MICROLATION - EDSL zum Simulieren von komplexen Microservice-Anwendungen zur Bewertung ihrer Resilienz

Bjarne Valentin Rentz[1]

Abstract: Microservice-Anwendungen müssen aufgrund der Kommunikation über das Netzwerk mit neuen Fehlerquellen im Vergleich zu klassischen Anwendungen umgehen können. Dafür gibt es verschiedene Entwurfsmuster (Retry, Timeout etc.), die unterschiedliche Einsatz- und Konfigurationsmöglichkeiten bieten. Es ist jedoch schwer für konkrete Anwendungen die optimale Verwendung und Konfiguration dieser herauszufinden. Deshalb stellt diese Arbeit eine Embedded Domain Specific Language (EDSL) vor, durch welche Microservice-Anwendungen definiert und simuliert werden können, um das eben genannte Problem zu lösen. MICROLATION ermöglicht damit die Erforschung der Muster und ihrer Konfigurationen im Rahmen von Microservices.

Keywords: Microservices; Simulation; Resilienz; EDSL

1 Einleitung

Für Microservice-Architekturen besteht ein aktuelles Interesse in der Industrie [Zi17] und Forschung [Ja18]. Als verteilte Systeme bringen Microservices neue Herausforderungen mit sich. Eine davon ist die Kommunikation über das Netzwerk und die damit verbundene Fehlerbehandlung. Einzelne Microservices können ausfallen oder nicht erreichbar sein. Auch können Aufrufe verloren gehen [Ja18; Ne15]. Im Sinne der *Antifragilen Organisation* [Ts13], sollten Organisationen daher von Anfang an mit Fehlern rechnen und ihr System dementsprechend planen und realisieren. Eine Antwort auf die Fehlerquellen sind Entwurfsmuster zur Erhöhung der Resilienz im Netzwerkverkehr.

Bei Betrachtung einer Microservice-Anwendung, die verschiedene Muster zur Erhöhung der Resilienz in der Kommunikation verwendet, ergibt sich folgende Fragestellung: Wie lässt sich die optimale Konfiguration für das System und insbesondere die der verwendeten Muster finden, um die Ausfallsicherheit respektive Resilienz zu maximieren? Ein Beispiel hierfür ist eine Anwendung mit zwei Microservices, wobei Microservice *A* Microservice *B* mit einem Retry-Muster aufruft , um eine Geschäftsfunktionalität zu implementieren. Hierbei stellt sich die Frage, wie die Anzahl an erneuten Versuchen und Wartezeit konfiguriert werden sollte, um eine hohe Resilienz zu erhalten.

Diese Arbeit zeigt einen simulationsbasierten Ansatz zum Bewerten verschiedener Entwurfsmuster und ihrer Konfiguration, welche die Resilienz im Netzwerkverkehr verbessern.

[1] NORDAKADEMIE, Angewandte Informatik, Köllner Chaussee, 25337 Elmshorn, Deutschland bjarnerentz@live.de

Die Simulation basiert auf einer EDSL in C#, durch welche die Systeme programmatisch definiert und simuliert werden können. In Listing 1 ist eine beispielhafte Definition zu sehen.

```
var ms1 = new Microservice();
var ms2 = new Microservice{
    Routes = {
        new Route{ Url = "Users", Faults = new TimeoutFault()}
    }
};
var policy = new Retry();
var call = ms1.call(ms2,
    new CallConfig{ Route = "Users", Policies = policy, Interval = TimeSpan.FromSeconds(1)});
```

Listing 1: Beispielhafte Definition von zwei Microservices durch *MICROLATION*

Hierbei werden folgende Themen behandelt:

- Welche Probleme bringen verbreitete Muster mit sich und welcher Lösungsansatz resultiert daraus (Abschnitt 2)?
- Wie können mithilfe von MICROLATION Microservices, ihre Aufrufe und Rückgabetypen und -werte definiert und simuliert werden können (Abschnitt 3)?
- Wie funktioniert MICROLATION und wie kann es erweitert und genutzt werden (Abschnitt 4)?
- Welche Ergebnisse können mit MICROLATION erzielt werden (Abschnitt 5) und wie steht dieser Ansatz im Vergleich zu anderen Arbeiten (Abschnitt 6)?

