Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Sven Allers

Entwicklung und Realisierung eines Systems zum Tracing aktorbasierter Software

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sven Allers

Entwicklung und Realisierung eines Systems zum Tracing aktorbasierter Software

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai von Luck
Zweitgutachter: Prof. Dr. Martin Hübner
Fachprüfer: Lutz Behnke

Eingereicht am: 13. März 2019

Sven Allers

Thema der Arbeit
Entwicklung und Realisierung eines Systems zum Tracing aktorbasierter Software

Stichworte
Verteilte Systeme, Tracing, Aktorbasierte Systeme

Kurzzusammenfassung
Diese Arbeit befasst sich mit der Analyse des Verhaltens von verteilten aktorbasierten Systemen. Zu diesem Zweck wurde im Kontext dieser Arbeit ErlViz entwickelt. Hierbei handelt es sich um ein System zum Tracing von aktorbasierten Systemen. Ein Ziel war hierbei vor allem die lokale sowie verteilte Nebenläufigkeit von aktorbasierten Systemen abbilden zu können. Deshalb wurde ein System geschaffen, das in der Lage ist, die Kommunikation von Aktoren aufzuzeichnen und zu visualisieren. Dies ermöglicht es den Programmfluss auf lokaler sowie verteilter Ebene nachvollziehen zu können. Neben der Entwicklung und Realisierung des Systems, wurde auch die Leistungsfähigkeit mittels verschiedener Experimente überprüft. Hierbei wurden Aspekte der Performance aber auch die Leistungsfähigkeit der Visualisierung betrachtet.
Sven Allers

Title of the paper
Development and Implementation of a System to Trace Actor Based Software

Keywords
Distributed Systems, Tracing, Actor Based Systems

Abstract
This work covers the analysis of the behaviour of distributed actor based systems. For this purpose, ErlViz was developped in the context of this work. This is a system for tracing actor based systems. One of the goals was to be able to represent the local and the distributed concurrency of actor based systems. That is why a system was created that is able to record the communication of actors and to visualise those. That makes it possible to comprehend the flow of the program on a local as well as on a distributed level. In addition to development and implementation of the system, experiments were made to verify the capabilities of the system. Therefore aspects of performance as well as the capabilities of the visualisation were considered.

Inhaltsverzeichnis

1 Einleitung

Verteilte Systeme sind allgegenwärtig. Mit einer höheren Verteilung geht jedoch auch eine höhere Nebenläufigkeit einher. Dies macht es besonders schwer Fehler in dieser Art von Systemen zu untersuchen und über das allgemeine Verhalten Rückschlüsse zu ziehen. Eine Form des Systems, bei dem es besonders zum Tragen kommt, sind zum Beispiel Multiplayer Online Spiele. Hier können zehntausende von Spielern zeitgleich miteinander interagieren [MMO14], während zur selben Zeit alle möglichen Aspekte der simulierten Welt berechnet werden müssen. Hier spielen Aspekte wie eine gute Lastverteilung eine zentrale Rolle [All15].

Sollte es in einem verteilten System zu einem unerwartetem Verhalten kommen, so erfolgt die Analyse von Fehlern oft mittels der vorliegenden Log-Dateien. Dies kann sich jedoch als sehr aufwendig herausstellen [BWBE16]. Denn im Gegensatz zu einem einzelnen Programm, kann in einem Verteiltem System eine hohe Heterogentität vorliegen. Vor allem zwischen den verschiedenen Softwarekomponenten. Aber auch die Hardware kann sich zwischen den einzelnen Elementen unterscheiden. Hinzu kommt, dass Abläufe in einem verteilten System nebenläufig stattfinden. Dies kann zu Deadlocks oder Race Conditions führen [BWBE16]. Insbesondere ein falsches Timing von Nachrichten zwischen den Systemen, stellt sich als einer der Hauptursachen für Fehler in verteilten Systemen heraus [LLL+17, LLLG16]. Zwar gibt es eine Reihe von Arbeiten, die sich mit der Analyse von Logs, zum Beispiel mittels Machine Learning, beschäftigen [NKN12, KDJ+12], allerdings haben Log-Nachrichten den großen Nachteil, dass diese während der Entwicklung definiert werden müssen. Damit muss der Entwickler im Vorhinein antizipieren, welche Informationen wichtig sind aufzuzeichnen. Dies stellt sich jedoch zur Analyse von verteilten Systemen als Fehleranfällig heraus [LGW+08, SBB+10]. Deswegen setzen andere Verfahren auf die Möglichkeit Aufzeichnungen dynamisch zur Laufzeit einbinden zu können [LGW+08, MRF15]. Es ist zudem bekannt, dass nur 10% aller Nebenläufigkeitsprobleme überhaupt durch Unittests gefunden werden. Alle anderen Fehler werden erst zum Zeitpunkt des Deployments oder später gefunden [LLLG16]. Deswegen ist es sinnvoll Methodiken an der Hand zu haben, mit denen das Verhalten auch im Nachhinein noch nachvollzogen werden kann. Neben einer verteilten Nebenläufigkeit

kommt bei modernen Anwendungen auch die lokale Nebenläufigkeit zum Tragen, weswegen es Bestrebungen gibt, neben der Kommunikation der Systemkomponenten auch lokale Daten aufzuzeichnen [LSZE11]. Insbesondere bei aktorbasierten [HBS73] Systemen existiert ein hohes Interesse daran, die lokalen Kommunikation der Aktoren nachvollziehen zu können. Somit könnte neben dem verteilten Programmfluss auch der lokale Programmfluss nachvollzogen werden.

Innerhalb dieser Arbeit wurde ErlViz entwickelt. Hierbei handelt es sich um ein System zur Untersuchung von aktorbasierten verteilten Systemen. Dieses System soll genau diese Möglichkeit realisieren, den verteilten sowie den lokalen Programmfluss nachvollziehen zu können. Es geht darum ein Werkzeug zur Analyse von aktorbasierten Systemen bereitzustellen, dass dem Nutzer die Möglichkeit gibt, die lokale sowie verteilte Nebenläufigkeit des Systems zu analysieren. Neben der Aufzeichnung aktorbezogener Daten wird auch eine Visualisierung bereitgestellt, die bei der Analyse des Systems unterstützen soll. Diese Visualisierung nutzt einen Tracing-Graph, der die zeitliche Abfolge von Ereignissen darstellt. Diese Art der Visualisierung ist bereits der Analyse von verteilten Systemen bekannt [BWBE16] und wird durch die Visualiserung eines Aktorsystems um eine lokale Komponente erweitert. Neben Konzeption und Umsetzung von ErlViz wurde das System auch evaluiert. Hierbei wurde sowohl der Overhead betrachtet als auch die Fähigkeiten der Visualisierung analysiert.

1.1 Abgrenzung

Innerhalb dieser Arbeit wird eine erste Version von ErlViz betrachtet. Damit richtet sich das System vor allem an Testumgebungen. Eine Variante, die die Fähigkeit besitzt, auch auf einem produktiven System eingesetzt zu werden, ist nicht das Ziel der Arbeit. Stattdessen geht es darum zu verstehen inwiefern das Konzept trägt. Eventuelle Optimierungen können Bestandteil folgender Arbeiten sein.

1.2 Struktur der Arbeit

In Kapitel 2 soll zunächst in die Thematik dieser Arbeit eingeführt werden. Dafür wird als Erstes die vorliegende Problemstellung aufgezeigt und allgemeine Anforderungen an das zu entwickelnde System definiert. Anschließend wird auf die bisherige Forschung in dem Gebiet der Analyse von verteilten Systemen eingegangen um abschließend ein Fazit zu ziehen. In Kapitel 3 wird das ErlViz System auf konzeptioneller Ebene vorgestellt. Dazu gehören die

einzelnen Komponenten, die Architektur sowie die Funktionalitäten, die die Visualiserung dem Nutzer bereitstellt. Kapitel 4 beschäftigt sich mit der Umsetzung von ErlViz. Hierzu gehören technische Details der Implementierung aber auch die Evaluation des Tracing Tool Builders [Erlb] als eine Option der Umsetzung des Tracings von ErlViz. Außerdem wird die Experimentierumgebung vorgestellt, in die das System eingebunden wurde und auf der Basis im folgenden Kapitel die Experimente stattfinden. Entsprechend geht es in Kapitel 5 darum, das entwickelte System auf Basis von einer Reihe von Experimenten zu untersuchen. Hierbei werden technische Aspekte, wie die Auswirkung auf die Geschwindigkeit und die Speicherauslastung des beobachteten Systems aber auch die Visualisierung untersucht. Kapitel 6 bildet den Abschluss und fasst die Arbeit noch einmal zusammen. Außerdem wird ein Gesamtfazit gezogen und es gibt einen Ausblick auf mögliche Nachfolgearbeiten.

2 Analyse

Bevor näher auf ErlViz eingegangen wird, soll zunächst in den Untersuchungsgegenstand eingeführt werden. Dazu wird als erstes die vorliegende konkrete Problemstellung vorgestellt, die als Motivation dieser Arbeit dient und an derer das System validiert und untersucht wird. Aus der beschriebenen Problemstellung werden anschließend die Anforderungen an das System herausgearbeitet. Nachdem die allgemeine Problemstellung und die daraus resultierenden Anforderungen vorgestellt wurden, werden die bisherigen Arbeiten, die bereits in diesem Bereich stattgefunden haben, betrachtet. Zum Abschluss dieses Kapitels wird in einem Fazit dargestellt was die bisherigen Lösungen leisten können und wo deren schwächen liegen, um die Motivation hinter ErlViz weiter zu untermauern.

2.1 Problemstellung

Bevor eine tiefere Analyse bisheriger Arbeiten durchgeführt wird, soll an dieser Stelle die vorliegende Problemstellung näher erläutert werden. Im Allgemeinen, soll diese Arbeit aufzeigen, wie kommunikative Eigenschaften in einem verteilten System ermittelt werden können, um auf der Basis Rückschlüsse über das Verhalten des Systems ziehen zu können. Hierbei wird der Schwerpunkt vor allem auf aktorbasierte Systeme gesetzt. Dies soll anhand von einem vorliegendem Problem aufgezeigt werden. Hierzu wird die Lastverteilung in Timadorus [Tim] näher Untersucht. Bei Timadorus handelt es sich um eine Umgebung, die es ermöglichen soll Massive Multiplayer Online Games zu betreiben. Das Verfahren zur Verteilung der Last wurde bereits in vergangenen Arbeiten entwickelt und erweitert [BAW+16, All17b]. Dieses System wurde komplett mittels Erlang [Erlc] realisiert. Dadurch liegt ein Aktorsystem vor. Dies hat die Besonderheit, dass neben der Verteilung des Systems auch eine starke lokale Nebenläufigkeit vorliegt. Im Folgenden sollen deshalb zunächst die Verfahren und die aufgetretenen Probleme beschrieben werden.

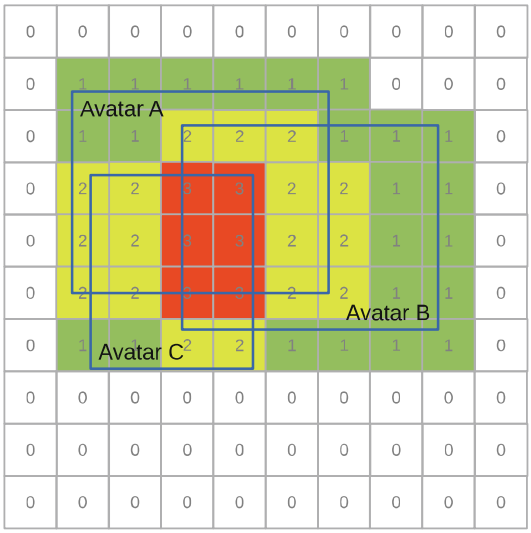


Abbildung 2.1: Heat Map mit den AoIs verschiedener Avatare [BAW+16]

2.1.1 Avatarbasierte Lastverteilung

Die avatarbasierte Lastverteilung ist das grundlegende Verfahren mit dem die Lastverteilung in Timadorus stattfindet. Aufgabe von diesem Verfahren ist es, die Avatare die einem Spiel beitreten auf die verschiedenen Spielserver des Timadorus-Systems zu verteilen. Hierzu wird die simulierte Spielwelt zunächst in ein Raster eingeteilt. Diese werden genutzt um zu analysieren wie die Area of Interests (AoI) der einzelnen Avatare verteilt sind. Bei den AoIs handelt es sich um die Interessengebiete der einzelnen Avatare, die beschreiben über welches Gebiet ein Avatar Informationen der Spielwelt braucht, weil dieser potentiell damit interagieren kann. Entsprechend müssen Informationen von Elementen, die sich in diesem Gebiet befinden, auf den Spielserver geladen werden, der den Avatar verwaltet. Daher wird von dem Verfahren, unter Berücksichtigung der Auslastung des Spielservers, versucht, möglichst Spieler auf einem Spielserver zusammen zu bringen, deren AoIs sich überschneiden oder zumindest nah beieinander sind. In jeder Zelle des Rasters wird deshalb festgehalten, wie viele AoIs von verschiedenen Avataren sich auf einem Spielserver innerhalb dieses Gebietes befinden. Für einen Spielserver lässt sich dann daraus eine sogenannte Heat Map bilden (Abb. 2.1). Für neue Avatare wird nun die Summe der AoIs in den Zellen, mit denen die eigene AoI Überschneidungen hat, gebildet und dieser Wert mit der Auslastung des Spielservers verrechnet. Sollte keine Überschneidung vorliegen, so wird die Manhattan-Distanz zur nächsten Zelle berechnet in der sich ein AoI befindet. Dieser errechnete Wert wird nun genutzt um einen Avatar einem Spielserver zuzuordnen. Das besondere an dem Verfahren ist, dass ein Avatar nicht zwingend dem Spielserver

zugeordnet wird, bei dem die stärkste Überschneidung besteht, sondern bei starker Auslastung auch einem alternativen Spielserver zugeordnet werden kann. [BAW+16]

