THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

Generierung und Ordnung von Events in verteilten Systemen mit asynchroner Kommmunikation

Bachlorthesis
Studiengang Informatik

vorgelegt von
Simon Stockhause

Mai 2020

Referent der Arbeit: Prof. Dr. Harald Ritz
Korreferent der Arbeit: M.Sc. Pascal Bormann

Hier Steht Erklärung

Zusammenfassung

Contents

List of Figures

Listings

1 | Einleitung

1.1 Motivation

Die heutigen Bedürfnisse der Anwender ein stets erreichbaren, fehlerfreien und ??schnellen?? Service zur Verfügung zu haben, stellt hohe Erwartung an Unternehmen und deren Entwickler und Operatoren. Um den Ansprüchen der Nutzer gerecht zu werden, müssen Systeme gewährleisten, dass ein gewisser Grad von Beobachtbarkeit des Systems erreicht wird. Die Beobachtbarkeit sorgt für die nötige reaktionsfähigkeit der Entwickler und Operatoren, um möglicherweise auftretende Komplikationen, die die Benutzerbedürfnisse beeinträchtigen, schnell, präzise und langfristig beheben zu können.

Die Komplexität des Gesamtsystems, welches aus vielen kleinen Komponenten bestehen kann, ist eine große Herausforderung für Entwickler und Operatoren. Die enorme Skalierbarkeit einzelner Komponenten und die ausgezeichnete Resourcennutzung der Hardware löst zwar viele Probleme der Vergangenheit, wie zum Beispiel Überbelastung einzelner Knoten, Ausfall von Komponenten und Latenzprobleme. Allerdings schafft diese Umstellung neue Schwierigkeiten, die es zu bewältigen gilt.

Die Instrumentalisierungsbibliothek *Traktor* soll Events innerhalb eines Systems generieren und ordnen, sodass während der Entwicklungsphase eines Systems Fehlerquellen lokalisiert werden können. Bei der Entwicklung der Instrumentalisierungsbibliothek sollen Standards ermittelt, analysiert und umgesetzt werden. Es sollen Erfahrungswerte in der Domain der Beobachtbarkeit von Systemen, durch die Entwicklung einer Instrumentalisierungsbibliothek, gewonnen werden.

1.2 Problemstellung

Im Rahmen dieser Bachlorarbeit sollen Events in einem verteilten Rendering-System generiert und untersucht werden. Die Kommunikation innerhalb eines verteilte Systems über Prozessgrenzen spielt eine zentrale Rolle. Die Intrasystem - Netzwerkkommunikation bedarf Konzepte zur Nachvollziehbarkeit von Events und deren Beziehungen zueinander.

Das verteilte Rendering-System generiert asynchron *Frames* und sendet diese in Intervallen über eine Websocketverbindung an einen Client. Frames werden verworfen, sobald neuere Frames generiert wurden, d.h. innerhalb eines Intervals können mehrere Frames entstehen, aber nur eines ist relevant. Die Einhaltung von zeitlichen Grenzwerten, wie z.B. die Framegenerierungszeit oder der End-zu-End Latenz, gilt zu überprüfen. Der Klient kann asynchron Einfluss auf die zu generierenden Frames durch Übermittlung von Daten nehmen. Die bei diesem Datenaustausch entstehenden Events wie z.B. das Starten einer Framegenerierung, dem Beending einer Framegenerierung, dem Senden eines fertiggestellten Frames und dem Empfangen eines Frames sollen erstellt werden. Die zeitliche Einordnung der Events hat dabei eine zentrale Rolle einzunehmen. Die Eventgenerierung muss sich mit den zeitlichen Rahmenbedingungen vereinbaren lassen. Das Konstrukt von Events die zueinander in Verbindung stehen, soll untersucht werden. Dazu stellt sich folgende Frage: *Ist es möglich Events in einem verteilten System zu generieren und miteinander in Verbindung zu setzen, die es erlauben, einen Stream von Frames als eine Anordnung von Events, die kausal miteinandern verbunden sind, darzustellen?*

1.3 Foschungsstand

Es gibt diverse Konzepte und Werkzeuge zur Erhebung von Tracingdaten. Darunter zählen Instrumentalisierungsbibliotheken z.B.:

- Zipkin
- Jaeger
- Opentracing
- Brown Tracing Framework
- X-Trace

Das Paper *End-to-End Tracing Models: Analysis and Unification* beschreibt das spanbasierte Modell als eine Sammlung von *spans*, welche jeweils eine Block von Rechenarbeit. Die Brown University präsentiert in ihrer Veröffentlichung *Universal Context Propagation for Distributed System Instrumentation* eine Schichten-Architektur zur Übermittlung von Tracingdaten in einem spezifizierten *Baggage Context*. Der Baggage Context wird als Metadata mitgereicht und stellt somit eine Form des Piggybacking dar. darstellt.[1] Abgesehen von dem Brown Tracing Framework und dem X-Trace verwenden alle genannten Ansätze das spanbasierte Datenmodell.

- beschreibe, was metriken sind: bezug zu überwachung
- beschreibe was logs sind: bezug zu überwachung

[1] [Lea14] "End-to-End Tracing Models: Analysis and Unification". 2014.

- Kombination neuartig: OpenTelemtry

1.4 Thesisübersicht

2 | Themenüberblick

2.1 Verteilte Systeme

Verteilte Systeme dienen als Anwendungsfeld für die Instrumentalisierungsbibliothek. Auch das in dieser Arbeit vorgestellte Testbett, sowie das verteilte Rendering-System, in welches Traktor zum Einsatz kommt, sind solche verteilten Systeme. Dafür ist es wichtig, verschiedene Eigenschaften und Konzepte von verteilten Systemen zu definieren. Diese Eigenschaften und Konzepte nehmen einen entscheidenden Faktor bei der Ermittlung, Analyse und Umsetzung von Anforderungen der Instrumentalisierungsbibliothek ein. Dementsprechenend wird zunächt der Begriff des verteilten Systems definiert:

> Ein verteiltes System ist eine Kollektion unabhängiger Computer, die den Benutzern als ein Einzelcomputer erscheinen[2]

2.1.1 Eigenschaften eines verteilten Systems

Die Definition von van Steen und Tanenbaum geht mit zwei charakteristischen Eigenschaften von verteilten Systemen einher. Die erste Eigenschaft äußert sich darin, dass alle Komponenten eines Systems unabhängig voneinander agieren können. Komponenten werden in diesem Zusammenhang auch Knoten genannt. Knoten sind Hardwarekomponenten, also pysikalische Recheneinheiten. Auch Prozesse innerhalb einer Hardwareeinheit können Knoten sein. Dabei ist es möglich, dass mehrere Knoten auf einer Hardwareheinheit sind.[3].

Knoten sind miteinandern über Netzwerke verknüpft. Innerhalb eines Netzwerks kommunizieren Knoten mittels Nachrichten miteinander. Beim Ansprechen eines verteilten Systems soll dem Anfragesteller, zum Beispiel ein Client eines Webservers, nicht ersichtlich sein, dass mehrere Komponenten ein Gesamtsystem bilden, welches die Anfrage verarbeitet und entsprechende Vorgänge ausführt.

[2] [TS06] *Distributed Systems: Principles and Paradigms (2nd Edition)*. 2006.
[3] [ST17] *Distributed Systems*. 2017, p. 2.

2.1.2 Beobachtbarkeit von verteilten Systemen

Unter Beobachtbarkeit versteht man die Möglichkeiten der Betrachtung eines System. Die Symptome, die entstehen können, falls unerwünschte oder unvorhergesehene Zustände in dem System entstehen, sind oft schwer nachvollziehbar. Speziell verteilte Systeme erschweren die Reproduktion solcher Symptome, da das komplexe Ineinandergreifen der Komponenten die Erfassung der Vorgänge erschwert.

Die Beobachtbarkeit von verteilten Systemen kann in erster Linie durch Überwachung diverser Komponenten des Systems ermöglicht werden. Dabei lassen sich diese Komponenten in drei Kategorien unterteilen. Aufzuzählen sind (i) **Hardware** auf denen die Knoten angesiedelt sind, (ii) **Netwerke**, in denen die Kommunikation der einzelnen Knoten stattfindent und die (iii) **Anwendungen**, welche sich auf alle Knoten verteilten kann.

Hardware Aufgrund der Vielzahl an Hardwarekomponenten in einem System gilt es, besonders interessante Komponenten, im Kontext der Fehlerfindung beziehungsweise der Performanceanalyse, zu beobachten. Als Hauptkomponenten der meisten Systeme zählen zum Beispiel die *Central Processor Unit (CPU)*, die *Graphical Processor Unit (GPU)* oder auch die verschiedenen Speicher wie zum Beispiel der *Random-Access Memory (RAM)* oder die *Hard Disk Drive (HDD)*. Aus der Beobachtung dieser Komponeten gewinnt man allgemeine Informationen über das Gesamtsystem. Diese Informationen können zum Beispiel dazu genutzt werden, die Auslastung einzelner Knoten zu ermitteln und eventuell auf unbalancierte Nutzung der Konten reagieren zu können oder auftretende *bottlenecks*, das heißt stark auffällige performancebeeinträchtigende Komponenten im System, erkennen zu können. Diese könnten beispielsweise langsame HDDs sein, auf die oft zugegriffen wird.

Netzwerk Die Kommunikation innerhalb des Systems wird durch das Netzwerk, also die Verknüpfung der jeweiligen Knoten miteinandern, ermöglicht. Verschiedene Protokolle zur Regelung des Nachrichtenaustauschs kommen dabei zum Einsatz. Besonders nennenswert sind die Protokolle *Transmission Control Protocol (TCP)* und *Internet Protocol (IP)*. Diese beiden Protokolle dienen als Grundlage für zwei darauf aufbauende Protokolle. Zum einen das Hypertext Transfer Protocol (HTTP) und das Websocket Protocol. Der Anwendungsfall der Protokolle wird im Kapitel 5 genauer betrachtet.

Auch das Netzwerk generiert aussagekräftige Daten über das verteilte System. Die in dem Netzwerk verkehrenden Datenmengen sind dabei zu betrachten. Diese Datenmengen können auf unterschiedliche Weise gemessen und Schlussfolgerungen aus den Ergebnissen gezogen werden. Gemessen wird Netzwerkverkehr beispielsweise anhand der Größe der Datenpakete, der Geschwindigkeit der Datenpakete vom Sender zum Empfänger oder einer Knotenerreichbarkeitsprüfung. Diese Daten für sich können schon informativ sein. Allerdings lassen sich auch weitere Informationen durch Korrelation gewinnen. Bespielsweise könnte in einem Anwendungsfall eine Korrelation zwischen Geschwindigkeitsanomalien und Tageszeit bestehen. Auch denkbar wären Anomalien,

die sich in Paketverlusten äußern. Diese könnten zum Beispiel durch erhöhte Beanspruchung eines bestimmten Knotens beziehungsweise einer bestimmten Komponente ausgelöst werden. Durch Feststellung von Korrelationen können Maßnahmen durchgeführt werden, die der auftretende Anomalie entgegensteuert oder gar ganz beseitigt.

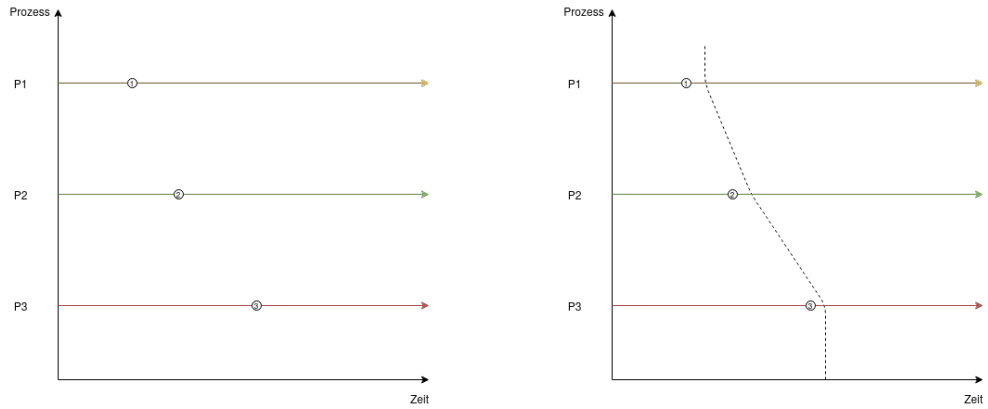
Anwendung Die Beobachtung der Anwendung ist, im Zusammenspiel mit dem Nachrichtenaustausch über das Netzwerk, zentral. Die Beobachtbarkeit der Anwendung sorgt dafür, dass Anwendungsdaten erhoben, ver- und aufgearbeitet und anschließend präsentiert werden. Die Präsentation der Daten hilft den Verantwortlichen Informationen über die internen Vorgänge des Systems zu gewinnen und entsprechend agieren zu können. Es ist zudem möglich gewisse Daten interpretieren zu lassen. Die daraus gewonnenen Informationen können von weiteren Systemen dazu genutzt werden, automatisiert zu steuern. Grundsätzlich kann man auch hier zwischen drei verschiedenen Datenquellen unterscheiden. Diese drei Datenquellen sind:

- Metriken[4], z.B.:
  - Systemdaten
  - Anzahl von Instanzen
  - Anfrageanzahl
  - Fehlerrate
- Applikationslogs, z.B.:
  - Fehler
  - Warnungen
  - Applikationsinformationen
- Traces, z.B:
  - Segmente
  - Kontext

2.1.3 Ordnung von Events

Ein Trace ist die Sammlung von Events die im Laufe des Weges durch ein verteiltes System generiert wurden. Die Knoten, die diesen Weg umfassen, generieren Events, indem sie Programmcode ausführen, welcher Instrumtalisiert ist. Diese Events sind die kleinsten Einheiten eines Traces und unterliegen einer Kausalordnung. Das *Happend Before Model* nach Lamport beschreibt die Kausalordnung. Die Kausalordnung ist eine strikte partielle Ordnung.[5].

[4] [Wat17] *8 Key Application Performance Metrics & How to Measure Them.* 2017.
[5] [Gar02] *Elements of distributed computing.* 2002.



(a) Zeigt Prozesse P1, P2 und P3. Diese generieren jeweils ein Event.

(b) Events (1), (2) und (3) finden parallel statt.

Abbildung 2.1

Die Abb. 2.1a stellt eine Situation dar, in der drei Prozesse jeweils ein Event erzeugen. Intuitiv würde man interpretieren, dass (1) von P1 vor (2) von P2 erzeugt wurde und das darauf (3) von P3 folgt. Dies mag stimmen, in der Annahme, das es nur eine globale Zeit als Richtwert gäbe. Allerdings lässt sich die Ordnung, in dem Kontext von verteilten Systemen und Nebenläufigkeit, nicht ohne Berücksichtigung verschiedener Einflussfaktoren feststellen. Lamport erläutert, dass in einer Umgebung, in der eine Ordnung anhand eines Zeitstempels physikalischer Zeit festgelegt wird, eine physikalische Uhr vorhanden sein muss.[6] Die Einbindung einer physikalischen Uhr in ein verteiltes System ist eine aufwendige und komplexe Aufgabe. Probleme wie Synchronisation verschiedener Uhren mit keiner absoluten Präzision und dem dazugehörigen Drift, dem algorithmisch entgegengewirkt werden muss, treten auf. Abb. 2.1b zeigt, dass die Events, unter der Berücksichtigung der Definition von Lamport, nebenläufig stattfindet. Aus diesem Grund ist es naheliegend zu versuche eine Lösung zu finden, die auf physikalische Uhren verzichtet.

Drei Bedingungen müssen nach Lamport erfüllt sein, damit eine Beziehung zwischen Events partiell geordnet ist.

> „(1) Wenn a und b Events im selben Prozess sind und a vor b stattfindet, dann $a \rightarrow b$. (2) Wenn a das Senden einer Nachricht eines Prozesses ist und b das Empfangen einer Nachricht eines anderen Prozesses ist, dann $a \rightarrow b$. (3) Wenn $a \rightarrow b$ und $b \rightarrow c$, dann $a \rightarrow c$.“ [7]

Um die Feststellung von Lamport zu verdeutlichen wird Abb. 2.2 untersucht. Die Abbildung stellt zwei Prozesse dar, dabei wird angenommen, dass diese in unterschiedlichen Rechensystemen angesiedelt sind. Das bedeutet, dass auf eine globale physikalische Uhr verzichtet wird. Diese Prozesse können sich mittels dem Senden einer Nachricht und

[6] [Lam78] “Time, Clocks, and the Ordering of Events in a Distributed System”. 1978, S. 559.
[7] [Lam78] “Time, Clocks, and the Ordering of Events in a Distributed System”. 1978, S. 559.

