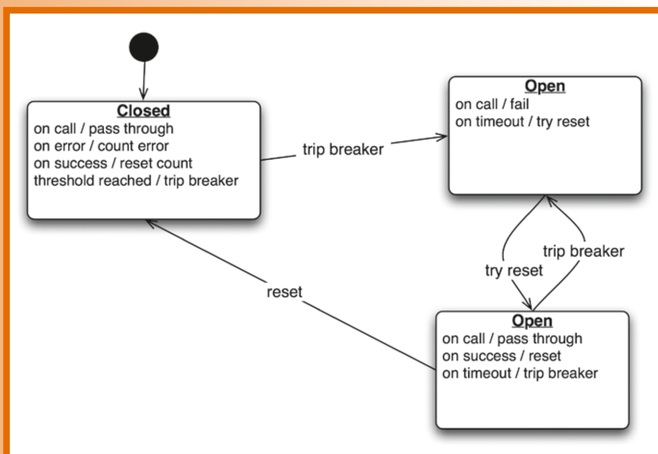


Abb. 2: Zustandsautomat des Leistungstrenners (Circuit Breakers)

Auf Fehler reagieren

Wenn im System einzelne Komponenten oder Services nicht zur Verfügung stehen, sollten wir darauf angemessen reagieren. Wie oft haben wir schon Kaskaden von Exception-Handlern geschrieben, die unsere schöne Fachlogik umlagerten? Auch wenn man sie seit Java 7 dank Multi-Catch abkürzen kann, möchte man doch eigentlich keine catch-Klauseln schreiben.

Hystrix bietet dafür Framework-seitig das Konzept des Fallbacks. Analog zur in Listing 1 dargestellten Methode `run()` können wir in unserem Hystrix-Command eine Methode `getFallback()` definieren, die im Falle einer Exception oder eines Timeouts dann das Ergebnis im Fehlerfalle zurückliefert.

```
@Override
protected Document getFallback() {
  return null;
}
```

Listing 2: Die Methode getFallback() der FetchDocumentCommand-Klasse

Listing 2 zeigt die Methode, die im Fehlerfalle aufgerufen werden soll. In unserem Beispiel geben wir einfach null zurück. Hier könnte man sich auch den Aufruf eines Backup-Connectors vorstellen, von dem die Dokumente geladen werden sollten.

Gleich, welche Funktionalität benötigt wird: Wir können sie einfach als Fallback implementieren und brauchen dafür keinen try-catch-Block.

Abbildung 3 zeigt schematisch, wie Hystrix die Stabilitätsmuster Bulkhead und Circuit Breaker einsetzt und wie es das Monitoring integriert.

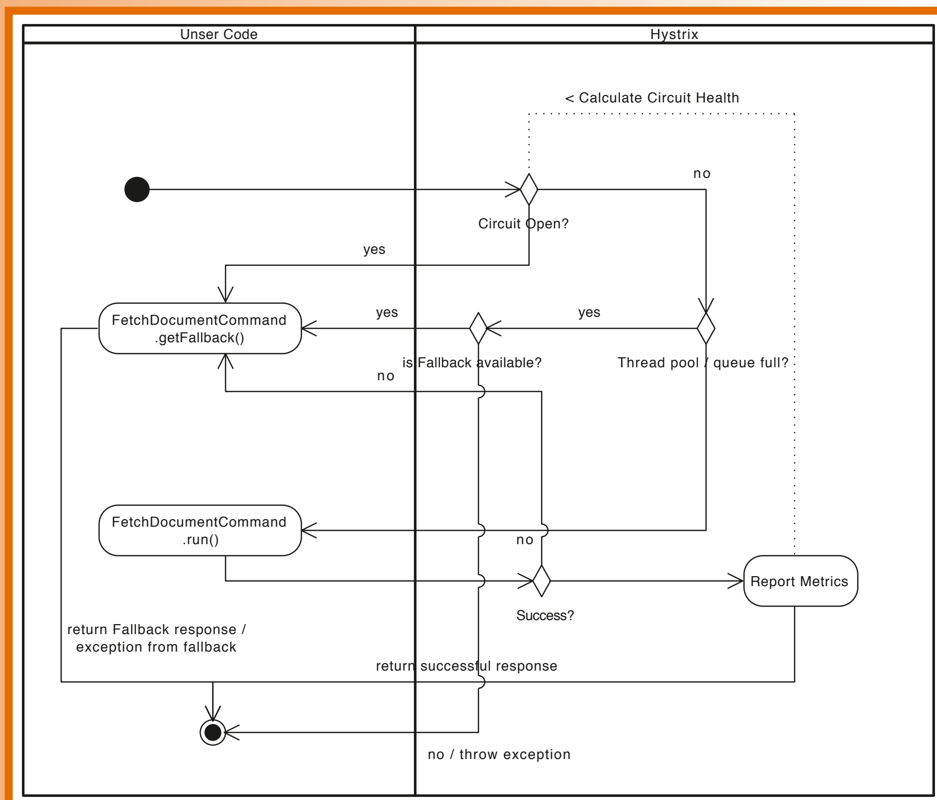


Abb. 3: Ablaufplan für den Aufruf eines Hystrix-Kommandos, zu sehen sind die verschiedenen Schutzwälle und der Leistungstrenner

RXJava geht auch

Wenn wir über neue Daten informiert werden wollen, können wir auf den Kommandos anstelle der Methode `execute()` die Methode `observe()` aufrufen, das ein Observable zurück liefert. Hystrix verwendet für das Reactive Programming, das bereits in dieser Kolumne vorgestellte RXJava [Gha13].

