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

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 Tracingbibliothek 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

In einem System werden Nachrichten ausgetauscht. In Falle eines verteilten Systems spielt dabei die Kommunikation über Prozessgrenzen eine zentrale Rolle. Entscheidende Ereignisse und deren zeitliches Auftreten sind von besonderem Interesse. Diese Ereignisse

werden Events genannt. Events bilden einzelne Zeitpunkte ab. Die Intrasystem - Netzwerkkommunikation bedarf Konzepte zur Nachvollziehbarkeit von Events und deren Beziehungen zueinander.

Das 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. Das System generiert Frames innerhalb von 16ms. Die Antwortzeiten betragen ca. 100ms. Der Client 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 von z.B.:

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

Abgesehen von dem Brown Tracing Framework und dem X-Trace verwenden alle genannten Ansätze das spanbasierte Datenmodell. 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. 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 darstellt.[1]

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

- beschreibe, was metriken sind: bezug zu überwachung
- beschreibe was logs sind: bezug zu überwachung
- 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 des Netzwerk 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 erschweren.

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) **Anwendung**, welches 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.

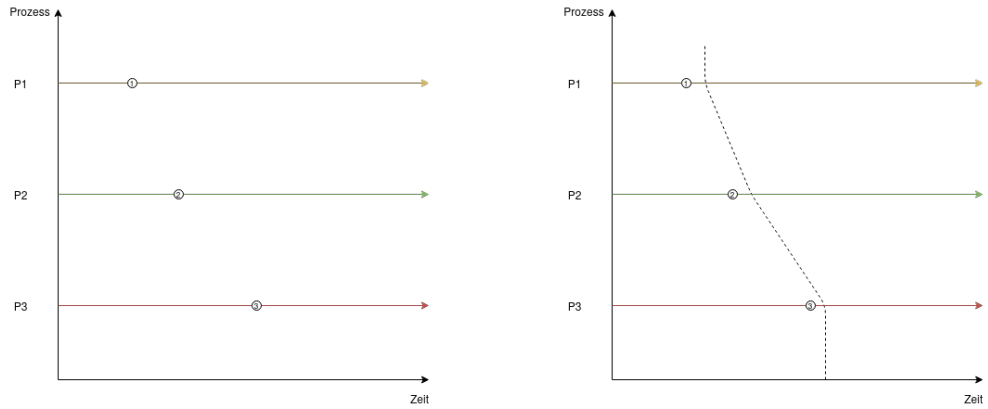
Anwendung Die Beobachtung der Anwendung ist, im zusammenspiel mit dem Nachrichtenaustausch über das Netzwerk, zentral. Die Beobachtbarkeit der Anwendung sorgt dafür, dass Anwendungsdaten erhoben, ver- und aufgearbeitet und anschließend präsentiert werden. Die Präsentation der Daten hilft den Verantwortlichen 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 Berücksichtignug 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

[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.

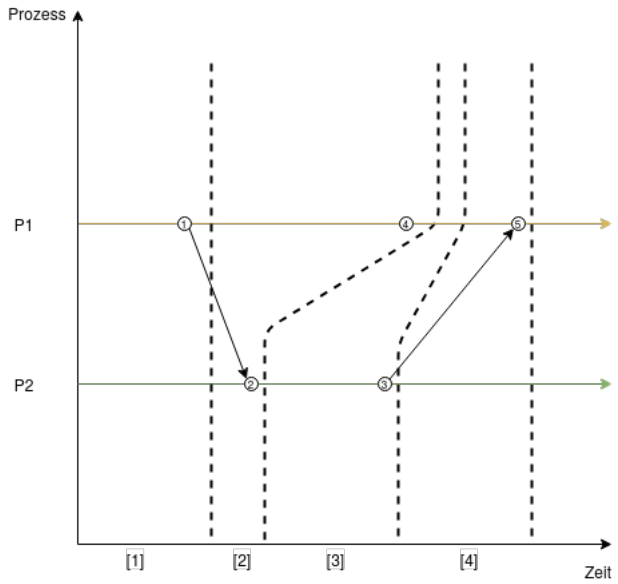


Abbildung 2.2: Zeigt Prozesse P1 und P2. Diese generieren jeweils Events. Logische Reihenfolge des Auftretens der Events anhand der gestrichelten Linien [1], [2], [3] und [4] ersichtlich.