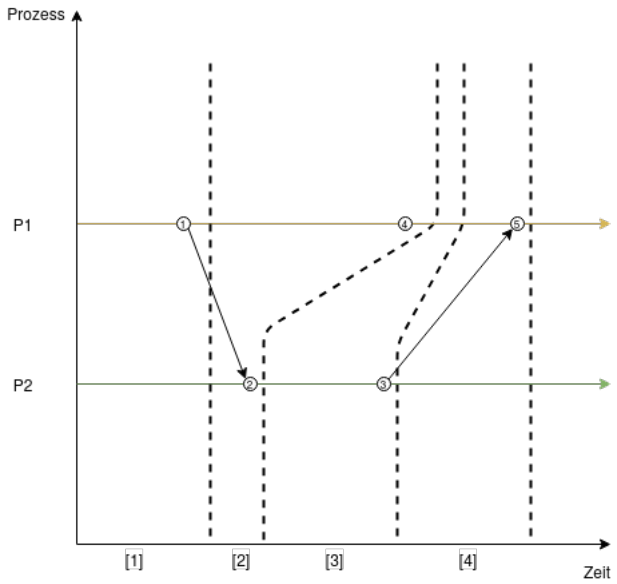


Abbildung 2.2: Zeigt Prozesse P1 und P2. Diese generieren jeweils Events. Logische Reihenfolge des Auftretens der Events anhand der gestrichelten Linien [1], [2], [3] und [4] ersichtlich.

dem Empfangen einer Nachricht verständigen. Das verteilte System generiert insgesamt fünf Events. P1 erzeugt (1) in der logischen Zeitspanne [1] und sendet anschließend eine Nachricht an P2. Das Empfangen wird durch (2) dargestellt. Daraus folgt, dass eine *Happens Before Relation* zwischen (1) und (2) besteht, also (1) $\rightarrow$ (2). Nebenläufig dazu, findet in P1 Event (4) statt. (4) kann durch P2, in dieser Zeitspanne, in keiner Weise beeinflusst werden. Das heißt, dass (4) ein von (2) kausal unabhängiges Ereignis ist, (2) $\not\rightarrow$ (4). (4) ist aber kausal von (1) abhängig, da es das direkt folgende Event im gleichen Prozess ist, also (1) $\rightarrow$ (4). (4) bildet dabei, als neues abhängiges Event, Zeitspanne [2]. Weitergehend zu folgern ist, (2) $\rightarrow$ (3), weshalb [3] entsteht. Auch hier ist eine kausale Unabhängigkeit von (3) zu (4) zu sehen. An dieser Stelle spielt eine weitere Bedingung eine Rolle. Es muss die *Clock Condition*[8] von Lamport berücksichtigt werden. Diese Bedingungen besagt:

$$\text{Clock Condition}: \text{ wenn } a \rightarrow b \text{ dann } C(a) < C(b)$$

Diese Bedingung führt C als logische Uhr ein. Eine logische Uhr ist ein von der physikalischen Zeit unabhängiger Zähler, der einem Event eine Zahl zuweist. Die Clock Condition ist erfüllt, sobald (1) darauf geachtet wird, dass zwischen zwei Events eines Prozesses die logische Uhr voranschreitet und (2) das einem Event ein Zeitstempel zugewiesen wird,

[8] [Lam78] "Time, Clocks, and the Ordering of Events in a Distributed System". 1978, S.5 560.

der folgende Eigenschaften hat[9]:

$$T_m = C_i\langle a\rangle \text{ dann } C_{j+1} = \max[C_j, T_m]$$

Dabei ist a das Event. $C_i\langle a\rangle$ ist der Zeitpunkt der Uhr i während Event a. T_m der Zeitstempel eines Events, welcher eine Nachricht m zu einem anderen Prozess sendet, zum Zeitpunkt $C_i\langle a\rangle$. Beim empfangenen Prozess mit der Uhr j, wird das erzeugte Event mit dem Zeitstempel $C_{j+1} = \max[C_j, T_m]$ versehen. Das bedeutet für die Darstellung, dass (3) $\not\rightarrow$ (4), (3) $\rightarrow$ (5) und $C_{P1}\langle 4\rangle < C_{P2}\langle 3\rangle$. Diese Annahme kann getroffen werden, weil (2)$\rightarrow$ (3) und somit diese beiden Events nicht zu einem Zeitpunkt mit (4) stattfinden können, weil es $C_{P2}\langle 2\rangle < C_{P2}\langle 3\rangle$ widersprechen würde.

Die **Transitivität** ist durch (1) $\rightarrow$ (2) und (2) $\rightarrow$ (3) gegeben. Das heißt eine Relation zwischen (1) und (3) besteht, also (1) $\rightarrow$ (3). Die zweite Charakteristik, dass ein Event nicht vor sich selbst stattfinden kann, ist gegeben.(1) $\rightarrow$ (1) kann also nicht bestehen und ist somit **irreflexiv**. Zuletzt die ist durch die Tatsache, dass ein Event nicht vor und nach einem anderen parallel bestehen kann, also (1) $\rightarrow$ (2) und (2) $\rightarrow$ (1), gezeigt, dass die Relation **antisymmetrisch** ist. Die Kombination der drei Charakterisitken machen eine strikte partielle Ordnung aus, eine sogenannte Kausalordnung.

Die gezeigte Kausalordnung von Events stellt das Fundament zur Erhebung von Tracedaten in verteilten Systemen dar und ist somit unerlässlich, um ein verteiltes System beobachtbar zu machen, also einen Teil der *Oberservability* zu ermöglichen.

2.2 Distributed Tracing

Der Begriff des verteilten Systems wurden bereits nach Tanenbaum definiert. Allerdings lohnt es sich eine alternative Definition zu betrachten, die eine andere Perspektive ermöglicht.

> Ein verteiltes System ist ein System, mit dem ich nicht arbeiten kann, weil irgendein Rechner abgestürzt ist, von dem ich nicht einmal weiß, dass es ihn überhaupt gibt.[10]

Diese Definition lässt sich so auffassen, dass Lamport im Jahr 1987 die Problematik der Fehlersuche während der Laufzeit, des Debuggings während der Entwicklungsphase und des organisatorischen Aufwands im Allgemeinen, also damit der grundsätzlich hohen Unübersichtlichkeit und Komplexität von verteilten Systemen, beschreibt. Diese Probleme und Herausforderungen, mit denen man in der heutigen Zeit der serviceorientierten Architektur beziehungsweise der Microservicearchitektur konfrontiert wird,

[9] [Lam78] "Time, Clocks, and the Ordering of Events in a Distributed System". 1978, S. 560.
[10] [Lam87] *Distributed Systems Definition by Lamport.* 1987.

können durch *distributed tracing* angegangen werden. Distributed tracing ist das verfolgen und analysieren von kausalzusammenhängenden Events in einem verteilten System. So kann zum Beispiel Fehlerquellenermittlung beschleunigt werden. Tracing ist ein Konzept, welches als Werkzeug umgesetzt werden kann. Durch das Erheben von Tracedaten lassen sich detailierte Schlussfolgerungen über einzelne *Requests* und ihren Weg durch das verteilte System schließen. Dies verlangt allerdings das direkte Eingreifen in den Quellcode. Der alternative Ansatz des *Blackbox Monitoring*, also das Überwachen eines Systems mit eingeschränkten Informationen, wird in sich überschneidenden Anwendungsbereichen eingesetzt. Das Blackbox Monitoring versucht das System von Außen zu betrachten. Dabei wird keine Instrumentalisierung von Quellcode vorgenommen. Dieser sammelt Daten aus bestehenden Logs und Netzwerkschnittstellenüberwachung. Der Ansatz des Blackbox Monitoring ist sinnvoll, wenn mit vielen Softwarekomponenten gearbeitet wird, auf deren Entwicklung man keinen direkten Einfluss hat. Dazu zählen beispielsweise proprietäre Software, ohne Zugang zu dem Quellcode. Allerdings ist die Rekonstruktion der Requestpfade unzuverlässig und die Fehlerfreiheit, der gesamten Kausalordnung aller erzeugten Events, aufgrund der eingeschränkten Informationen, nicht ohne Nachteile, gegeben.

Distributed tracing umfasst zwei Teilbereiche der im Abschnitt 2.1.2 beschriebenen Beobachtbarkeit von verteilten Systemen. Das ist zum einen die verteilte Anwendung mit ihren einzelnen Service-Komponenten und zum anderen das Netzwerk über das Nachrichten ausgetauscht werden. In Kapitel 3 wird anhand eines minimalbeispiels beschrieben, welche Rollen diese beiden Aspekte in der Generierung und Ordnung von Events zu spielen haben.

Ziel von distributed tracing soll sein, mit möglichst wenig *Overhead* und *minimalem Aufwand* tiefgreifenden Einblick in eine verteilte Anwendung zu gewinnen. Mit wenig Overhead ist gemeint, dass das System nicht zu stark beeinträchtigt wird. Das heißt, dass es zu keiner unvereantwortlichen Einbüßung von Leistung kommen darf. Das Bedürfnis nach minimalem Aufwand stammt von der Natur der Microservice-Architektur. Das Prinzip von lose gekoppelten und jederzeit austauschbaren Microservices fordert eine schnelle und einheitliche Möglichkeit, Einblick in die Komplexität von verteilten Systemen zu gewinnnen. Dabei ist es erforderlich und aus Entwicklersicht verständlich, dass der Instrumentalisierungsanteil des Services nur ein kleinen Teil der Gesamtlogik ausmachen darf, denn die Entwickler wollen sich auf die Businesslogik konzentrieren und distributed tracing soll den Entwicklungsprozess unterstützen und nicht durch übermäßige Investitionsanforderungen beeinträchtigen.

2.3 Entwicklung einer Tracingbibliothek

Das Konzept von *distributed tracing* ist als Bibliothek in der Programmiersprache C# umgesetzt. Drei Konzepte sind dabei zu definieren. Diese sind **Profiling**, **Tracing**, und **Instrumentalisierung**.

Profiling ist der Prozess des Analysierens einer Anwendung durch Erhebung von anwendungsbezogenen Daten. Dabei kommen Profilingwerkzeuge wie z.B. *perf*[11] zum Einsatz, die Profilingdaten erheben. Diese Daten bestehen unter anderem aus aufgerufenen Funktionen, CPU Auslastung und Laufzeiten von Funktionen.

Tracing ist eine Konzept zur Anwendungsüberwachung. Dabei werden Events in Relation zueinander gesetzt, die die chronologische Reihenfolge der Events repräsentieren.

Instrumentalisierung ist das Implementieren von Anwendungslogik, die dafür sorgt, das Daten erhoben werden können, die zur Performanceanalyse und zur Fehlerdiagnose dienen. Die Daten werden z.B. von Tracingwerkzeugen genutzt.

Die Bibliothek wird als Tracingwerkzeug konzipiert, die sich Instrumentalisierung zu nutzen macht, um ihre Daten zu erheben. Die Bibliothek ist über den Packetmanager *Nuget* veröffentlicht. Dadurch soll minimaler Integrationsaufwand in Systeme entstehen. Dies ermöglicht in den Entwicklungsumgebungen wie zum Beispiel *Unity* auf Linux, wie auch auf Windows, eine einfache und schnelle Integration. Innerhalb des Entwicklungsprozess ist eine Continuous Integration Pipeline aufgebaut worden, die dafür sorgt, dass die Bibliothek über Nuget verfügbar gemacht wird.

[11] **perf**.

3 | Problembeschreibung

In diesem Kapitel wird die Problematik, um das Generieren und Ordnen von Events in einem verteilten System mit asynchroner Kommunikation, beschrieben. Dabei betrachtet man die Relevanz der Problemstellung. Ausserdem werden Fragen aufgestellt, die diese Problemstellung umfassen. Anschließend wird eine Anforderungsanalyse durchgeführt.

3.1 Anwendungsüberwachung

Viele Bereiche der Wirtschschaft, der Wissenschaft und grundsätzlich des alltäglichen Lebens sind durch Software unterstützt. Trends wie beispielsweise Internet of Things (IoT), Hausautomatisierung, Mobile Geräte, etc. sind Anwendungsbeispiele. Diese sind aus ihrer Natur heraus stark verteilte Anwendungen. Aber auch potentiell neue Anwendungsbereiche, wie zum Beispiel verteiltes Rendering, benötigen detaillierte Einsicht in die internen Vorgänge der Anwendung.

Dabei spielen zwei Eigenschaften in der Überwachung der Anwendung eine zentrale Rolle. Zum einen ist das die (i) *Performance* und zum anderen die (ii) *Korrektheit*.

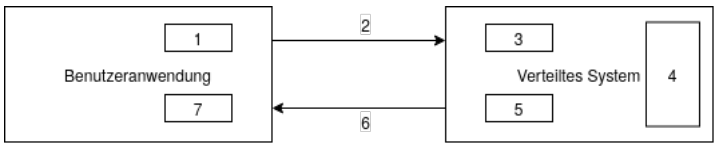


Abbildung 3.1: Kausaler Pfad einer Vorgangs in dem verteilten rendering System

Performance Viele Anwendungsbereiche setzten gewisse Rahmenbedingungen, die erfüllt werden müssen. Nutzererwartungen im Bezug auf interaktive Systeme, welches einer der beiden Anwendungsfälle der Instrumentalisierungsbibliothek ist, äußern sich beispielsweise in der Reaktionszeit der Anwendung auf Benutzereingaben. Das Rendering nimmt dabei nur einen Teil der Gesamtlatenz ein. Ein beispielhafter Gesamtpfad, der durch die verteilte Renderinganwendung genommen werden kann, besteht aus dem

Senden der Benutzereingabe, der Übermittlung der Benutzereingabe zum verteilten System, der Verarbeitung der Eingabe und der Übermittlung des Ergebnisses an die Benutzeranwendung. Abb. 3.1 verdeutlicht diesen Pfad von kausal relatierenden Events. Dabei ist jede Komponente des Pfades ein generiertes Event. Zu sehen ist, dass das (1) Senden der Benutzereingabe vor dem (2) Übermitteln der Benutzereingabe stattfinden. Anschließend wird die Eingabe (3) empfangen. Auch die (4) Verarbeitung im verteilten System, das für den Benutzer, wie in Abschnitt 2.1.1 definiert, nicht als solches kenntlich sein muss, generiert in diesem Beispiel ein Event. Die Antwort wird (5) gesendet und die (6) Übermittlung wird durchgeführt. Zuletzt wird die Antwort (7) empfangen. Das Empfangen schließt den Pfad ab. Die Gesamtdauer des Pfades wird als Latenz eines Frames bezeichnet. Daraus kann eine Durschschnittslatenz über eine Zeitspanne berechnet werden, welches als Performanceindikator dient. Die Zeitspanne zwischen den einzelnen Events können verglichen werden. Dabei ist es möglich sog. *Bottlenecks* zu identifizieren. Bottlnecks sind Vorgänge, die einen Großteil der Gesamtdauer ausmachen. Sie werden durch die Zeitspanne zwischen zwei Events, die auf dem *kritischen Pfad* liegen, bestimmt. Diese Art der Anwendungsüberwachung soll die Möglichkeit bieten, Bottlenecks zu identifizieren. Wie in Abschnitt 2.1.3 beschrieben, verlangt eine Messung der Zeit über verschiedene physikalisch Entitäten entweder eine globale physikalische Uhr oder jeweils eine physikalische Uhr in jeder Entität, die über alle Entitäten synchrone sind beziehungsweise synchronisiert werden. Dabei stellt sich die Frage **F1**:

> **F1:**
> Inwiefern lässt sich eine Zeitmessung von Eventzeitspannen konzipieren?

Korrektheit Die Korrektheit eines Systems ist dann gegeben, wenn die Eigenschaften eines Systems einer *Spezifikation* entsprechen. Das bedeutet, dass die Beobachtung des Verhaltens einer (verteilten) Anwendung nicht ausreicht, um seine Korrektheit zu beweisen. Tracing soll also nicht die Korrektheit einer verteilten Anwendung beweisen. Tracing kann aber dabei unterstützen, indem es das Verhalten einer Anwendung beobachtbar macht. Insbesondere die Zusammenhänge der Komponenten und die entstehenden Nebenläufigkeiten sind erschwerende Faktoren in der Verifikation. So stellt sich die Frage **F2**:

> **F2:**
> Welche Visualisierungsformen können genutzt werden, um kausal zusammenhängende Events derart darzustellen, sodass anhand der Visualisierung feststellbar ist, ob das Verhalten der Anwendung starke Ausreißer, die auf Fehlimplmenentierung deuten könnten, aufweist

3.2 Zusammenführung von Events

Man definieren ein minimales verteiltes System, welches das verteilte Rendering-System vereinfacht darstellt. Dieses besteht aus zwei Komponenten. Abb. 3.2 bildet ein solches

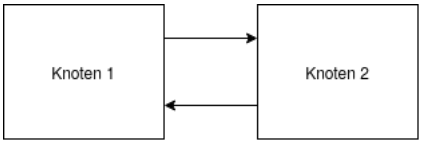
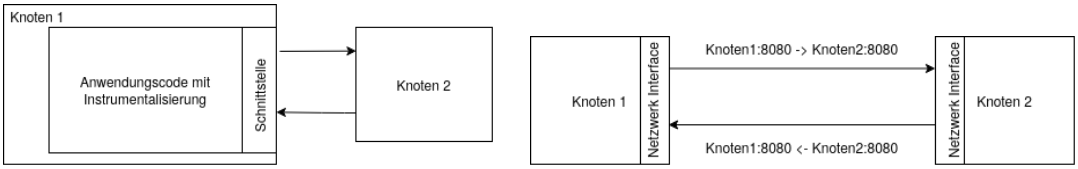


Abbildung 3.2: Minimale Struktur eines verteilten Systems, bestehend aus zwei Komponenten



(a) zeigt Knoten mit instrumentalisiertem Anwendungscode

(b) Netzwerkkommunikation über TCP/IP

Abbildung 3.3

System ab. Die Knoten beinhalten zwei für das Generieren und Ordnen von Events interessante Aspekte. Dies ist zum einen die in Abb. 3.3a dargestellte verteilte Anwendung mit ihrem instrumentalisiertem Code und zum anderen das in Abb. 3.3b dargestellte Netzwerk, über welches Nachrichten ausgetauscht werden.

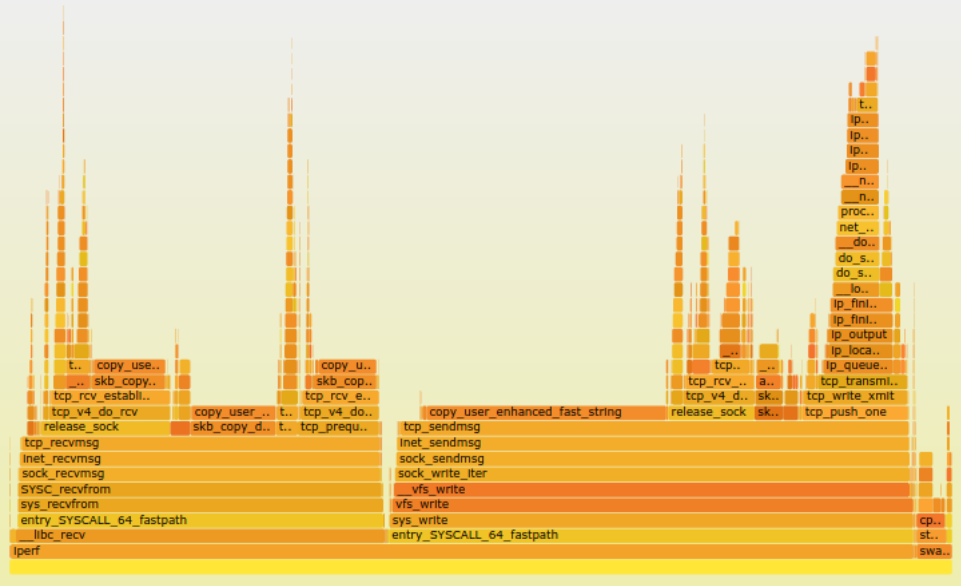


Abbildung 3.4: Visualisierung von CPU Performancedaten dargestellt als Flammengraph von Brendan D. Gregg [Bre15]

Die Komponenten des Systems besitzen jeweils einen Linux Kernel. Der Kernel bietet die Funktionalität *perf_events* zu erheben.

