Reaktive Architekturen

Leonard Thiele
Grundseminar Informatik Master
Department Informatik
Hochschule für Angewandte Wissenschaften Hamburg

28. Februar 2019

Mit zunehmenden Datenmengen und gewachsenen Bedürfnissen der Endnutzer werden auch die Anforderungen an moderne Software immer komplexer. In dieser Ausarbeitung sollen reaktive Architekturkonzepte vorgestellt werden. Diese bieten gegenüber imperativen Ansätzen oftmals Vorteile wie z.B. bei der Skalierung, Ausfallsicherheit und Wartbarkeit von Softwaresystemen.

1 Einführung

Die Komplexität von Software nimmt kontinuierlich zu. Das liegt vor allem daran, dass die Anforderungen an moderne Dienste stetig gewachsen sind. Das äußert sich sowohl bei der Entwicklung auf der Client- und Serverebene.

In der Anwendungsentwicklung findet sich eine Vielzahl komplexer Anforderungen über die reine Fachlichkeit hinaus. Die Programme sollen stets auf Nutzereingaben reagieren können und nicht für hintergründige Operationen blockieren. Auch kontinuierliche Kommunikation mit Hardwarekomponenten (Kamera, GPS etc.) und der Außenwelt (andere Clients, Server) sind keine Seltenheit. Diese Faktoren machen eine große Flexibilität der Anwendung erforderlich. Das imperative Programmiermodell mit seiner manuellen, sukzessiven Abarbeitung der anfallenden Events erscheint vor diesem Hintergrund nicht praktikabel. Die Implementation erfordert viel Code zur Verbindung der Komponenten. Die langfristige Wartung wird sehr aufwendig, da die gegenseitigen Aufrufe sehr komplex werden können.

In der Backend-Entwicklung sieht die Problematik ähnlich aus. Hier kommt es unter anderem darauf an, dass die Architektur gut skaliert. Herkömmliche, monolithische Architekturen sind in der Regel nicht in der Lage, bei einer großen Zahl von Clients akzeptable Antwortzeiten zu liefern. Die Verwaltung der einzelnen Verbindungszustände ist aufwendig, bremst das System und skaliert nicht. Ein synchroner Ansatz ist auch hier also eher hinderlich.

Eine Lösung für die oben genannten Probleme versprechen reaktive Architekturkonzepte. Hier ist die Kommunikation zwischen den einzelnen Komponenten asynchron und hochgradig flexibel organisiert. Diese Ausarbeitung soll einen Überblick geben, bei welchen Problemstellungen der reaktive Ansatz sinnvoll erscheint. Darauf folgend werden die Grundlagen reaktiver Systeme vorgestellt und zum Schluss ein Fazit zu deren Verbreitung gezogen.

2 Probleme synchroner Ansätze

In der Anwendungsentwicklung ist es seit längerem als „best practice“ angesehen, den Code für Geschäfts- und Oberflächenlogik sauber zu trennen. Das hat mehrere Vorteile: Zum einen können mehrere Teams ohne größere Interferenzen an verschiedenen Teilen der Software arbeiten. Zum anderen wird die Software besser wartbar und ist bei erfolgreicher Abstraktion in komplexeren Szenarien übersichtlicher. Zwischen den Komponenten erfolgt die Kommunikation in klassischen imperativen Programmieransätzen über Methodenaufrufe. Das kann schnell zu unübersichtlichen Kommunikationswegen und Abhängigkeiten führen und macht so die gewonnenen Vorteile zunichte.

In der Serverentwicklung spielen die verwendeten Entwurfsmuster ebenfalls eine große Rolle. Am Beispiel eines Webservers wird dies besonders deutlich. Klassische Webserver arbeiten die eingehenden Anfragen synchron ab. Um mehrere Anfragen gleichzeitig bedienen zu können, wird auf Multithreading zurück gegriffen. Dadurch ist das Programm begrenzt skalierfähig, denn für jede neue Anfrage kann ein neuer Thread gestartet werden, bis kein Arbeitsspeicher mehr zur Verfügung steht. Die Menge des vorhandenen Arbeitsspeichers lässt sich durch Threadpools noch effizienter nutzen. Hier werden Threads, die mit einer Anfrage fertig sind, direkt mit der nächsten betraut. Das führt dazu, dass der generelle Verwaltungsaufwand kleiner wird. Trotzdem stellen Threads zur Parallelisierung und Flexibilisierung von vielen gleichzeitigen Anfragen an einen Webserver keine Option dar.

