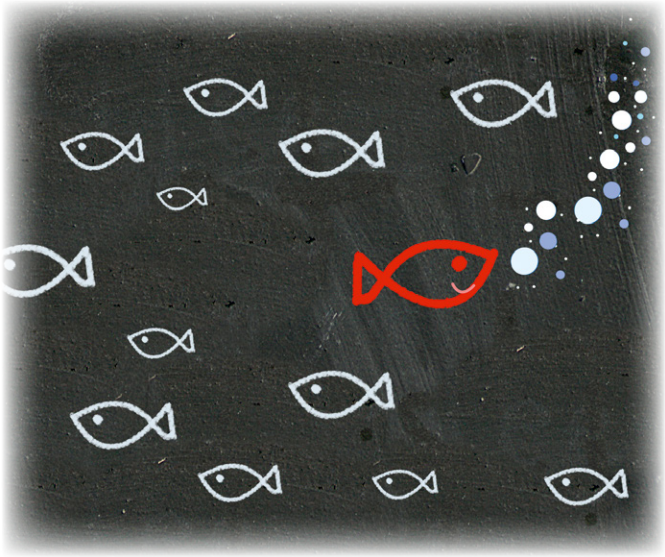
Gegen den Strom

Datenströme asynchron verarbeiten mit Reactive Streams

Rüdiger Grammes, Kristine Schaal



Be reactive! Das ist die Antwort auf die gestiegenen Anforderungen an Performanz, Skalierbarkeit und Ausfallsicherheit in allen Bereichen der IT. Für asynchrone, eventgetriebene, nicht-blockierende Architekturen sind in den letzten Jahren zahlreiche neue Frameworks wie Akka, Vert.x und Reactor entstanden. Ein weiterer konzeptioneller Baustein kommt nun dazu: Reactive Streams bieten die passende Unterstützung, um mit Bottlenecks im Datenfluss umzugehen.

Reactive Streams sind eine Spezifikation zur asynchronen, nicht-blockierenden Verarbeitung von (potenziell unbegrenzten) Datenströmen innerhalb einer JVM. Die Spezifikation definiert einen Publish-Subscribe-Ansatz inklusive Backpressure. Reactive Streams ergänzen konzeptionell die Architektur von reaktiven Systemen und Anwendungen, deren Grundlagen im Reactive Manifesto [ReactiveManifesto] beschrieben sind.

Hinter Reactive Streams steht eine Reihe namhafter Firmen, darunter Netflix, Pivotal, Typesafe und RedHat. Entsprechend existieren bereits einige Implementierungen, obwohl die erste finale Version des Standards – Version 1.0.0 – erst Ende April 2015 veröffentlicht wurde. Dazu gehören Akka Streams [AkkaStreams], Spring Reactor ab Version 2 [SpringReactor], Vert.x ab Version 3 [Vert.x3] und RxJava (Versionen 1 und 2) [RxJava]. Neben Frameworks existieren auch Treiber-Implementierungen für beispielsweise MongoDB und RabbitMQ.

In diesem Artikel erläutern wir zunächst die Motivation und stellen die Spezifikation vor. Im Anschluss zeigen wir Akka Streams als Beispiel für ein Framework, das die Spezifikation implementiert.

Stau an der richtigen Stelle mit Backpressure

Betrachten wir eine einfache asynchrone Verarbeitungskette, bestehend aus einem Sender und einem Empfänger. Schickt der Sender dem Empfänger schneller Daten, als dieser sie verarbeiten kann, kommt es irgendwann zu Problemen. Es gibt zwei naive, „einseitige" Ansätze, damit umzugehen:

- *Ansatz 1:* Ausschließlich der Empfänger reagiert auf die Überlast: Das kann ein Speicherüberlauf sein, wenn der Empfänger einen Puffer ohne Größenbeschränkung hat, oder Datenverlust.
- *Ansatz 2:* Der Sender begrenzt die Datenmenge, die er dem Empfänger schickt. Aber der Sender weiß in der Regel nicht, wie viel der Empfänger verarbeiten kann. Es kann also trotzdem zu Überlast kommen. Geht der Sender zu pessimistisch vor, so ist es möglich, dass er weniger schickt, als der Empfänger verarbeiten könnte, womit der Durchsatz sinkt.