> perf_events ist ein eventorientiertes Überwachungswerkzeug, welches helfen kann, Leistung zu verbessern und Fehelerquellen von Funktionen zu lokalisieren.[12]

Von dem Szenario ausgehend, dass man Events innerhalb des Minimalbeispiels analysieren möchte, eignen sich Flammengraphen[13]. Der in Abb. 3.4 gezeigte Flammengraphe stellt perf_events dar, die während einer TCP Kommunikation erhoben worden sind. Dabei ist die Länge der Balken, die Zeit, die das Event, relativ zur Gesamtzeit der Messung, insgesamt eingenommen hat. Die Profilingdaten weisen keine Kausalität auf. Die erhobenen perf_event Daten sind Stichproben. Man geht in diesem Beispiel allerdings davon aus, das alle Events aufgenommen worden sind. Diese Darstellung erlaubt es, die Events *eines* Systems genau zu beschreiben. Nun kommt das zweite System hinzu, mit dem die Kommunikation stattgefunden hat. Auch hier sei gegeben, dass das zweite System Daten generiert hat, welche zu einer ähnlich aufgebaute Visualisierung führt. Die beiden Flammengraphen werden miteinander verbunden. Dies führt zu einer dreidimensionalen Darstellung von Flammengraphen, gezeigt in Abb. 3.5. Wie in einer TCP-Verbindung üblich, wird eine Kommunikationskanal aufgebaut. Über den Kanal können Nachrichten ausgetauscht werden. Anschließend wird die Verbindung mit einem Vier-Wege-Handschlag beendet. Die obersten Blöcke und ihre systemübergreifenden Verbindungslinien, dargestellt durch die gestrichelten Linien mit Pfeilrichtung, stellen die Terminierung der TCP-Verbindung dar.

Der Terminierungsprozess wird genauer betrachtet. Abb. 3.6 zeigt vier Events. **A** ist die *FIN* Markierung des Initiators. Sie leitet die Terminierung ein. **C** stellt das Empfangen und Beantworten mittels *ACK* und *FIN* dar.
B ist der Terminierungsmoment des Initatorsystems. Dieser findet nach dem Zeitpunkt des Eintreffens von *FIN* des Empfänger statt. Dieser Zeitpunkt ist das Senden des letzten *ACK* des Initiators, addiert mit einer Konstante *Timeout*. Event **B** ist also definiert als:

$$\mathrm{B}: \; Ack_{init} + Timeout$$

D ist der Terminierungsmoment des Empfängersystems. Dieser Zeitpunkt ist das Erhalten der letzten, vom Initiatorsystem gesendeten, *ACK* Markierung. Die unbekannte Variable *Übertragungszeit* nimmt Einfluss auf den Zeitpunkt. Event **D** ist also definiert als:

$$\mathrm{D}: \; Ack_{init} + \mathrm{Ü}bertragungszeit$$

Zu untersuchen sind die Relationen zwischen diesen vier Events. Dabei sind zwei Relationen, wie in Abschnitt 2.1.3 beschrieben, als A $\rightarrow$ B und C $\rightarrow$ D definiert. Durch die kausale Abhängigkeit von C von A gilt A $\rightarrow$ C. Durch die *Transitivität* ist entsprechend A $\rightarrow$ D gegeben. Es ist zu untersuchen, ob B $\rightarrow$ D gilt. Dabei sind die Bedingungen, die

[12] [Bre] *Linux perf Examples.*
[13] [] *Flame Graphs.*

von Lamport definiert worden sind, zu betrachten. Da zwei Systeme miteinander kommunizieren, muss folgende Bedingung erfüllt sein, sodass eine *Happens-Before* Relation gegeben ist.

> (i) Wenn a das Senden einer Nachricht ist und b das Empfangen derselben Nachricht in einem anderen System ist, dann a $\rightarrow$ b[14]

Nach der Definition von **D** ist es mit b gleichzusetzen, somit ist ein Teil der Bedingung erfüllt. **B** ist jedoch nicht das Senden der Nachricht, also des letzten *Ack*, sonder ein Event, welches darauf folgt. Die Events sind nebenläufig.

Aus dieser Darstellung folgert Fragestellungen F3:

> **F3:**
> Welche Konzepte für Tracingwerkzeuge gibt es, um über Prozessgrenzen hinaus Relationen bilden zu können, bei der eine Reihenfolge der Events feststellbar ist

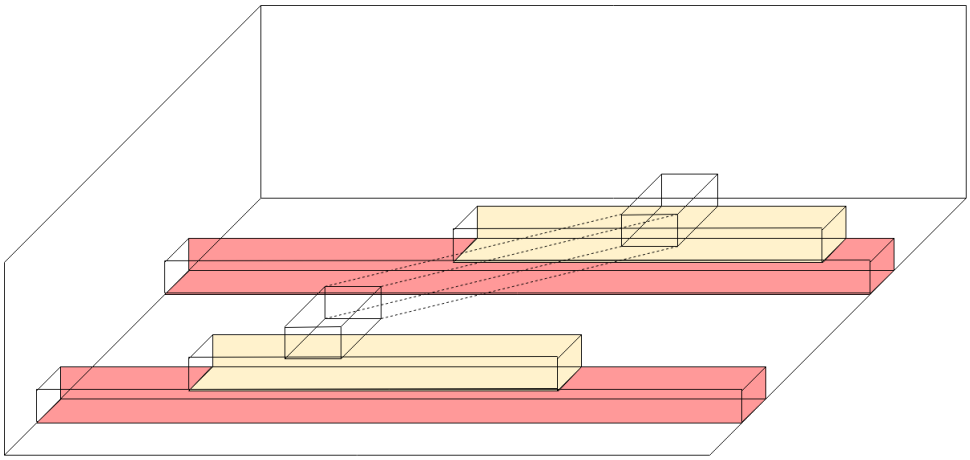


Abbildung 3.5: Skizzierung eines dreidimensionalen Flammengraphs mit Nachrichtenaustausch

[14] [Lam78] "Time, Clocks, and the Ordering of Events in a Distributed System". 1978.

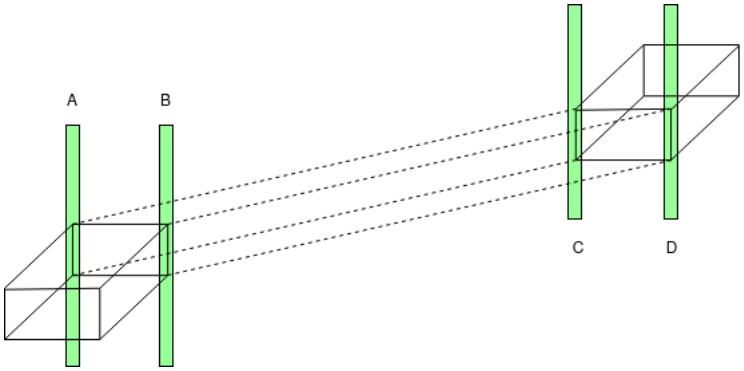


Abbildung 3.6: Detailierte Betrachtung des in Abb. 3.5 gezeigte Nachrichtenaustauschs. Stellt die Schließung einer TCP Verbindung dar

3.3 Anforderungsanalyse

Das Systems, für das die Instrumentalisierungsbibliothek entwickelt wird, ist ein System für verteiltes Rendering. Die Instrumentalisierungsbibliothek muss notwendige Funktionalitäten spezifizieren, die es ermöglichen ein Modell aus kausal abhängigen Events darzustellen. Zur Erstellung des Modells muss sich mit den Funktionalitäten der **Eventgenerierung**, der **Eventrelation**, der **Synchronisation von Eventgeneratoren**, der **Eventübermittlung** und der **Ordnung von Events** beschäftigt werden. Im Fokus der Interpretation des Modells soll die **end-zu-End Latenz**, sowie die **Generierungszeit eines Frames** stehen. Rahmenbedingungen wie die eingeschränkte **Nachrichtenmodifikation** sind zu berücksichtigen.

Semantisch relevante Ereignisse sind zu definieren. Eine Funktionalität muss geschaffen werden, die es erlaubt, diese Ereignisse als ein Event abzubilden. Die Generierungsfunktionalität muss dafür sorgen, dass die Events einen spezifizierten Aufbau aufweisen, um weiterverarbeitet und ausgewertet werden zu können. Die Nutzung einer standardisierten und erprobten Application Programming Interface (API) ist wünschenswert.

3.3.1 Funktionalitäten

3.3.1.1 Eventgenerierung

Events müssen in einem für die Anwendung semantisch relevanten Bereich generiert werden können. Die Generierung eines Event, welches den Startpunkt eines Traces darstellt, muss dafür sorgen, dass der Traces eindeutig identifizierbar ist. Alle Events, die auf ein anderes Event folgen, müssen eine Relation auf das Elternevent aufweisen.

3.3.1.2 Eventrelation

Es muss ein Modell für Events konzipiert werden. Das Modell muss in der Lage sein, Relationen abbilden zu können. Diese Relationen sollen die kausalen Zusammenhänge der Events darstellen.

3.3.1.3 Synchronisation von Eventgeneratoren

Eventgeneratoren sind oftmals auf verschiedenen Komponenten des verteilten Systems angesiedelt. Dabei ist es zu untersuchen, wie ein solches Konzept aussehen kann, dass dafür sorgt, dass Events geordnet werden können.

3.3.1.4 Eventübermittlung

Damit ein Kausalpfad erstellt werden kann, müssen die Events, die in einem anderen System generiert werden und zum Kausalpfad gehören, in einer Form zusammengeführt werden können.

3.3.1.5 Eventkontext

Das erste generierte Event bildet den Startpunkt eines neuen Traces. Mehrere Anfragen können zur gleichen Zeit von dem in Abb. 3.2 beschriebenen System angenommen werden. Dadurch ist gegeben, dass mehrere Traces parallel existieren. Um diese eindeutig zu machen muss ein Tracekontext definiert werden, der es erlaubt, Traces eindeutig, über Prozessgrenzen hinaus, zu identifizieren und Events diesen zuzuordnen.

3.3.2 Rahmenbedingen

3.3.2.1 End-zu-End Latenz

Es sind Latenzgrenzwerte definiert. Die Zeitspannen, wie zum Beispiel die Generierungszeit eines Events oder der End-zu-End Antwortzeit, sind zu messen. Die Generierungszeit eines Frames soll 16ms nicht überschreiten, damit dem Anwender eine flüssige Darstellung geboten werden kann. Die Instrumentalisierung des Quellcodes fügt zusätzliche Programmlogik hinzu, weshalb eine erhöhte Verarbeitungszeit entsteht. Diese Verarbeitungszeit soll das verteilte rendering System möglichst geringfügig belasten. Vorallem die Generierungszeit der Frames muss unverändert sein.

3.3.2.2 Generierungszeit eines Frames

Die Renderinggeschwindigkeit wird anhand der Zeit gemessen, also wieviele *ms* gebraucht werden, um ein Bild zu generieren. Der Generierungsprozess eines *Frames* umfasst vier Ebenen. Diese Ebenen sind die Applikationsebene, die Geometrieprozessierung, die Rasterung und die Pixelprozessierung. Die Verarbeitung wird, abhängig von der bearbeitet Ebene, von der CPU oder der GPU durchgeführt. Es ist wünschenswert GPU und CPU Aktivitäten überwachen zu können.

3.3.2.3 Nachrichtenmodifikation

Die Generierung von Events kann von zwei Perspektiven aus betrachtet werden. Zum einen die *Blackbox* Perspektive und zum anderen die *Whitebox* Perspektive.

Bei dem Blackboxansatz, wird die Generierung angestoßen, sobald Schnittstellen angesprochen werden. Dabei werden betriebssystemspezifische Funktionalitäten genutzt, um diese Betriebssystemereignisse zu erkennen. Diese Ereignisse können erkannt, aufgearbeitet und als Events gespeichert werden. Betriebssystemspezifische Ereignisse sind vor allem ausgehende und eingehende Nachrichten, die von den Netzwerkschnittstellen verarbeitet werden. Abb. 3.3b zeigt eine auf dem TCP/IP Stack basierende Nachricht. Die Daten der Senderaddresse, der Empfängeradresse und einem Zeitstempel könnten genutzt werden. Allerdings ist das Fehlen von Applikationsinformationen ein entscheidendes Problem. Das Ziel des Blackboxansatz ist die minimale Voraussetzung von *a priori* Informationen über Kommunikationswege, über den Aufbau von Applikationsnachrichten, die Semantik der Anwendung und den Aufbau des verteilten Systems.[15]. Allerdings sind diese Daten äußert wichtig, um ein tiefgreifendes Verständnis des verteilten Systems zu gewinnen.

Der Whiteboxansatz nutzt Instrumentalisierung des Quellcodes, um die Eventgenerierung anzustoßen. Dabei wird vorausgesetzt, dass die Semantik der Anwendung, Informationen über den Aufbau von Nachrichten, den Aufbau des Systems und die Kommunikationswege zwischen Komponenten bekannt sind. Bei der Notwendigkeit einer Modifizierung von Nachrichten weist dieser Ansatz jedoch auch Schwächen auf. Im Anwendungsfall des verteilten Rendering-Systems fehlt die Möglichkeit Nachrichten innerhalb der Anwendung um Tracingdaten zu erweitern.

[15] [Agu+03] "Performance debugging for distributed systems of black boxes". 2003.

4 | Design

In diesem Kapitel werden die in Kapitel 3 beschriebenen Problemstellung bearbeitet. Anfänglich werden Designziele definiert, die den Rahmen für die Konzipierung bilden. In Abschnitt 4.2 wird ein Datenmodell vorgestellt, welches sich an Standards von bereits existierenden Tracingmodellen orientiert. Abschnitt 4.3 präsentiert ein Konzept zur Verarbeitung der erhobenen Tracingdaten. Der Abschnitt 4.4 beschäftigt sich mit der Darstellung der Tracingdaten zur Informationsgewinnung durch den Anwender der Bibliothek. Somit wird ein Konzept für die Tracinginfrastruktur **Traktor** zur Erhebung, Verarbeitung und Visualisierung von Tracingdaten vorgestellt.

4.1 Designziele

Aus den in Abschnitt 3.3 beschriebenen Anforderungen folgen Designziele, die an die Implementierung gestellt werden. Die von Google erstellte Fachpublikation *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*[16] dient vielen Tracingsystemen als konzeptionelle Grundlage. In der Puplikation werden Designziele aufgeführt, die neue Tracingsysteme bewerten sollten. Aus den Anforderungen des verteilten Rendering-Systems ergeben sich Designziele. Diese Designziele umfassen die (i) **Verarbeitungskosten**, die (ii) **Benutzbarkeit**, die (iii) **Portabilität** und die (iv) **Datenverfügbarkeit**. Ausserdem werden Nicht-Ziele definiert. Zu diesen gehören die (v) **Anwendungs-Level Transparenz** und die (vi) **Skalierbarkeit**.

4.1.1 Ziele

Verarbeitungskosten Ein für die Performance der Anwendung kritisches Designziel, in der Tracing eingeführt werden soll, stellt der **Overhead** dar. Der Overhead, der durch die Instrumentalisierung entsteht, soll möglichst gering sein. So kann etwa in spezialisierten hochperfomanten Services kleinster, durch Instrumentalisierung entstehender Mehraufwand, deutlich merkbar sein.[17].

[16] [Sig+10] "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure". 2010.
[17] [Sig+10] "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure". 2010.

Benutzbarkeit Die Benutzbarkeit des Tracingsystems soll durch die Verwendung von Standards gewährleistet sein. Die von *Opentracing* veröffentlichte API soll der Instrumentalisierungsbibliothek eine vertraute und bewährte Anwendererfahrung liefern.

Portabilität Ein weiteres Designziel soll eine gegebene **Portabilität** sein. Die Umgebung für die das Tracingsystem entwickelt wird, ist eine Mischung aus plattformabhängigen und plattformunabhängigen Komponenten. Durch den verminderten Mehraufwand der bei plattformunabhängigen Komponenten entsteht, wird eine verbesserte Nutzerfreundlichkeit gewährleistet. Vor allem bei der Integration in bestehende Systeme wird dies bemerkbar, da Bauprozesse von Projekten plattformabhängig sind. Der platformabhängige Bauprozess soll im Falle des verteilten rendering System nicht beeinflusst werden.

Datenverfügbarkeit Die Datenverfügbarkeit soll zeitnah stattfinden. Die von der Tracinginfrastruktur generierten Tracingdaten sollen zur Laufzeit dargestellt werden.

4.1.2 Nicht-Ziele

Die von Dapper genannten Designziele der *Anwendungs-Level Transparenz* und der *Skalierbarkeit* spielen für das verteilte Rendering-System eine untergeordnete Rolle. Diese Bewertung hat ihren Ursprung aus einer interpretierten Form eines Sprichworts.

> Du bist nicht Google, also versuch auch nicht Google zu sein

Dapper ist für eine Infrastruktur konzipiert, die globalen Maßstäben entspricht. Das verteilte rendering System entspricht nicht diesen Maßstäben, somit soll auch die Tracinginfrastruktur diese nicht erfüllen müssen. Die beiden Deisgnziele von Dapper werden als Nicht-Ziele für das verteilte rendering System bewertet.