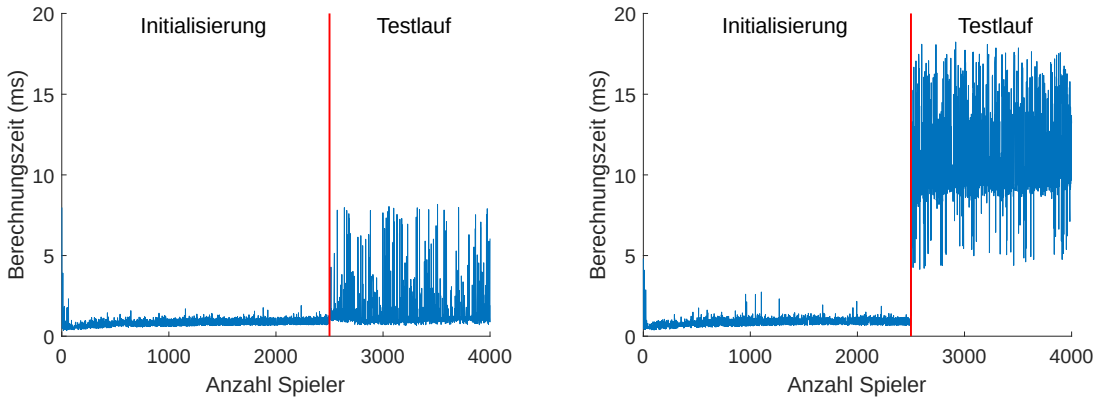
Damit die Last auch auf längere Zeit gut verteilt ist, muss auch eine Möglichkeit existieren, die Avatare bei Bedarf auf andere Server zu migrieren. Deshalb wird für jeden Avatar in festen Abständen der Spielserver erneut berechnet. Unterscheidet sich der berechnete Server von dem aktuell zugewiesenem Server, so wird der Avatar dem neu berechneten Server zugewiesen. [BAW+16]

Realisiert wurde das System in zwei Komponenten. Zum einen den Load Balancer, der die Lastverteilung übernimmt, sowie dem Monitor, der in die Spielserver integriert wird. Aufgabe des Monitors ist es den Zustand des Spielservers zu überwachen und diesen an den Load Balancer zu übermitteln. Darunter fallen Lastinformationen und die Belegung der Zellen im Raster der Spielwelt. [BAW+16]

2.1.2 Zellbasierte Migration

Die zellbasierte Migration war der Versuch die Neuberechnung zur Migration der Avatare zu optimieren. Da bei der periodischen Neuberechnung die Berechnung pauschal für jeden Avatar durchgeführt wird, finden diese auch für Avatare statt, bei denen ein Wechsel extrem unwahrscheinlich ist. Ziel war es deshalb die Anzahl der Neuberechnungen zu verringern. Deswegen werden zunächst die Avatare ermittelt bei denen ein Wechsel wahrscheinlich ist. Dazu werden sämtliche mit AoIs belegte Zellen innerhalb des Rasters der Spielwelt betrachtet. Sollten die AoIs innerhalb einer Zelle mehreren Spielservern zugewiesen sein, so wird für jeden Avatar, zu dem die AoIs gehören, der Spielserver neu berechnet. Dadurch findet die Neuberechnung für deutlich weniger Avatare statt. [All17b]

Mittels verschiedener Tests wurde untersucht, wie sich diese Anpassung auf die Performance auswirkt. Hierzu wurde verglichen, wie sich die Zeit zur Berechnung des Spielservers für einen neuen Avatar verändert, sobald sich die Avatare in der Spielwelt bewegen. Dabei stellte sich heraus, dass die Performance durch die zellbasierte Migration deutlich abnimmt (Abb. 2.2). So hat die Berechnung des Spielservers, bevor sich die Avatare bewegt haben (Initialisierung), bei beiden Verfahren zwischen 0,8 und 0,9 ms betragen. Sobald sich die Avatare jedoch bewegt haben (Testlauf), war eine deutliche Differenz zu sehen. So erhöhte sich bei der periodischen Berechnung die durchschnittliche Zeit zur Berechnung des Spielservers auf ca. 1,54 ms. Bei der zellbasierten Migration erhöhte sich die Berechnungszeit sogar auf durchschnittlich 10,85 ms.



(a) Periodische Neuberechnung

(b) Zellbasierte Migration

Abbildung 2.2: Berechnungszeiten pro Avatar (95er Perzentile) [All17b]

Somit sind bei beiden Einbußen erkennbar, doch sind diese bei der zellbasierten Migration deutlich stärker. Denn die Dauer der Berechnung ist dort mehr als sieben mal so lang. [All17b]

Dies bedeutet, dass die Performance trotz weniger Neuberechnungen schlechter geworden ist. Innerhalb dieser Arbeit soll dies als Anlass genommen werden, zu untersuchen wie scheinbare Fehler in verteilten Systemen ermittelt werden können. Das entwickelte Verfahren wird dieses Problem als Anwendungsfall nehmen, um zu ermitteln, wie es zu den Performanceproblemen kommen konnte.

2.1.3 Nachrichtenflüsse

Neben dessen Verteilung ist auch die starke lokale Nebenläufigkeit eine große Herausforderung bei der Untersuchung des Systems. Dies beruht auf der Tatsache, dass das System mittels Erlang als Aktorsystem realisiert wurde. Im Folgenden soll diese Problematik anhand von Beispielen verdeutlicht werden. Hierbei wird zur Vereinfachung nur das interne Verhalten des Load Balancers betrachtet. Bei allen weiteren Systemen wird nur die Kommunikation untereinander aufgezeigt.

Zuweisung von Avataren

Die Zuweisung der Avatare, ist der Einstiegspunkt an dem ein neuer Avatar in die Spielumgebung eintritt. Das heißt in dieser Phase muss auch entschieden werden welchem Spielserver der Avatar zugewiesen wird. Deshalb ist unter anderem die initiale Berechnung eines Spielservers für den Avatar Bestandteil dieses Prozesses. Zusätzlich werden hier auch die Neuberechnungs-

prozesse für die Avatare initialisiert. Sei es durch das starten eines neuen Aktors, wie bei der periodischen Neuberechnung oder durch den Eintrag in eine Tabelle, wie es bei der zellbasierten Migration geschieht. Insofern hat der Zuweisungsprozess einen starken Einfluss auf das System, weil sich dessen Zustand hier deutlich ändern kann. Da es sich bei diesem Prozess zudem um denjenigen handelt bei dem die Performance der zellbasierten Migration zusammengebrochen ist (siehe Abschnitt 2.1.2), ist dies einer der Bereiche die innerhalb dieser Arbeit stärker untersucht werden sollen. Deshalb soll dieser Prozess im Folgenden näher betrachtet werden.

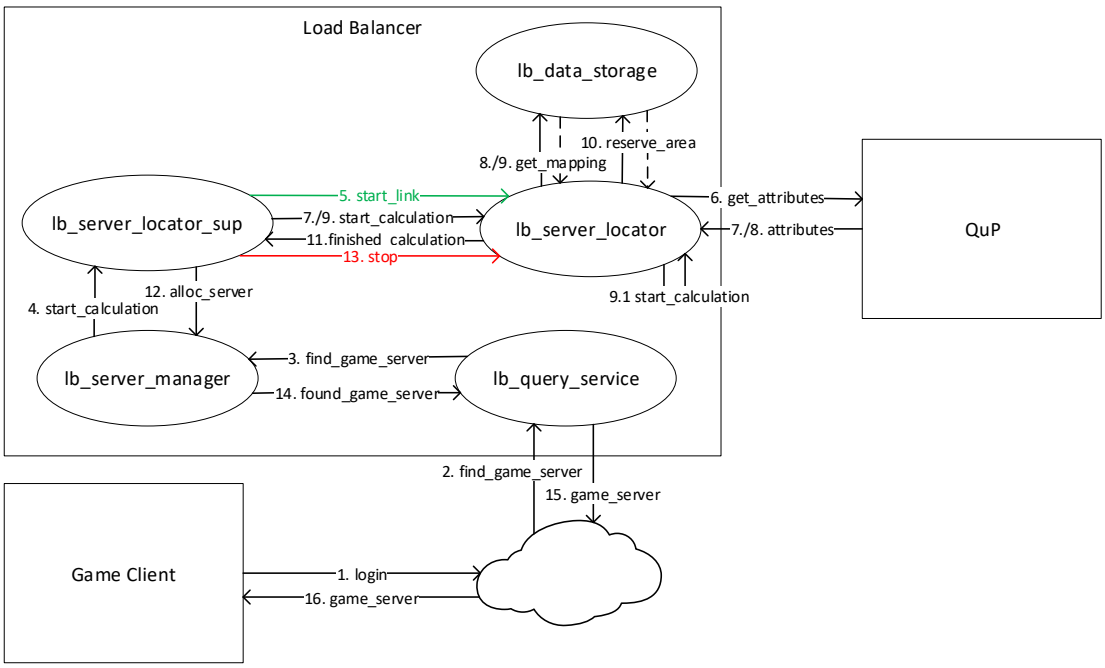
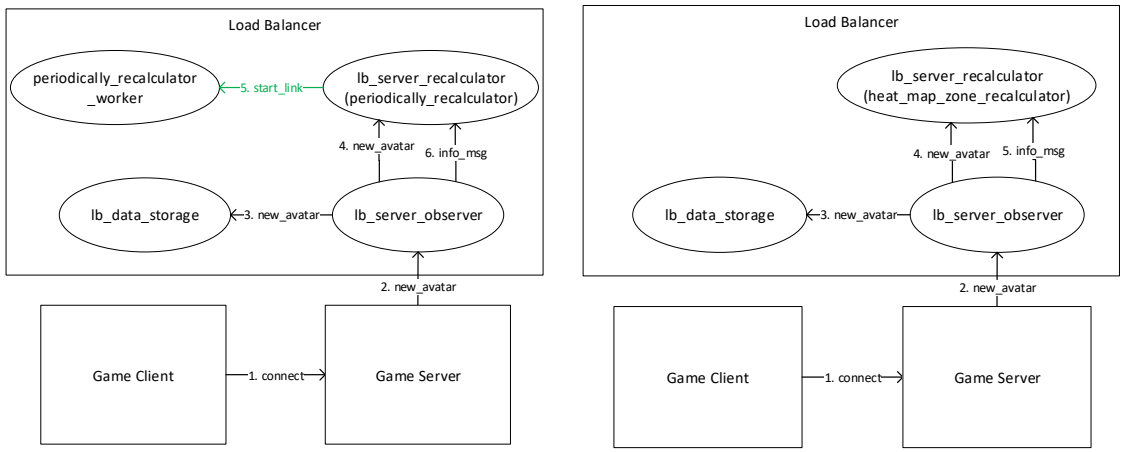


Abbildung 2.3: Kommunikation um initialen Server zu finden

Ermittlung des Spielservers Die Zuweisung neuer Avatare (Abb. 2.3) wird von den Spielclients der Spieler ausgelöst. Dies geschieht wenn dieser sich im System anmeldet (1.). Hierzu benachrichtigt der Client die Umgebung von Timadorus, dass dieser sich anmelden möchte. Daraufhin wird von dieser ein Anfrage an den Load Balancer weitergeleitet um den passenden Spielserver zu ermitteln (2.). Innerhalb des Load Balancers wird die Anfrage über den lb_server_manager an den lb_server_locator_sup weitergeleitet (3. & 4.). Dieser startet wiederum einen neuen lb_server_locator-Aktor. Dessen Aufgabe ist es, den Spielserver für den Avatar des Clients zu berechnen. Dazu wird zunächst die aktuelle Position des Avatars über QuP ermittelt (6.). Bei QuP handelt es sich um das System innerhalb von Timadorus mit dessen Hilfe

die komplette Spielwelt persistiert wird [BGL14]. Sobald die Attribute des Avatars ermittelt wurden (7./8.), wird die aktuelle Heat Map zusammen mit den Auslastungsinformationen der Spielserver (8./9.) bezogen. Auf Basis dieser Informationen wird der passende Spielserver ermittelt. Ist dies geschehen, so wird zunächst der zugewiesene Server in dem Bereich des Avatars reserviert (10.). Anschließend wird der lb_server_sup benachrichtigt und der lb_server_locator Prozess beendet (11. & 13.). Abschließend wird der Client über den zugewiesenen Spielserver informiert. Hierzu wird das Ergebnis über den lb_server_manager und dem lb_query_service an den Client übertragen (12., 14., 15. & 16.).