Reactive Streams lösen das Problem durch Backpressure (zu Deutsch: Gegendruck oder Rückstau). Dem Empfänger werden nur so viele Daten zugestellt, wie er verarbeiten beziehungsweise in seinem Puffer zwischenspeichern kann. Dabei fordert der Empfänger (Subscriber) die Daten explizit entsprechend seiner Kapazitäten an, der Sender (Publisher) darf nur maximal so viele Daten schicken, wie der Empfänger angefordert hat. Der Datenfluss und die Datenanforderung in Gegenrichtung des Datenstroms sind in Abbildung 1 dargestellt.

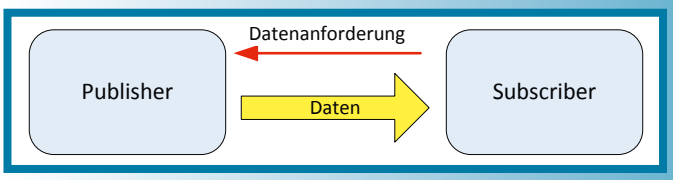


Abb. 1: Datenanforderung und Datenstrom

Das Vorgehen kann man auf eine Verarbeitungskette mit beliebig vielen Schritten verallgemeinern. In einer längeren Kette ist Backpressure „ansteckend": Der Gegendruck wird also – sofern er nicht im betroffenen Schritt behandelt wird – nach vorne in der Kette weitergegeben. Das ist in Abbildung 2 dargestellt. Dabei agiert ein Glied in der Mitte der Kette sowohl als Empfänger als auch als Sender.

Damit haben wir eine Überlast beim Empfänger verhindert, nicht jedoch im Gesamtsystem. Aber wir haben bereits konzeptionelle Verbesserungen erreicht:

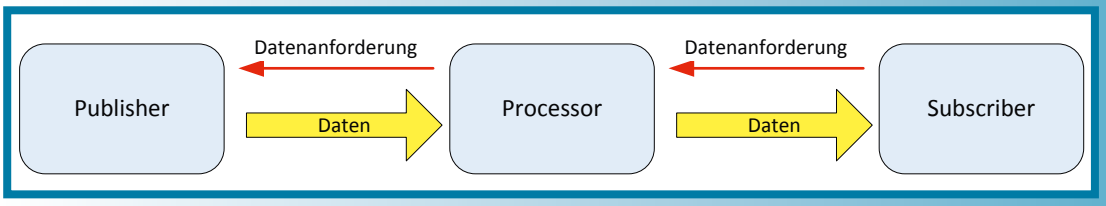


Abb. 2: Verarbeitungskette mit Backpressure

- Der Überlastschutz lässt sich dynamisch steuern: Ist ein Glied in der Verarbeitungskette temporär langsam, fordert es zeitweise weniger Daten an, kann aber später mehr Daten anfordern, sobald es wieder schneller arbeitet.
- Wir können kontrollieren, wo wir mit Überlast umgehen, indem wir diese gezielt an die Stelle in der Kette verlagern, die die beste Strategie implementieren kann – oder bis ganz vorne in der Kette.

Reactive Streams ermöglichen ein dynamisches push/pull (push, wenn der Empfänger schneller als der Sender ist; pull, wenn der Empfänger langsamer als der Sender ist). Oder anders ausgedrückt: Reactive Streams ermöglichen ein selbstregulierendes System bezüglich der Überlast.

Reactive Streams definieren immer 1:1-Beziehungen zwischen Sender und Empfänger. Ein Broadcast ist nicht möglich.

Backpressure – ein alter Hut?

Strategien zur Flusskontrolle sind im Bereich der Kommunikationsprotokolle – beispielsweise dem TCP-Protokoll – lange bekannt. Es gibt dort verschiedene Ansätze für Flusskontrolle, unter anderem:

- *Sliding Window:* Die Anzahl der noch nicht quittierten Pakete ist durch eine Fenstergröße im Sender beschränkt. Der Empfänger steuert die Übertragungsrate durch die Geschwindigkeit, mit der er Pakete quittiert. Der Sender kann die Fenstergröße dynamisch anhand von Informationen in den Quittungen des Empfängers anpassen. Ein Spezialfall von Sliding Window ist „Stop and Wait" mit einem Fenster der Größe eins.
- *NAK oder NACK* (für „Negative Acknowledgement"): Der Empfänger meldet Probleme (z. B. in einer Überlastsituation) mit einer negativen Quittierung.