verzichtet wird. Diese Prozesse können sich mittels senden einer Nachricht und 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* 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 Conditions 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,

der folgende Eigenschaften hat[8]:

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

Dabei ist 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 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 Verteilte 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.[9]

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, können durch distributed tracing angegangen werden. So kann zum Beispiel Fehlerquellenermittlung beschleunigt werden. Tracing ist ein Konzept, welches als Werkzeug umgesetzt

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

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 Blackboxmonitoring ist sinnvoll, wenn mit vielen Softwarekomponenten gearbeitet wird, auf deren Entwicklung man keinen direkten Einfluss hat. Dazu zählen beispielsweise proparitäre 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 Serviceskomponenten 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 loose 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. Den 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* zum einsatz,

die Profilingdaten erheben. Diese Daten bestehen unter anderem aus aufgerufenen Funktionen, CPU Auslastung und Laufzeiten von Funktionen. Die Events, die durch Profiling erhoben werden, weisen keine Relationen zueinander auf.

Tracing ist eine Konzept zur Anwendungsüberwachung. Dabei werden Events in relation zueinader 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 Bilbiothek 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 Continues Integration Pipeline aufgebaut worden, die dafür sorgt, dass die Bibliothek über Nuget verfügbar gemacht wird.

3 | Problembeschreibung

In diesem Kapitel wird die Problematik, um das Generieren und Ordnen von Events in einem verteilten System mit asynchroner Kommunikation, beschrieben. Dabei betrachten 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 Wisschenschaft und grundsätzlich des alltäglichen Lebens sind 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 detailierte 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*.

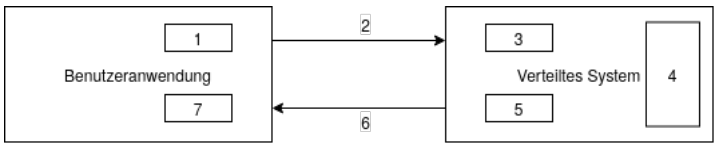


Abbildung 3.1: Kausaler Pfad einer Vorgangs in dem verteilten rendering System

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. Sind 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 darzugestellen, sodass anhand der Visualisierung feststellbar ist, ob das Verhalten der Anwendung starke Ausreißer, die auf Fehlimplmenentierung deuten könnten, aufweist

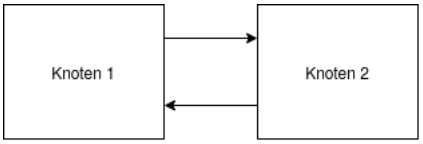
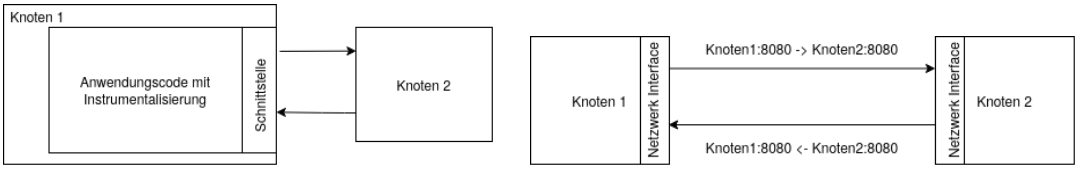


Abbildung 3.2: Minimale Struktur eines verteilten Systems, bestehend aus zwei Komponenten



(a) zeigt Knoten mit instrumentalisiertem Anwendungscode

(b) Netzwerkkommunikation über TCP/IP

Abbildung 3.3

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 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.