(a) Bei periodischer Neuberechnung

(b) Bei zellbasierter Migration

Abbildung 2.4: Kommunikation um Spieler im System anzumelden

Anmelden auf Spielserver Nachdem der Spielserver ermittelt wurde, meldet sich der Spielclient bei dem entsprechenden Spielserver an (Abb. 2.4). Dazu verbindet sich dieser zunächst mit diesem (1.). Anschließend teilt der Spielserver dem Load Balancer mit, dass sich ein neuer Avatar auf ihm angemeldet hat (2.). Der lb_server_observer informiert daraufhin den lb_data_storage, dass sich auf dem Spielserver ein neuer Avatar befindet (3.). Dadurch wird die zuvor beschriebene Reservierung bestätigt. Anschließend wird auch dem lb_server_recalculator mitgeteilt dass sich ein neuer Avatar auf dem Spielserver befindet. Abhängig davon welche Form der Neuberechnung konfiguriert ist, wird die Nachricht nun unterschiedlich verarbeitet. Bei der periodischen Neuberechnung wird ein neuer Prozess gestartet, der in regelmäßigen Abständen eine Neuberechnung für den Avatar initialisiert (Abb. 2.4a). Dies Bedeutet, dass für jeden registrierten Avatar ein eigener Aktor gestartet wird. Dadurch existieren eine Vielzahl von Aktoren, die parallel eine regelmäßige Neuberechnung für die einzelnen Avatare übernehmen.

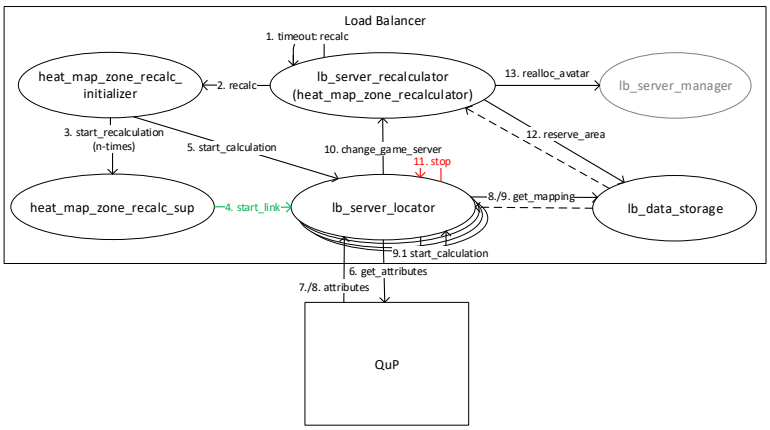
Bei der zellbasierten Migration wird der Avatar in eine Tabelle eingetragen, die später genutzt wird, um zu ermitteln für welche Avatare eine Neuberechnung stattfinden muss (Abb. 2.4b).

Neuberechnung & Migration

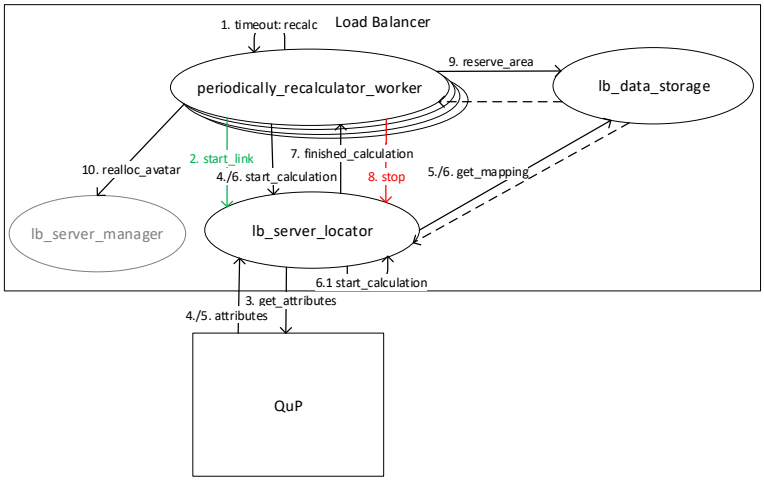
Die Neuberechnung der Spielserver für die Avatare ist der Bereich, in dem sich die periodische Neuberechnung und die zellbasierte Migration am stärksten unterscheiden. Denn hier sollte die Optimierung der Performance stattfinden [All17b]. Entsprechend muss vor allem hier nach Gründen für die Performance-Einbußen gesucht werden. Deshalb wird im Folgenden dieser Prozess näher betrachtet. Mit der Neuberechnung einhergehend ist die Migration des Avatars, wenn ein neuer Spielserver ermittelt wurde. Im Folgenden wurden diese beiden Elemente separiert. So wird zunächst der Prozess der Neuberechnung beschrieben und anschließend die Migration eines Avatars.

Neuberechnung Der Prozess der Neuberechnung unterscheidet sich zwischen der periodischen Neuberechnung und der zellbasierten Migration deutlich (Abb. 2.5). Zwar findet in beiden Fällen die Neuberechnung in zuvor definierten regelmäßigen Abständen statt. Bei der periodischen Neuberechnung findet dies jedoch für jeden Avatar separat statt, wohingegen die Neuberechnung bei der zellbasierten Migration für alle ermittelten Avataren zusammen stattfindet. Entsprechend wird die Neuberechnung bei beiden Vorgängen durch eine *timeout* Nachricht gestartet (1.). Bei der zellbasierten Migration (Abb. 2.5a) wird nun der Neuberechnungsprozess initialisiert. Dazu wird der heat_map_zone_recalc_initializer benachrichtigt (2.). Dieser sucht nach potentiellen Kandidaten, bei denen eine Neuberechnung stattfinden soll. Wurden welche gefunden, so wird für diese der Berechnungsprozess gestartet (3., 4. & 5.). Dazu wird für jeden Avatar, für den eine Berechnung durchgeführt werden soll, ein eigener Aktor gestartet. Bei diesem Aktor handelt es sich um einen lb_server_locator. Dieser wird auch für die initiale Zuweisung der Avatare genutzt. Entsprechend wird genauso beim QuP-System nach der aktuellen Position des Avatars gefragt und anschließend die aktuelle Heat Map sowie die Lastinformationen der Spielserver vom lb_data_storage bezogen (6., 7./8. & 8./9.). Nachdem der Spielserver berechnet wurde, wird zunächst überprüft, ob sich das Ergebnis vom aktuell zugewiesenen Spielserver unterscheidet. Sollte dies der Fall sein, so wird der heat_map_zone_recalculator benachrichtigt (10.). Dieser reserviert dann den neuen Spielserver in den Zellen des Avatars im lb_data_storage und benachrichtigt den lb_server_manager, welcher die weiteren Schritte zur Migration des Avatars einleitet (12. & 13.).

Bei der periodischen Neuberechnung (Abb. 2.5b) existiert für jeden Avatar ein Aktor, der den



(a) Bei zellbasierter Migration



(b) Bei periodischer Neuberechnung

Abbildung 2.5: Neuberechnung der Server für Spieler

Zeitpunkt der Berechnung für seinen Avatar selbstständig steuert. Gibt es hier einen *timeout* innerhalb eines Aktors, so wird für den zugehörigen Avatar die Berechnung durchgeführt (1.). Dazu wird auch hier ein lb_server_locator Aktor gestartet, der die Berechnung übernimmt. Auch hier handelt es sich um den selben Aktor, wie bei der initialen Berechnung, sodass auch hier zunächst die Position vom QuP bezogen wird und die Berechnung anschließend mit der Heat Map und den Lastinformationen aus dem lb_data_storage durchgeführt wird (3., 4./5. & 5./6.). Nachdem die Berechnung abgeschlossen wurde, wird der für den Avatar zuständige Aktor über das Ergebnis informiert (7.). Unterscheidet sich der berechnete Spielserver von dem aktuell zugewiesenen Spielserver, so wird der Spielserver im Bereich des Avatars reserviert

und der Migrationsprozess über den lb_server_manager eingeleitet (9. & 10.).

Der Hauptunterschied der Verfahren liegt vor allem in dem Prozess, wie die Avatare ausgewählt werden für die Neuberechnungen stattfinden sollen. Bei der zellbasierten Migration gibt es einen einzelnen Aktor, der in Regelmäßigen Abständen nach Kandidaten sucht die potentiell den Server wechseln und für diese eine Neuberechnung startet. Dadurch soll vor allem die Anzahl der Berechnungen minimiert werden. Bei der periodischen Neuberechnung wird für jeden Avatar ein eigener Aktor gestartet, der in regelmäßigen Abständen eine Neuberechnung für diesen Avatar initialisiert.

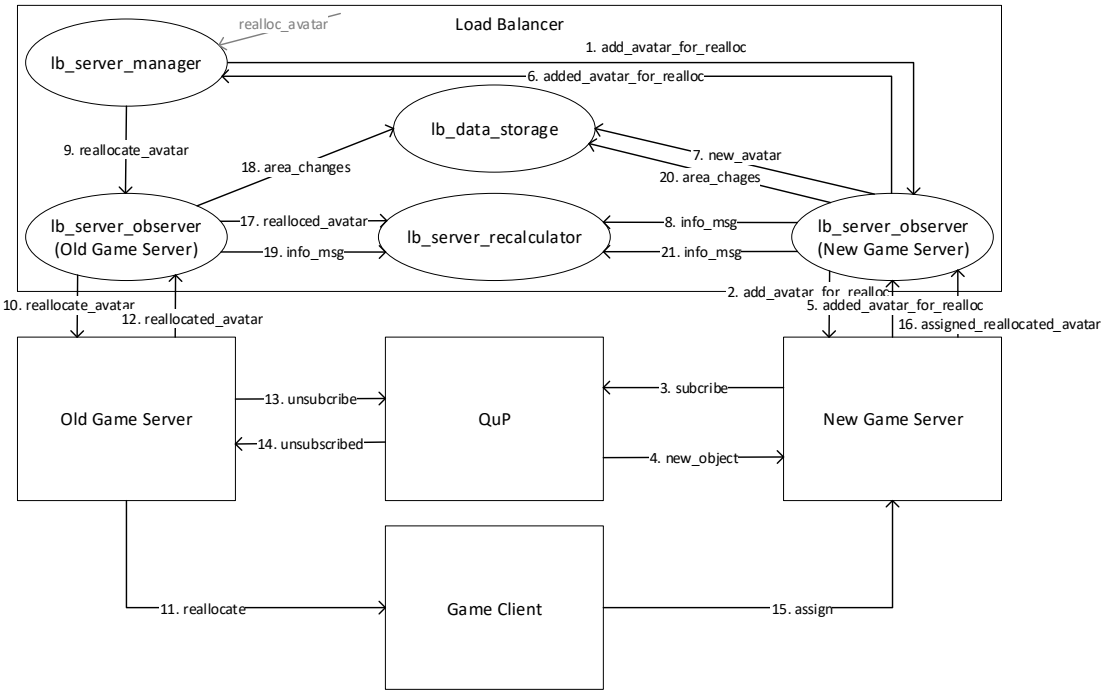


Abbildung 2.6: Migration eines Avatars vom alten auf den neuen Spielserver

Migration von Avataren Die Migration eines Avatars erfolgt immer dann, wenn während der Neuberechnung ein anderer Spielserver ermittelt wurde, als der Avatar gerade zugeordnet ist. In einem solchen Fall wird zunächst der lb_server_manager über die Veränderung informiert (Abb. 2.6). Dieser initialisiert dann den Migrationsprozess. Dazu wird als erstes der lb_server_observer informiert, der den Spielserver beobachtet, dem der Avatar zugewiesen werden soll (1.). Beim lb_server_observer handelt es sich um den Aktor, der die Kommunikation mit einem speziellen Spielserver übernimmt. Der Aktor, der dem neu zugewiesenen Spielserver

des Avatars zugewiesen ist, informiert im nächsten Schritt den Spielserver, damit dieser bereits die Daten des Avatars vom QuP beziehen und abonnieren kann (2. & 3.). Nachdem dieser die Daten für den Avatar bezogen hat, wird der zuständige lb_server_observer informiert, dass die Daten des Avatars bezogen wurden (4. & 5.). Daraufhin wird die Reservierung beim lb_data_storage bestätigt und das erfolgreiche Beziehen der Daten dem lb_server_manager mitgeteilt (6. & 7.). Dieser leitet dann den Wechsel des Avatars vom alten zum neuen Spielserver ein (9.). Der lb_server_observer des alten Spielserver benachrichtigt dazu den alten Server darüber, dass der Avatar den Spielserver wechseln soll (10). Der alte Spielserver benachrichtigt entsprechend den Spielclient des Avatars über den Wechsel, sodass sich dieser beim neuen Spielserver anmelden kann (11. & 15.). Hier kommt zum tragen, dass die Daten des Avatars bereits bezogen und abonniert wurden. Dadurch kann der Wechsel aus Sicht des Spielclients flüssiger vollzogen werden. Nachdem der Spielclient benachrichtigt wurde, wird der lb_server_observer des alten Spielservers über den Wechsel informiert und der Avatar vom Spielserver deabonniert (12., 13. & 14.). Zeitgleich informiert der neue Spielserver seinen lb_server_observer über den erfolgten Wechsel des Avatars (16.). Durch Benachrichtigung des lb_server_recalculator wird der Wechselprozess abgeschlossen (17.). Abschließend werden die mitgelieferten Zustandsinformationen der Spielserver verarbeitet (18., 19., 20. & 21.).