2 Problemstellung

Für die Erhöhung der Resilienz im Netzwerkverkehr existieren verschiedene Muster. In der Literatur wird dabei häufig auf die Muster aus *Release It* [Ny07] wie Retry, Timeout oder Circuit Breaker Bezug genommen. Wie in der Einleitung gezeigt wurde, bringen Muster für eine erhöhte Resilienz in der Netzwerkkommunikation mehrere Unbekannte mit sich. Jedes hat eigene Konfigurationsmöglichkeiten und unterschiedliche Aufgabenbereiche. Insbesondere komplexere Muster wie Circuit Breaker können oftmals in der Anzahl an Fehlern bis zum Auslösen, der Zeit, bis sie in den nächsten Zustand fallen, oder auch die Fehler, auf welche sie reagieren, konfiguriert werden. Zusätzlich können die Entwurfsmuster kombiniert werden, sodass beispielsweise Retry und Circuit Breaker verschachtelt werden. Dadurch ergibt sich eine hohe Anzahl an möglichen Kombinationen zwischen den Mustern selbst und entsprechenden ihren Konfigurationen, aus denen die möglichst optimale gefunden werden soll. Fachliche Aspekte und Vorgaben wie maximales Alter der angezeigten Daten können die Komplexität zusätzlich erhöhen. Forschung und Industrie stehen beide vor dem Problem, wie effektiv eine optimale bzw. gut passende Kombination aus Mustern und

Konfigurationen für den konkreten Einsatz gefunden werden kann. Aufgrund der daraus resultierenden hohen Komplexität verfolgt diese Arbeit einen simulationsbasierten Ansatz.

3 MICROLATION

MICROLATION ermöglicht es, mithilfe einer EDSL zuerst Microservice-Anwendungen mit verschiedenen Entwurfsmustern sowie Fehlerquellen zu definieren und anschließend das Gesamtsystem zu simulieren. Der Fokus der Bewertung liegt auf der Resilienz des Systems. Als EDSL besteht MICROLATION aus Klassen, die bereitgestellt werden und durch Interaktionen miteinander die benötigten Funktionalitäten implementieren. Die Details der Implementierung und Erstellung der EDSL werden in Abschnitt 4 behandelt. Die Funktionsweise wird im Folgenden anhand eines Beispiels erläutert.

Eine Microservice-Anwendung bestehend aus zwei Microservices: Der *Aufrufer* Microservice ruft unter Verwendung des Retry-Musters den *Ziel* Microservice auf. Mithilfe von MICROLATION soll nun eine gut geeignete Anzahl an erneuten Versuchen herausgefunden werden. Um den Retry auszulösen, werfen 10 % der Aufrufe des Ziels einen Fehler. Die zu durchsuchende Lösungsmenge ist die Anzahl an erneuten Versuchen von 0 bis 10.

```
1  int[] retryCounts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2
3  var caller = new Microservice("Caller");
4  var target = new Microservice("Target")
5  {
6      Routes =
7      {
8          new Route<int> { Url = "Target", Value = () => 1,
9              Faults = MonkeyPolicy.InjectException(with => with.Fault(new Exception())
10                 .InjectionRate(0.1)
11                 .Enabled())
12                 .AsPolicy<int>() }
13     }
14 };
15 var call = caller.Call(target, new CallOptions<int>
16 {
17     Route = "Target",
18     Interval = _ => TimeSpan.FromMilliseconds(500)
19 });
20 var simulation = new Simulation()
21 {
22     Microservices = { caller, target }
23 };
```

Listing 2: Definition einer Microservice-Anwendung in *MICROLATION*

In Listing 2 ist die Definition besagter Microservice-Anwendung zu sehen. Zuerst wird die Lösungsmenge (Zeile 1) und der aufrufende Microservice (Zeile 3) definiert. In den

Zeilen 4–14 wird der *Ziel* Microservice definiert. Dieser hat eine Route unter der URL *Target*, welche immer den Wert 1 zurückgibt (Zeile 8). Mithilfe der Zeilen 9–12 wird dafür gesorgt, dass 10 % der Aufrufe fehlschlagen, wodurch das Retry-Muster ausgelöst wird. Anschließend wird der Aufruf zwischen den beiden Microservice erstellt (Zeilen 15–19). Dafür stellt die Microservice-Klasse die *Call*-Methode bereit, welche als Parameter den aufzurufenden Microservice und *CallOptions* entgegennimmt. Die CallOptions definieren die aufzurufende Route (Target) und das Intervall wie oft der Aufruf ausgeführt werden soll, in diesem Fall beträgt es 500 Millisekunden. Die Definition der Anwendung wird durch die Erstellung einer Simulation (Zeilen 20–23) abgeschlossen. Dieser werden die zu simulierenden Microservices übergeben.

```
1 var results = new Dictionary<int, List<CallResult>>();
2 foreach (var retryCount in retryCounts)
3 {
4     call.CallOptions.Policies = Policy<int>.Handle<Exception>().Retry(retryCount);
5     var result = await simulation.Run(TimeSpan.FromSeconds(60));
6     results.Add(retryCount, result.First().Value);
7 }
```