Backpressure in Reactive Streams lässt sich mit diesen Ansätzen nur eingeschränkt vergleichen, da die Kommunikation auf eine JVM begrenzt ist, und damit komplett In-Memory erfolgt. Explizite Quittierung von Daten wird daher nicht benötigt. Anders als in den genannten Protokollen liegt die Flusskontrolle bei Backpressure komplett in der Hand des Empfängers: Es werden keine Daten zugestellt, wenn der Empfänger keine Daten anfordert. Bei Sliding Window sendet der Sender dagegen Daten ohne Anforderung bis zur initialen Fenstergröße. Eine Überlastsituation im Empfänger kann dann also nicht ausgeschlossen werden.

Die Reactive-Streams-Spezifikation

Reactive Streams definieren ein Set von Java-Interfaces in Kombination mit einer Spezifikation des Verhaltens (Protokoll) [ReactiveStreamsSpecification]. Ein TCK („Technology Compatibility Kit") steht zur Verfügung, um eine Implementierung auf Konformität zur Spezifikation zu testen.

Die Spezifikation enthält Interfaces für diese Bestandteile der Streams: Publisher, Subscriber, Processor (der sowohl Publisher als auch Subscriber ist) und Subscription (über sie teilt der Subscriber dem Publisher seine Datenanforderungen mit). Im Folgenden beschreiben wir die Interfaces, die Methoden erläutern wir detailliert im anschließenden Sequenzdiagramm.

Der *Publisher* enthält nur eine einzige Methode zur Übergabe eines Subscribers:

```
public interface Publisher<T> {
  public void subscribe(Subscriber<? super T> s);
}
```

Der *Subscriber* enthält Methoden, die der Publisher zur Übergabe von Daten oder zum Signalisieren von Fehler- oder Endzuständen nutzt:

```
public interface Subscriber<T> {
  public void onSubscribe(Subscription s);
  public void onNext(T t);
  public void onError(Throwable t);
  public void onComplete();
}
```

Eine *Subscription* ist die Verbindung zwischen einem Publisher und einem Subscriber. Über die Subscription wird eine 1:1-Beziehung zwischen Publisher und Subscriber hergestellt:

```
public interface Subscription {
  public void request(long n);
  public void cancel();
}
```

Der *Processor* erbt von beiden Interfaces, Subscriber und Publisher:

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Abbildung 3 beschreibt beispielhaft den Ablauf der Interaktion:

1. Zu Beginn der Verarbeitung wird ein Subscriber beim Publisher mit der subscribe-Methode registriert.
2. Der Publisher ruft daraufhin Subscriber.onSubscribe() auf und übergibt eine Subscription für die Kommunikation vom Subscriber zum Publisher.