Fazit

Wie an den Beispielen gesehen werden konnte, kommunizieren die Aktoren des Systems in einem hohen Maße miteinander. Hierbei finden Teile der Kommunikation parallel, d.h. zeitgleich mit anderen Teilen der Kommunikation, statt. Vor allem an Stellen wo Aktoren mehrfach gestartet werden, entsteht ein hoher Grad der Nebenläufigkeit. Dadurch steigen die Kommunikationsmuster zusätzlich an. Hinzu kommt, dass nicht alle vorhandenen Prozesse aufgezeigt wurden, sondern nur eine Auswahl von Prozessen dargestellt wurden. So gibt es z.B. noch die regelmäßige Zustandsübermittlung der Spielserver. All dies geschieht nebenläufig. Schon bei der Betrachtung der vorgestellten Nachrichtenflüsse ist erkennbar, dass die Kommunikation unübersichtlich und in Teilen kaum nachvollziehbar ist. Hier kommt auch der Nichtdeterminismus von Nebenläufigen Nachrichten zum tragen. Durch die Kombination von lokaler und verteilter Nebenläufigkeit wird dieses Problem noch verstärkt. Hinzu kommt, dass viele Vorgänge dieselben Aktoren aufrufen. So werden regelmäßig der lb_server_manager aber auch der lb_data_storage aufgerufen. Dadurch können Flaschenhälse entstehen, die, wenn die Posteingänge dieser Aktoren zu schnell anwachsen, die Anwendung blockieren können.

2.1.4 Hypothesen

Die Performanceprobleme können eine Vielzahl von Ursachen haben. Da es jedoch eine Reihe von Aktoren gibt, die als einzige im System dedizierte Aufgaben übernehmen liegt ein starker Verdacht darin, dass diese Flaschenhälse bilden und sich dort Nachrichten anstauen. Zu diesen Aktoren gehören unter anderem der lb_server_manager sowie der lb_data_storage. Diese sind zentrale Elemente im Load Balancer und werden von einer Vielzahl von Aktoren aufgerufen. Wenn diese bei den beiden Verfahren unterschiedlich angesprochen werden, so kann dies zu einer Differenz in der Performance führen. Es könnte allerdings auch eine Überlastung des QuP-Systems vorliegen. Denn dieses wird für jede Berechnung nach den Werten eines Aktors befragt. Sollte es hier zu Einflüssen durch andere Berechnungen kommen, so kann auch dies die Performance beeinflussen.

Aus diesen hypothetischen Aussagen wird schnell ersichtlich, dass eine Möglichkeit benötigt wird, um das Verhalten des Systems ermitteln zu können. Deshalb beschäftigt sich diese Arbeit vor allem mit Systemen, die es ermöglichen, die Kommunikation innerhalb eines Systems zu beobachten. Hierbei spielt nicht nur die lokale Nebenläufigkeit durch die Aktoren eine Rolle, sondern auch die Kommunikation zwischen den Servern.

2.2 Anforderungen

Für die Beobachtung der Kommunikation des Systems stellen sich verschiedene Anforderungen. Deshalb sollen diese im Folgenden zunächst beschrieben werden.

2.2.1 Lokale Nebenläufigkeit

Das System zur Lastverteilung ist Aktorbasiert [HBS73]. Dies hat zur Folge, dass auf den einzelnen Knoten eine Vielzahl von Aktoren nebenläufig arbeiten, die zudem miteinander kommunizieren. Da auch die Kommunikation der Aktoren innerhalb der Knoten einen starken Einfluss auf das Verhalten des Systems hat, muss es möglich sein, auch diese lokale Nebenläufigkeit zu beobachten und zu analysieren.

2.2.2 Verteilte Nebenläufigkeit

Damit der Load Balancer in der Lage ist, Entscheidungen zu treffen, benötigt dieser bestimmte Informationen von den Spielservern. Darunter fallen deren Auslastung aber auch Informationen zur aktuellen Verteilung der Spieler innerhalb der Spielwelt. Durch den Austausch

dieser Informationen werden bestimmte Prozesse auf dem Load Balancer angestoßen, die die Verteilung der Avatare auf die einzelnen Spielservern stark beeinflussen. Dies geht soweit, dass durch diese Informationen eine Umverteilung der Avatare eingeleitet werden kann. Entsprechend müssen auch diese verteilten Prozesse für eine Analyse der Lastverteilungsverfahren betrachtet werden können.

2.2.3 Tracing-Graph

Ein Tracing-Graph bietet die Möglichkeit, dass die Kommunikation zwischen den einzelnen Aktoren und den Servern visualisiert werden kann. Da innerhalb dieser Arbeit keine automatisierte Analyse vorgesehen ist, wird dies auch einer der wichtigsten Möglichkeiten sein um die Interaktion der Aktoren zu analysieren. Ziel soll es sein, dass die Interaktion in einem Zeitstrahl sichtbar wird. Ein Beispiel für diese Art von Graph stellt ShiViz [BWBE16] dar. Zusätzlich soll der Graph mit Performancemetriken angereichert werden können.

2.2.4 Happens Before Order

Eine Happens Before Order innerhalb der Aufzeichnung stellt sicher, dass nachvollzogen werden kann, in welcher Reihenfolge Nachrichten im System gesendet wurden. Hierbei kann zwischen Ereignissen unterschieden werden, die nacheinander ($a \rightarrow b$) und die nebenläufig ($a \nrightarrow b$) geschehen sind. Dies ist unabdingbar, damit der Tracing Graph erstellt werden kann. Zudem bietet eine solche Ordnung die Möglichkeit für weitere zukünftige Analysemöglichkeiten, wie z.B. dem Happened-Before Join [MRF15].

2.2.5 Kein Anpassen von Code

Das Analysetool soll möglichst leicht zu integrieren sein. Deshalb soll zur Integration kein Code angepasst werden müssen. Dadurch kann das Tool durch alle in Erlang verfassten Anwendungen genutzt werden. Ein explizites definieren von Punkten zum Aufzeichnen ist zudem fehleranfällig [LGW+08, SBB+10]. Gerade in einer Umgebung, wie der von Timadorus, ist ein aus Anwendungssicht transparenter Ansatz nützlich, da diverse Systeme für diese Umgebung entwickelt wurden und diese dann nicht extra zum erheben der Daten angepasst werden müssen. So ließe sich eine Aufzeichnung der Teilsysteme, die mit den Load Balancer interagieren, mit einem geringem Aufwand realisieren. Denn insbesondere die von den Spielservern übermittelten Informationen an den Load Balancer beeinflussen dessen Berechnungen stark. Daher ist auch deren Verhalten von besonderem Interesse. Ein Werkzeug, das im Kontext von Timadorus hierfür nützlich sein kann, ist der Erlang Trace Tool Builder [Erlb], da hier-

mit zahlreiche Informationen über die Aktoren innerhalb eines Erlang-Systems abgerufen werden können. Sollte eine Integration ohne Anpassungen nicht möglich sein, so muss der Aufwand möglichst gering sein. Dies könnte z.B. durch den systemübergreifenden Einsatz einer Standardbibliothek, wie dies bei Dapper geschieht [SBB+10], erreicht werden.

2.2.6 Performance-Metriken

Um die Daten besser analysieren zu können sollen bestimmte Performance-Metriken erfasst werden. Hierbei soll es sich um Folgende Metriken handeln:

- CPU-Auslastung
- Speicherauslastung
- Systemzeit bestimmter Nachrichten
- Fülle des Posteingangs der Aktoren

Bei der Systemzeit der Nachrichten, soll vor allem erfasst werden, wann diese eingetroffen sind und wann auf diese geantwortet wurde. Dies soll zumindest für Nachrichten geschehen, die aus dem Netzwerk stammen. Wie bereits in Abschnitt 2.1 beschrieben, benötigen Anfragen zur Berechnung eines Spielservers für Avatare, bei Einsatz der zellbasierten Migration, ungefähr 8 mal länger sobald sich die Avatare auf den Spielservern bewegen. Durch Erfassen der Zeit wäre es möglich, die Anfragen zu ermitteln, die länger benötigen und so zu erkennen, welche Besonderheiten in der Kommunikation dort stattgefunden haben. Die Fülle des Posteingangs soll ermittelt werden, damit erkannt werden kann, ob sich Nachrichten an einem Aktor aufstauen und dieser nicht in der Lage ist, diese schnell genug abzuarbeiten.

2.2.7 Queries

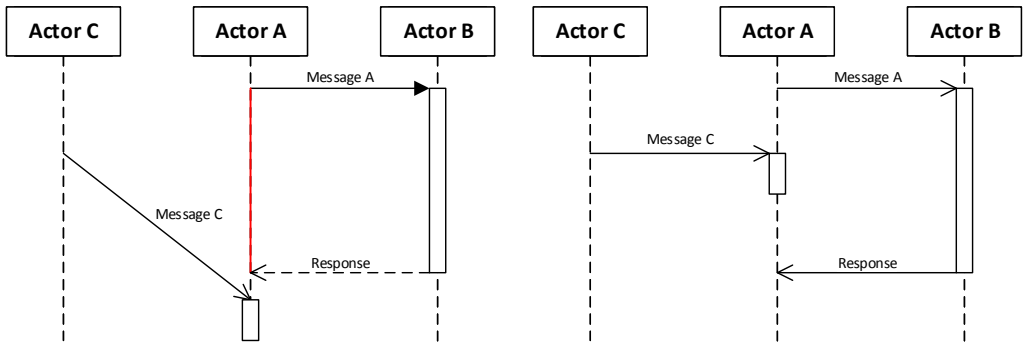
Aufgrund der Menge an Daten, die ermittelt werden, muss eine Möglichkeit zur Verfügung stehen, um die betrachteten Daten zu reduzieren. Sei es durch filtern oder durch Daten heraussuchen, die untersucht werden sollen. Mittels Queries kann eine solche Suche gestaltet werden. Zudem können auf Basis solcher Queries auch Aggregationen durchgeführt werden [LSZE11, CMF+14]. Folgende Funktionalitäten sollte eine Abfrage Schnittstelle bieten:

- Filtern einzelner Aktoren.
  - Hierbei muss die Happens Before Order eingehalten werden.
- Pfad einer Anfrage durch das Aktorensystem ermitteln.

- Dauer einer Anfrage bis zu dessen Antwort ermitteln.
- Der Zeitraum, der betrachtet wird, muss einschränkbar sein.

ShiViz bietet z.B. bereits die Möglichkeit, Anfragen an einzelne Knoten zu filtern und dabei die Reihenfolge der Nachrichten beizubehalten [BWBE16]. Da die Visualisierung jedoch in der Lage sein muss, die Kommunikation von Aktoren darzustellen, kann die Lösung nicht einfach übernommen werden. Der Pfad den eine Anfrage durch das Aktorsystem nimmt, ist wichtig zu visualisieren, weil so sichtbar wird, welche Interaktionen während dieser Anfrage stattfanden. Dadurch wird besser erkennbar, welche Besonderheiten auftreten, wenn eine Anfrage länger dauert. Um zu wissen, welche Anfragen zu diesen längeren Anfragen gehören, muss deshalb auch die Dauer einer Anfrage ermittelt werden können. Sollte bekannt sein in welchem Zeitraum sich eine Anfrage ungefähr befunden hat, so kann durch das Einschränken des Zeitraumes gezielt nach diesen gesucht werden. Zudem fördert dies die Übersichtlichkeit bei großen Datenmengen.