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.[10]

Von dem Szenario ausgehend, dass man Events innerhalb des Minimalbeispiels analysieren möchte, eignen sich Flammengraphen. 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 miteinandener verbunden. Dies führt zu einer

[10] [Bre] *Linux perf Examples.*

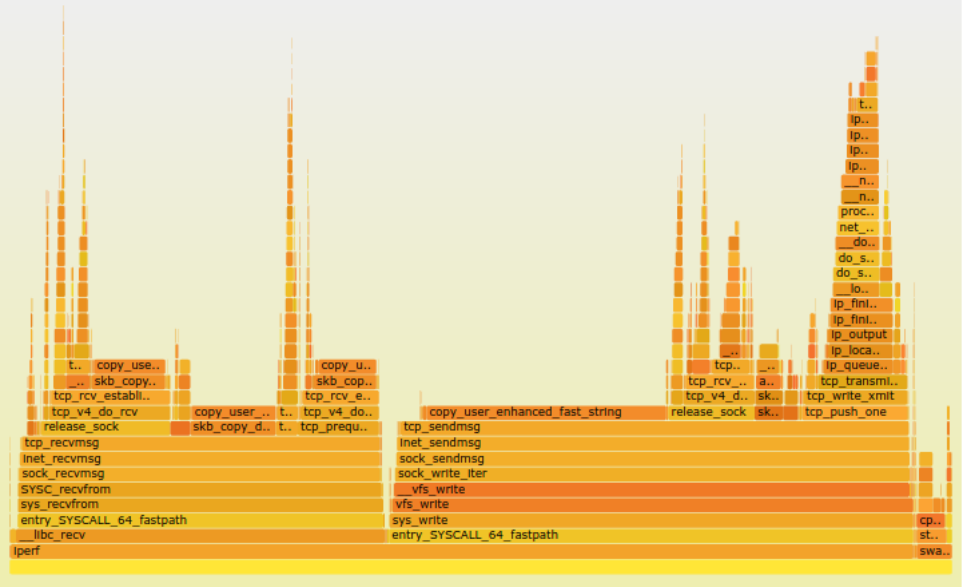


Abbildung 3.4: Visualisierung von CPU Performancedaten dargestellt als Flammengraph von Brendan D. Gregg [Bre15]

dreidimensionalen Darstellung von Flammengraphen, gezeigt in Abb. 4.7. 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, dargetellt durch die gestrichelten Linien mit Pfeilrichtung, stellen die Terminierung der TCP-Verbindung dar.

Der Terminierungsprozess wird genauer betrachtet. Abb. 3.5 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 Initators, 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* Makierung. Die unbekannte Variable *Übertragungszeit* nimmt Einfluss auf den Zeitpunkt. Event **D** ist also definiert als:

$$\mathrm{D}: \; Ack_{init} + \text{Übertragungszeit}$$

Zu untersuchen sind die Relationen zwischen diesen vier Events. Dabei sind zwei Relationen, wie in Abschnitt 2.1.3 beschrieben, als A → B und C → D definiert. Durch die kausale Abhängigkeit von C von A gilt A → C. Durch die *Transitivität* ist entsprechend A → D gegeben. Es ist zu untersuchen, ob B → D gilt. Dabei sind die Bedingungen, die 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 → b[11]

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

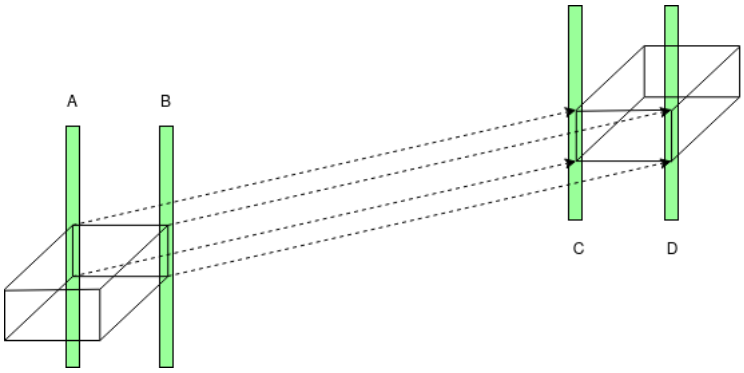


Abbildung 3.5: Detailierte Betrachtung des in Abb. 4.7 gezeigte Nachrichtenaustauschs. Stellt die Schließung einer TCP Verbindung dar