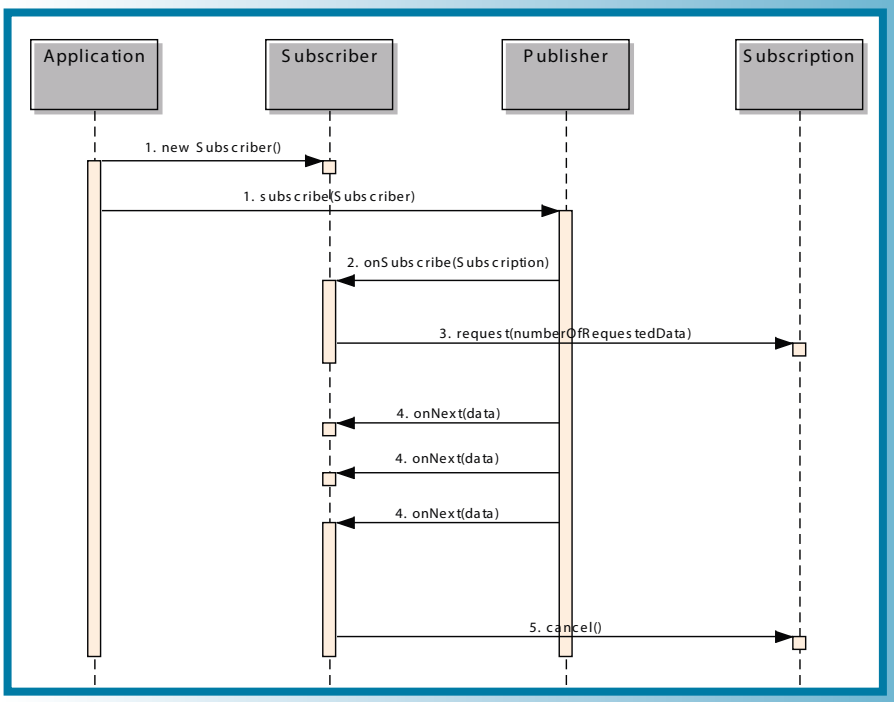


Abb. 3: Interaktion von Publisher, Subscriber und Subscription

3. Der Subscriber fordert Daten an, indem er Subscription.request() aufruft. Er spezifiziert dabei die Anzahl der Datensätze, die er aktuell maximal verarbeiten kann.
4. Zur Übergabe von Daten ruft der Publisher Subscriber.onNext() auf und übergibt dabei einen Datensatz. Er darf das maximal so oft tun, wie Datensätze angefordert wurden.
5. Durch den Aufruf von Subscription.cancel() wird die Beziehung durch den Subscriber beendet. Noch ausstehende Daten aus einem Request kann der Publisher aber noch mit `onNext()` übergeben.

Der Subscriber kann `Subscription.request()` jederzeit aufrufen. Er muss damit nicht warten, bis er alle Datensätze aus dem letzten Request erhalten oder verarbeitet hat. Damit kann der Subscriber sicherstellen, dass seine „Pipeline" immer gut gefüllt ist. Die Anzahl der angeforderten Datensätze addiert sich: Wenn der Subscriber zum Beispiel zunächst fünf Datensätze anfordert und die nächsten fünf bereits anfordert, wenn er erst drei Datensätze bekommen hat, bekommt er insgesamt zehn Datensätze.

Der Publisher kann seinerseits mit `Subscriber.onComplete()` signalisieren, dass er keine weiteren Daten senden wird. Das beendet die Beziehung zwischen Subscriber und Publisher endgültig.

Der Publisher kann dem Subscriber durch den Aufruf von `Subscriber.onError(Throwable t)` einen Fehler signalisieren. Diese Methode ist nicht zur Validierung gedacht, sondern zeigt eine Fehlersituation an, die so schwerwiegend ist, dass keine sinnvolle Weiterverwendung des Streams mehr möglich ist. Die Beziehung zwischen Publisher und Subscriber endet damit ebenfalls endgültig.

Implementierungen

Obwohl Reactive Streams ein neuer Standard ist, wird er bereits in einigen Frameworks implementiert, unter anderem da Framework-Hersteller an der Spezifikation maßgeblich mitgewirkt haben. In Tabelle 1 ist eine Auswahl von Frameworks mit den Klassen, die die Reactive-Streams-Interfaces implementieren beziehungsweise – im Fall von Akka Streams – die entsprechende Implementierung kapseln, aufgelistet.

	Akka Streams	Vert.x	Reactor	Java 9 (Vorschlag)
Publisher	Source	ReactiveWriteStream	Stream	Flow.Publisher
Subscriber	Sink	ReactiveReadStream	Action	Flow.Subscriber
Processor	Flow		Action	Flow.Processor
Subscription				Flow.Subscription

Tabelle 1: Frameworks für Reactive Streams

Reactive Streams in Akka

Als Beispiel für eine Reactive-Streams-Implementierung schauen wir uns Akka Streams an. Akka Streams bilden Verarbeitungsschritte eines Streams auf Akka-Aktoren [Akka] ab, die asynchron über Nachrichten kommunizieren. Das Konzept der Reactive Streams sorgt dabei für die Flusskontrolle zwischen den Aktoren.

Die Laufzeitumgebung eines Akka Streams besteht aus einem Aktor-System (Zeile 1 in Listing 1) und einem Materializer (Zeile 2), der die Abbildung des Streams auf Aktoren durchführt:

Akka Streams beginnen mit einer Source (Zeile 3-4), verlaufen über eine (potenziell leere) Sequenz von Flows (Zeile 5) und enden in einem Sink (Zeile 6-7). Dies entspricht dem in

```
1: final ActorSystem system = ActorSystem.create("streams");
2: final Materializer mat = ActorMaterializer.create(system);

3: final Source<Integer, BoxedUnit> source = Source
4:   .from(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
5:   .filter(i -> i % 2 == 0);
6: Sink<Integer, Future<BoxedUnit>> sink =
7:   Sink.foreach(i -> format("Verarbeitet: %d", i));
8: final RunnableGraph<BoxedUnit> runnable = source.to(sink);
9:
10: runnable.run(mat);
```