2.2.8 Blockierende synchrone Nachrichten



(a) Synchroner Aufruf (b) Asynchroner Aufruf

Abbildung 2.7: Synchroner vs. asynchroner Aufruf

Einige Aktorbibliotheken stellen die Möglichkeit bereit synchrone Aufrufe durch zuführen. So ist dies z.B. auch innerhalb von Erlang/OTP [Erlc] möglich. Neben der Eigenschaft, dass es sich um einen synchronen Aufruf handelt, kommt hier noch hinzu, dass dieser Aufruf blockierend ist. D.h. dass während der Sender auf die Antwort wartet, dieser keine weiteren Nachrichten empfangen kann. Damit wird der Aktor davon abhängig, wie lange ein anderer Aktor zum Bearbeiten einer Anfrage benötigt. Denn solange dieser nicht geantwortet hat, kann der Prozess keine weiteren Nachrichten entgegen nehmen. Nachrichten, die nicht verarbeitet

werden können, landen bei Aktoren zunächst in einem Posteingang. Erst wenn der Aktor den aktuellen Aufruf verarbeitet hat, entnimmt dieser die nächste Nachricht, um diese zu verarbeiten [HBS73]. Abb. 2.7a zeigt ein Beispiel in dem Aktor A eine synchrone Nachricht an Aktor B sendet. Während Aktor B die Nachricht verarbeitet sendet Aktor C eine Nachricht an A. Dieser kann die Nachricht jedoch nicht entgegennehmen, weil Aktor B noch nicht geantwortet hat. Somit landet die Nachricht im Posteingang von dem Aktor und wird erst verarbeitet, wenn der Aktor bereit ist. Im Gegensatz dazu könnte Aktor A bei einer asynchronen Nachricht, die parallel eintreffende Nachricht verarbeiten (Abb. 2.7b), weil nicht auf die Antwort von Aktor B gewartet werden muss. Bei synchronen Aufrufen verlängert sich somit die Zeit in der keine Nachrichten entgegen genommen werden können. Sollten zusätzlich synchrone Nachrichten in einem aufgerufenem Aktor stattfinden, so entstehen weitere Abhängigkeiten zu anderen Aktoren. Treten zyklische Aufrufe auf, so kann dies sogar zu einem Deadlock führen. Innerhalb von Erlang werden diese mithilfe eines Timeouts aufgelöst. Überschreitet ein synchroner Aufruf diesen Timeout, so wird davon ausgegangen, dass es ein fehlerhaftes Verhalten gibt. Der Aufruf wird daraufhin abgebrochen und der Aktor beendet sich. Das weitere Verhalten hängt von der Supervisor-Strategie ab, die der Supervisor von dem Aktor, der sich beendet, nutzt. Dieser kann entscheiden, ob der Aktor neu gestartet wird oder nicht. Dies macht alle synchronen Aufrufe zu einer potentiellen Schwachstelle, wenn diese falsch eingesetzt werden. Deshalb sollen diese auch gesondert betrachtet werden können.

2.2.9 Weiterführende Anforderungen

Neben den bereits genannten gibt es auch optionale Anforderungen. Hierbei handelt es sich um Anforderungen, die den Umfang dieser Arbeit übersteigen. Einige von diesen würden vor allem das Debugging erleichtern oder Daten bereits analysieren. Andere sind zwingend nötig, damit das System auch in einer produktiven Umgebung genutzt werden kann.

Log and Replay

Log and Replay ermöglicht es die Ereignisse mithilfe der vorliegenden Log-Dateien zu wiederholen [LSZE11]. Dadurch wäre es möglich vergangene Ereignisse besser nachvollziehen zu können. So könnte z.B. ein vorhandener Debugger genutzt werden, um sich schrittweise durch die Codebasis zu arbeiten. Dies wäre insbesondere bei Aktorsystemen attraktiv, da so das Verhalten einer Nachricht besser nachvollzogen werden könnte. Evtl. könnten sogar erste Codeanpassungen getestet werden, indem die neue Codebasis mit der fehlerhaften Nachricht in dem aufgezeichneten Zustand aufgerufen wird. Daraus ergibt sich jedoch auch, dass für

Log and Replay der Zustand des Systems festgehalten werden müsste, was einen erheblichen Overhead darstellen würde.

Critical Path Analysis

Eine solche Analyse würde es ermöglichen, den kritischen Pfad von Anfragen im Netzwerk zu ermitteln [CMF+14]. In diesem Kontext könnte es auch dazu dienen, zu ermitteln, welche Pfade der Kommunikation innerhalb des Aktorsystems dafür sorgen, dass Anfragen länger dauern. Dies würde die Analyse der Kommunikation vereinfachen.

Sampling

Damit das System auch produktiv genutzt werden könnte, müssten zwingend Optimierungen vorgenommen werden. Dies kommt u.a. daher, dass eine hohe Menge an Daten über das beobachtete System gesammelt werden, wodurch die Performance des Systems beeinträchtigt werden kann. Mittels Sampling könnte die Beeinträchtigung minimiert werden [LGW+08, CMF+14]. Denn hierbei werden die Daten nur stichpunktartig aufgezeichnet. Da bekannt ist, dass bei einem hohen Datenaufkommen bereits bei einer geringe Samplingrate ein hoher Erkenntnisgewinn erzielt werden kann [CMF+14], könnte diese sogar entsprechend der Auslastung angepasst werden. Es entstehen jedoch auch neue Problemstellungen. So wird durch das unvollständige Aufzeichnen der Graph unvollständig und es müsste ein Weg gefunden werden, um die Happens Before Order dennoch erhalten zu können.

Live Verfügbarkeit der Daten

Würden die Daten direkt nach dem Erfassen zur Verfügung stehen, so könnte eine erste Analyse bereits erfolgen während das System noch läuft. Sollte es automatisierte Analysemöglichkeiten geben, so wäre es durch die direkte Verfügbarkeit auch möglich, ein Warnsystem aufzusetzen, sobald das System in einen kritischen Zustand gerät. Die direkte Verfügbarkeit von Daten wäre damit vor allem von Interesse, wenn das System auch produktiv genutzt werden soll.

Dynamisch zur Laufzeit einbinden

Die Möglichkeit die Aufzeichnung dynamisch zur Laufzeit einzubinden setzt voraus, dass auf Seiten des zu beobachtenden Systems keine spezifischen Vorkehrungen getroffen werden müssen damit dieses beobachtet werden kann. Dadurch wäre es möglich ein System spontan zu beobachten, wenn z.B. festgestellt wird, dass dieses sich nicht wie erwartet verhält.

2.3 Bisherige Arbeiten

Bevor in die Umsetzung eines Systems zur Aufzeichnung der Kommunikation von Aktoren eingestiegen wird, sollen zunächst bisherige Arbeiten auf dem Gebiet vorgestellt werden. Hierbei wird vor allem ein Augenmerk auf das Ermitteln des Verhaltens von verteilten sowie nebenläufig arbeitenden Systemen gelegt.

2.3.1 Fehler in verteilten Systemen

Zunächst soll auf die verschiedenen Fehlerszenarien in verteilten Systemen eingegangen werden. In der Cloud Bug Study von Gunawi et al. [GMS+14] wurden die am häufigsten auftretenden Probleme innerhalb von ausgewählten Open Source Projekten untersucht. Dabei stellte sich heraus, dass sich die größten Probleme bei der Zuverlässigkeit (45%), Performance (22%) und Verfügbarkeit (16%) von Programmen ergeben. Allein Race Conditions deckten hierbei 12% aller Fehler ab. Innerhalb dieser Race Conditions waren ca. 50% verteilte Race Conditions. Aufgedeckt wurden auch sogenannte „Killerbugs“. Hierbei handelt es sich um Fehler, wie positive Feedbackschleifen, bei denen sich die Situation durch Neustarts weiter verschlechtert, weil diese noch mehr Last erzeugen oder verteilte Deadlocks. Auch in TaxDC [LLLG16] wurden verschiedene Open Source Projekte betrachtet um Fehler in verteilten Systemen zu untersuchen. Dabei konnte ermittelt werden, dass 63% aller verteilten Nebenläufigkeitsprobleme durch Hardwarefehler sichtbar und 60% dieser Fehler durch eine einzige falsch getimte Nachricht ausgelöst werden können. Hierbei traten 44% aller Timingfehler alleine durch eine falsche Reihenfolge von Nachrichten auf. Zudem konnte gezeigt werden, dass sich ein Großteil verteilten Nebenläufigkeitsfehler mit 3 Knoten reproduzieren lässt, eine Erkenntnis, die auch in anderen Arbeiten bereits festgestellt wurde [YLZ+14, LLLG16]. In verteilten Systemen kann es in der Folge von Fehlern zu einer Vielzahl von weiteren Problemen kommen. Darunter fehlerhafte Operationen, Performanceproblemen, Datenverlust oder auch korrupte Daten [GMS+14]. Allerdings können auch Fehlerbehandlungen zu Fehlern führen. So werden in verteilten Systemen häufig Timeouts eingesetzt. Diese können jedoch auch zu False-Positives führen wenn dieser zu kurz ist bzw. den Eindruck einer hängen gebliebenen Anwendung vermitteln, wenn diese zu groß sind [GMS+14].

2.3.2 Log & Replay

Log & Replay bietet die Möglichkeit aufgezeichnete Daten wieder abzuspielen, um so weitere Erkenntnisse gewinnen zu können. Bei Recon [LSZE11] handelt es sich um so ein System. Hier wurde das Logging Verfahren strikt vom Replay Mechanismus getrennt. Dadurch sollen

alle aufwendigen Operationen nur noch während der Replayphase stattfinden. Zum Logging kommt hierbei Jockey [Sai05] zum Einsatz. Soll das System nun analysiert werden, so wird eine spezielle Anfragesprache genutzt. Mit dieser lässt sich eine Anfrage schreiben, die anschließend in Programmbefehle kompiliert und daraufhin ausgeführt wird [LSZE11].

2.3.3 Tracing

Beim Einsatz von Tracing wird das Verhalten des Systems zunächst aufgezeichnet. Anschließend können die Daten ausgewertet werden. Bei ShiViz [BWBE16] handelt es sich um ein System, dass es ermöglicht, die Kommunikation eines verteilten Systems zu visualisieren. Dazu muss diese in einem speziellen Format aufgezeichnet werden, damit sie anschließend eingelesen und ausgewertet werden kann. Um die Reihenfolge der Nachrichten sicher zu stellen, werden bei ShiViz Vektoruhren eingesetzt. Neben der Visualisierung stellt ShiViz auch eine Reihe von Werkzeugen bereit. Unter anderem eine spezielle Abfragesprache, die genutzt werden kann, um das Verhalten des Systems zu analysieren. DCatch [LLL+17] hat den Anspruch verteilte Nebenläufigkeitsfehler im System zu erkennen bevor sie auftreten. Dazu wurde ein eigenes Happens-Before-Modell erstellt, dass definiert wie ein valides Verhalten aussieht. Um das untersuchte System validieren zu können, werden u.a. sämtliche Speicherzugriffe sowie die Kommunikation zwischen den Knoten aufgezeichnet. Aus diesen Informationen wird ein Happens-Before-Graph aufgebaut, der mit den Regeln aus dem Happens-Before-Model untersucht werden kann. Sollten sich aus diesen Informationen konfliktbehaftete Zugriffe oder Kandidaten für verteilte Nebenläufigkeitsprobleme ergeben, so wird dies durch DCatch berichtet. Bei D^3S [LGW+08] wurde vor allem das Problem angegangen, dass Entwickler antizipieren müssen was wichtig ist aufzuzeichnen. Deshalb wurde ein System entwickelt, das dynamisch zur Laufzeit eingebunden werden kann. Dazu werden durch den Nutzer zunächst Prädikate definiert, die beschreiben, was verifiziert werden soll. Diese werden kompiliert und können dynamisch eingebunden werden. Dabei wird Code in das Programm injiziert, der die zu beobachtende Zustände freilegen kann. Sollten Fehler ermittelt werden, so ist D^3S in der Lage, eine Sequenz der Probleme anzuführen, die zu dem Problem geführt haben. Bei Dapper [SBB+10] handelt es sich um eine Tracinginfrastruktur von Google. Auch hier war es das Ziel ein Tracingsystem zu haben, das Transparent für den Entwickler agiert, da alles andere als fragil betrachtet wurde. Im Gegensatz zu D^3S wird das Tracing jedoch nicht dynamisch eingebunden. Stattdessen wird eine RPC-Bibliothek genutzt, die die Aufzeichnungen im Hintergrund durchführt. Zwar steht es den Entwicklern frei programatisch noch weitere Daten durch Dapper aufzeichnen zu lassen, jedoch wird standardmäßig nur die Kommunikation zwischen den Knoten aufgezeichnet. Da das System in Produktion eingesetzt wurde, lag der

Fokus hier vor allem auf einem geringen Overhead und einer hohen Skalierbarkeit. Da es sich bei Dapper um eine ganze Infrastruktur handelt, werden auch weiterführende Werkzeuge bereitgestellt, um die Aufzeichnung zu analysieren. Darunter auch den Dapper Trace Tree, der die Kommunikation zwischen den verschiedenen Knoten darstellt. Damit die Daten genutzt werden können, werden die Daten zunächst lokal abgelegt und von dort gezogen, um in einer Bigtable Datenbank [CDG+08] gespeichert zu werden. Andere Systeme wie Zipkin [Zip] haben sich von Dapper inspirieren lassen und nutzen einen ähnlichen Ansatz. Bei Pivot Tracing [MRF15] handelt es sich um ein weiteres System, dass die Möglichkeit bieten will erst zur Laufzeit zu definieren was aufgezeichnet werden soll. Dazu können spezielle auf LINQ [MBB06] basierende Anfragen gestellt werden, die die Aufzeichnung definieren. Ein Hauptbeitrag von Pivot Tracing ist der sogenannte Happened Before Join der es ermöglicht mehrere Anfragen basierend auf Lamports Happened Before Relation [Lam78] zu vereinen. Magpie [BIMN03] nutzt Machine Learning um Anomalien feststellen zu können. Dazu wird stetig der Resourcenverbrauch beobachtet und für jede Anfrage ein Audit-Trail gebildet, der die Kommunikation im System repräsentiert. Mit diesen Daten werden sogenannte *Bayesian Watchdogs* erstellt. Hierbei handelt es sich um probabilistische Modelle, die beschreiben wie ein normales Verhalten aussehen sollte.