Listing 3: Sequenzielle Ausführung der Simulationen durch *MICROLATION*

Die anschließende Ausführung der Simulation ist in Listing 3 zu sehen. Zuerst wird in Zeile 1 ein Dictionary zum Speichern der Ergebnisse erstellt. Als Schlüssel wird die jeweils verwendete Anzahl an erneuten Versuchen verwendet. Anschließend wird für jede zuvor definierte Anzahl an erneuten Versuchen eine Richtlinie – in diesem Fall das Retry-Muster – erstellt und dem Aufruf zugewiesen (Zeile 4). Daraufhin wird die Simulation für 60 Sekunden ausgeführt (Zeile 5) und das Ergebnis dem Dictionary hinzugefügt (Zeile 6). MICROLATION ermöglicht auch eine parallele Ausführung verschiedener Simulationen. Aufgrund der Einfachheit ist dieses Beispiel sequenziell.

Eine exemplarische Auswertung der Simulation ist in Tabelle 1 zu sehen. Dafür wurde für jede Anzahl an erneuten Versuchen die durchschnittliche Aufrufzeit und die Anzahl an Fehlern, die der aufrufende Microservice erhalten hat, bestimmt. Dabei ist deutlich zu erkennen, wie die Anzahl an erhaltenen Fehlern stark abnimmt und ab zwei erneuten Versuchen keine Fehler mehr erhalten wurden. Die durchschnittliche Verbindungszeit nimmt ebenfalls zuerst stark ab und erreicht im Bereich von 0,04ms ein Plateau. Die Aussagekraft der Ergebnisse sind jedoch fraglich, da die durchgeführte Simulation und Auswertung nicht für eine Bewertung ausreicht.

# Erneute Versuche	Durchschnittliche Aufrufzeit	# Fehler
0	0,128ms	13
1	0,0533ms	1
2	0,0457ms	0
3	0,0451ms	0
4	0,0428ms	0
5	0,0398ms	0
6	0,0412ms	0
7	0,0471ms	0
8	0,0455ms	0
9	0,0443ms	0
10	0,0375ms	0

Tab. 1: Ergebnisse des Beispiels

Der primäre Verwendungszweck ist die Simulation der Netzwerkaufrufe von Microservice-Anwendungen. Diese können flexibel definiert werden, wobei Entwurfsmuster zur Erhöhung der Resilienz eingesetzt werden können. Muster können frei konfiguriert werden sodass einer der Zwecke die gezielte Bewertung von unterschiedlichen Konfigurationen ist, wie im Beispiel gezeigt wurde. Ebenso kann der Einsatz von verschiedenen Mustern bzw. die Kombination dieser bewertet werden. Da die Microservices flexible definiert werden können, ist es ebenfalls möglich verschiedene Architekturen zu vergleichen. Zusammengefasst soll durch MICROLATION die Bewertung verschiedener Microservice bezogener Aspekte wie der Einsatz von Entwurfsmustern, Konfiguration und Architektur ermöglichen werden. Somit kann MICROLATION auch in der Forschung und Industrie für verschiedene Einsatzmöglichkeiten verwendet werden. MICROLATION ist keine eigenständige Sprache. Als eingebettet Sprache bietet sie explizit die Möglichkeit weitere Auswertungen sowie Automatisierungen in C# zu implementieren.

4 Details

Dieser Abschnitt behandelt die technischen Details. Die Implementierung von MICROLATION orientiert sich an den Richtlinien für die Erstellung von DSLs von *Karsai et al.* [Ka14]. Nach der Implementierung wird zudem auf die Erweiterbarkeit eingegangen. Der Sourcecode ist auf Github[2] zu finden.