Listing 1: Reactive Streams mit Akka Streams

Abbildung 2 gezeigten Zusammenspiel von Publisher, Processor und Subscriber in Reactive Streams. Source, Flow und Sink sind keine Implementierungen der Spezifikations-Interfaces, stattdessen sind diese innerhalb der Akka-Streams-Implementierungsklassen intern gekapselt.

Mit der Verknüpfung von Source und Sink erhält man einen ausführbaren Stream (Zeile 8). Bis hier ist der Stream nur eine Struktur – die Ausführung beginnt erst, wenn man den Stream auf dem Materializer ausführt (Zeile 10). Erst mit diesem Schritt werden die Aktoren erzeugt, die die einzelnen Verarbeitungsschritte asynchron ausführen.

Akka Streams sind typsicher – in unserem Fall produziert die Source einen Stream aus Integer-Werten, die Sink verarbeitet Integer-Werte. Der Typ kann sich dabei im Rahmen der Verarbeitung im Stream auch ändern. Neben dem Typ der Elemente im Stream kann die Ausführung des Streams selbst auch ein Resultat ergeben. Der Typ dieses Resultats wird durch den zweiten Typparameter der Sink (Zeile 6) beschrieben. „BoxedUnit" ist ein Void-Typ, unser Stream erzeugt kein Resultat.

Backpressure in Action

Um die Abläufe besser zu verdeutlichen, simulieren wir in einem Beispiel eine Überlastsituation:

- In Zeile 13 von Listing 2 bremsen wir den Stream mit einer langsamen Berechnung aus. Die Berechnung benötigt eine Sekunde und gibt die Eingabe unverändert zurück.
- Akka Streams stellen Puffer-Komponenten zur Verfügung, die Werte im Stream puffern und auf Buffer-Overflows entsprechend einer konfigurierbaren Strategie reagieren. In Zeile 11 bauen wir einen Puffer der Größe 1 ein, der bei Überlauf das älteste Element verwirft (Strategie „dropHead")*. Der Puffer ist absichtlich so klein gewählt, um einen Überlauf zu provozieren.

Vor dem Puffer und vor der Berechnung geben wir das Element auf der Konsole aus.

In der Ausgabe des Programms (s. Listing 3) erkennt man, dass alle Elemente innerhalb kurzer Zeit die Stufe vor dem Puffer durchlaufen (Zeile 1-14). Vier Elemente durchlaufen die Stufe vor de r Berechnung und liegen damit im Puffer der langsamen Berechnung (Zeile 3-10). Nach einer Sekunde ist das erste Element verarbeitet (Zeile 15), weitere folgen im Abstand von einer Sekunde. Das Element Nummer 9 kommt nicht in der Verarbeitung an.

* Bei der Puffergröße 1 macht es genaugenommen keinen Unterschied, ob wir das neueste oder das älteste Element verwerfen, da nur genau ein Element existiert.

```
 1: final ActorSystem system = ActorSystem.create("streams");
 2: final Materializer mat = ActorMaterializer.create(system);
 3:
 4: final Source<Integer, BoxedUnit> source =
 5:   Source.from(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
 6: Sink<Integer, Future<BoxedUnit>> sink =
 7:   Sink.foreach(i -> format("Verarbeitet: %d", i));
 8:
 9: RunnableGraph<BoxedUnit> runnable = source
10:   .map(f -> format("Vor dem Buffer: %d", f))
11:   .buffer(1, OverflowStrategy.dropHead())
12:   .map(f -> format("Vor der Berechnung: %d", f))
13:   .map(f -> slowComputation(f))
14:   .to(sink);
15:
16: runnable.run(mat);
```