Anwendungs-Level Transparenz Instrumentalisierung erfordert Eingreifen in Anwendungsquellcode. Dapper löst dies durch Nutzung von Bibliotheken. Diese werden instrumentalisiert und in der Anwendungslogik verwendet. Somit ist die Tracinginfrastruktur für den Anwendungsentwickler nicht wahrnehmbar. Die Instrumentalisierung des verteilten Rendering-Systems soll an semantisch relevanten Bereichen stattfinden und flexibel sein. Dies gelingt durch direktes Modifizieren der Anwendungslogik. Der Anwendungsentwickler muss sich dementsprechend selbst um die Instrumentalisierung kümmern.

Skalierbarkeit Der Aufbau des verteilte rendering System ist, in seiner einfachsten Form, statisch. Das bedeutet, dass eine Skalierbarkeit keine zentrale Rolle in dem Design der Tracinginfrastruktur darstellt. Die Skalierbarkeit soll allerdings bewertet werden.

4.2 Datenmodell

In diesem Abschnitt wird der Lösungsansatz der Frage **F1** diskutiert. Diese Frage erfordert die Konzipierung eines Datenmodells, welches drei Aspekte berücksichtigt. Der erste Aspekt wird durch die Zeitspanne des Events selbst dargestellt. Dabei muss das Event derart repräsentiert werden, sodass eine Zeitspanne als Information daraus gewonnen werden kann. Die detaillierte Darstellung wird in Abschnitt 4.2.1 erläutert. Der zweite Aspekt entsteht durch die Gegebenheit eines lokalen Kontexts. Dieser wird in Abschnitt 4.2.2 beschrieben. Der dritte Aspekt ist durch die Kommunikation von Komponenten des Systems gegeben. Dieser wird in Abschnitt 4.2.3 dargestellt.

Das Datenmodell beruht auf dem Spanmodell, welches in der vorher beschrieben Fachpublikation *Dapper* vorgestellt, durch die Spezifikation von *Opentracing* erweitert und dem Anwendungsfall des verteilten Rendering-Systems und des Entwicklungsprojekts angepasst wird.

Ein Überblick über das Spanmodell wird gezeigt. Ein *Trace* bildet die größte Einheit des Modells ab. Traces sind Eventsammlungen, die den Weg einer Anfrage durch ein verteiltes System darstellen. Ein Trace ist die Reise der Anfrage, wohingegen einzelne Events die Etappen dieser Reise sind. Die Events werden weiterführend als Spans bezeichnet. Ein Trace beinhaltet ein bis beliebig viele *Spans*. Ein Span ist die kleinste Einheit eines Traces. Spans bilden einen Arbeitsprozess innerhalb eines Prozesses ab. Innerhalb eines *Threads* ist zu jeder Zeit nur ein Span *aktiv*.

4.2.1 Spans

Um Daten zu repräsentieren und zusammenzufassen, braucht es einen Datencontainer. Die Datencontainer eines Traces sind Spans. Spans beinhalten die Daten die nötig sind, um eine Reihenfolge herzustellen. Außerdem besteht die Möglichkeit Anwendungsinformationen, wie zum Beispiel Anwendungslogs, in Spans mitzuführen.

Ein Span kapselt folgende Informationen:

- Ein Operationsnamen
- Ein Startzeitpunkt
- Ein Endzeitpunkt
- Ein Spankontext

Der Spankontext wiederum ist auch ein Datencontainer. Dieser beinhaltet eine *SpanID* und eine *TraceID*. Auch ein Beziehungstyp wird mitgeführt. Der Beziehungstyp kann zum einen eine *Child-of* Beziehung oder eine *Follows-From* Beziehung annehmen. Diese beiden Beziehungstypen ergeben sich aus der zeitlich Anordnung von Events. Die in Abschnitt 2.1.3 beschriebene Ordnung bieten die Grundlage für die Unterscheidung dieser

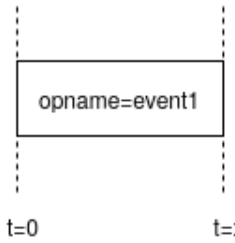
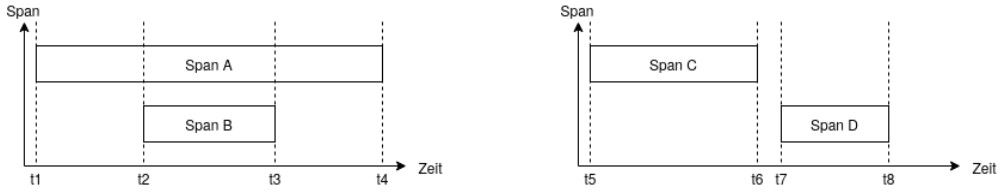


Abbildung 4.1: Darstellung eines Spans mit Anfangszeit, Endzeit und Operationsnamen



(a) Span B hält eine *Child-Of* Beziehung zu dem Elternspan

(b) Span D hält eine *Follows-From* Beziehung zu Span C

Abbildung 4.2

beiden Beziehungstypen. $t1$ ist der Beginn von Span A. Innerhalb dieses Events findet ein weiteres Event statt. Dieses wird durch Span B repräsentiert und ist zeitlich durch $t2$ und $t3$ markiert. Anschließend beendet Span A zum Zeitpunkt $t4$. Span B ist ein Kindspan von Span A, weil sie von Span A ausgelöst worden ist und noch vor Span A beendet. Eine Follows-From Beziehung äußert sich dahingehend, dass ein Span auf einen anderen folgt und eine Kausalität gegeben ist. Span C startet zum Zeitpunkt $t5$ und endet zum Zeitpunkt $t6$. Durch die Anwendungslogik, die Span C umfasst, wird Span D generiert, nachdem Span C beendet wurde. Span D erbt den Kontext von Span C. Die Zeitspanne von Span D ist durch $t7$ und $t8$ dargestellt.

Die SpanId ist eine eindeutig identifizierbare Nummer, die bei der Erstellung eines Spans generiert und in dem Spankontext gespeichert wird. Die TraceId ist eine eindeutig identifizierbare Nummer, die bei der Erstellung des ersten Spans eines Traces erstellt wird. Die TraceId wird auch in dem Spankontext gespeichert. Nachfolgende Spans, die zu diesem Trace gehören, übernehmen die TraceId. Dadurch ist eine Zuordnung jedes Spans zu einem Trace gegeben. Die zeitliche Einordnung des Startzeitpunktes und des Endzeitpunktes eines Spans ist in Abb. 4.1 dargestellt. Die Konzipierung der Start- und Endzeit sind die zentralen Daten, mit der sich Fragestellung **F1** lösen lässt. Durch die Relationbildung mittels einer Beziehungstypdefinition und der Zeitstempel, ist eine Zeitmessung von Eventzeitspannen gegeben. Das Beispiel aus Abb. 4.3 verdeutlicht dies. Die Gesamtzeit eines Events bzw. eines Spans lässt sich durch die Differenz zweier Zeitstempel ermitteln. Die Zeitspanne die von dem Span *FormatiereNachricht* eingenommen wird, ergibt sich aus $t3 - t2$. Dies wäre ein Event mit einer Lebensdauer von 0.5. Die Relationstypen werden dazu genutzt, um den Zustand des Traces zu definieren. Ein Trace

ist dann abgeschlossen, sobald der letzte Span eines Traces, ohne *Child-Of* Beziehung beendet ist. *EmpfangeAntwort* ist in diesem Fall der letzte Span. Außerdem besitzt dieser keinen übergeordneten Span, gegeben aus einer *Follows-From* Beziehung zum Ursprungsspan. Daraus folgert der Abschluss des Traces.

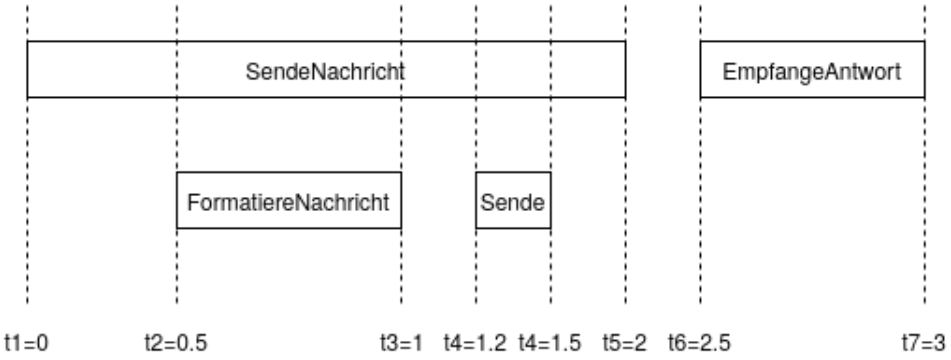


Abbildung 4.3: Darstellung eines Trace mit mehreren Spans. Anfangszeit und Endzeit ergeben die Gesamtzeit eines Traces

4.2.2 Tracingcontext innerhalb eines Systems

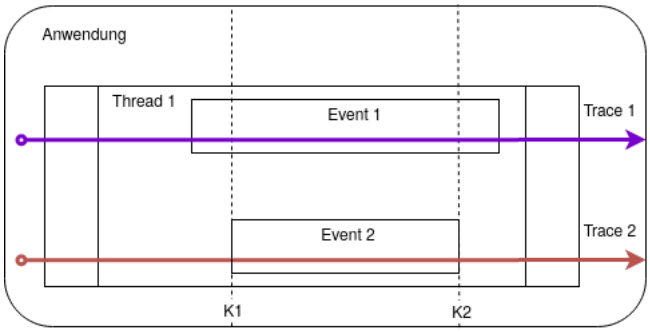


Abbildung 4.4: Anwendung mit einem Thread. Ein von der Instrumentalisierung veranlasster Kontextwechsel innerhalb eines Threads

Ein Prozess kann *Threads* implementieren. Die Events, die in einem Thread stattfinden, können ihren Lebenszyklus entweder in diesem beginnen und beenden oder in einem Thread beginnen und in einem anderen Thread enden. Der Tracer, also die übergeordnete Verwaltungseinheit, muss jedoch jederzeit feststellen können, welches Event *aktiv* ist. Der Grund dafür ist beispielsweise das Erstellen von Relationen für zu generierende Events, die ihr Elternevent kennen müssen. Eine Umgebung ist notwendig, in der der aktive Span festgestellt werden kann. Diese Umgebung wird als *Scope* bezeichnet. Der Kontext eines Traces ergibt sich aus dem aktiven Span. Der Kontext ist notwendig, um folgende Spans zuordnen zu können. Anhand der aufgeführten Beispiele wird das Konzept des Scopes verdeutlicht. Dabei beinhaltet das erste Beispiel, dargestellt in Abb. 4.4, einen Thread, indem zwei Traces parallel existieren. Die Abb. 4.5 stellt die Situation eines Traces über mehrere Threads dar.

Im ersten Beisiel sei gegeben, dass eine Anwendung ein Thread implementiert. Dabei existieren zwei parallele Traces. Trace 1 beinhaltet das Event 1. Event 1 sendet asynchron eine große Datei an ein Frontend, wie dies in dem verteilten Rendering-System der Fall sein könnte. Trace 2 beinhaltet Event 2. Event 2 liest eine kleinen Datei aus einem Speicher. Es wird davon ausgegangen, dass das Schreiben deutlich mehr Zeit beansprucht, als das Lesen. Der Ablauf des Kontextwechsels beginnt mit der Erstellung und Aktivierung des Spans, welcher Event 1 umfasst. Die Aktivierung des Spans sorgt dafür, dass dieser in den *Scope* wandert. Der Scope verwaltet den aktiven Span. Die Startzeit des Spans wird gespeichert und die Anwendungslogik asynchron bearbeitet. Event 2, welches zu Trace 2 gehört, startet und generiert den zweiten Span. Dabei findet der Kontextwechsel K1 statt. K1 tauscht den Span von Event 1 mit dem Span von Event 2. Der Span von Event 2 ist nun aktiv und der Span von Event 1 nimmt implizit den Zustand *nicht beendet* ein. Die Anwendungslogik, die Event 2 darstellt, wird bearbeitet. Das Event 2 beendet, woraus die Schließung des dazugehörigen Spans folgert. Die Endzeit wird gespeichert. Trace 2 schließt, da der Span von Event 2 keinen *ElternId* beinhaltet und somit der Wurzelspan ist. K2 wechselt wieder den aktiven

Span. Danach beendet Event 1, da der *Callback*, durch die Beendigung des Schreibens, aufgerufen wird. Entsprechenden schließt Span von Event 1. Auch hier ist dieser der Elternspan und schließt Trace 1 ab. Zu jedem Zeitpunkt kann des Tracerobjekt feststelle, welches das aktive Span ist.

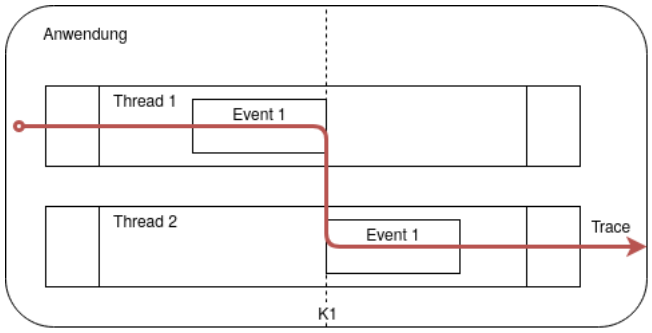


Abbildung 4.5: Anwendung mit zwei Threads. Ein von der Instrumentalisierung veranlasster Kontextwechsel zwischen Threads

Im zweiten Beispiel sei gegeben, dass eine Anwendung zwei Threads implementiert. Es exisitert ein Trace, welches Event 1 beinhaltet. Thread 1 beinhaltet Event 1. Event 1 sorgt für die die Erstellung eines Scopes in dem der Span von Event 1 erstellt und aktiviert wird. Kontextwechsel K1 findet statt. Der Scope in Thread 1 schließt. Da Event 1 in Thread 2 weitergeführt wird, beendet der Span nicht bei K1. In Thread 2 findet die Weiterführung von Event 1 statt, weshalb ein Scope eröffnet wird. In diesem Scope wird der in dem vorherigen Scope aktive Span eingeführt. Dies stellt die Kontextübermittlung dar. Event 1 endet anschließend und Trace 1 wird geschlossen.

Diese beiden Beispiele zeigen die Relevanz von Scopes zur Verfolgung des Kontexts innerhalb eines Systems ohne Netzwerkkommunikation. Scopes lösen das Problem der lokalen Kontextverfolgbarkeit. Damit lassen sich Endzeiten von Spans innerhalb eines Systems, auch über Threadgrenzen hinaus, bestimmen.

4.2.3 Tracingcontext über Prozessgrenzen

Die Kontextverfolgung über Prozessgrenzen stellt wohl das schwierigste Problem im verteilten tracing dar. Im Fokus steht hierbei die Fragestellung **F3**. Es werden zwei Konzepte vorgestellt, die es ermöglichen sollen, den Tracingkontext über Prozessgrenzen hinaus zu transportieren.

4.2.3.1 Registry

Das erste Konzept ist die **Traktor Registry**. Der Infrastrukturaufbau, bestehend aus der Registry und den Tracern, die mit der Registry verbunden sind, erfüllt die Auf-

gabe der Kontextpropagierung. Die Traktor Registry dient als Nachrichtenproxy für die Tracer-Instanzen. Bei der Initialisierung einer Traktorinstanz stellt diese eine Websocketverbindung zur Registry her. Bei eingeleiteter Kontextpropagierung wird eine Websocketnachricht, die den Tracekontext beinhaltet, an die Registry gesendet. Die Registry empfängt die Nachricht und sendet diese an ihr Zieltracer weiter. Der Zieltracer, der auch eine Verbindung zur Registry aufgebaut hat, empfängt die Nachricht. Der Nachrichteninhalt wird extrahiert und für die Erstellung für kommende Spans des aktuellen Traces und der damit verbundenen Relationsbildung genutzt. Die Verarbeitungsablauf dieses Registryservices lässt sich aus Abb. 4.6 erschließen.