2.3.4 Model Checking

Beim Model Checking wird versucht Fehler zu finden, bevor sie in der produktiven Umgebung landen. Dynamische Model Checker probieren hierfür systematisch verschiedene Szenarien aus. Für verteilte Systeme wird hierbei in aller Regel eine Schicht bei der Übertragung von Nachrichten eingebunden. Dadurch kann der Model Checker verschiedene Ereignisse auslösen, die sich auf die Nachrichten auswirken. Ziel ist es hierbei möglichst viele Szenarien abbilden zu können. Insbesondere bei der Reihenfolge der verschiedenen Nachrichten sollen möglichst viele Permutationen überprüft werden [LHJ+14]. Bei Deligiannis et al. [DMT+16] muss hierfür zunächst eine spezielle Testspezifikation mittels P# geschrieben werden, die das angestrebte Verhalten beschreiben. Außerdem muss ein sogenannter *Test Harness* bereitgestellt werden, der durch das nichtdeterministische Auslösen von Events, interessantes Verhalten im zu testendem System auslöst. Anschließend wird die Spezifikation systematisch durchgetestet. Damit dies möglich ist, muss das Versenden von Nachrichten über von dem Testsystem bereitgestellten Methoden stattfinden. Dadurch gehen die Nachrichten durch das Testsystem. Sollte ein Fehler gefunden werden, so stellt das System die Aufzeichnung bereit, bei der dieser Fehler aufgetreten ist. Neben logischen Fehlern ist das System auch in der Lage zu erkennen wenn das getestete System hängen geblieben ist. Die Entwickler von Aspirator [YLZ+14] fanden in einer eigenen

Studie heraus, dass die meisten Fehler mit einfachen Regeln hätten verhindert werden können. Deswegen haben sie einen statischen Checker entwickelt, der Programmcode auf Basis ihrer Erkenntnisse untersucht und eine Warnung ausgibt, wenn potentielle Probleme gefunden werden. Da Model Checker für verteilte Systeme versuchen alle möglichen Folgen von Events abzuarbeiten, kann es hier zu einer Zustandsexplosion kommen. Deshalb versucht SAMC [LHJ+14] die zu überprüfenden Folgen von Events intelligent zu minimieren. Dazu werden semantische Informationen herangezogen, um so redundante Tests zu verhindern. Um diese Informationen zu erhalten wird überprüft, wie die verschiedenen Nachrichten verarbeitet werden. Innerhalb der Arbeit wurden zudem verschiedene Regeln vorgestellt, wie die Menge der überprüften Eventfolgen reduziert werden kann. So kann zum Beispiel eine permutation von Nachrichten ignoriert werden, wenn diese vollkommen unabhängig voneinander verarbeitet werden. MODIST [YCW+09] betrachtet das beobachtete System komplett als Blackbox. Es arbeitet hierbei zwischen dem Betriebssystem und der eigentlichen Anwendung. Somit ist der Model Checker komplett transparent für das getestete System. Auf dieser Ebene ist MODIST in der Lage Nachrichten umzusortieren oder Fehler zu injizieren, indem es z.B. ein unzuverlässiges Netzwerk simuliert. Es werden dabei zwei Testszenarien unterstützt. Zum einen generische Tests die automatisiert durchgeführt werden und zum anderen systemspezifische Tests, die extra geschrieben werden müssen.

2.3.5 Monitoring

Beim Monitoring werden erfasste Daten beobachtet. Bovenzi et al. [BCPC11] haben ein System entwickelt, das zunächst aus den funktionellen Tests erlernt, wie sich das System verhalten soll. Dazu werden Parameter, wie das Interaktionsmuster zwischen den Komponenten oder Betriebssysteminformationen für das erwartete Verhalten herangezogen. Neben dem positiven Verhalten kann das System auch Muster für typische Fehlersituationen erlernen. Diese können z.B. aus fehlgeschlagenen Tests hergeleitet werden. Die Mystery Machine [CMF+14] ist ein Analysewerkzeug zur Untersuchung der Performance von heterogenen verteilten Systemen und stammt von Facebook. Hierbei wird auf die Daten von ÜberTrace zugegriffen. Dieses Werkzeug liest die Log-Dateien der verschiedenen Dienste ein und ist so in der Lage Traces von Anfragen an das System zu erstellen. Dies erfordert allerdings, dass die verschiedenen Log-Inhalte in einem einheitlichen Format eingelesen werden können. Da innerhalb des Systems Sampling zur Aufzeichnung der Daten genutzt wird, muss zudem sichergestellt werden, dass alle zusammenhängenden Aufrufe aufgezeichnet werden, damit die Traces überhaupt erstellt werden können. Deswegen kann ÜberTrace für alle Systeme entscheiden, welche Anfragen aufgezeichnet werden. Die Mistery Machine nutzt die aufgezeichneten Daten um ein kausales

Model der Interaktion der verschiedenen Komponenten zu erstellen. Auf diesem Model können verschiedene Analysen durchgeführt werden. Darunter die Erkennung von Anomalien oder eine Analyse von kritischen Pfaden im System.

2.4 Fazit

Tabelle 2.1: Verschiedenen Verfahren zur Untersuchung verteilter Systeme mit ihren aus den Arbeiten herleitbaren Eigenschaften

	Lokale Nebenläufigkeit	Verteilte Nebenläufigkeit	Tracing-Graph	Happens Before Order	Kein Anpassen von Code	Performance Metriken	Queries	Sampling	dynamisch zur Laufzeit	transparent für Program
Recon [LSZE11]	x	x			x					
Jockey [Sai05]	x				x					
ShiViz [BWBE16]		x	x	x			x			
DCatch [LLL+17]		x		x						
D^{3}S [LGW+08]		x		x	x		x[1]	x	x	x
Dapper [SBB+10]		x	x	x		x	x			x
Pivot Tracing [MRF15]		x		x	x		x[1]		x	
Magpie [BIMN03]		x	x[2]	x	x	x				
Testing mittels P# [DMT+16]		x								
Aspirator [YLZ+14]					x					
SAMC [LHJ+14]		x			x					
MODIST [YCW+09]		x			x					
Behaviour Monitoring [BCPC11]		x			x					
Mistery Machine [CMF+14]		x	x[2]	x		x	x	x		

Es gibt bereits eine ganze Reihe von Arbeiten, die das Analysieren der Interaktion von verteilten Systemen untersuchen. Werden dem allerdings der Kontext sowie Anforderungen dieser Arbeit entgegengestellt, so ist ersichtlich, dass keines die kompletten Anforderungen erfüllt (siehe Tabelle 2.1). Stattdessen können bestimmte Aspekte aus diesen Arbeiten genutzt werden. Insbesondere die Kombination aus lokaler und verteilter Nebenläufigkeit findet in der

[1] Abfrage nur zur Laufzeit möglich
[2] Nur als interne Datenstruktur (keine Visualisierung)

aktuellen Literatur wenig Beachtung. Wenn dann auch noch ein Tracing über diese beiden Ebenen betrachtet werden soll, dann konnte die betrachtete Litaratur dies nicht abdecken. Da wir uns aber innerhalb eines aktorbasierten Systems befinden sind gerade diese beiden Aspekte von hohem Interesse. Deshalb soll ein eigenes System zur Untersuchung der Umgebung entstehen. Hierbei soll auch auf bestehendes Wissen zurückgegriffen werden und dies für die benötigten Zwecke kombiniert werden.

3 ErlViz System

Mittels der ErlViz Plattform werden Kommunikationsdaten von Aktoren aufgezeichnet sowie visualisiert. Abgeleitet ist der Name von ShiViz [BWBE16], da das System zur Visualisierung der Daten auf diesem System aufbaut. Durch seinen starken Fokus auf Aktorsysteme, insbesondere auf Erlang, leitet sich der Name ErlViz ab. Im Folgenden soll zunächst die Architektur des Systems aus einer konzeptionellen Sicht betrachtet und eingeordnet werden. Zunächst gibt es eine allgemeine Beschreibung der Architektur. In den anschließenden Abschnitten werden die einzelnen Komponenten näher beschrieben.

3.1 Allgemeiner Aufbau

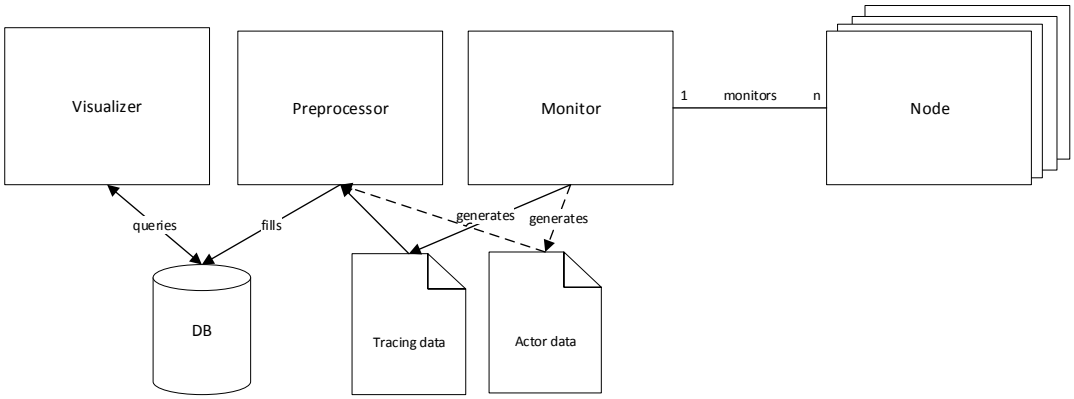


Abbildung 3.1: Überblick über das ErlViz System

In Abb. 3.1 ist das System in einer stark vereinfachten Version abgebildet. Die Plattform besteht aus mehreren Elementen. Diese lassen sich in drei Bereiche einteilen: dem Tracing, dem Preprocessing sowie der Visualisierung. Der Monitor ist die Komponente, die das Tracing der verschiedenen Knoten übernimmt. Hierbei fallen je nach Konfiguration ein bis zwei verschiedene Artefakte an. Diese beinhalten die aufgezeichneten Traces sowie eventuell zusätzlich aufgezeichnete Daten über die Aktoren. Bevor die angefallenen Daten visualisiert werden

können, müssen sie durch den Preprocessor verarbeitet werden. Hierbei werden die Inhalte der verschiedenen Dateien zusammengeführt, in ein durch die Visualisierung lesbares Format überführt und in einer Datenbank hinterlegt. Diese überführten Daten können vom Visualizer zur Visualisierung genutzt werden, um die ermittelte Kommunikation der Aktoren darzustellen.

3.2 Tracing

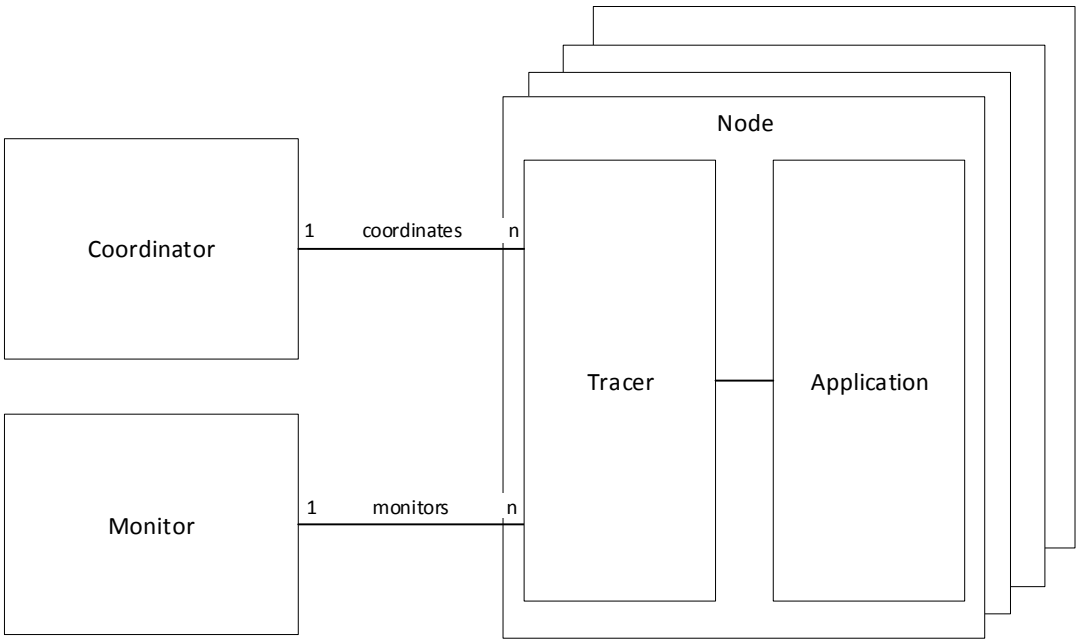


Abbildung 3.2: Architektur des Tracers

Das Ziel des Tracings ist es die Kommunikation zwischen den Aktoren zu ermitteln. Das Tracingsystem umfasst drei Module: den Monitor, den Coordinator sowie den Tracer auf dem Erlangknoten (Abb. 3.2). Der Monitor hat vor allem die Aufgabe, die Daten, die der Tracer aufzeichnet entgegenzunehmen und abzuspeichern. Zudem wird das Tracing der Anwendungen auf den verschiedenen Erlangknoten von ihm aus initialisiert und auch beendet. Das Tracing der verschiedenen Aktoren erfolgt durch Manipulation des Bytecodes. Dies ermöglicht es die Nachrichten, die durch die Aktoren versendet werden, mit zusätzlichen Informationen anzureichern. Diese können später genutzt werden, um die Kommunikation zwischen den Aktoren zu ermitteln. Zudem ermöglicht es die Manipulation gezielt Code einzuschleusen, der Tracingdaten sowie zusätzliche Aktormetriken aufzeichnet. Der Coordinator verwaltet die

verschiedenen Tracer. Dieser führt dazu intern eine Liste über alle beobachteten Knoten und koordiniert das Hinzufügen sowie das Entfernen von verschiedenen Knoten. Dadurch kann sichergestellt werden, dass Nachrichten nur angereichert werden, wenn diese an beobachtete Knoten versendet werden. Dazu benachrichtigt dieser die verschiedenen Tracer über die Veränderungen. Außerdem sichert der Coordinator ab, dass sich ein Tracer erst beendet, wenn das ganze System informiert ist. Das stellt sicher, dass keine angereicherten Nachrichten an einen Knoten gesendet werden auf dem sich der Tracer bereits beendet hat.