Listing 2: Stream mit Überlastbehandlung

```
 1: 15:46:40 [sys-akka.actor.default-dispatcher-8] Vor dem Buffer: 1
 2: 15:46:40 [sys-akka.actor.default-dispatcher-8] Vor dem Buffer: 2
 3: 15:46:40 [sys-akka.actor.default-dispatcher-9] Vor der Berechnung: 1
 4: 15:46:40 [sys-akka.actor.default-dispatcher-8] Vor dem Buffer: 3
 5: 15:46:40 [sys-akka.actor.default-dispatcher-9] Vor der Berechnung: 2
 6: 15:46:40 [sys-akka.actor.default-dispatcher-8] Vor dem Buffer: 4
 7: 15:46:40 [sys-akka.actor.default-dispatcher-8] Vor dem Buffer: 5
 8: 15:46:40 [sys-akka.actor.default-dispatcher-4] Vor der Berechnung: 3
 9: 15:46:40 [sys-akka.actor.default-dispatcher-8] Vor dem Buffer: 6
10: 15:46:40 [sys-akka.actor.default-dispatcher-4] Vor der Berechnung: 4
11: 15:46:40 [sys-akka.actor.default-dispatcher-3] Vor dem Buffer: 7
12: 15:46:40 [sys-akka.actor.default-dispatcher-3] Vor dem Buffer: 8
13: 15:46:40 [sys-akka.actor.default-dispatcher-4] Vor dem Buffer: 9
14: 15:46:40 [sys-akka.actor.default-dispatcher-4] Vor dem Buffer: 10
15: 15:46:41 [sys-akka.actor.default-dispatcher-5] Verarbeitet: 1
16: 15:46:41 [sys-akka.actor.default-dispatcher-5] Vor der Berechnung: 5
17: 15:46:41 [sys-akka.actor.default-dispatcher-5] Vor der Berechnung: 6
18: 15:46:42 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 2
19: 15:46:43 [sys-akka.actor.default-dispatcher-8] Vor der Berechnung: 7
20: 15:46:43 [sys-akka.actor.default-dispatcher-8] Vor der Berechnung: 8
21: 15:46:43 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 3
22: 15:46:44 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 4
23: 15:46:45 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 5
24: 15:46:45 [sys-akka.actor.default-dispatcher-9] Vor der Berechnung: 10
25: 15:46:46 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 6
26: 15:46:47 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 7
27: 15:46:48 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 8
28: 15:46:49 [sys-akka.actor.default-dispatcher-7] Verarbeitet: 10
```

Listing 3: Programmausgabe

Um den Ablauf zu verstehen, muss man zwei Details zu Akka Streams kennen, die in der Default-Konfiguration zum Tragen kommen:

- Jede Stufe in einem Akka Stream besitzt einen internen Puffer, der per Default vier Elemente groß ist. Das schließt die Map-Schritte, die wir zur besseren Nachvollziehbarkeit eingebaut haben, mit ein (nicht aber den expliziten Puffer, der nur ein Element enthält).
- Eine Stufe fordert per Default neue Elemente an, wenn ihr Puffer zur Hälfte oder weniger gefüllt ist.

Initial fordern die Stufen Daten entsprechend ihrer Pufferkapazität an. Durch die langsame Berechnung und durch Backpressure füllen sich die Puffer der Stufen, bis die in Abbildung 4 dargestellte Situation erreicht ist

Element 1 befindet sich in der Berechnung, Elemente 2 bis 4 im Puffer der Berechnung. Da der Puffer mehr als zur Hälfte gefüllt ist, wird zunächst kein weiteres Element angefordert. Die Stufe vor der Berechnung puffert die Elemente 5 bis 8 und fordert ebenfalls keine weiteren Daten an. Die Puffer-Stufe puffert das Element 9, fragt aber aufgrund der Puffer-Strategie „dropHead“ weitere Elemente an. Das Element 10 verdrängt dadurch das Element 9. Da die Source keine weiteren Elemente liefert, verbleibt Element 10 im Puffer. Mit Beginn der Verarbeitung von Element 2 ist der Puffer der Berechnung nur noch halb gefüllt, und es werden dadurch automatisch zwei weitere Elemente angefragt (Zeile 16 und 17 in Listing 3). Das setzt sich fort, bis alle Elemente durch den Stream gelaufen sind.

Explizite Puffer-Elemente sind eine Möglichkeit in Akka Streams, um Strategien für die Überlastbehandlung zu definieren. Durch den Puffer mit der Strategie „dropHead“ behandeln wir Überlast durch das Verwerfen der ältesten gepufferten Elemente – das ist sinnvoll in Fällen, in denen Daten verloren gehen dürfen und die neuesten Daten relevanter sind. Neben „dropHead“ werden die Strategien „dropTail“ (neueste verwerfen) und „backPressure“ angeboten.