3 Reaktiver Ansatz

Die Darstellung reaktiver Architekturen in dieser Ausarbeitung orientiert sich maßgeblich am Reaktiven Manifest [1]. Dieses Dokument wurde von Jonas Bonér, Dave Farley, Roland Kuhn, und Martin Thompson erstellt und durch viele Community-Beiträge ergänzt. Es trägt zusammen, wie moderne Softwaresysteme aufgebaut sein sollen, um als *reaktiv* zu gelten. Softwaresysteme sollen „stets antwortbereit, widerstandsfähig, elastisch und nachrichtenorientiert sein“ [1].

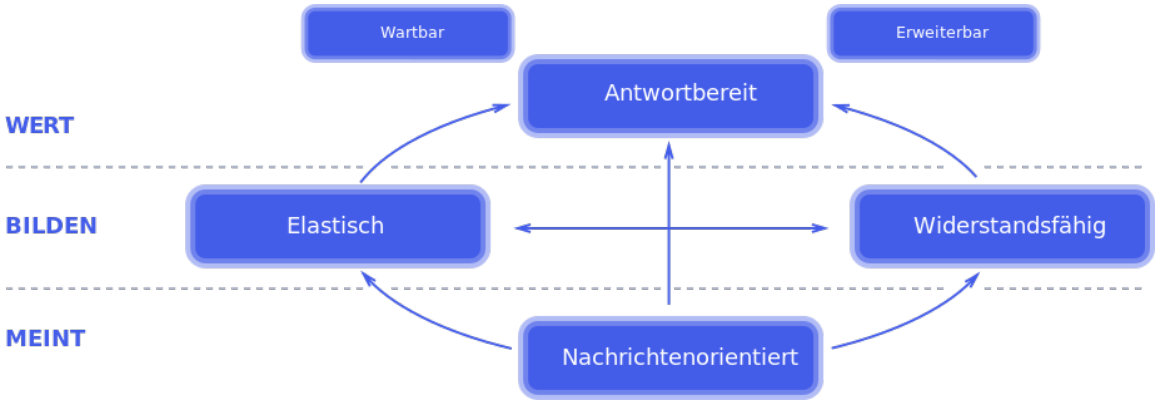

Abbildung 3.1: *Reaktive Merkmale*

Von **Antwortbereitschaft** *(engl. reponsiveness)* wird gesprochen, wenn die Systeme innerhalb von festgelegten Zeitlimits antworten. So lässt sich ein Ausfall bei Verlust einer Nachricht schnell feststellen. Dies führt zu vorhersehbarem und transparentem Verhalten des Systems. Der Nutzer weiß auch im Fehlerfall, ob von dem System noch eine Nachricht zu erwarten ist oder nicht und kann ggf. geeignete Maßnahmen ergreifen. Diese werden im nächsten Punkt erläutert.

Von **Widerstandsfähigkeit** *(engl. resilience)* ist die Rede, wenn die Systeme auch mit partiellen oder totalen Ausfällen umgehen können. Die Widerstandsfähigkeit lässt sich durch Konzepte wie der Replikation und Delegation von Services erhöhen. Bei der Replikation werden bestimmte Komponenten du-

pliziert und können im Fehlerfall füreinander einspringen. Je mehr einzelne Teilaufgaben an speziell dafür vorgesehene Komponenten delegiert werden, umso leichter wiegt ein einzelner Ausfall. Bei einem Ausfall wird das System von selbst und/oder durch eine vorgeschaltete Komponente neu gestartet und nimmt seinen Dienst wieder auf.

Elastizität *(engl. elasticity)* beschreibt die Eigenschaft eines Systems, auf zu bearbeitende Lastmengen von variablem Umfang reagieren zu können. Dazu darf es intern keine Flaschenhälse geben, welche die Bearbeitung großer Datenmengen beschränken. Weiterhin ist es von Vorteil, das System modular zu gestalten. Bei Bedarf lassen sich dann einzelne Module verteilen und/oder replizieren, um Engpässe auszugleichen.

Als letztes sollen reaktive Systeme **nachrichtenorientiert** *(engl. message driven* sein. Dieser Punkt ist von besonderer Wichtigkeit. Sämtliche Aufrufe innerhalb des Systems laufen asynchron über Nachrichten ab. Dazu bedarf es sehr klaren Schnittstellendefinitionen. An den Schnittstellen lässt sich das Verhalten des Systems sehr genau überwachen und Fehler werden schnell sichtbar. Dieses Kriterium beim Entwurf führt zu geringer Kopplung und hoher Kohäsion der einzelnen Komponenten. Diese können dann theoretisch sogar an getrennten Orten ausgeführt werden, was die Ausfallsicherheit erhöhen kann. Außerdem erleichtert dieses Entwurfsmodell die zuvor genannten Methoden Replikation und Delegation und somit die Skalierung eines reaktiven Systems.