[2] https://github.com/BjarneRentz/Microlation

4.1 Implementierung

Zuerst wurden anhand des Verwendungszwecks die Schlüsselwörter der Sprache abgeleitet und als Klassen implementiert. Die Funktionalitäten von MICROLATION sind folglich durch die Interaktionen der Klassen untereinander realisiert. Bei den Schlüsselwörtern handelt es sich um `Simulation`, `Microservice`, `Route`, `CallConfig`, `Call` und `CallResult`. Ihre genaue Bedeutung sowie die Interaktionsmöglichkeiten werden im Folgenden behandelt.

Die Simulation dient als zentrale Klasse dem zentralen Ausführen einer Simulation über die `Run`-Methode. Die Methode gibt ein Dictionary zurück, mit dem jeweiligen `Call` als Schlüssel und einer Liste vom Typ `CallResult` als Wert. Somit kann jeder Aufruf eigenständig ausgewertet werden. Dafür werden der Simulation alle zu simulierenden Microservices übergeben.

Die Microservice-Klasse ist ein zentrales Element von MICROLATION. Fachlich repräsentiert sie einen zu simulierenden Microservice. Die Klasse kann zudem Routen und Calls definieren. Über die `Simulate`-Methode werden alle Calls des Microservices für die angegebene Dauer parallel ausgeführt, sodass Calls sich nicht untereinander beeinflussen. Aufrufe bzw. Calls zwischen zwei Microservices können über die `Call`-Methode der Microservice-Klasse erstellt werden. Diese nimmt als Parameter den aufzurufenden Microservice, sowie die `CallOptions` entgegen und erstellt anhand dieser einen Call welcher einer internen Liste hinzugefügt und ebenfalls als Rückgabewert zurückgegeben wird. Die Rückgabe ermöglicht es Calls manuell aufzurufen, sie als Schlüssel für die Ergebnisse der Simulation zu verwenden und das Erweitern des Calls um Validatoren. Wie in Listing 4 zu sehen können diese einem Call über Fluent-Ausdrücke hinzugefügt werden (Zeile 8). Dies ermöglicht die Verkettung mehrere Validate-Ausdrücke. In dem Beispiel ist der erhaltene Wert des Calls valide, wenn er größer als null ist.

```
1 var callWithoutPolicies = ms1
2     .Call(ms2,
3         new CallOptions<int>
4         {
5             Route = "Id",
6             Interval = _ => TimeSpan.FromMilliseconds(Random.Shared.Next(100, 700))
7         })
8     .Validate(v => v > 0);
```

Listing 4: Beispielhafte Verwendung von Validatoren

Ein weiterer elementarer Bestandteil ist die Klasse `Route`. Wie bereits erwähnt repräsentiert sie einen Endpunkt bzw. eine Route eines Microservices, die aufgerufen werden kann. Routen haben ein Rückgabewert und sind typisiert, damit jede Route einen unterschiedlichen Rückgabetypen haben kann. Damit sie trotzdem innerhalb einer Liste verwaltet werden können, implementieren sie das Interface `IRoute`. Routen verfügen zudem über die ebenfalls typisierten Attribute `Value` und `Faults`. Ersteres ist eine Funktion, welche den Rückgabewert

der Route zurückgibt. Damit ist es beispielsweise möglich zufällige Werte für jeden Aufruf der Route zu definieren. Faults sind optional und ermöglichen das injizieren von Fehlern bei Aufrufen der Route. Auf die genaue Funktionsweise wird im Rahmen von Calls detaillierter eingegangen.

CallOptions dienen lediglich der Sammlung von Parametern für den Call. Sie beinhalten die `URL` der aufzurufenden Route, eine Funktion `Interval`, welche abhängig von der Iteration die Zeit zwischen zwei Aufrufen zurückgibt und die `Policies` (Richtlinien), welche der Aufruf nutzt. Wie die Faults werden sie im Folgendem detaillierter erläutert.

CallResults repräsentieren das Ergebnis eines Aufrufs. Sie beinhalten die Dauer des Aufrufs (`ConnectionTime`), ob der Aufruf ein valides Ergebnis geliefert hat (`Valid`) und – falls aufgetreten – eine Ausnahme (`Exception`). Die Verbindungszeit ist eine gängige Metrik in Verschiedenen Arbeiten [Me20; ZGD19] und ermöglicht zudem eine effektive Bewertung des Fail-Fast-Ansatzes. Durch die Ausnahmen und Validierung wird zudem eine inhaltliche Bewertung ermöglicht.