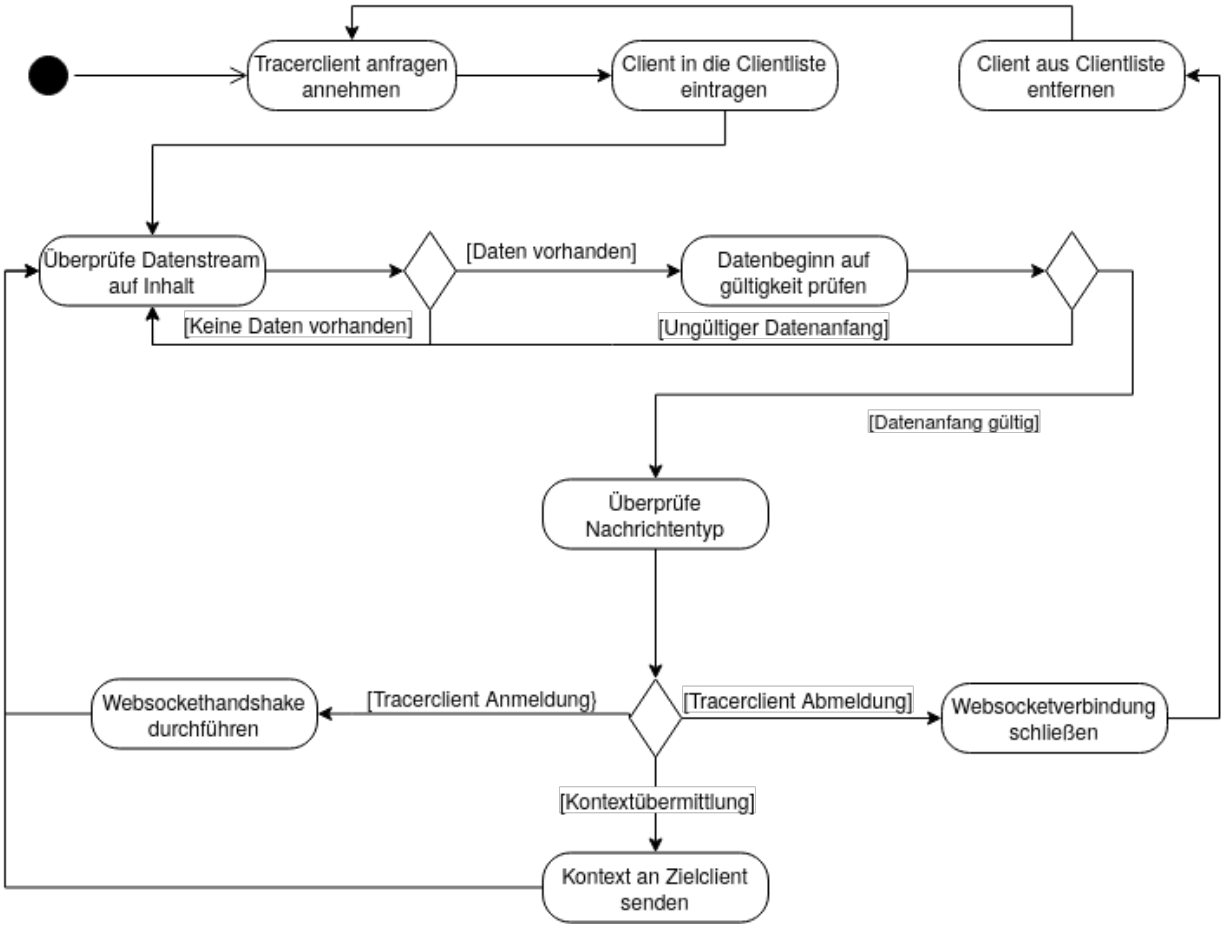


Abbildung 4.6: Aktivitätendiagramm der Traktor Registry

4.2.3.2 Interceptor

Das zweite Konzept beruht auf dem Abfangen und dem Modifizieren von spezifizierten Nachrichten, die aus dem System gesendet werden. Es werden betriebssystemabähngige

Funktionalitäten genutzt, um ausgehende Nachrichten zu identifizieren und einzuordnen. Dabei wird auf Nachrichten gelauscht, die von einem Port ausgehen, die die Anwendungskomponente belegt. Entspricht diese Nachricht den Vorgaben, kann diese um Kontextdaten erweitert werden. Anschließend wird die Nachricht in modifizierter Form an ihren Empfänger weitergeleitet. Auf dem Empfängersystem sitzt ein gleicher *Interceptor*. Dieser lauscht auf auf eingehende Nachrichten und filtert, wie beim Senden, die gewünschten Nachrichten heraus. Der Tracingkontext wird aus der Nachricht extrahiert. Anschließend wird die Nachricht, so wie sie von der Senderanwendung geschickt wurde, an die Empfängeranwendung weitergeleitet. Abb. 4.7 zeigt die Nachrichtenmodifikation, bei der eine Anwendung auf dem Sendersystem Nachrichten über die Ports 8080 und 8090 an eine Empfängeranwendung sendet. Ports die nicht von der Anwendung beansprucht werden, wie zum Beispiel Port 1337, sind von der Modifikation nicht betroffen. Der Interceptor erweitert die Nachrichten um den Trancingkontext, welches von dem Emfpängerinterceptor entpackt we alität lässt sich die Registry als potentielle zentrale Fehlerq erden dazu auf jedem System zusätzliche, neben der Anwend igt.

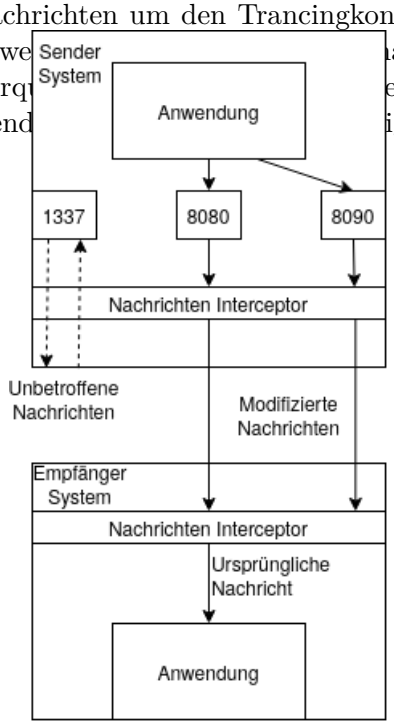


Abbildung 4.7: Design des Nachrichteninterceptor

4.3 Verarbeitungsmodell

Das Verarbeitungsmodell ist das Bindeglied zwischen Visualisierung und Instrumentalisierung. Dabei sind zwei Komponenten zu designen. Die erste Komponente hat dafür zu sorgen, dass die Daten aus der Anwendung gelangen und zu einem Service übermittelt werden, der die Daten sammelt. Dafür werden *Reporter* eingesetzt. Die zweite Komponente ist der *Kollektor*. Dieser erhält alle Daten der Reporter. Die gesammelten Daten werden in eine strukturierte Form aufgearbeitet. Anschließend können die Daten, abhängig von der Konfiguration, gespeichert oder direkt visualisiert werden.

4.3.1 Reporter

Bei Abschluss eines Spans wird dieser an den Reporter übermittelt. Durch eine UDP Verbindung zu dem Kollektor werden die Spans versendet. Der Trace wird von der Verwaltungseinheit parallel weitergeführt, bis dieser abgeschlossen ist. Der Reporter kennt den Zustand des gesamten Traces nicht.

4.3.2 Kollektor

Der Kollektor ist ein eigener Service im verteilten System. In dem Kollektor werden die reporteten Spans gesammelt. Der Kollektor stellt einen UDP Endpunkt bereit, über den die Reporter daten übermitteln können. Diese können in einer relationalen Datenbank gespeichert werden. Das Datenmodell der Traces und Spans bietet sich dafür an. Anhand der TraceId lassen sich die dazugehörigen Spans ermitteln. Jede TraceId ist ein Eintrag in der Datenbank. Die Datenverfügbarkeit ist entsprechend gegeben.

4.4 Visualisierung

In diesem Kapitel sollen Darstellungsformen von Tracingdaten präsentiert werden. Dabei werden bestehende Visualisierungsmöglichkeiten von Performancedaten aufgegriffen und dem Kontext von distributed tracing, sowie dem verteilten Rendering-System angepasst.

4.4.1 Frame Galerie

Das verteilte Rendering-System generiert Frames. Die Instrumentalisierung des System hat den Zweck einzelne Frames untersuchen zu können. Eine abstrakte Darstellung der Tracingdaten zur Untersuchung der Frames bietet sich daher an. Bereits existierende Tracingdarstellungen, bietet eine sehr eingeschränkte und spezifizierte Ansicht der Daten. Das Konzept der Frame Galerie soll die Daten interpretieren und für den Anwender eine Abstraktion der Tracingdaten bereitstellen.

Das Visualisierungskonzept ist durch drei Ebenen gekennzeichnet. Jede Ebene liefert einen gewissen Grad von Informationen. Dabei bietet die oberste Ebene, dargestellt in Abb. 4.8, eine Übersicht über alle Frames. Grenzwertüberschreitungen könne beispielsweise ein Indiz für fehlerhafte Frames sein, die es weiter zu untersuchen gilt. Da nur ein

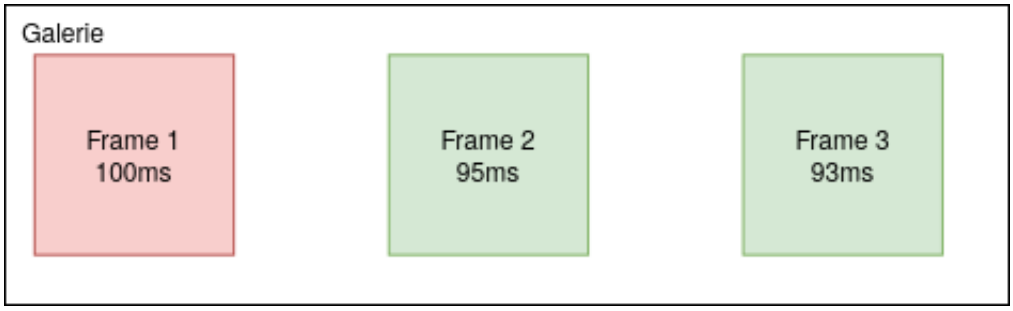


Abbildung 4.8: Skizzierung der oberste Ebene der Frame Galerie

Da das verteilte Rendering-System konzeptionell in der Lage sein soll, Frames aus Teilframes zusammenzustellen, ist es sinnvoll, den Generierungsprozess der fertigestellten Frames nachvollziehen zu können. Dieser Visualisierungsansatz ist nur dann sinnvoll, sobald die Architektur mehrere Renderer und Aggregatoren umfasst. Aggregatoren wären Systemkomponenten, die die Teilframes sammelt und zusammenführt. Die mittlere Ebene, dargestellt in Abb. 4.9, veranschaulicht den Ursprung durch ein Stammbaum. Es müssen nicht alle Teilframes, aufgrund ihres zeitlichen Auftretens in der Generierungsphase, bei der Zusammenstellung verwendet werden. Zusätzlich sind die Generierungszeiten aus den Tracingdaten hergeleitet und an die jeweiligen Frames geheftet. Bei einem Grenzwertübertritt werden diese farblich hervorgehoben.

Die unterste Ebene ist Detailansicht der Spans. Hierzu wird die klassische Visualisierungform von Tracingdaten verwendet. Die Servicenamen, die Operationsnamen und die dazugehörigen Verarbeitungszeiten der Spans sind erkenntlich. Zusätzlich können *Breakpoints* gekennzeichnet werden. Durch Visualisierung von beispielsweise gestrichelten Linien, wie in Abb. 4.10 gezeigt, ist ersichtlich, dass Frames, die ab diesem Zeitpunkt generiert werden, für die Zusammenführung ungenutzt bleiben.

Dem Anwender der Visualisierung ist durch die Ebenenaufteilung möglich, für ihn interessante Aspekte auszuwählen und gegebenenfalls Fehlerquellen weitergehend zu untersuchen.

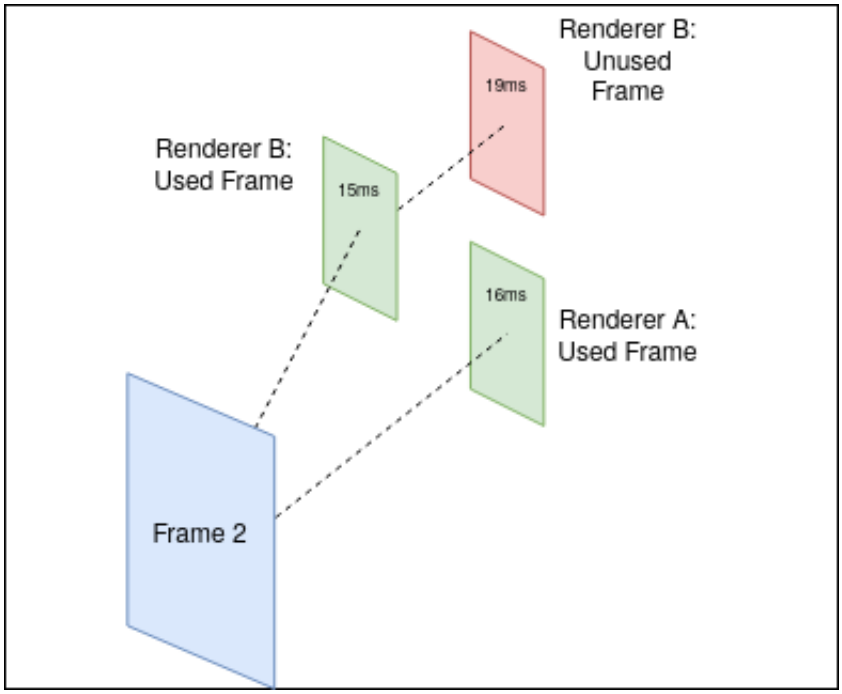


Abbildung 4.9: Skizzierung der mittleren Ebene der Frame Galerie

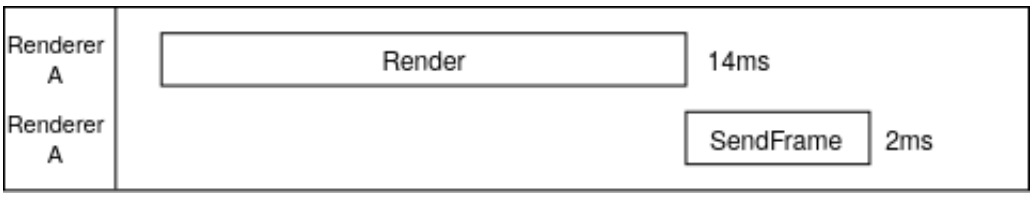


Abbildung 4.10: Skizzierung der unteren Ebene von Teilframe aus Renderer A

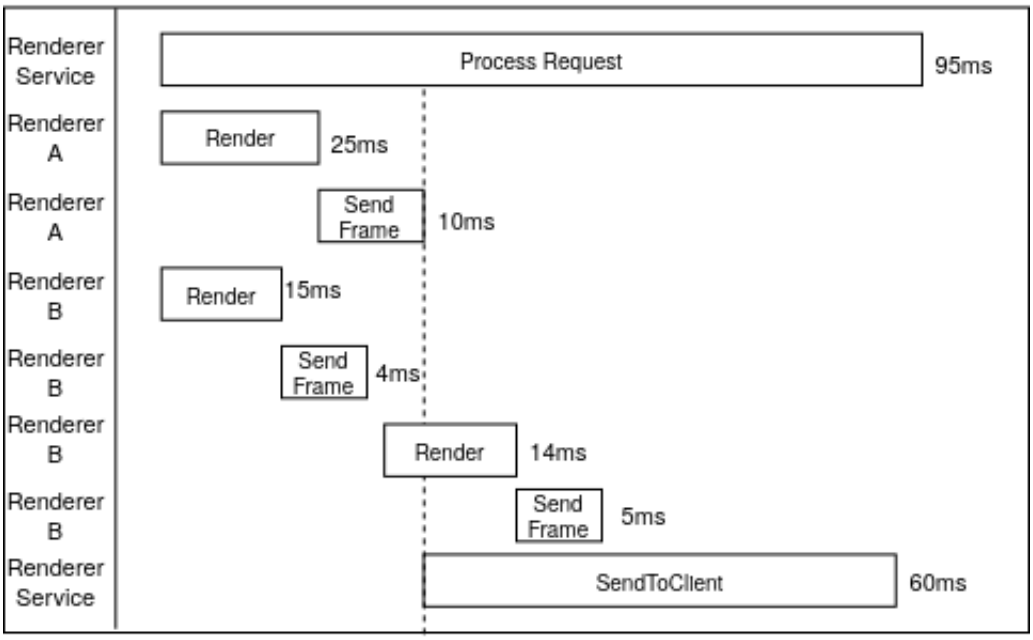


Abbildung 4.11: Skizzierung der unteren Ebene von Frame 2

4.4.2 Dreidimensionaler Flammengraph

Das zweite Visualisierungskonzept nimmt sich Flammengraphen, gezeigt in Abb. 3.5, und Sequenzdiagramme als Vorbild.

Dabei werden Spans aufeinandergestapelt. Spans werden als rechteckige Flächen dargestellt und farblich hervorgehoben. Diese werden anhand ihrer Berarbeitungsreihenfolge geordnet. Je tiefer die Operation in dem Tracestack ist, desto höher ist diese im Graph. Diese Eigenschaft wird mit dem klassichen Flammengraph geteilt. Spans werden zudem durch Services, in denen sie generiert wurden, aufgeteilt. Die Relationen zueinander, sind durch Linien gekennzeichnet. Dadurch lassen sich Kausalitäten zwischen Spans, sowie die dazugehörigen Services feststellen.

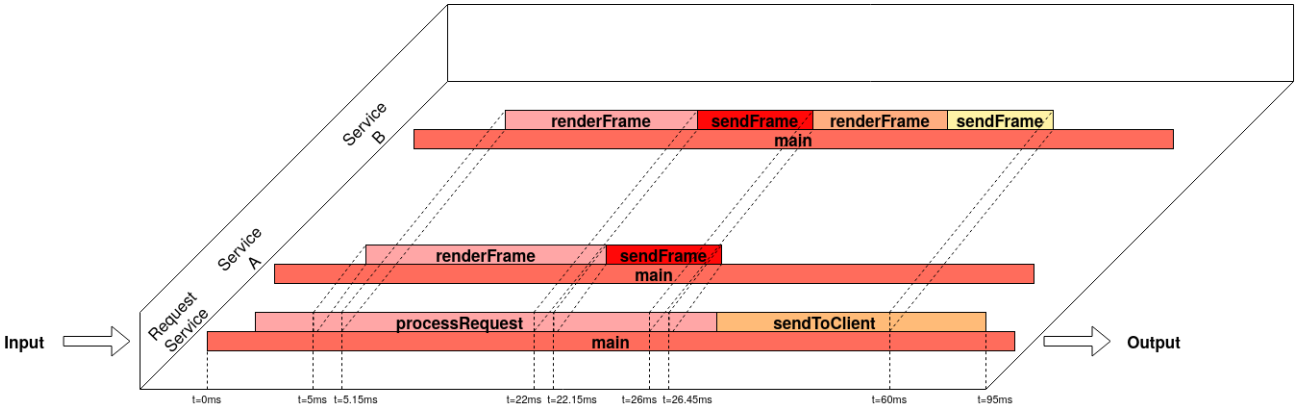


Abbildung 4.12: Darstellungsbeispiel eines 3D-Flammengraph

Diese Visualisierungsform soll ein Vergleich zwischen Traces ermöglichen. Die Traces werden zusammen auf eine Zeitachse gestellt. Der Anweder kann die Traces anschließend vergleichen. Dadurch sollen nicht verwendete Frames identifizerbar sein. Auch die Ursache soll damit ermittelt werden können. Spezielle Zustände des verteilten verteilten rendering System, wie zum Beispiel dem Setzen einer neuen Kameraposition, ausgelöst durch den Client, können näher untersucht werden. Das Setzen ist eine asynchrone Funktionalität, die im rendering System dazu führt, dass eine neue Framegenerierung angestoßen wird. Dabei werden allerdings aktuell bearbeitete Frames nicht gestoppt, sondern fertiggestellt und gesendet. Diese werden nur nicht weiter verwendet. Um die zeitliche Einordnung dieser Events darstellen zu können, müssen die Traces miteinander verglichen werden. In Abb. 4.13 wird eine solche Situation gezeigt. Der Trace startet im Requestservice. Der Span mit dem Operationsnamen *processRequest* sorgt im Render Service A für zwei Framegenerierungen. Der erste *generateFrame* Span schließt zum Zeitpunkt **t1** erfolgreich ab, entsprechend wird der Frame gesendet und verwendet. Die Generierung wird mit dem zweiten generateFrame fortgesetzt. Anschließend wird der erste ProcessRequest durch eine Benutzereingabe zum Zeitpunkt **t2** unterbrochen. Der zweite Trace beginnt. Der Requestservice veranlasst die Generierung des Frames mit

den neuen Benutzerdaten im Render Service B zum Zeitpunkt **t3**. Der Render Service A stellt seinen Frame zum Zeitpunkt **t4** fertig. Da der Frame auf Benutzereingaben beruht, die zu diesem Zeitpunkt veraltet sind, wird der Frame nicht weiter verwendet. Trace 1 schließt ab. Währenddessen wird Trace 2 fortgesetzt.