Durch Umstellen des Puffers auf die Strategie „backPressure“ geht in unserem Beispiel kein Element mehr verloren, da der Puffer erst dann weitere Elemente anfragt, wenn Pufferkapazitäten frei werden. Die Behandlung der „Überlast“ muss dann aber in einem vorigen Schritt erfolgen.

Interoperabilität

Verschiedene Reactive-Streams-Implementierungen können kombiniert und deren individuelle Vorteile so ausgenutzt werden. In Akka Streams kann eine Implementierungsklasse für Publisher (z. B. ein Reactor Stream oder ein Vert.x Reactive-WriteStream) verwendet werden, um eine Source zu erzeugen. Analog kann eine Klasse, die Subscriber implementiert, verwendet werden, um einen Sink zu erzeugen.

Im Beispiel erzeugen wir einen Sink mit einer eigenen Subscriber-Implementierung. Unser Subscriber fordert initial ein Element an (Zeile 11 in Listing 4). Bei Empfang eines Elements wird dieses ausgegeben. Wurde weniger als ein vorgegebenes Maximum an Elementen empfangen, so wird ein

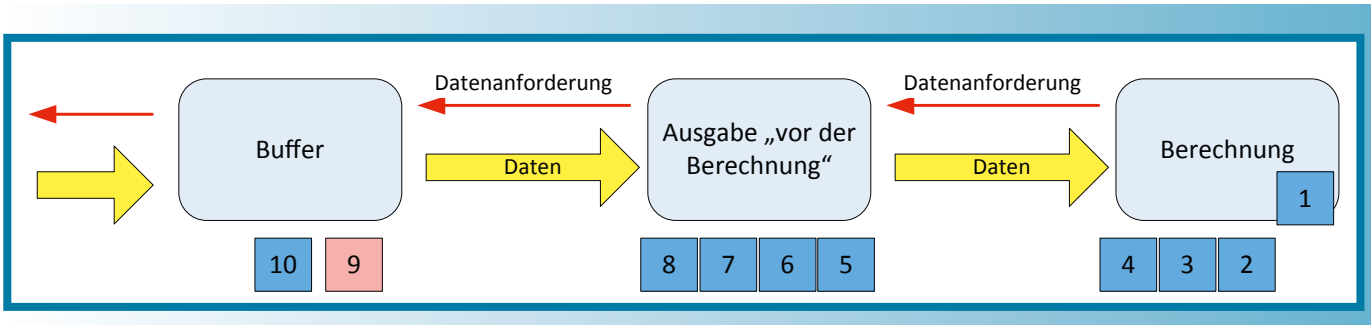


Abb. 4: Verarbeitung mit Backpressure

```
 4: Sink<Integer, BoxedUnit> sink=Sink.create(new Subscriber<Integer>(){
 5:   private Subscription sub;
 6:   private static final int MAX_RECEIVED = 3;
 7:   private int received = 0;
 8:
 9:   public void onSubscribe(Subscription sub) {
10:     this.sub = sub;
11:     sub.request(1);
12:   }
13:
14:   public void onNext(Integer i) {
15:     format("Verarbeitet: %d", i);
16:
17:     received++;
18:     if (received < MAX_RECEIVED) {
19:       sub.request(1);
20:     } else {
21:       sub.cancel();
22:     }
23:   }
24:   ... // weitere Implementierungen von onComplete, onError
          // nicht gezeigt
30: });
```

Listing 4: Stream mit eigener Subscriber-Implementierung

weiteres Element angefordert (Zeile 19), ansonsten wird die Subscription gecancelled (Zeile 21).

Die in diesem Beispiel absichtlich sehr einfach gehaltene Form der Flusskontrolle („Stop and Wait“) ist sehr ineffizient. Frameworks wie Akka Streams nehmen dem Nutzer die Dimensionierung der Datenanfragen ab, indem sie automatisch anhand der verbleibenden Pufferkapazitäten Daten anfordern. Die Dimensionierung der Puffer ist Aufgabe des Nutzers und muss entsprechend der erwarteten Mengengerüste und der fachlichen Anforderungen vorgenommen werden. Beispielsweise kann es bei Daten, die schnell veralten (z. B. ein einmal pro Sekunde gelieferter Messwert eines Sensors), sinnvoller sein, die Puffergröße klein zu halten und bei Verzögerungen in der Verarbeitung alte Daten zu verwerfen.