3.3 Anforderungsanalyse

Das Systems, für das die Instrumentalisierungsbilbiothek entwickelt wird, ist ein System für verteiltes Rendering. Die Instrumentalisierungsbilbiothek 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

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

Eventübermittlnug 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 Ereginisse als ein Event abzubilden. Die Generierungfunktionalität muss dafür sorgen, dass die Events einem spezifizierten Aufbau aufweisen, um weiterverarbeitet und ausgewertet werden zu können. Die Nutzung einer standarisierten 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 Synchronistion von Eventgeneratoren

Eventgeneratoren sind oftmals auf verschiedenen Komponenten des verteilten Systems angesiedelt. Wie kann also ein Konzept aussehen, 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 generiert Event bildet den Startpunkt eines neuen Traces dar. Mehrere Anfragen können zur gleichen 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 diesem zuzuordnen.

3.3.2 Rahmenbedingen

3.3.2.1 end-zu-end Latenz

Die durchschnittliche Antwortzeit des verteilten rendering Systems ist im Durchschnitt 100ms. Die durchschnittliche Generierungszeit eines Frames beträgt 16ms. 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. gemessen 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 Ereginisse können erkannt, aufgearbeitet und als Events gespeichert werden. Betriebssystemspezifische Ereignisse sind vorallem 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 Senderadresse, der Empfängeradresse und einem Zeitstempel könnten genutzt werden. Allerdings ist das Fehlen von Applikationsinformationen ein entscheidendes Problem. Das Ziel des Blackboxansatz ist die minimale Vorraussetzung von *a*

priori Informationenen über Kommunikationswege, über den Aufbau von Applikationsnachrichten, die Semantik der Anwendung und den Aufbau des verteilten Systems.[12]. Allerdings sind diese Daten äußert wichtig, um ein tiefgreifendes Verstädnis 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.

[12] [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 Informationsgewinnen 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 folgern Designziele, die an das Design gestellt werden. Die von Google erstellte Fachpublikation *Dapper, a Large-Scale Distributed Systems Tracing Infrastrcture* dient vielen Tracingsystemen als konzeptionelle Grundlage. In der Puplikation werden Designziele aufgeführt, die neue Tracingsysteme bewerten sollten. Es werden Designziele bewertet, die von Dapper genannt werden. Ausserdem sind zusätzliche Designziele aufgezeigt, die aus den Anforderungen des verteilten rendering Systems hervorkommen. 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.[13].

[13] [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 entwicklet wird, ist eine Mischung aus Platformabhängigen und Platformunabhängigen Komponenten. Durch den verminderten Mehraufwand der bei platformunabhängigen Komponeten entsteht, wird eine verbesserte Nutzerfreundlichkeit gewährleistet. Vorallem bei der Integration in bestehende Systeme wird dies bemerkbar, da Bauprozesse von Projekten platformabhängig sind. Der platformabhängige Bauprozess soll im Falle des verteilten rendering System nicht beeinflusst werden. Die

Datenverfügbarkeit Die Datenverfügbarkeit soll zeitnah stattfinden. Die von der Tracinginfrastruktur generierten Tracingdaten sollen zur Laufzeit darsgestellt 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 Ausnutzung 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

F1: Inwiefern lässt sich eine Zeitmessung von Eventzeitspannen konzipieren

In diesem Abschnitt wird das Problem **F1** diskutiert. Dieses Problem erfordert die Konzipierung eines Datenmodells, welches drei Aspekte berücksichtig. 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 detailierte 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 Kommuniktion von Komponenten des Systems gegeben. Dieser wird in Abschnitt 4.2.3 dargestellt.