Zur Erinnerung: Bei RXJava können Beobachter an sogenannten Observables registriert werden, die vom Observable für jedes Element durch den Aufruf von `onNext()` des Observers benachrichtigt werden. Ist ein Observable am Ende des Datenstroms angelangt, benachrichtigt es die registrierten Beobachter über deren Methode `onComplete()`. Fehler werden per Aufruf der Beobachter-Methode `onError()` propagiert.

Hystrix-Kommandos können per `toObservable()` oder auch `observe()` in RXJava-Observables konvertiert werden.

Gruppieren von Kommandos

Eingangs haben wir bereits die Konfigurierbarkeit von Hystrix erwähnt. Abhängigkeiten können je nach Bedarf mit unterschiedlichen Strategien vom Rest des Systems isoliert werden. Die hierarchische Konfiguration ermöglicht beispielsweise das detaillierte Protokollieren von externen Abhängigkeiten einerseits und das grobgranulare Protokollieren andererseits.

Darüber hinaus können die Isolationsstrategien und Thread-Pools für eine Gruppe von Kommandos individuell gesetzt werden. Jedes Kommando definiert seine Gruppenzugehörigkeit beim Instanziieren über den Super-Aufruf im Konstruktor. In Listing 1 kann man in der ersten Zeile des Konstruktors sehen, dass `FetchDocumentCommand` zur Gruppe „DocArchive“ gehört.

Threads und Semaphore

Hystrix bietet zwei Isolationsstrategien an, mit denen die Ausführung eines Kommandos isoliert werden kann: das Entkoppeln eines Aufrufs in einem eigenen Thread sowie das Beschränken der nebenläufigen Ausführung von Aufrufen mit Hilfe von Semaphoren. Hystrix verwendet standardmäßig eigene Threads für den Aufruf der `run()`-Methode. Das kostet zwar ein wenig Overhead, erlaubt aber das Setzen von Timeouts und das frühzeitige Fortsetzen der restlichen Verarbeitung. Die beiden Isolationsstrategien können die Anzahl der parallel ausführbaren Aufrufe begrenzen.

Wollten wir beispielsweise sicherstellen, dass ein Kommando nicht wie üblich im separaten Thread ausgeführt wird, könnten wir dafür als Strategie Semaphore konfigurieren. Dadurch wird der Overhead eingespart, den die Verwendung von Threads mit sich bringt. Allerdings dient ein Semaphor nur dazu, die Anzahl paralleler Aufrufe zu begrenzen. Die Fähigkeit Service-Aufrufe mit Timeouts zu versehen, bieten Semaphore nicht. Dazu ist dann doch ein separater Thread nötig.

Im Rating-System bietet sich zum Beispiel das Begrenzen der parallelen Aufbereitung von Reports an, da deren Erstellung recht viel Speicher benötigt. Daher würden wir dafür in der Initialisierungsroutine konfigurieren

```
cfg.setProperty(
  "hystrix.command.casereports."+
  "execution.isolation.semaphore.maxConcurrentRequests", 3);
```

Optimierte Client-Aufrufe

Das Command-Muster erlaubt Hystrix nicht nur die Isolation von Client-Abhängigkeiten, sondern ermöglicht auch das Sammeln von Client-Aufrufen, um die zugehörigen Service-Aufrufe en block auszuführen.

Dazu können in Hystrix sogenannte Collapser konfiguriert werden. Am Beispiel des Document-Archives könnte ich mir vorstellen, dass man nicht jedes Dokument einzeln vom Archiv laden möchte, wenn in einem Rating-Vorgang stets alle benötigt werden. Dafür würden wir dann also konfigurieren

```
hystrix.collapser.docArchive.maxRequestsInBatch = 15
hystrix.collapser.docArchive.timerDelayInMilliseconds = 10
```