Calls sind das Gegenstück von Routen und repräsentieren einen Aufruf dieser. Zudem beinhalten sie den simulativen Part von MICROLATION und führen die Aufrufe aus, die sie repräsentieren. Deshalb sind sie wie die Routen ebenfalls typisiert. Zudem existiert analog zu den Routen das Interface `ICall`, um sie trotz Typisierung als Schlüssel für das Dictionary zu verwenden, welches von der `Simulate`-Methode der Microservice-Klasse als Ergebnis zurückgeben wird. Über die `Execute`-Methode lässt sich der Call ausführen. Diese Methode baut zunächst die Aufrufkette über die Policies und Faults auf. Policies stehen für die eingesetzten Entwurfsmuster wie beispielsweise Retry oder Timeout. Faults hingegen verfolgen den diametralen Zweck und erzeugen künstliche Fehler wie Latenz, Ausnahmen oder anderer Rückgabewerte. MICROLATION nutzt dafür die Bibliothek *Polly*[3], welche verschiedene der Muster bereits implementiert sowie über Interfaces verfügt, um neue Muster zu erstellen. Für die Fehler existiert die auf Polly basierende Bibliothek *Simmy*[4], welche ebenfalls verschiedene Muster bereitstellt. Dadurch sind zum einen bereits gängige Muster wie Retry, Timeout oder Circuit Breaker implementiert. Zum anderen kann MICROLATION durch die Interfaces mit eigenen Mustern erweiter werden.

In Listing 5 ist die erste Hälfte der `Execute`-Methode zu sehen. Zuerst wird in Zeile 1 der bereits erwähnte Intervall abgewartet, bevor mit der Ausführung fortgeschritten wird. In den Zeilen 5 bis 9 wird der Aufruf aufgebaut. Da sowohl `Policies` als auch `Faults` optional sind, ergeben sich verschiedene Konstellationen für den Aufbau der Aufrufkette. Die in Zeile 9 verwendete Methode `Policy.Wrap` ermöglicht das verschachteln mehrerer Instanzen von `ISyncPolicy`. Dadurch ist dafür gesorgt, dass die Fehler innerhalb der Richtlinien ausgeführt werden und somit entsprechend abgefangen werden können.

[3] `https://github.com/App-vNext/Polly`
[4] `https://github.com/Polly-Contrib/Simmy`

```
await Task.Delay(CallOptions.Interval(iteration), token);
watch.Reset();

// Build the callChain based on the provided policies and faults.
ISyncPolicy<T>? callChain = null;
if (CallOptions.Policies != null)
    callChain = CallOptions.Policies;
if (TypedRoute.Faults != null)
    callChain = callChain == null ? TypedRoute.Faults : Policy.Wrap(callChain,
    ↪ TypedRoute.Faults);
```

Listing 5: Aufbau der Aufrufkette in der `Execute`-Methode

Die Durchführung des Aufrufs ist in Listing 6 abgebildet. Innerhalb eines Try-Catch-Blocks wird die Stoppuhr gestartet (Zeile 4) und anschließend der Aufruf ausgeführt. Dafür wird zunächst überprüft, ob die Aufrufkette existiert. Falls ja wird über diese die `Value`-Methode der Route aufgerufen. Andernfalls wird die Methode direkt aufgerufen, um das Ergebnis zu erhalten (Zeile 5). Daraufhin wird die Stoppuhr gestoppt und das Ergebnis validiert, falls Validatoren vorhanden sind. Sind keine vorhanden wird das `Valid`-Feld von `CallResult` auf `false` gesetzt. Tritt eine Ausnahme während des Aufrufs ein, wird diese gefangen (Zeilen 10–14). In diesem Fall wird ebenfalls die Stoppuhr gestoppt. Anstelle des `Valid`-Felds wird die Ausnahme dem Ergebnis hinzugefügt. Abschließend wird das Ergebnis zurückgegeben. Durch die Ausführung der Simulation auf der Aufruf-Ebene ist garantiert, dass Calls sich nicht untereinander beeinflussen. Zudem ist es dadurch auch möglich, auf jeder Ebene manuell Simulationen sowie Änderungen durchzuführen, wie bereits in dem Beispiel gezeigt werden konnte.

```
var result = new CallResult();
try
{
    watch.Start();
    var value = callChain != null ? callChain.Execute(() => TypedRoute.Value()) :
    ↪ TypedRoute.Value();
    watch.Stop();
    result.CallDuration = watch.Elapsed;
    result.Valid = Validators.Any() && Validators.Aggregate(true, (curr, next) => curr &&
    ↪ next(value));
}
catch (Exception e)
{
    watch.Stop();
    result.Exception = e;
    result.CallDuration = watch.Elapsed;
}
return result;
```