Das Datenmodell beruht auf dem Spanmodell, welches in der vorher beschrieben Fachpuplikation *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

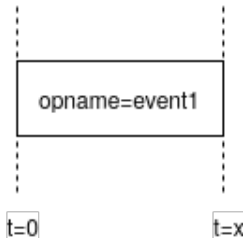


Abbildung 4.1: Darstellung eines Spans mit Anfangszeit, Endzeit und Operationsnamen

Der Spankontext wiederum ist auch ein Datencontainer. Dieser beinhaltet eine *SpanID* und eine *TraceID*. Auch ein Beziehungstyp mitgeführt. Der Beziehungstyp kann zum einen eine *Child-of* Beziehung oder eine *Follows-From* Beziehung annehmen. 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 Startzetpunktes 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. 4.2 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.

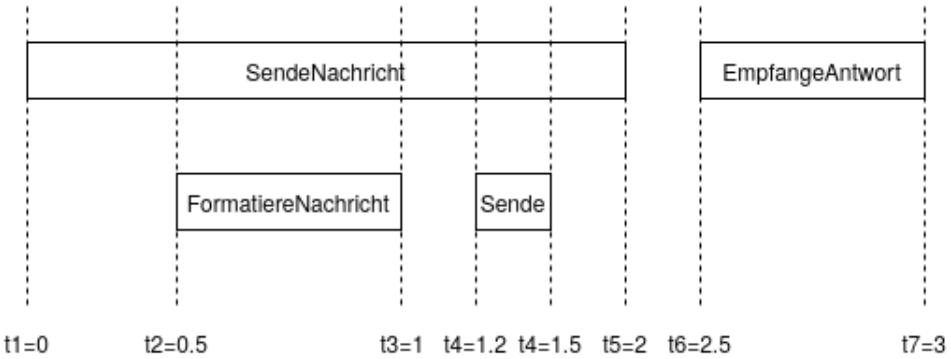


Abbildung 4.2: Darstellung eines Trace mit mehreren Spans. Anfangszeit und Endzeit ergeben die Gesamtzeit eines Traces

4.2.2 Tracingcontext innerhalb eines Systems

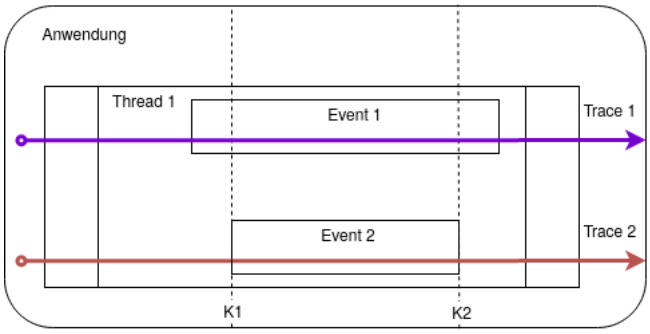


Abbildung 4.3: Anwendung mit einem Thread. Ein von der Instrumentalisierung veranlasster Kontextwechsel innerhalb eines Threads

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. 4.3, einen Thread, indem zwei Traces parallel exisitieren. Die Abb. 4.4 stellt die Situation eines Traces über mehrere Threads dar.

Im ersten Beisiel sei gegeben, dass eine Anwendung ein Thread implementiert. Dabei exisiteren 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 Erstellnug und Aktivierung des Spans, welcher Event 1 umfasst. Die Aktivierung des Spans sorgt dafür, dass dieser in dem *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 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.

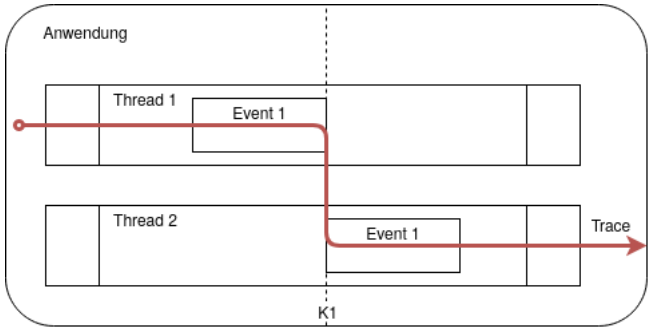


Abbildung 4.4: Anwendung mit zwei Threads. Ein von der Instrumentalisierung veranlasster Kontextwechsel zwischen Threads

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 schließt 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

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

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, Tracingkontext über Prozessgrenzen hinaus zu transportieren.