Abgrenzung zu Java Streams

Java implementiert seit Version 8 ebenfalls Streams mit dem neuen Collection-API [GüLe13]. Im Gegensatz zu Reactive Streams sind Java Streams aber zur Auswertungszeit in ihrer Größe beschränkt – mit der Abarbeitung der Collection ist die Stream-Verarbeitung beendet. Man spricht hier auch von „Cold Streams“. Reactive Streams unterstützen dagegen auch sogenannte „Hot Streams“, die kein definiertes Ende haben: beispielsweise ein Stream von UI-Events oder ein Realtime-Feed.

Fazit und Ausblick

Mit Frameworks wie RxJava und Reactor sind in den letzten Jahren Möglichkeiten entstanden, asynchrone Verarbeitung in Form eines Datenstroms durchzuführen. Reactive Streams führen nun ein standardisiertes Protokoll ein, dass die Interoperabilität zwischen Frameworks dieser Art ermöglicht. Der Standard ist dabei sehr schlank gehalten, leicht verständlich und wirkt in seiner ersten Version bereits sehr reif, da viele Framework-Hersteller an dem Standard mitgearbeitet haben. Reactive Streams füllen dabei insbesondere mit Backpressure eine Lücke im Bereich der Flusskontrolle und Überlastbehandlung, um die man sich bisher selbst kümmern musste.

Implementierungen von Reactive Streams sind technisch sehr unterschiedlich und reichen bis zu generischen, konfigurierbaren Frameworks wie Akka Streams. Zu der Komplexität einer asynchronen Verarbeitung kommt durch Pufferung und Flusskontrolle zusätzliche Komplexität hinzu, zumal diese Aspekte in den Implementierungen sehr unterschiedlich behandelt werden und für den Entwickler nicht direkt sichtbar sind. Trotz theoretischer Interoperabilität sollte man sich daher in einer Architektur auf wenige Implementierungen beschränken. Sinnvolle Kombinationen sind zum Beispiel das Zusammenspiel eines Treibers mit Reactive-Streams-Unterstützung (z. B. für MongoDB) mit einem generischen Framework wie Akka Streams.

Reactive Streams arbeiten bisher nur innerhalb einer JVM zusammen, nicht JVM-übergreifend. Diesen Aspekt adressiert reactive-streams-io [ReactiveStreamsIO], das sich in der Entstehung befindet.

Literatur

[Akka] http://www.akka.io

[AkkaStreams] http://doc.akka.io/docs/akka-stream-and-http-experimental/1.0/java.html

[GrLeSc14] R. Grammes, M. Lehmann, K. Schaal, Vert.x im Einsatz für hochskalierbare Architekturen, in: JavaSPEKTRUM, 05/2014. s. a. http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2014/05/grammes_lehmann_schaal_JS_05_14_gVcQ.pdf

[GüLe13] M. Günther, M. Lehmann, Lambda-Ausdrücke in Java 8, in: JavaSPEKTRUM, 03/2013, s. a. http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2013/03/guenther_lehmann_JS_03_13_va54.pdf

[ReactiveManifesto] http://www.reactivemanifesto.org/

[ReactiveStreams] http://www.reactive-streams.org/

[ReactiveSstreamsIO] https://github.com/reactive-streams/reactive-streams-io/

[ReactiveStreamsSpecification] https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.0/README.md#specification

[RxJava] https://github.com/ReactiveX/RxJava/wiki

[SourcenGithub] https://github.com/accso/reactive-streams-examples

[SpringReactor] http://projectreactor.io/docs/reference/

[Vert.x3] http://vertx.io/

Dr. Rüdiger Grammes ist seit November 2011 als Principal bei der Accso – Accelerated Solutions GmbH und dort als Softwarearchitekt in verschiedenen Projekten unterwegs.
E-Mail: grammes@accso.de

Dr. Kristine Schaal ist als Softwarearchitektin bei der Accso – Accelerated Solutions GmbH tätig. Sie arbeitet seit fast zwanzig Jahren in der Softwareentwicklung und ist in Projekten der Individualentwicklung für Kunden verschiedener Branchen unterwegs.
E-Mail: schaal@accso.de