Listing 6: Durchführung des Aufrufs in der `Execute`-Methode

4.2 Erweiterbarkeit

Insbesondere im Bereich der Policies sowie Faults ist MICROLATION durch die Verwendung von Polly respektive Simmy und deren Interfaces erweiterbar. Dies wird dadurch bestärkt, dass bereits verschiedene Erweiterungen für Polly existieren[5]. Für die Auswertung lassen sich zudem verschiedene Bibliotheken wie LINQ von Microsoft verwenden, da es sich bei CallResult um ein Plain Old Class Object (POCO) handelt.

5 Evaluation

Für eine abschließende Evaluation soll mithilfe von MICROLATION folgende Problemstellung gelöst werden: Ein Aufruf eines anderen Microservices verwendet die Muster Retry und Timeout, um die Resilienz zu erhöhen. Welche Konfigurationen erreichen abhängig zur Erreichbarkeit des aufzurufenden Microservices die besten Ergebnisse? Ähnlich wie in der Arbeit [Me20] sollen die Muster bzw. ihre Konfiguration erforscht werden.

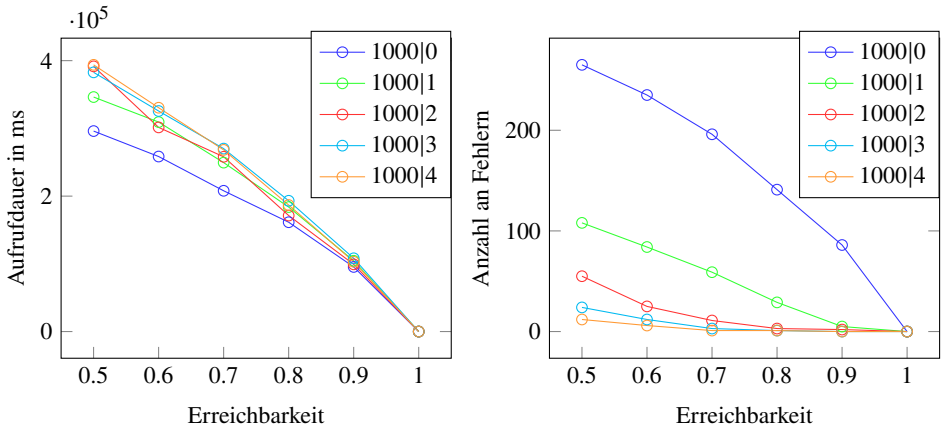
Für die Beantwortung werden zwei Microservices definiert, wobei einer eine entsprechende Route zur Verfügung stellt. Dieser wird von dem anderen Microservice unter Verwendung von Retry und Timeout aufgerufen. Dabei ist Timeout in Retry verschachtelt. Dadurch reagiert der Retry auf Fehler, die der Timeout wirft. Der Aufruf wird alle 500 Millisekunden ausgeführt. Für die Simulation der Erreichbarkeit, wird dem aufgerufenem Microservice eine Latenz von 4 Sekunden injiziert. Die Wahrscheinlichkeit, mit welcher die Latenz injiziert wird, ergibt sich durch folgende Berechnung: $1 - \textit{Erreichbarkeit}$, wobei *Erreichbarkeit* ein Wert zwischen 0 und 1 ist und die Erreichbarkeit in Prozent angibt.

Dieses System wird mit allen Kombinationen aus Timeout in Millisekunden (1000, 1500, 2000, 2500, 3000), Anzahl an erneuten Versuchen (0–4) und Erreichbarkeiten (50 %, 60 %, 70 %, 80 %, 90 %, 100 %) für zehn Minuten simuliert. Die Erhebung der Metriken beschränkt sich aufgrund des begrenzten Umfangs auf die gesamte Dauer der Aufrufe und wie viele Fehler der Aufrufer erhalten hat. Der Sourcode für die Evaluation ist ebenfalls über Github bereitgestellt[6].