Das erste Konzept ist die **Traktor Registry**. Der Infrastrukturaufbau, bestehend aus der Registry und der Tracer, die mit der Registry verbunden sind, erfüllt die Aufgabe 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 Kontextpropagieren 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. 4.5 erschließen.

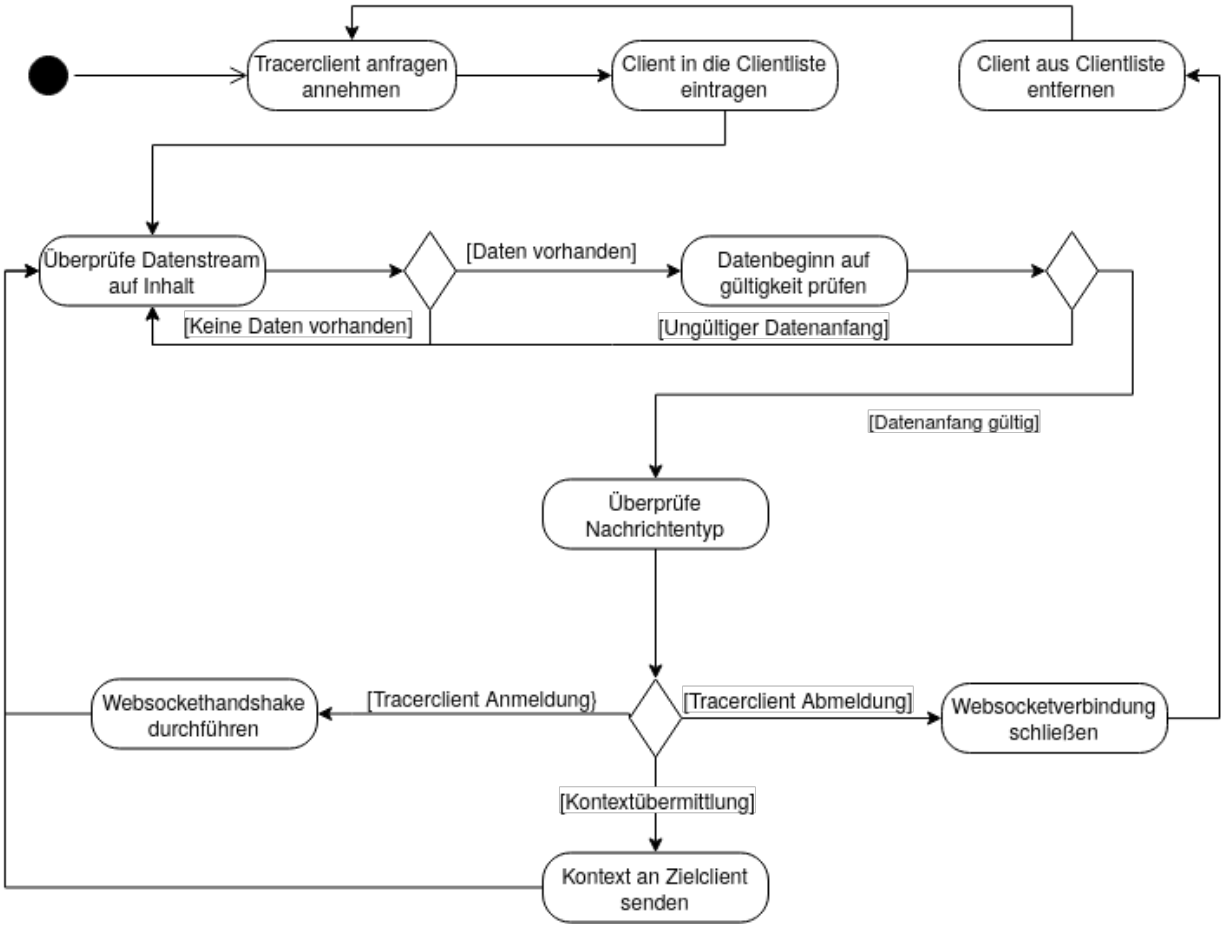


Abbildung 4.5: Aktivitätendiagramm der Traktor Registry

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

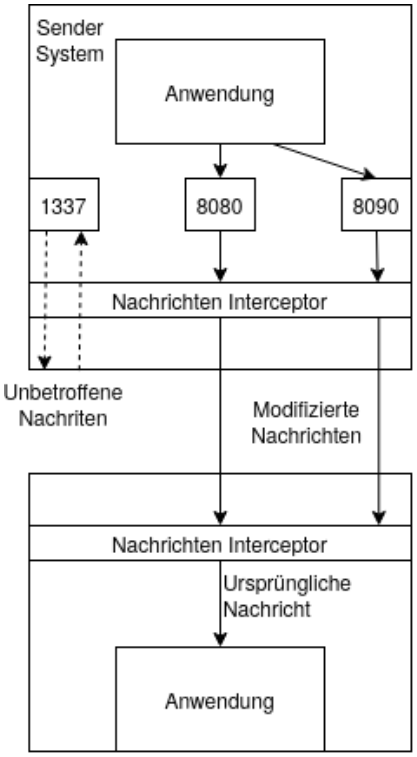


Abbildung 4.6: Design des Nachrichteninterceptor

an ihren Empfänger weitergeleitet. Auf dem Empfängersystem sitzt der gleiche *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. 4.6 zeigt das die Nachrichtenmodifikation, bei der ein 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 werden. Durch diese Funktionalität lässt sich die Registry, also einem potentiellen *single point of failure*, ersetzten. Allerdings werden dazu auf jedem System zusätzliche, neben der Anwendung laufende, Scripte benötigt.