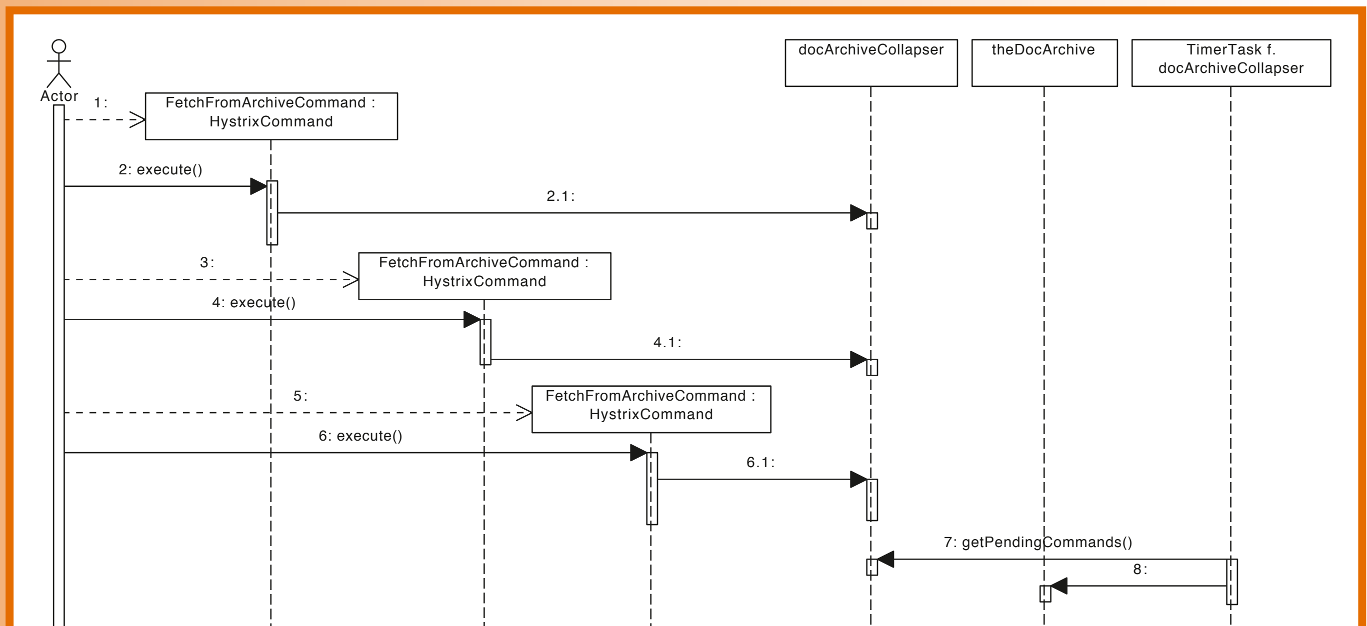


Abb. 4: Wird der RequestCollapser konfiguriert, werden in definierten Intervallen die Requests gesammelt und bei Erreichen eines Limits oder Intervallendes gesendet

Mit dieser Konfiguration gibt Hystrix unsere Aufrufe von `execute()` oder `queue()` nicht direkt an das Document-Archive weiter, sondern sammelt sie, bis das Batch-Limit (`maxRequestsInBatch`) erreicht ist oder der Timer wieder die noch nicht verarbeiteten Requests im Batch ans Document-Archive weiter gibt. Das Sequenz-Diagramm in Abbildung 4 zeigt das Szenario, in dem ein Client drei Dokumente vom Archiv holen möchte und sie im Batch abgeholt werden.

Analog zum Zusammenlegen von Requests kann Hystrix auch so konfiguriert werden, dass Anfrage-Ergebnisse innerhalb eines Request-Zyklus gecached werden können.

Fazit

Hystrix ist ein tolles Framework zum Realisieren von verteilten Systemen. Dank der integrierten Isolationsmuster kann die Reaktionsfähigkeit des Systems auch bei partiellen Ausfällen aufrechterhalten werden. Dank der Integration von Metriken samt Dashboard zum live Überwachen des Systems während des Betriebs und des Konfigurationsframeworks, das die Anpassung an bestehende Rahmenparameter ebenfalls während des Betriebs erlaubt, stehen die Chancen gut, das System zur Laufzeit anzupassen und lange Ausfallzeiten zu vermeiden.

Ein weiterer Vorteil ist das geradlinige Programmiermodell: Jede Art von Interaktion mit einem zuliefernden System wird in einem Kommando gekapselt. Optimiert wird unter der Haube. Hystrix Request Collapser erlauben es, fachlich sinnvolle Client-Modelle zu bauen und nachträglich die Optimierung mit Batch-Requests zu konfigurieren.

Literatur und Links

[Gha13] Ph. Ghadir, Reactive Extensions in Java, in: JavaSPEKTRUM, 5/2013

[GhaSch14] Ph. Ghadir, Ph. Schirmacher, Domain-Driven Design in Clojure, in: JavaSPEKTRUM, 2/2014

[GoF] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley, 1996

[Graphite] `http://de.wikipedia.org/wiki/Graphite`

[Hystrix] `https://github.com/Netflix/Hystrix`

[HystrixWiki] `https://github.com/Netflix/Hystrix/wiki`

[Metrics] `http://metrics.codahale.com/getting-started/`

[Nyg07] M. Nygard, Release It!, Pragmatic Bookshelf, 2007

[RXJava] Java VM implementation of Reactive Extensions, `https://github.com/Netflix/RxJava`

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com