In Abbildung 1 sind für fünf der 150 simulierten Konfigurationen die gesamte Aufrufdauer (Abbildung 1a) und die Anzahl an erhaltenen Fehlern (Abbildung 1b) zu sehen. In der Legende gibt der erste Wert den konfigurierten Timeout und der zweite Wert die Anzahl an erneuten Versuchen an. Abgebildet sind fünf Konfigurationen mit gleichem Timeout (1000 ms) aber unterschiedlicher Anzahl an erneuten Versuchen (0–4). In Abbildung 1a ist zu sehen, wie sich eine erhöhte Anzahl an erneuten Versuchen negativ auf die Gesamtdauer aller Aufrufe auswirkt. Ebenfalls ist zu erkennen, dass bei sinkender Erreichbarkeit die Differenz steigt. Bei der Durchführung mit zwei erneuten Versuchen ist auffällig, dass bei

[5] https://github.com/Polly-Contrib

[6] https://github.com/BjarneRentz/Microlation/blob/main/Examples/SampleEvaluation.cs



(a) Gesamtdauer der Aufrufe

(b) Anzahl an erhaltenen Fehlern

Abb. 1: Ausschnitt der Ergebnisse von fünf Konfigurationen

einer Erreichbarkeit von 60 % die Aufrufdauer geringer ist als bei einem erneuten Versuch. Im Bereich der erhaltenen Fehler (Abbildung 1b) fällt auf, dass die Graphen – im Gegensatz zur Aufrufdauer – unterschiedliche Formen je nach Konfiguration annehmen. Während der Graph bei der Durchführung ohne erneuten Versuch bei steigender Erreichbarkeit zunehmend stärker fällt, ist bei einem erneuten Versuch eine lineare Entwicklung zu sehen. Bei zunehmenden erneuten Versuchen wird der Effekt bei besserer Erreichbarkeit stetig weniger. Ebenfalls sind die Differenzen zwischen den Konfigurationen stärker ausgeprägt, nimmt jedoch bei zunehmender Anzahl an erneuten Versuchen ab.

Ein Vergleich der Ergebnisse ist aufgrund mangelnder Arbeiten in diesem Bereich schwierig. Die Arbeit von [Me20] bestätigt trotz unterschiedlicher Herangehensweise und Aufbau der Microservice-Anwendung die Tendenzen: Der Einsatz von Mustern erhöht die Dauer der Aufrufe und verringert die Anzahl an auftretenden Fehlern. Daher sollte prinzipiell eine Konfiguration verwendet werden, welche auch die fachlichen Anforderungen erfüllt. Beispielsweise hat ein erneuter Aufruf einen geringeren Einfluss auf die Zeit, aber einen vergleichsweise hohen Einfluss auf die erhaltenen Fehler.

Durch die Evaluation wird deutlich, dass sich durch MICROLATION auch Kombinationen von unterschiedlichen Mustern und Ausgangssituationen simulieren lassen. Durch den programmatischen Ansatz können die Ergebnisse zudem automatisiert gesammelt und verarbeitet werden. Dies ist insbesondere bei größeren Datenmengen hilfreich, die durch die Kombinatorik entstehen. Im Rahmen der Evaluation sind zudem Limitationen deutlich geworden. Derzeit existiert keine Möglichkeit, verkette Aufrufe abzubilden. Auch die Möglichkeit, die Ergebnisse zu validieren, ist in ihrer Auswirkung begrenzt. Da die Validierung erst nach dem Ausführen des Aufrufs durchgeführt wird, können die eingesetzten Muster nicht darauf reagieren.

6 Verwandte Arbeiten

Die bereits erwähnte Arbeit von *Mendonça et al.* [Me20] behandelt die Bewertung verschiedener Konfiguration unterschiedlicher Entwurfsmuster anhand eines stochastischem Modells und unterscheidet sich damit in der Herangehensweise. Zudem verfolgt diese Arbeit das Ziel, ein Framework zur Untersuchung von Mustern zu erarbeiten und nicht die konkrete Bewertung dieser. Die Simulation von Microservice-Anwendungen wird ebenfalls in [GIM17] behandelt. Die Arbeit hat jedoch das Ziel, den Einfluss der Anwendung auf die Performance und daraus resultierenden Hardwareabhängigkeiten zu untersuchen. Einen ähnlichen Ansatz wie MICROLATION verfolgt *μqSim* [ZGD19]. Dies erlaubt es, komplexe Microservice-Anwendungen in ihrem kompletten Verhalten in Bezug auf Antwortzeiten zu simulieren. Jedoch werden die Microservices dafür über eine JSON-Datei definiert und anschließend durch μqSim simuliert. MICROLATION bietet im Vergleich einen programmiertechnischeren Ansatz mit Fokus auf die Bewertung der Resilienz und nicht dem Latenzverhalten der gesamten Anwendung.