4 Reaktive Umsetzungen und deren Verbreitung

Reaktive Architekturen und Programmierung fallen oftmals mit von funktionaler Programmierung inspirierten Konzepten zusammen. In der Anwendungsentwicklung ist es nicht unüblich mit sog. *Callbacks* zu arbeiten. Diese werden beim Aufruf einer entfernten Komponente übergeben und nach deren Ausführung abgearbeitet. So entsteht zwar reaktiver Code, bei mehreren geschachtelten Aufrufen mit deren Callbacks wird es aber schnell sehr unübersichtlich. Moderne Frameworks wie die ReactiveX-Familie [2] bieten daher sog. *Streams*, mit deren Hilfe sich Komponenten deutlich zielgerichteter vernetzen lassen. Streams arbeiten Event-basiert und können einen oder mehrere Erzeuger sowie Verbraucher haben. Über die Events werden Informationen zwischen den einzelnen Programmteilen ausgetauscht, ohne dass diese direkte Kenntnis voneinander haben müssen. Viele Multiparadigma-Sprachen wie Java oder Ruby bieten inzwischen auch ohne die Einbindung externer Libraries die Nutzung von Streams an. So kommen auch Konzepte aus funktionalen Sprachen zunehmend im Mainstream der Entwicklungswerkzeuge an.

Auf der Serverseite gibt es verschiedene fertige Frameworks, um die genannten reaktiven Eigenschaften umzusetzen. Zur zustandslosen Kommunikation zwischen verschiedenen Services beispielsweise bieten sich Protokolle wie *REST* an, auf die hier nicht weiter eingegangen werden soll.

Trotz der offensichtlichen Vorteile ist pures funktionales Programmieren bei der Entwicklung großer Softwarearchitekturen nicht besonders stark verbreitet. Das hat mehrere Gründe. In der Ausbildung wird oft mit objektorientierter Programmierung begonnen, da dieses eher der menschlichen Sicht auf die Welt entspricht. Das Formulieren von Problemen und Lösungen im funktionalen Kontext erfordert denn zu einem späteren Zeitpunkt eine stärkere Umgewöhnung gewohnter Denkmuster. Damit einher geht auch das Problem, dass funktionale

Sprachen trotz ihres teilweise beachtlichen Alters nicht zu den populärsten am Markt gehören [3]. Da reaktive Entwurfsmuster und Modelle erst in jüngerer Zeit mit der Einführung in populäre Sprachen und Frameworks an Relevanz gewinnen, sind Einstiegshilfen im Moment noch seltener in hoher Qualität zu finden als z.B. für imperatives oder objektorientiertes Programmieren. Gerade die Möglichkeit, sich schnell in eine Technologie einarbeiten zu können, ist aber oft ein wichtiges Entscheidungskriterium.

5 Zusammenfassung & Ausblick

Reaktive Konzepte haben bei der Planung von Software-Architekturen für Client- wie auch Server-Anwendungen oftmals entscheidende Vorteile. Sie stellen eine ernst zunehmende und oftmals elegantere Lösung als imperatives oder objektorientiertes Programmieren dar. Trotzdem haben sie nach wie vor keine große Verbreitung gefunden, obwohl funktionales Programmieren und reaktive Konzepte keine wirkliche Neuheit darstellen. Mit dem Einzug von reaktiven Konzepten in viele oft genutzte Sprachen ist aber eine Verbreitung in Entwicklerkreisen zu erwarten.

Literatur

[1] Jonas Bonér u. a. *Das Reaktive Manifest*. 25. Feb. 2019. URL: https://www.reactivemanifesto.org/de.

[2] o.V. *ReactiveX*. 27. Feb. 2019. URL: http://reactivex.io/.

[3] TIOBE Software BV. *TIOBE Index*. 27. Feb. 2019. URL: https://www.tiobe.com/tiobe-index/.

[4] André Staltz. *The introduction to Reactive Programming you've been missing*. 25. Feb. 2019. URL: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754.

[5] Michael Menzel. *Reaktive Architekturen mit RxJava*. 25. Feb. 2019. URL: https://blog.senacor.com/reaktive-architekturen-mit-rxjava/.

[6] Roland Kuhn, Brian Hanafee und Jamie Allen. *Reactive Design Patterns*. 2016.

[7] Uwe Friedrichsen, Stefan Toth und Eberhard Wolff. *Resilience - Wie Netflix sein System schützt*. 2015.

[8] Debasish Ghosh. *Functional and Reactive Domain Modeling*. 2016.