3.2.1 Architektur

Wie bereits in Abschnitt 3.2 zu sehen war, besteht das Tracing aus drei zentralen Komponenten: dem Monitor, dem Coordinator und dem Tracer auf dem beobachteten Knoten. Der Monitor startet und beendet alle Tracingvorgänge. Der Coordinator hat die Aufgabe alle Knoten zu synchronisieren, wenn zusätzliche Knoten getraced bzw. wenn Knoten aus dem Tracing entfernt werden. Dazu melden sich neue bzw. zu entfernende Knoten bei dem Coordinator und dieser reicht die Informationen an alle anderen Knoten weiter. Der Tracer ist für die Aufzeichnung der Aktoren auf dem Knoten zuständig. Dieser übernimmt die Manipulation des Bytecodes und reichert die Nachrichten mit Tracinginformationen an bzw. ermittelt alle benötigten Metriken. Dazu werden die ermittelten Daten zunächst lokal persistiert. Nach Abschluss des Tracings werden diese Daten an den Monitor übermittelt, welcher die verschiedenen Informationen der verschiedenen Knoten zusammenführt und bereits vorverarbeitet, damit der Preprocessor die Informationen verarbeiten kann.

Der Monitor

Der Monitor startet und beendet das Tracing. Er ist darüber hinaus auch für die Konfiguration der Aufzeichnung verantwortlich. D.h. er teilt während des Starts allen Tracern mit, wie die Aufzeichnung stattfinden soll, welche Metriken erfasst werden sollen und wie die Adresse des Coordinators ist. Nach dem Ende der Aufzeichnung liest der Monitor zudem die lokalen Aufzeichnungen der einzelnen Knoten ein und überführt diese in ein für den Preprocessor lesbares Format.

Der Coordinator

Der Coordinator hat die Aufgabe, die verschiedenen Tracer miteinander zu synchronisieren. Deshalb melden sich alle Tracer bei dem Coordinator an und auch wieder ab, wenn sie mit der Aufzeichnung beginnen bzw. diese wieder beenden. Diese Ereignisse werden wiederum

allen vorhandenen Tracern mitgeteilt. Hierbei geht es darum, dass alle Tracer über die Knoten informiert sind auf denen Aufzeichnungen stattfinden. Das soll sicherstellen, dass Nachrichten an diese Knoten mit Tracinginformationen angereichert werden sobald die Aufzeichnung beginnt und wiederum keine angereicherten Nachrichten mehr versendet werden sobald die Aufzeichnung beendet ist

Der Tracer

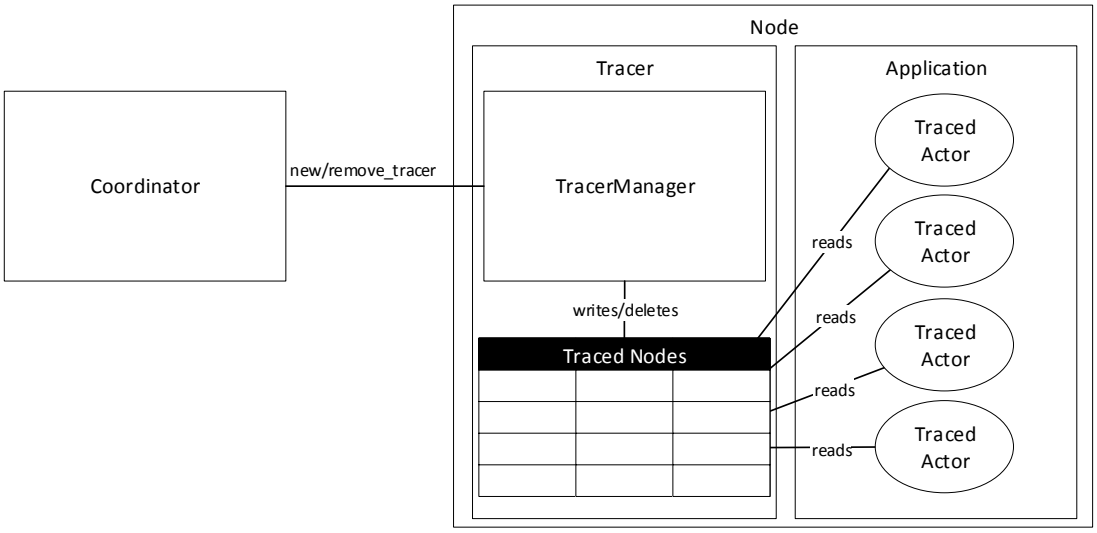


Abbildung 3.3: Verwaltung und Zugriff auf den Shared Memory zur Verwaltung getraceter Knoten

Der Tracer ist die Komponente, die die Verwaltung des Tracings auf dem Knoten übernimmt. Dazu führt dieser die Manipulation des Bytecodes durch und schleust so gezielt Code zum Tracing ein. Dies ermöglicht es, getracete Nachrichten mit zusätzlichen Informationen anzureichern, die benötigt werden, um die Kommunikation der Aktoren zu rekonstruieren. Zusätzlich werden empfangene Nachrichten, die mit Zusatzinformationen angereichert wurden wieder entpackt, damit die vorliegende Programmlogik weiterhin die Nachrichten verarbeiten kann. Hierbei ist es wichtig, dass nur Nachrichten angereichert werden, die an Knoten gesendet werden, bei denen das Tracing aktiviert ist. Denn nur diese sind in der Lage, die verpackten Nachrichten zu erkennen und entsprechend wieder zu entpacken. Deshalb hält jeder Tracer die beobachteten Knoten in einem lokalen Shared Memory fest. Wenn der Tracer über Veränderungen durch den Coordinator informiert wird, werden entsprechend Knoten in diesen Speicher hinzugefügt bzw. aus diesem wieder entfernt. Alle Aktoren in denen Nachrichten

angereichert und entpackt werden, haben Zugriff auf diesen Speicher und können so entscheiden bei welchen Nachrichten ein Anreichern stattfinden muss. Der Shared Memory Ansatz ermöglicht einen effizienten Zugriff auf die Daten. Verwaltet wird dieser Speicher über den Tracing Manager (siehe Abb. 3.3). Dieser ist die einzige Instanz, die den Inhalt des Speichers manipuliert. Alle anderen Aktoren haben nur das Recht aus diesem Speicher zu lesen.

3.2.2 Hinzufügen eines Knotens

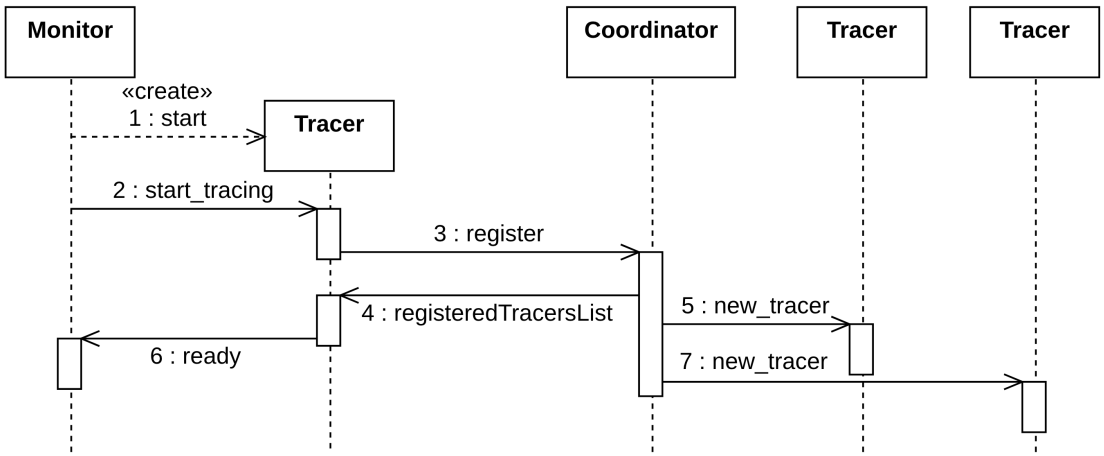


Abbildung 3.4: Hinzufügen eines neuen Knotens

Beim Hinzufügen eines neuen Knotens (Abb. 3.4) geht es vor allem darum, dass alle anderen Knoten darüber informiert werden, dass es einen weiteren Knoten gibt, an denen sie angereicherte Nachrichten senden sollen, damit auch diese getraced werden können. Das Hinzufügen eines neuen Knotens wird immer durch den Monitor initialisiert. Dieser startet den Tracer auf dem Knoten der beobachtet werden soll (1). Nachdem der Monitor dem Tracer mitgeteilt hat, dass dieser das Tracing starten soll (2), ist dieser direkt in der Lage angereicherte Nachrichten zu empfangen und zu verarbeiten. Dies stellt sicher, dass Nachrichten sofort verarbeitet werden können, sobald andere Knoten über den neuen Knoten informiert wurden. Diese müssen somit nicht auf eine Bestätigung warten, dass der Knoten bereit ist, sondern können direkt mit dem Senden von angereicherten Nachrichten beginnen. Zum Start des Tracings registriert sich der Tracer beim Coordinator (3), welcher wiederum die Tracer auf den anderen Knoten über den neuen Knoten informiert (5 & 7). Durch das Registrieren beim Coordinator, wird dieser dort in eine Liste aller vorhandenen Tracer aufgenommen, die informiert werden müssen, wenn es Veränderungen gibt. Außerdem übermittelt der Coordinator dem neuen Tracer alle aktuell registrierten Tracer (4), sodass dieser weiß an welche Knoten Nachrichten angereichert werden

müssen. Zum Abschluss informiert der Tracer den Monitor, dass der Prozess abgeschlossen ist (5).

3.2.3 Entfernen eines Knotens

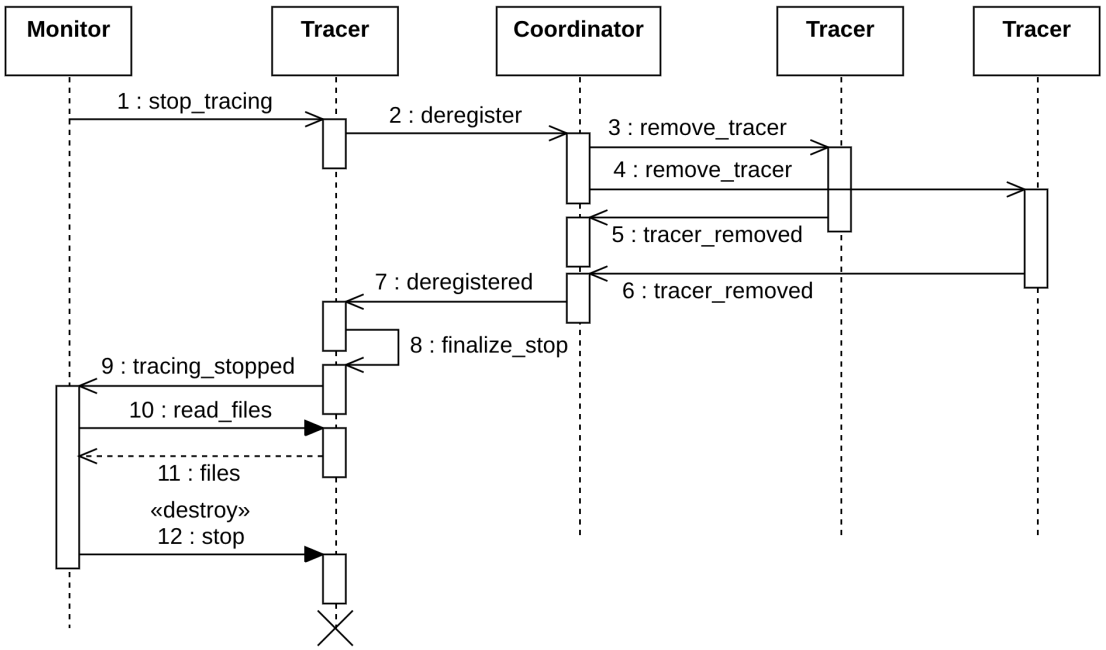


Abbildung 3.5: Entfernen eines Knotens

Das Entfernen von Knoten ist deutlich aufwendiger als das Hinzufügen eines neuen Knotens (Abb. 3.5). Denn hier muss sichergestellt werden, dass der Knoten zunächst von allen anderen Knoten entfernt wird, bevor sich der Tracer beendet. Das hängt damit zusammen, dass andere Knoten potentiell noch angereicherte Nachrichten senden könnten. Diese können jedoch nicht mehr korrekt entpackt werden, wenn der Tracer bereits beendet wurde. Zum Beenden sendet deshalb der Monitor zunächst eine Nachricht an den Tracer, der beendet werden soll (1). Der Knoten sendet anschließend eine Nachricht an den Coordinator, dass er aus dem System entfernt werden möchte (2). Bevor sich der Tracer nun beendet, wartet dieser zunächst auf eine Bestätigung, dass der Knoten des Tracers von allen anderen Knoten entfernt wurde. Damit dies geschieht, sendet der Coordinator eine Nachricht an alle vorhandenen Knoten (3 & 4). Jeder Knoten sendet anschließend eine Bestätigung, sobald der Knoten entfernt wurde (5 & 6). Hat der Coordinator von allen Knoten eine Bestätigung erhalten, informiert dieser wiederum den zu entfernenden Knoten, dass der Prozess beendet ist (7). Zwar senden ab diesem Zeitpunkt