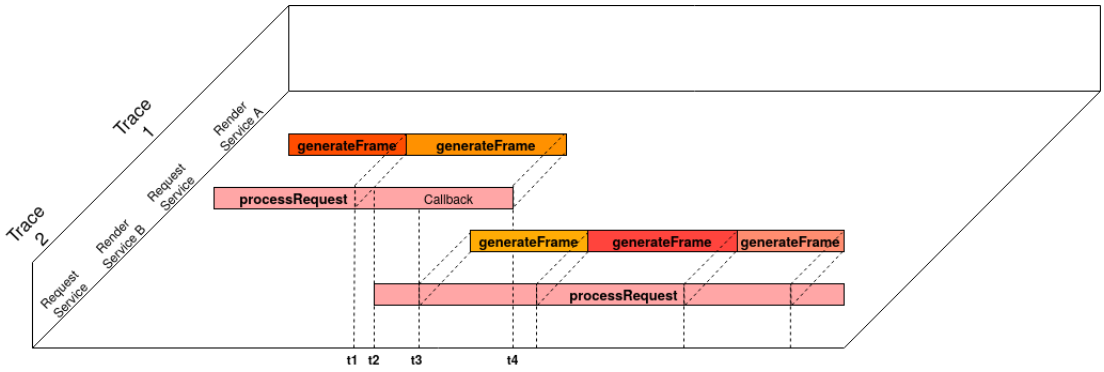


Abbildung 4.13: Tracevergleich mit einem 3D Flammengraph

4.5 Trakorentwicklungsumgebung

In diesem Kapitel wird die Traktorentwicklungsumgebung spezifiziert. Die Umgebung soll die verschiedenen Anwendungsfälle der Traktorbibliothek abdecken. Eine Entwicklungsumgebung ist dahin gehend sinnvoll, dass der Anwendungsfall des verteilten rendering Systems komplexe abhängigkeiten aufweist, bei der eine effiziente Entwicklung, z.B. durch die platformabhängigen Plugins oder der Unityumgebung, beeinträchtigt wird.

Die Traktorentwicklungsumgebung soll folgende Eigenschaften aufweisen:

- **TE**.1 Die Entwicklungsumgebung soll aus mehreren Webservern bestehen
- **TE**.2 Die Infrastruktur soll auf genau einem Host-System ein verteiltes System abbilden.
- **TE**.3 Die Webserver-Komponenten der Entwicklungsumgebung soll mit der Traktor-API instrumentalisiert sein.
- **TE**.4 Das Ansprechen der Entwicklungsumgebung soll durch einen Kommandozeilen HTTP Client ansprechbar und bedienbar sein.

Diese Spezifikation ermöglicht das Testen von Funktionalitäten der Traktor Tracingbibliothek. Dabei sollen die (i) Eventgenerierungfunktionalitäten ausgeführt werden. Ausserdem soll durch durch die Instrumentalisierung von zwei miteinander kommunizierenden Komponenten (ii) die Kontextpropagierung durchgeführt werden. Das umfasst sowohl die lokale Kontextverwaltung, als auch die Kontextpropagierung über Prozessgrenzen hinaus. Zudem sollen die leichtgewichtigen Services, also den Reporter und die

Registry, getestet werden. Dabei ist die (iii) Implementierung des Websocketprotokolls innerhalb der Registry, als auch der (iv) Websocketclient des Tracers zu untersuchen. Mit dieser Spezifikation wird ein Untersuchungssystem beschrieben, mit der teilweise die Designziele überprüft werden können.

5 | Implementierung

In diesem Kapitel werden Implementierungen von Prototypen gezeigt, die im Kapitel Kapitel 4 besprochen sind. In Abschnitt 5.1 werden die implementierungsdetails der Instrumentalisierungsbilbiothek **Traktor** vorgestellt. In Abschnitt 5.2 und in Abschnitt 5.3 werden die Traktor Services beschrieben. Anschließend werden zwei Anwendungsfälle gezeigt. Der Anwendungsfall des rendering System in Abschnitt 5.4 und der Anwendungsfall einer verteilten Webanwendung in Abschnitt 5.5

5.1 Instrumentalisierungsbibliothek: Traktor

In diesem Kapitel wird die Implementierung der Instrumentalisierungsbibliothek Traktor vorgestellt. Es wird das Datenmodell umgesetzt, welches in Abschnitt 4.2 konzipiert ist und gezeigt, inwiefern Traktor die Anforderungen erfüllt.

Traktor ist eine Instrumentalisierungsbibliothek, die sich die OpenTracing API als Vorbild nimmt. Die OpenTracing API ist eine herstellerneutrale Instrumentalisierungs-API, die unter der Schirmherschafft der Cloud Native Computing Foundation (CNCF) steht, welches Teil der gemeinnützig agierenden Linux Foundation (LF) ist.

Die zwei zentralen Einheiten der Traktor Instrumentalisierungsbibliothek sind der in Abschnitt 5.1.1 gezeigte **Tracer** und der in Abschnitt 5.1.2 **Spans**. Weitere Klassen, die den in Abschnitt 4.2.2 beschriebenen Designansatz implementieren, sind der in Abschnitt 5.1.5 vorgestellte Scope und der in Abschnitt 5.1.6 vorgestellte ScopeManager. Die Kontextpropagierung, die mit einer Websocketverbindung innerhalb des Tracer und der, in Abschnitt 5.3 gezeigten, Traktor Registry, umgesetzt ist, implementiert den in Abschnitt 4.2.3 erläuterten Designansatz. Ein Gesamtüberblick mit allen Relationen der Klassen, wird in Abschnitt 5.1.8 gegeben.

5.1.1 Tracer

Der Tracer ist die zentrale Verwaltungseinheit der Instrumentalisierung. Dieser verwaltet die Verbindung der Anwendungsinstrumentalisierung zu dem Agenten und der Reg-

istry. Der Tracer kümmert sich um das Generieren von Spans und die Herstellung von Kausalzusammenhängen zwischen den Spans. Ausserdem stellt der Tracer die Funktionalitäten zur Kontextpropagierung über die Registry bereit. Die Spans sind Darstellungen von ausgeführter Arbeit in einer instrumentalisierten Anwendung.

Die Tracer API erweitert die OpenTracing API um vier Funktionen, die notwendig sind, um sich mit der Tracinginfrastuktur zu verbinden. Die *Configure* Funktion sorgt für einen Verbindungsaufbau, abhängig von den übergebenen Parametern. Ein vollständiger Verbindungsaufbau ist nötig, um eine Kontextpropagierung zu ermögliche. Ausserdem wird der User Datagramm Protocol (UDP)-Klient initzialisiert, damit beendete Spans reportet werden können. Der UDP-Klient hat einen Port zu belegen, der angegeben werden muss. Die IP-Adressen der Services sind dem Tracer bekannt zu geben. Auch der Port auf der die Anwendung lauscht, muss übergeben werden. Die Configure Funktion verwendet die *Register* Funktion zur Erstellung einer **WebSocketClient** Instanz. Die Websocketverbindung wird über die Lebensdauer des Tracer offen gehalten. Über diesen Kommunikationskanal werden die Kontextinformationen übermittelt. Listing 5.2 zeigt eine Beispielhafte Konfigurierung. Dabei wird eine Tracerinstanz instanziiert und mit den Adressen der Traktorservices konfiguriert.

```
1  using Traktor;
2
3  var registryAddress = "localhost";
4  var registryPort = "8080";
5  var agentAddress = "localhost"
6  var agentPort = 13337;
7  var reporterPort = 13338;
8
9  Tracer tracer = new Tracer()
10 tracer.Configure(registryAddress,registryPort,agentAddress,
11    agentPort,reporterPort);
```

Listing 5.1: Tracer Verbindungsaufbau mit der Registry und dem Reporter

Die Kontextdaten werden in einer *Carrier* Instanz gespeichert und über die Leitung gesendet. In der Implementierung gibt es genau einen Carriertyp. Der BinaryCarrier ist ein Datencontainer, der die Kontextinformationen als MemoryStream speichert. Der MemoryStream speichert die Daten im Arbeitsspeicher als Bytearray. Das Format wurde für die Übertragung über eine TCP/IP Verbindung ausgewählt. Die Spangenerierung wird durch eine *SpanBuilder* Instanz geregelt. Der Operationsname wird der SpanBuilder Instanz mitgegeben und für die Generierung des Spans genutzt. Die Kontextpropagierung wird durch die beiden sich ergänzenden asynchronen Funktionen *SendContext* und *ReceiveContext* implementiert. Diese nutzen die *Extract* und *Inject*, um die BinaryCarrier zu erstellen, beziehungsweise die Daten aus dem Carrier zu extrahieren.

Der Tracer hält eine **Scopemanager** Instanz. Der Scopemanager verwaltet die Spans. Über den Scopemanager kann der Tracer den aktuell aktiven Span ermitteln.

5.1.2 Span

Der Span beinhaltet alle relevanten Daten, die für das Repäsentieren eines Events in einem System notwendig sind. Darunter zählt ein Operationsname. Der Operationsname kann beispielsweise die Sematik des Events beschreiben. Auch bietet sich ein Funktionsname innerhalb der Anwendung an, falls der Span genau diesen umfasst. Der **Startzeitstempel** und der **Endzeitstempel** beschreibt die Zeitspanne, in der das Event stattfindet. Diese sind in der Implementierung *Datetime*-Instanzen. Das Datetimeformat, welches für die Tracinginfrastruktur genutzt wird, hat folgenden Aufbau:

```
1 MM/dd/yyyy hh:mm:ss.ffff tt
```

Listing 5.2: Zeitstempelformat der Spans

Der Aufbau orientiert sich an dem amerikanischen Zeitformat. **MM** steht für den Monat, **dd** für den Tag und **yyyy** für das Jahr. Interessanter wird es bei dem Tageszeitformat. In einem System, bei dem Zeit eine kritische Rolle spielt, sind kleinste Zeitunterschiede von Bedeutung. **hh** und **mm** sind entsprechend die Stunden und Minunten. **ss.ffff** entspricht den Sekunden und dem Tausendstel einer Sekunde. Die **Systemuhr Resolution** ist an dieser Stelle erwähnenswert. Die Resolution einer Uhr beschreibt die kleinste Einheit von Zeit, die akkurat von einer Uhr gemessen werden kann. Die Resolution der Systemuhr hängt von dem Betriebsystem ab. In einem Windows 8 Betriebsystem ist die Standardresolution bei 15.6 ms. Das .NET Framework, welches für die Implementierung genutzt wird, behandelt verschiedene genutzte Softwaretimer wiederum eigenständig. Diese kann manuell auf 0.5 reduziert werden. In einem Linux-basierenden Betriebssystem hängt die Resolution von der Software Clock ab. Dabei wird die Zeit in sog. **Jiffies** gemessen. Die Linux-Kernel version 2.6.0 führt eine jiffiegröße von 0.001 sekunden ein[18]. Dementsprechen sind die gemessenen Zeitstempel mit diesen Hintergrundinformationen zu bewerten.

Ein Span besitzt einen Spankontext und eine Referenzliste. Der Inhalt des Spankontext wird in Abschnitt 5.1.4 beschrieben. Die Referenzliste beinhaltet alle Spans, die eine **Happens-Before Relation** zu dem Span aufweisen. Das bedeutet, dass alle Elternspans, beispielsweise aus *Child-Of* und *Follows-From* resultierenden Beziehungen, in dieser Liste referenziert werden.

Schlussendlich kennt ein Span seinen Tracer. Dies ist erforderlich, da beim fertigstellen des Spans die *Finish* Funktion aufgerufen wird. Diese sorgt dafür, dass der Endzeitstempel gesetzt wird. Ausserdem wird die *Report* Funktion der Reporterinstanz, bekannt durch den Tracer, aufgerufen.

[18] [20] *perf(1) Linux User's Manual.* 2020.

5.1.3 SpanBuilder

Der SpanBuilder ist die Bindegliedentität zwischen dem Tracer und der Spangenerierung. Der Anwender der Instrumentalisierungsbibliothek kann durch die *BuildSpan* Funktion des Tracer die Spangenerierung einleiten. Dabei wird eine SpanBuilder Instanz erstellt, die die übergebenen Parameter zur Spangenerierung nutzt. Der SpanBuilder verlang, während der Initialisierung, nach einem Operationsnamen und der Referenz des aufrufenden Tracers. Diese werden als Felder in der SpanBuilder Instanz gespeichert. Durch die Funktionen *AddReference* und *AsChildOf* kann das Feld *references* bearbeitet werden. Dabei ist AsChildOf eine spezialisierte Form der AddReference Funktion. Die AddReference Funktion erwartet einen Referenztypen und einen SpanContext als Paramenter. Diese werden dann für den zu bauenden Span genutzt.

Eine weitere Grundfunktion ist die *Start* Funktion. Diese beinhaltet die nötige Anwendungslogik, für die Traceidentifikationsnummergenerierung. Auch die Spanidentifikationsnummer wird generiert. Anschließend wird der Span gebaut und mit den gesammelten Daten initialisiert.

Ähnlich wie bei der AddReference Funktion, ist die *StartActive* Funktion eine Spezialisierungen der Start Funktion. Die überladene StartActive Funktion kann Parameterlos oder mit dem Boolean *finishSpanOnDispose* aufgerufen werden. Der Boolean bestimmt, wie mit einem Span umgegangen wird, nachdem die entsprechende Finish Funktion aufgerufen worden ist. StartActive setzt den gebauten Span auf den Zustand **Aktiv**. Das bedeutet, dass der Scopemanager diesen verfolgt und gegebenfalls den vorherig aktiven Span zwischenspeichert.

Eine beispielhafte Nutzung des Spanbuilders sieht folgendermaßen aus:

```
1 var operationname = "example_function";
2 var spanBuilder = tracer.BuildSpan(operationname);
3 var span = spanBuilder.Start();
```

Listing 5.3: Beispielhafte Anwendung des SpanBuilder

Ein Operationsname wird definiert und als Paramenter für die BuildSpan Funktion genutzt. Anschließend wird durch die Start Funktion der SpanBuilder Instanz ein Span auf den Zustand Aktiv gesetzt.

5.1.4 SpanContext

Die SpanContext Klasse beinhaltet eine TraceId, eine SpanId und einen Rerenztypen als Felder. Diese werden bei der Initialisierung gesetzt. Die SpanContext-Klasse ist ein reine Datenkapselung. Semantisch sind die in der SpanContext enthaltenen Daten jene, die in einem BinaryCarrier, codiert als Bytearray, über die Leitung gesendet werden.

Ein Auszug eines Spankontexts aus dem Entwicklungssystem zeigt den Aufbau. Der Auszug stammt aus dem Registryservice. In dem Service wird der Nachricht auf die Konsole ausgebenen, sobald eine Propagiererung ausgelöst worden ist.

```
1 traktor-registry_1 | Client-Message: WIoJiNwldhLM;ZtDdH4lQ1a1b;child_of
```

Listing 5.4: Ein über die Registry propagierter Spankontext

Der Spankontext beinhaltet drei Daten. Eine TraceId, eine SpanId und einem Relationstyp.

5.1.5 Scope

Ein Scope ist ein lokaler Kontext, welcher von dem ScopeManager verwaltet wird. In einem Scope werden Spans verwaltet. Dadurch wird das in Abschnitt 4.2.2 beschriebene Problem der Kontextverfolgbarkeit innerhalb eines Systems gelöst. Ein Scope wird in einem Prozess geführt. Während eines Kontextwechsel wird der aktive Span in einer Variable gespeicher und anschließend wird der aktive Span mit dem zu aktivierenden Span ersetzt. Dadurch ist bei einer Beendigung des aktiven Spans der vorherige Span wiederherstellbar. Traktor nutzt die AsyncLocalScope CSharp Implementierung des OpenTracing-Projekts.

5.1.6 ScopeManager

Ein ScopeManager verwaltet Scopes. Bei einem Kontextwechsel, zum Beispiel bei einer Multi-Thread Implementierung, werden die Scopes durch den Manager verfolgt. Ein Scope ist nur innerhalb seines Prozesses relevant und beinhaltet einen aktiven Span. Über den Spanmanager lässt sich der Span, welcher durch einen aktiven Scope umfasst wird, ermitteln. Der Tracer ist dadurch jederzeit in der Lage, den aktiven Span und seinen Kontext zu nutzen. Wie auch der Scope, wird die ScopeManager Implementierung des OpenTracing-Projekts genutzt.

5.1.7 Reporter

Die Reporter Klasse kapselt die UDP-Verbindung des Tracers zu dem Agentenservice. Dabei wird die Netzwerkddresse, sowie der Port gespeichert. Bei der Initialisierung des Reporters, wird die Verbindung mit den gegebenen Parametern hergestellt.

Die Reporter Klasse umfasst drei Funktionen. Die Funktion *Connect* stellt mit gegebenen Parametern die Verbindung zu dem UDP-Service her. Der UDP-Klient wird als Feld gespeichert und kann zur weiteren Datenübertragung genutzt werden. Die Funktion *Report* sendet eine kodierte Form des Spans zu dem Agentenservice. Die Kodierung wird

durch die Funktion *BuildMessage* implementiert. Traktor nutzt eine UTF-8 Kodierung für den Nachrichtenaustausch.