7 Fazit

Die Simulation von Microservice-Anwendungen ist ein breites Gebiet mit verschiedenen Ansätzen und Zielen. In dieser Arbeit wurde als Ansatz eine EDSL vorgestellt, die eine programmatische Definition und Simulation der Microservice-Anwendung erlaubt. Der Fokus liegt dabei auf dem Netzwerkverkehr zwischen den Microservices, wobei verschiedene Entwurfsmuster mit unterschiedlichen Konfigurationen zur Erhöhung der Resilienz eingesetzt und simuliert werden können. MICROLATION ermöglicht somit eine Erforschung bestehender und neuer Entwurfsmuster, die die Resilienz des Netzwerkverkehrs von Microservice-Anwendungen erhöhen sollen. Dazu zählt insbesondere auch, welchen Einfluss verschiedene Konfigurationen für bestehende Muster haben und welchen Einfluss die Kombination von unterschiedlichen Mustern hat.

Die zukünftige Arbeit sollte sich darauf konzentrieren, die aktuellen Limitation aufzuheben. Die Validierung kann durch eine injizierte Funktion in den Call, welche Ausnahmen wirft und damit die Muster auslöst, erweitert werden. Eine Implementierung von verketten Aufrufen, würde eine realitätsnahe Simulation von Microservice-Anwendungen ermöglichen, ist jedoch in der Umsetzung aufgrund der aktuellen Realisierung der Calls komplex. Der Bereich der Auswertung sollte durch eine standardmäßige Evaluation erweiter werden. Diese sollte die Ergebnisse der einzelnen Aufrufe während der Simulation bereits aggregieren, um die Anzahl der verwalteten Objekte zu reduzieren und verschiedene Möglichkeiten bieten, die Daten zu speichern (z. B. Datei oder Datenbank).

Danksagung

Ich möchte mich bei den Reviewern für das sehr ausführliche und hilfreiche Feedback bedanken. Die Vorschläge haben entscheidende Beiträge zur Qualität dieser Arbeit geleistet.

Literatur

[GIM17] Gribaudo, M.; Iacono, M.; Manini, D.: Performance Evaluation of Massively Distributed Microservices Based Applications. In: 31st European Conference on Modelling and Simulation, ECMS 2017. European Council for Modelling und Simulation, S. 598–604, 2017.

[Ja18] Jamshidi, P.; Pahl, C.; Mendonça, N. C.; Lewis, J.; Tilkov, S.: Microservices: The Journey so Far and Challenges Ahead. IEEE Software 35/3, Publisher: IEEE, S. 24–35, 2018.

[Ka14] Karsai, G.; Krahn, H.; Pinkernell, C.; Rumpe, B.; Schindler, M.; Völkel, S.: Design Guidelines for Domain Specific Languages. arXiv:1409.2378 [cs]/, 8. Sep. 2014, arXiv: 1409.2378, URL: http://arxiv.org/abs/1409.2378, Stand: 08. 05. 2022.

[Me20] Mendonça, N. C.; Aderaldo, C. M.; Cámara, J.; Garlan, D.: Model-Based Analysis of Microservice Resiliency Patterns. In: 2020 IEEE International Conference on Software Architecture (ICSA). IEEE, S. 114–124, 2020.

[Ne15] Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Beijing Sebastopol, CA, 2015, ISBN: 978-1-4919-5035-7.

[Ny07] Nygard, M. T.: Release It! Design and Deploy Production-Ready Software. Pragmatic Bookshelf, Raleigh, N.C, 2007, ISBN: 978-0-9787392-1-8.

[Ts13] Tseitlin, A.: The Antifragile Organization. Communications of the ACM 56/8, S. 40–44, Aug. 2013, ISSN: 0001-0782, 1557-7317, URL: https://dl.acm.org/doi/10.1145/2492007.2492022, Stand: 07. 05. 2022.

[ZGD19] Zhang, Y.; Gan, Y.; Delimitrou, C.: μqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, S. 212–222, 2019.

[Zi17] Zimmermann, O.: Microservices Tenets. Computer Science - Research and Development 32/3, S. 301–310, 1. Juli 2017, ISSN: 1865-2042, URL: https://doi.org/10.1007/s00450-016-0337-0.