4.3 Verarbeitungsmodell

4.3.1 Reporter

4.3.2 Kollektor

4.4 Visualisierung

4.4.1 Tracegraph

4.4.2 Dreidimensionale Flammengraphen

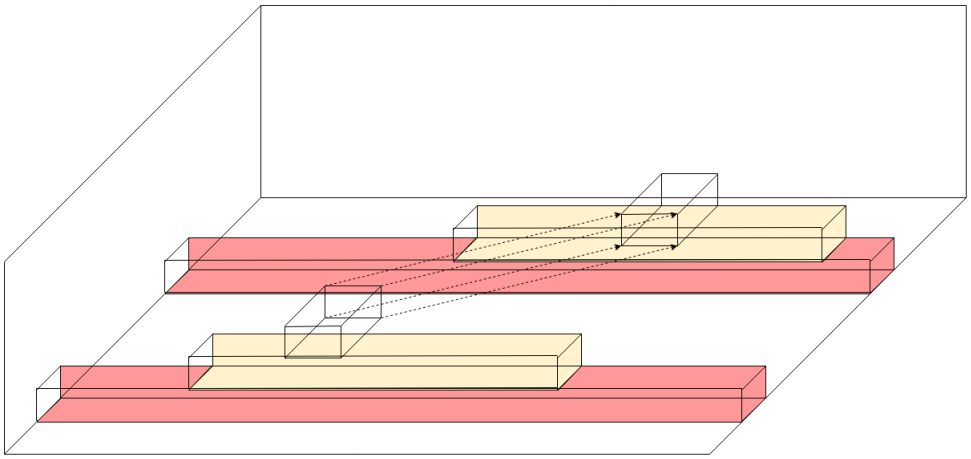


Abbildung 4.7: Skizzierung eines dreidimensionalen Flammengraphs mit Nachrichtenaustausch

5 | Implementierung

5.1 Instrumnetalisierungsbibliothek: Traktor

5.2 Traktor Agent

5.3 Traktor Registry

6 | Evaluierung

6.1 Genauigkeit der Eventgenerierung

6.1.1 Uhren und Zeit

6.2 Darstellung der Events

6.3 Vergleich mit Jaeger

6.3.1 Datenmodelle

6.3.2 Bereitstellung

6.3.3 Ergebnisse

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

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

Rendering render, a two-dimensional image, given a virtual camera,three-dimensional objects, light sources, and more 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

[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.

[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.

[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/.