5.1.8 UML-Klassendiagramm der Bibliothek

Die Instrumentalisierungsbibliothek ist in der Programmiersprache CSharp implementiert. Die Programmiersprache eignet sich für die Entwicklung von Systemkomponenten, die in einer verteilten System zum Einsatz kommen, wie aus der Spezifikation der Programmiersprache entnommen werden kann:

> C# ist gedacht als simple, moderne, universal einsetzbare, objektorientierte Programmiersprache. [...] Die Sprache ist für die Entwicklung von Softwarekomponenten gedacht, die in einer verteilten Umgebung bereitgestellt werden.[19]

Das Klassendiagramm, gezeigt in Abb. 5.1, stellt einen Überblick über die vorherig beschriebenen relevanten Entitäten der Bibliothek dar.

[19] [HWG03] *C# Language Specification.* 2003, S. xx.

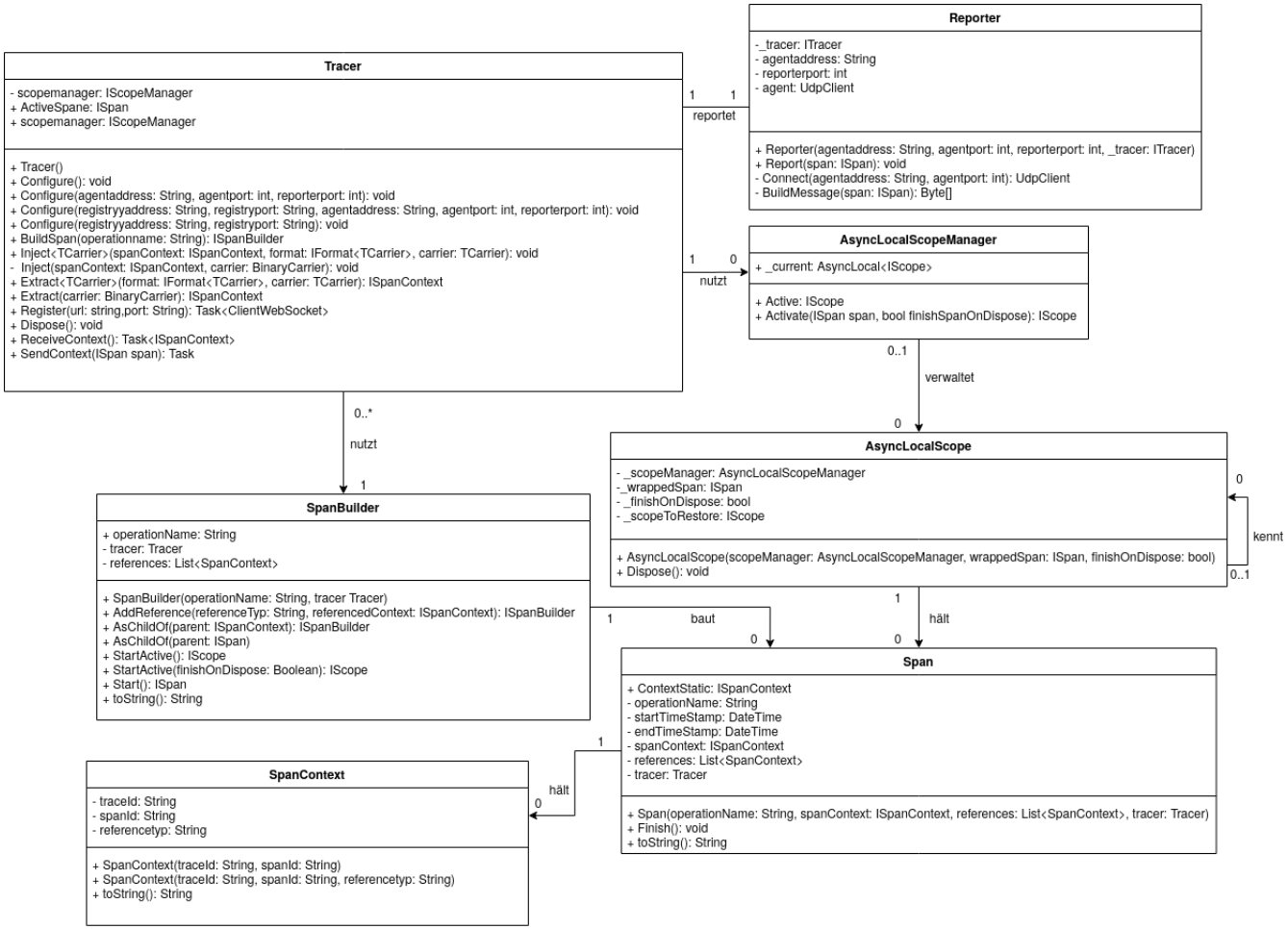


Abbildung 5.1: Klassendiagramm der Traktor Instrumentalisierungsbibliothek

5.2 Traktor Agent

Der Traktoragent ist ein Service, welcher ein UDP-Endpunkt bereitstellt. Der Service gibt alle erhaltenen Nachrichten auf seiner Konsole aus. Zwei Umgebungsvariablen werden genutzt, um den UDP-Endpunkt zu initialisieren. Die **UDP_IP** ist der *Localhost*, da der Service auf einer eigenständigen Komponeten bereitgestellt wird. Dies ist durch den Spezifikikationspunkt **TE**.2 gegeben. Der belegete Port wird durch die Umgebungsvariable **UDP_PORT** konfiguriert.

Ein reporteter Span kann folgendermaßen aussehen:

```
1 traktor-agent_1| recieved message:
2   b'Process Context;
3   04/14/2020 10:10:06.1791 PM;
4   WIoJiNwldhLM;ZtDdH4lQ1a1b;
5   child_of;
6   04/14/2020 10:10:06.4158 PM'
7 traktor-agent_1| from:  ('172.22.0.5', 13338)
```

Listing 5.5: Ein reporteter Span. Der gezeigte Span ist in der Webserver Entwicklungsumgebung generiert worden

Der Service *traktor-agent_1* erhält von der Netzwerkaddresse 172.22.0.5:13338 die dargestellte Nachricht eines Spans mit dem Operationsnamen *Process Context*. Der Startzeitstempel und der Endzeitstempel sind Teil des reporteten Spans. Auch der Spankontext ist dargestellt.

5.3 Traktor Registry

Die Traktor Registry ist ein Websocket Server. Verbindungsanfragen, initiiert von einer Tracerinstanz, werden als eigenständiger Thread in einer Klientenliste der Registry gespeichert. Die Anwendungslogik der *ClientHandler*-Threads implementiert die Websocket Handshakes. Das Websocketprotokoll wird in Abschnitt 5.3.1 beschrieben. Die Kontextpropagierung wird durch den Websocket Server umgesetzt. Die Implementierung wird in Abschnitt 5.3.2 vorgestellt.

5.3.1 Websocketprotokoll

Das Websocketprotokoll ermöglicht eine vollduplex Kommunikation zwischen der Registry und dem entsprechenden Websocketklienten eines Tracers. Das Protokoll nutzt dafür eine einzige TCP-Verbindung. Das Websocketprotokoll wird genutzt, um einen geringen **Overhead** der Kommunikation zu gewährleisten. Dementsprechend wird das

Designziel der **Verarbeitungskosten** berücksichtigt. Das Websocketprotokoll ist durch Request for Comments (RFC) mit der Nummer 6455 spezifiziert.

Das Websocketprotokoll verlangt einen öffnenden Handshake zur Etablierung der Verbindung. Dieser basiert auf einem HTTP Handshake. Die eröffnende HTTP-Anfrage beinhaltet eine *Upgrade*-Anforderung, der in diesem Moment genutzten HTTP-Verbindung. Der folgende Logausschnitt zeigt eine solche Eröffnungsnachricht:

```
1 GET / HTTP/1.1
2 Host: traktor-registry:8090
3 Connection: Upgrade
4 Upgrade: websocket
5 Sec-WebSocket-Version: 13
6 Sec-WebSocket-Key: s4VKefPmikGz1rJ24buoaQ==
```

Listing 5.6: Eröffnende Nachricht eines Websocket Handshake

Die HTTP-Nachricht beinhaltet sogenannte **Header**, die Daten beinhalten, die für den Protokollwechsel von HTTP zu Websocket nötig sind. Der Host-Header identifiziert den Ursprung des Handshakeinitiators. Dieser ist in diesem Falle die IP-Addresse die hinter dem Alias *traktor-registry* steht. Der Port 8090 der Anwendung, von welchem die Anfrage stammt, wird zusätzlich mitgesendet. Der Connection-Header gibt die bervorzugte Verbindungsart an. Diese beschreibt den gewünschten Vorgang eines Upgrades der Verbindung. In Verbindung mit dem Inhalt des Connections-Headers wird ein Upgrade-Header gesendet. Dieser ist ein Vorschlag an den Server ein anderes Protokoll zu nutzen. Entsprechend beinhaltet der Upgrade-Header den Vorschlag das Websocket Protokoll zu verwenden. Die letzen beiden Header Sec-WebSocket-Version und Sec-WebSocket-Key sind websocketspezifische Header.

Die Serverantwort sieht ähnlich aus. Diese besteht aus der Request-Line und dem Sec-WebSocket-Accept Header, der eine Bestätigung der zu nutzenden Websocketverbindung darstellt.

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
```

Listing 5.7: Serverantwort eines Websocket Handshake

Bei einem erfolgreichen Handshake bleibt die Verbindung kommunikationsbreit, bis der Tracer die Verbindung manuell schließt.

Die Implementierung des Websocket Servers basiert auf Grundlage eines Guides von Mozilla und wurde entsprechend den Implementierungsanforderungen angepasst und

erweitert.[20] Nachfolgend wird die Implementierung der Antwort auf einen Websocketverbindungsanfrage gezeigt:

```
1  string swk = Regex.Match(s, "Sec-WebSocket-Key: (.*)")
2           .Groups[1].Value.Trim();
3  string swka = swk + "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
4  byte[] swkaSha1 = System.Security.Cryptography.SHA1.Create()
5           .ComputeHash(Encoding.UTF8
6           .GetBytes(swka));
7  string swkaSha1Base64 = Convert.ToBase64String(swkaSha1);
8
9  byte[] response = Encoding.UTF8.GetBytes(
10          "HTTP/1.1 101 Switching Protocols\r\n" +
11          "Connection: Upgrade\r\n" +
12          "Upgrade: websocket\r\n" +
13          "Sec-WebSocket-Accept: " + swkaSha1Base64 +
14          "\r\n\r\n");
15
16 stream.Write(response, 0, response.Length);
```

Listing 5.8: Implementierung der Serverantwort eines Websocket Handshake

Der Wert von *Sec-WebSocket-Key* wird extrahiert und von führenden und folgenden Leerzeichen bereinigt. Anschließend wird der Wert mit einer speziellen Globally Unique Identifier (GUID) konkateniert. Dieser zusammengesetzte String wird schließend mit SHA-1, einem kryptographischen Hash Algorithmus, verschlüsselt. Der verschlüsselte Wert wird daraufhin mit Base64 kodiert, die Serverantwort gebaut und anschließend an den Tracerklienten gesendet.

Der Websocket Server muss in der Lage sein, auf eine Schließung der Websocketverbindung, reagieren zu können. Dementsprechend muss eine Schließungsnachricht gebaut werden. Diese Schließungsnachricht ist eine 2-Byte großer Frame. Die Abb. 5.2 visualisiert einen Websocketschließungs Frame. Der Frame besteht aus einem FIN-Bit, der auf 1 gesetzt sein muss. RSV1, RSV2 und RSV3 sind reservierte Bits, die auf 0 gesetzt sein müssen. Der Opcode ist ein 4-Bit großer Flag, der dem Dezimalwert 8 entspricht. Dieser steht für die Operation des Schließens des Websockets. Die Maske ist auf 0 gesetzt. Auch sind die Bits der Payload Length auf 0 gesetzt.

[20] [MDN] *Writing a WebSocket server in C# - Web APIs | MDN.*

Bitposition	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Flag	FIN	RSV1	RSV2	RSV3	opcode				MASK	Payload Length						
Wert	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Abbildung 5.2: Websocket Closing Frame

5.3.2 Klientenverwaltung

Bei erfolgreich hergestellter Websocketverbindung, kann die Registry auf Kontextpropagierung reagieren. Die Klienten werden in einem seperaten Thread behandelt. Ein Thread beinhaltet jeweils einen Datenstrom. Der Thread implementiert eine Dauerschleife. In dieser Dauerschleife wird der Datenstrom kontinuierlich überprüft. Bei eingehenden Daten wird überprüft, um welchen Datentypen es sich handelt. Dieser Prozess ist zum Teil in dem Aktivitätsdiagramm, gezeigt in Abb. 4.6, visualisiert. Es werden drei Nachrichtentypen unterschieden:

- Tracerklient Anmeldung
- Tracerklient Abmeldung
- Kontextübermittlung

Der Nachrichtentyp der Anmeldung, lässt sich anhand der Request-Line feststellen. Bei Abmeldungen und Nachrichten, die Kontextübermittlung einleiten sollen, kann dies durch den *opcode* festgestellt werden. Die Implementierungsdetails der An- und Abmeldung wurde in Abschnitt 5.3.1 ausgeführt. Im Falle der Kontextübermittlung wird die Nachricht zuerst decodiert. Da die Nachricht von einem Websocketklienten stammt, ist der **Mask** bit gesetzt. Dies ist in RFC 6455 spezifiziert. Dementsprechend muss die Nachricht umformatiert werden, damit sie weiterverarbeitet werden kann. Die Nachricht wird im Registry-Log ausgegeben und anschließend wird der Kontext an alle Teilnehmenden Tracerklienten versendet. In der Implementierung ist noch keine Addressierung der Kontextnachrichten umgesetzt. Das bedeutet, dass alle teilnehmenden Klienten die Kontextnachricht empfangen, abgesehen von dem Klienten, der die Nachricht versendet hat. Dies kann in Systemen mit mehr als zwei Tracerinstanzen zu Komplikationen mit dem Nachrichtenstrom führen, da abgesehen von dem Zieltracer des Kontext, keine Kontextnachrichten erwartet werden.

5.4 Unity Rendering System

Das verteilte rendering System dient zur Generierung von Frames auf Grundlage von dreidimensionalen geospatial Daten. Dabei wird ein Ansatz verwendet, um die Vorteile einer verteilten Anwendung zu nutzen. Das heißt, dass viele Komponenten des Systems Teilaufgaben bearbeiten. Die Arbeit die von den Komponenten verrichtet wird, ist das Generierung von Teilframes. Die Teilframes werden in einem Kollektor zusammengeführt und anschließend an den anfragenden Klienten gesendet. Die Anwendung verwendet die Unity Engine als Renderer. Die Instrumentalisierung findet innherhalb der Scripts statt, die für den Renderprozess ausgeführt werden. Es wird nur das Event des Renderprozesses eines Frames generiert. Dadurch wird die Zeit aufgezeichnet, die für die Generierung eines Frames benötigt wird. Aufgrund der Struktur des Projekts, lassen sich keine detailierteren Events, wie z.B. das Empfangen der Klientenanfrage oder das Senden des Frames, darstellen.

Die Instrumentalisierung ist durch das Hinzufügen eines Scripts und der Modifikation eines bestehenden realisiert. Das hinzugefügte Script *LoggingServerSideRendering* beinhaltet eine Klasse, dessen Objekt als Tracer innerhalb eines Scripts genutzt werden kann. Aufgrund der hoch asynchronen und parallelen Natur der Appliktion ist es notwendig eine dynamische Portermittlung für den Reporter zu ermöglichen. Dazu wurde ein Portbereich bekanntgegeben, in dem ein freier Port ausgewählt wird. Jeder parallele Prozess, der einen Tracer nutzt, wartet auf die Beendigung des Konfigurationsprozesses des Tracers. Dies hat zur Folge, dass die Prozesse auf den Tracer warten müssen. Das widerspricht dem Designziel der Verarbeitungskosten. Dieser Umstand ist jedoch als Kompromis zu sehen. Der Kompromis besteht darin, die Nutzung des Tracers innerhalb der Unity Anwendung möglichst Benutzerfreundlich zu machen, also dem Designziel der Benutzbarkeit zu entsprechen.

Die Instrumentalisierung befindet sich in der *GBufferSource* Klasse. Die Klasse umfasst den Renderingvorgang eines Frames. Dieser Vorgang wird als ein zu verfolgendes Event bestimmt. In Listing 5.9 ist der Codeabschnitt dargestellt, der instrumentalisiert ist. Die Instrumentalisierung umfasst das Rendern, durchgeführt durch ein *_camera* Objekt und die Frameextraktions aus dem nativen Plugin.

```
1 using (var scope = LoggingServerSideRendering.Tracer
2     .BuildSpan("Render Frame")
3     .StartActive(true))
4 {
5     _camera.Render();
6     _nativeGBufferExtraction.NewFrameRendered(Time.frameCount);
7 }
```

Listing 5.9: Instrumentalisierung der GBufferSource Klasse

5.5 Webserver Entwicklungsumgebung

In diesem Kapitel wird die Traktorentwicklungsumgebung vorgestellt. Diese Testumgebung gewährleistet, dass die Funktionalitäten der Instrumentalisierungsbibliothek in einer Beispielanwendung angewandt werden können. Das System besteht aus vier Komponenten. Eine Übersicht der Komponenten ist in der Abb. 5.3 gezeigt. Neben der bereits beschriebenen Traktor Registry und dem Traktor Agent, werden zwei Webserver eingeführt. Die beiden C# Webserver *Traktor-Fibonacci-Caller* und *Traktor-Fibonacci-Server* sind mit Traktor instrumentalisiert. Sie dienen dazu, einen Nachrichtenaustausch zu ermöglichen, der sich durch das ganze System zieht. Die Applikation ermöglicht es einem Anwender, eine Zahl der Fibonacci-Folge zu berechnen. Die Nachricht die der Anwender an den Traktor-Fibonacci-Caller-Service sendet, beinhaltet im *Body* eine Zahl mit dem Variablennamen N. Die Zahl repräsentiert die zu berechnende Stelle der Fibonacci-Folge. In Listing 5.10 wird eine beispielhafte Anfrage gezeigt. Es wurde sich für die Implementierung eines simplen und rekursiv anwendbaren Algorithmus entschieden, da die Testumgebung nicht durch die Anwendungslogik der Webserver verkompliziert werden sollte. Ausserdem ermöglich die Rekursivität des Algorithmus, eine schnell zu skalierende Anzahl an generierten Events, mit einem deutlich erkennbaren und messbaren zeitlichen Unterschied. Die Grundstruktur der beiden Webserverimplemetierungen ist gleich. Ein C# Programm startet ein Serverobjekt, welcher einen Thread beinhaltet, der eine API bereitstellt, die über HTTP ansprechbar ist. Nachfolgend wird auf die Instrumentalisierung der Webserver eingegangen.

```
1  PUT / HTTP/1.1
2  Accept: application/json, */*
3  Accept-Encoding: gzip, deflate
4  Connection: keep-alive
5  Content-Length: 8
6  Content-Type: application/json
7  Host: 172.22.0.5:8085
8  User-Agent: HTTPie/0.9.8
9
10 {
11 "N": 3
12 }
```

Listing 5.10: HTTP-Anfrage an den Traktor-Fibonacci-Caller durch den HTTP-Klient HTTPie

Traktor-Fibonacci-Caller Sowohl der Traktor-Fibonacci-Caller als auch der Traktor-Fibonacci-Server basieren auf dem gleichen Aufbau. Beim Starten der Anwendung wird, abhängig von der Betriebssystemfamilie, die Tracerkonfiguration durchgeführt. Auf einem Unix-basierenden Betriebssystem, wird die Konfiguration über Umgebungsvariablen gehandhabt. Da der Server einen Tracer beinhaltet, sind die für die Registry-Verbindung benötigten Umgebungsvariablen zu setzten. Diese sind die *REGISTRY_IP*

und der *REGISTRY_PORT*. Für die Verbindung zum Traktor-Agenten werden die entsprechenden Umgebungsvariablen *AGENT_IP* und *AGENT_PORT* erwartet. Für den einzunehmenden Port des Tracers, über den die reporteten Spans zum Agenten gesendet werden, wird der *REPORTER_PORT* zur Konfiguration genutzt. Der Server ist nach seiner Initialisierung einsatzbereit und erwartet Nachrichten. Sobald eine Nachricht eintrifft, wird die *Proccess* Funktion aufgerufen. Die Process Funktion ist erhält einen HTTP Kontext als Paramenter. Dieser beinhaltet die Nutzdaten des Anwenders, die mittels HTTP-Klienten gesendet wurden.

Der instrumentalisierte Codeabschnitt des Callers ist folgendermaßen implementiert:

```
1  private void Process(HttpListenerContext context){
2   using (var scope = Program.tracer.BuildSpan("Process Context")
3       .StartActive())
4   {
5    var body = new StreamReader(context.Request.InputStream, Encoding.UTF8)
6       .ReadToEnd();
7    var json = JsonSerializer.Deserialize<JsonObject>(body);
8    Program.tracer.SendContext(scope.Span).GetAwaiter().GetResult();
9    string response = SendRequest(fibo_ip,fibo_port,json);
10   context = BuildResponse(context, response);
11   context.Response.Close();
12  }
13 }
```

Listing 5.11: Process Funktion des Fibonacci-Caller Services

Das *using* Statement dient zum automatischen Schließen, der in ihr genutzten Ressource. Dies ist in diesem Fall das Scope Objekt. Der Tracer baut einen neuen Span mit dem Operationsnamen *Process Context* und makiert in als Aktiv. Die Nutzdaten werden aus der HTTP-Nachricht extrahiert. Die aktuell als JSON-Objekt vorliegenden Daten, werden deserialisiert. Da der Tracerkontextwechsel bevorsteht, wird der Kontext des Spans an die Registry, mittels *SendContext*, gesendet und auf einen erfolgreichen Abschluss gewartet. Es ist zu diesem Zeitpunkt davon auszugehen, dass der Zieltracer bereit ist, den empfangen Kontext zu verarbeiten. Aus diesem Grund werden die Nutzdaten an den Trakor-Fibonacci-Server im JSON-Format als weiterer Request gesendet. Sobald das Ergebnis vom Fibonacci Server eintrifft, wird die Antwort, welcher zum HTTP-Klienten zurückkehrt, gebaut und übermittelt. Aufgrund des using-Statements beendet die Lebensspanne des Scope Objekts und der Span wird an den Traktor-Agent reportet.

Traktor-Fibonacci-Server Der Fibonacci-Server beinhaltet den Algorithmus zur Berechnung der Fibonaccizahl. Innerhalb des Server sind zwei Funktionen instrumentalisiert. Dies ist zum einen, wie auch im Fibonacci-Caller, die *Process* Funktion, aufgeführt in Listing 5.12. Die Funktion zeigt das Empfangen des Tracingkontexts durch den Aufruf der Funktion *ReceiveContext*. Ein neuer Span wird manuell erstellt. Die Relation wird durch die Tracerfunktionalität *AsChildOf(parentContext)* hergestellt. Anschließend

wird der Span gestartet und aktiviert. Die Nutzdaten werden extrahiert und die zweite instrumentalisierte Funktion *CalculateFibonacci* wird aufgerufen.

```
1  private async void Process(HttpListenerContext context)
2  {
3   ISpanContext parentContext = await Program.tracer
4         .ReceiveContext();
5   string response;
6   var span = Program.tracer.BuildSpan("Server: Process Context")
7         .AsChildOf(parentContext)
8         .Start();
9   Program.tracer.ScopeManager.Activate(span,true);
10  var body = new StreamReader(context.Request.InputStream, Encoding.UTF8)
11        .ReadToEnd();
12  var json = JsonSerializer.Deserialize<JsonObject>(body);
13  response = CalculateFiboncacci(json.N).ToString();
14  response = "result=" + response;
15  byte[] b = Encoding.UTF8.GetBytes(response);
16  [...]
17  context.Response.Close();
18  span.Finish();
19 }
```

Listing 5.12: Process Funktion des Fibonacci-Server Services

Die CalculateFibonacci Funktion ist bewusst rekursiv und möglichst ineffizient implementiert. Die Funktion braucht zur Berechnung des $n$te Elements $2 * Fn - 1$ rekursive Aufrufe, wobei Fn die ermittelte Fibonaccizahl ist. Dadurch ist es möglich, einen großen Trace zu erzeugen. Die Funktion nutzt das using-Statement, um beim Beenden des Codeabschnitt den Span automatisch zu reporten. *CalculateFibonacci* ist als Operationsname gegeben.

```
1  private int CalculateFiboncacci(int n)
2  {
3   using (var scope = Program.tracer.BuildSpan("CalculateFiboncacci")
4        .StartActive())
5   {
6    if (n == 0) return 0;
7    else if (n == 1) return 1;
8    return  ( CalculateFiboncacci(n - 1) + CalculateFiboncacci(n - 2) );
9   }
10 }
```

Listing 5.13: Instrumentalisierte CalculateFibonacci Funktion des Fibonacci-Server Services

Abb. 5.3 bietet eine Architekturübersicht. Es werden die Kommunikationswege, sowie die Services dargestellt, die implementiert und beschrieben wurden.

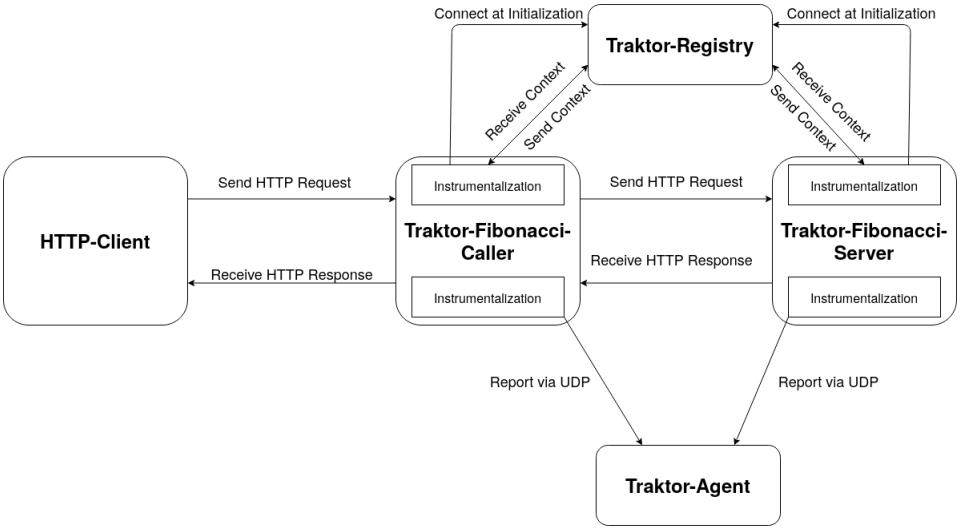


Abbildung 5.3: Systemübersicht der Traktorentwicklungsumgebung

6 | Evaluierung

In diesem Kapitel werden die Prozesse beschrieben, die durchgeführt wurden, um die Implementierung und die erhaltenen Ergebnisse zu evaluieren. In Abschnitt 6.1 wird die Traktorimplementierung auf ihre Einhaltung der Anforderungen untersucht. In Abschnitt 6.2 werden die Designziele herangezogen und auf ihre Gegebenheit in der Tracingbibliothek überprüft. Das Open-Source Projekt **Jaeger** wird als Vergleichswerkzeug in den Folgenden Abschnitten herangezogen. Jaeger ist eine auf der OpenTracing API basierenden *state-of-the-art* Distributed Tracing Implementierung. Sie setzt die aktuellsten Erkenntnisse der Distributed Tracing Gemeinschaft um. Dabei wird in Abschnitt 6.3 die Bereitstellung der Testumgebung, in Hinsicht auf beide Werkzeuge, diskutiert. Die Bereitstellungsunterschiede beider Werkzeuge werden aufgezeigt. In Abschnitt 6.4 werden die Ergebnisse der Spangenerierung verglichen. Es wird auf die Ausdruckskraft des Traktor-Datenmodells im Vergleich zu Jaeger eingegangen. Zuletzt werden die präsentierten Visualisierungansätze diskutiert. Es wird ein Vergleich zu den Visualisierungsmöglichkeiten der Jaeger UI durchgeführt.

6.1 Anforderungserfüllung

Es ist eine Analyse durchzuführen, bei der die erhobenen Daten interpretiert werden. Die daraus gewonnenen Informationen sollen die End-zu-End Latenz einer Anfrage durch ein verteiltes System und die Generierungszeit eines Frames, welches durch die Unity Anwendung generiert wurde, darstellen.

- Funktionalitäten
  - Eventgenerierung
  - Eventrelation
  - Synchronisation von Eventgeneratoren
  - Eventübermittlung
  - Ordnung von Events

- End-zu-End Latenz
- Generierungszeit eines Frames
- Nachrichtenmodifikation

6.2 Umsetzung der Designziele

6.3 Bereitstellung der Testumgebung

- Durch Docker-Compose wird eine verteiltes System auf einem Lokalen Rechner aufgebaut.
- gibt es architektonische unterschiede? Services etc.
- Konfigurationsunterschiede
- Ist traktor einfacher zu deployen? Wahrschienlich nicht da jaeger ein all in one image hat. Könnte aber auch umgesetzt werden

6.4 Ergebnissvergleich

- Bezug zur Problemstellung schaffen
- inwiefern hält sich jaeger an die Happens before relationship?
- wie setzt jaeger diese um?
- setzt es sie um?
- setzt traktor sie um? Wenn ja wie?
- Wie regelt Jaeger die Zeitresolution?
-

6.5 Visualisierungvergleich von Traktor und Jaeger

- jaeger bietet verschiedene visualisierungsmöglichkeiten
  - z.B. DAG
  - Tracediagramm
  - Service-Orientierter Ansatz?
  - Tracecompare
- Welche Vorteile bringen meine Visualisierungsansatze?

7 | Fazit

7.1 Ausblick

7.2 ZitatTest

[MRF15] [Sam+16] [MF18] [Bar+04] [Rey+06] [ACF12] [NCK19] [Bey+16] [Kal+17] [Bar+03] [Red] [SCR18] [Sig+10] [Ste01] [Fon+07] [Wat17] [Ope19]

Glossar

Continuous Integration Automatisierter und fortlaufend getätigter Integrationsprozess von Änderungen einer Anwendungen.. 10

Rendering Rendering ist das erzeugen von zwei-dimensionalen Bildern. Rohdaten, wie z.B. virtuelle Kameras, drei-dimensionale Objekte, Lichtquellen, etc. werden dazu genutzt, um diese Bilder zu generieren. 11

Abkürzungsverzeichnis

API Application Programming Interface. 16

CPU Central Processor Unit. 5, 17

GPU Graphical Processor Unit. 5, 17

HDD Hard Disk Drive. 5

HTTP Hypertext Transfer Protocol. 5

IoT Internet of Things. 11

IP Internet Protocol. 5

RAM Random-Access Memory. 5

TCP Transmission Control Protocol. 5

Bibliography

[] *Flame Graphs*. URL: http://www.brendangregg.com/flamegraphs.html (visited on 04/20/2020).

[20] *perf(1) Linux User's Manual*. 5.06. perf, Sept. 2020.

[ACF12] Mona Attariyan, Michael Chow, and Jason Flinn. "X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. USA: USENIX Association, 2012, pp. 307–320. ISBN: 9781931971966.

[Agu+03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. "Performance debugging for distributed systems of black boxes". In: *Operating Systems Review (ACM)*. Vol. 37. 5. Association for Computing Machinery, 2003, pp. 74–89. DOI: 10.1145/1165389.945454.

[Bar+03] P Barham, R Isaacs, R Moertier, and D Narayanan. "Magpie: online modelling and performance-aware systems". In: *Proceedings of USENIX HotOS IX* (2003).

[Bar+04] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. "Using Magpie for Request Extraction and Workload Modelling". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI'04. USA: USENIX Association, 2004, p. 18.

[Bey+16] B Beyer, C Jones, J Petoff, and N R Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated, 2016. ISBN: 9781491929124. URL: https://books.google.de/books?id=81UrjwEACAAJ.

[Bre] Brendan D. Gregg. *Linux perf Examples*. URL: http://www.brendangregg.com/perf.html (visited on 03/22/2020).

[Bre15] Brendan D. Gregg. *Flame Graph Search*. Aug. 2015. URL: http://www.brendangregg.com/blog/2015-08-11/flame-graph-search.html (visited on 03/20/2020).

[Fon+07] R Fonseca, G Porter, R Katz, S Shenker, and I Stocia. “X-Trace: A Pervasive Network Tracing Framework”. In: *Proceedings of USENIX NSDI* (2007).

[Gar02] Vijay K. (Vijay Kumar) Garg. *Elements of distributed computing*. Wiley-Interscience, 2002, p. 423. ISBN: 9780471036005.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154916.

[Kal+17] J Kaldor, J Mace, M Bejda, E Gao, W Kuropatwa, J O’Neill, K Win Ong, B Schaller, P Shan, B Viscomi, V Venkataraman, K Veeraraghavan, and Y Jiun Song. “Canopy: An End-to-End Performance Tracing And Analysis System”. In: *SOPS 2017* (2017).

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. ISSN: 15577317. DOI: 10.1145/359545.359563.

[Lam87] Leslie Lamport. *Distributed Systems Definition by Lamport*. 1987. URL: https://lamport.azurewebsites.net/pubs/distributed-system.txt (visited on 02/29/2020).

[Lea14] Jonathan Leavitt. “End-to-End Tracing Models: Analysis and Unification”. In: *D* (2014). URL: http://cs.brown.edu/%7B~%7Drfonseca/pubs/leavitt.pdf.

[MDN] MDNContributers. *Writing a WebSocket server in C# - Web APIs | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets%7B%5C_%7DAPI/Writing%7B%5C_%7DWebSocket%7B%5C_%7Dserver (visited on 04/17/2020).

[MF18] Jonathan Mace and Rodrigo Fonseca. “Universal Context Propagation for Distributed System Instrumentation”. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys ’18. New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190526. URL: https://doi.org/10.1145/3190508.3190526.

[MRF15] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 378–393. ISBN: 9781450338349. DOI: 10.1145/2815400.2815415. URL: https://doi.org/10.1145/2815400.2815415.

[NCK19] S Nedelkoski, J Cardoso, and O Kao. “Anomaly Detection and Classification using Distributed Tracing and Deep Learning”. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019, pp. 241–250. DOI: 10.1109/CCGRID.2019.00038.

[Ope19] OpenTracting. *opentracing trace overview figure*. 2019. URL: https://opentracing.io/docs/overview/.

[Red] Inc Red Hat. *Was ist IT-Automatisierung?* URL: https://www.redhat.com/de/topics/automation/whats-it-automation.

[Rey+06] Patrick Reynolds, Charles Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. "Pip: Detecting the Unexpected in Distributed Systems". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. USA: USENIX Association, 2006, p. 9.

[Sam+16] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. "Principled Workflow-Centric Tracing of Distributed Systems". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 401–414. ISBN: 9781450345255. DOI: 10.1145/2987550.2987568. URL: https://doi.org/10.1145/2987550.2987568.

[SCR18] MARIO SCROCCA. "Towards observability with (RDF) trace stream processing". PhD thesis. Politecnico Di Milano, 2018. URL: http://hdl.handle.net/10589/144741.

[Sig+10] Benjamin Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure". In: *Google Techical Report dapper-2010-1* (2010).

[ST17] Martinus Richardus van Steen and Andrew S Tanenbaum. *Distributed Systems*. 3rd. 2017. ISBN: 978-15-430573-8-6.

[Ste01] William Stearns. *Ngrep and regular expressions to the rescue*. http://www.stearns.org/. 2001. URL: http://www.stearns.org/doc/ngrep-intro.current.html.

[TS06] Andrew S Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[Wat17] Matt Watson. *8 Key Application Performance Metrics & How to Measure Them*. 2017. URL: https://stackify.com/application-performance-metrics/.