keine weiteren Knoten mehr angereicherte Nachrichten an den Knoten, dennoch kann es sein, dass sich bereits ältere Nachrichten für den Knoten, innerhalb des Netzwerks, in der Zustellung befinden. Um das Risiko dieser Race Condition zu minimieren, läuft ein Timeout, nach dessen Ende das Tracing erst komplett eingestellt wird[1]. Ist die Zeit abgelaufen, werden mit großer Wahrscheinlichkeit keine weiteren Nachrichten eintreffen, die angereichert sind. Dadurch kann das Tracing nun sicher eingestellt werden. Deshalb wird in diesem Moment das Tracing beendet und auch keine weiteren Nachrichten angereichert (8). Anschließend wird der Monitor darüber informiert, dass das Tracing beendet wurde (9). Nachdem dieser informiert wurde bezieht der Monitor alle Traceinformationen, die durch den Tracer gesammelt wurden (10 & 11). Ist die Übertragung der Informationen abgeschlossen so wird der Tracer endgültig beendet (12).

3.2.4 Ermitteln von Tracing- und Aktormetriken

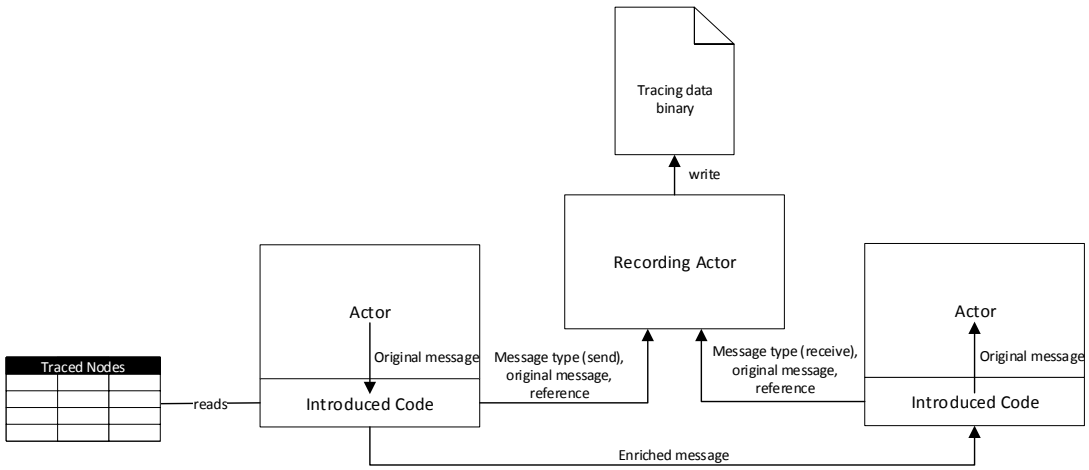


Abbildung 3.6: Kommunikation von Aktoren nachdem der Code durch ErlViz manipuliert wurde

Ziel beim Ermitteln der Tracing- und Aktormetriken war es vor allem, dass dies ohne weiteres Zutun eines Entwicklers möglich ist. Denn dies ist Fehleranfällig und ein vom Entwickler unabhängiges Tracing bereitet mehr Flexibilität für ein späteres spontanes Tracing [SBB+10]. Deshalb wird zum Tracing der Bytecode des Programms manipuliert und so gezielt spezieller Code zum Tracing eingeschleust. Auf diese Weise werden die Nachrichten zwischen den Aktoren mit speziellen Informationen angereichert, mit denen die Kommunikation zwischen den

[1] Ein ausgereifteres Verfahren wäre es, die Tracinginformationen der anderen Knoten auszulesen und zu warten bis die letzte Nachricht an den Knoten angekommen ist. Das würde jedoch den Umfang dieser Arbeit übersteigen.

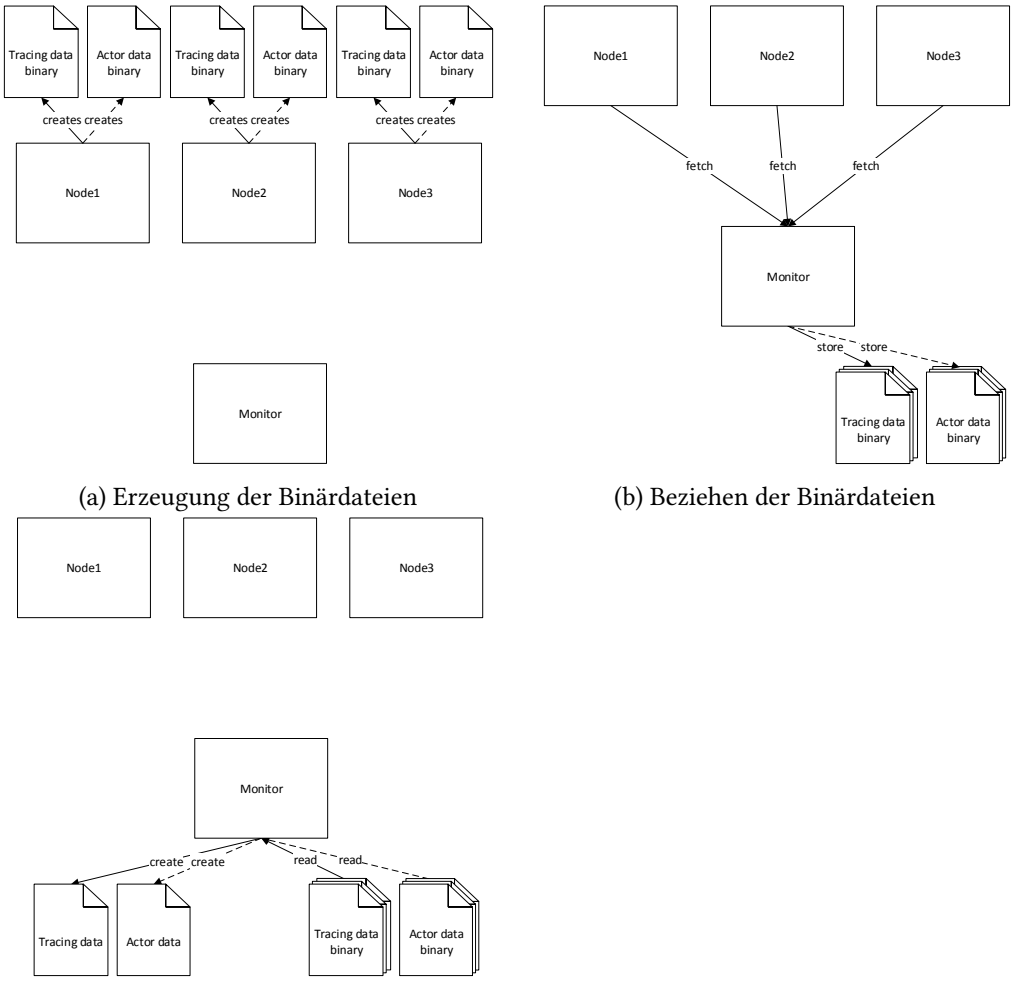
Aktoren rekonstruiert werden kann. Eine Möglichkeit die Nachrichten anzureichern ist die direkte Manipulation des Nachrichteninhalts (Abb. 3.6). Dieser wurde auch bei der Umsetzung von ErlViz gewählt (siehe Abschnitt 4). Dies hat zur Folge, dass diese Art von Nachrichten beim Empfangen erkannt und wieder in die ursprüngliche Form umgewandelt werden müssen, damit der Prozess Transparent für das beobachtete Programm bleibt.

Beim Anreichern einer Nachricht wird diese mit einer speziellen Referenz versehen, die beim Empfangen wieder ausgelesen werden kann. Diese Referenz wird nun sowohl beim Versenden als auch beim Empfangen der Nachricht aufgezeichnet. Durch das Zusammenführen der Aufzeichnungen, der verschiedenen Knoten, kann nach dem Tracing eine Zuordnung von der versendeten zur empfangenen Nachricht stattfinden. Um den Programmfluss möglichst wenig zu stören, findet die Aufzeichnung der Nachrichten in einem separatem Aktor statt, der durch das Versenden bzw. Empfangen von Nachrichten getriggert wird. Das Nutzen einer einfachen Referenz ist bei allen Systemen möglich, bei denen gilt, dass die Reihenfolge von Nachrichten zwischen den Aktoren garantiert ist. Denn nur so kann sichergestellt werden, dass die Nachrichten in der selben Reihenfolge aufgezeichnet werden, wie sie versendet bzw. eingetroffen sind. Eine alternative Technik, die angewandt werden kann, wenn diese Regel nicht gilt, die Aufzeichnung jedoch in einem separaten Aktor stattfinden soll, wäre der Einsatz von Lamportuhren [Lam78] mit denen die Reihenfolge der Nachrichten auf Aktorebene wieder hergestellt werden kann. Dazu müsste die Nachricht zwischen den Aktoren zusätzlich mit dem aktuellen Wert der Lamportuhr des sendenden Aktors angereichert werden, damit der empfangene Aktor basierend darauf seine eigene Uhr korrekt inkrementieren kann. Neben dem Versenden und dem Empfangen von Nachrichten wird auch aufgezeichnet, wenn ein Aktor gestartet oder terminiert wird. Neben dem eigentlich gestarteten oder terminierten Aktor wird hierbei auch aufgezeichnet, durch wen diese Aktionen jeweils ausgelöst werden. Somit existiert auch in diesen Anwendungsbereichen so etwas wie ein Sender und ein Empfänger.

Zusätzlich zum Tracing der Kommunikation können auch spezielle Aktormetriken abgegriffen werden. So kann der Zustand des Aktors oder die Länge des Posteingangs eines Aktors aufgezeichnet werden. Diese Aktormetriken werden zeitgleich wie die Nachrichten der Aktoren aufgezeichnet. So kann die Referenz und die Information um welche Art von Nachricht es sich handelt genutzt werden, um eine direkte Relation zwischen den Aktormetriken und den aufgezeichneten Nachrichten herzustellen. Zudem kann zur Aufzeichnung von zusätzlichen Aktormetriken Sampling [SBB+10, LGW+08, CMF+14] angewendet werden, um dass System

zu entlasten. Sollte die Anzahl der Nachrichten hoch sein, so kann bereits mit einer geringen Samplingrate ein hoher Erkenntnisgewinn erzielt werden [CMF+14].

3.2.5 Persistierung der Aufzeichnungen



(a) Erzeugung der Binärdateien

(b) Beziehen der Binärdateien

(c) Überführen in ein plattformunabhängiges Format

Abbildung 3.7: Erstellung der verschiedenen Tracingdateien

Alle Aufzeichnungen die ErlViz tätigt, werden zunächst direkt auf den einzelnen Knoten persistiert (Abb. 3.7a). Dadurch wird das System nicht zusätzlich durch Nachrichten belastet, die an den Monitor übermittelt werden müssen. Stattdessen werden diese Dateien nach der Aufzeichnung an den Monitor übertragen (Abb. 3.7b), um dort weiterverarbeitet zu werden.

Damit das laufende Programm möglichst gering durch die Aufzeichnung unterbrochen wird, ist ein separater Aktor, der vom Tracingsystem zur Verfügung gestellt wird, für das Überführen in die persistente Datei zuständig. Hierbei handelt es sich zunächst um Binärdateien, um den Overhead möglichst gering zu halten. Wobei das Aktortracing und die Aktormetriken in zwei separaten Dateien abgelegt werden.

Nachdem die Dateien an den Monitor übertragen wurden, tätigt dieser eine erste Verarbeitung der Dateien. So werden diese zunächst in ein plattformunabhängiges Format überführt (Abb. 3.7c). Dadurch kann der Preprocessor ohne größeren Aufwand auf Basis einer anderen Technologie entwickelt werden, als die beobachtete Plattform.

3.3 Preprocessing

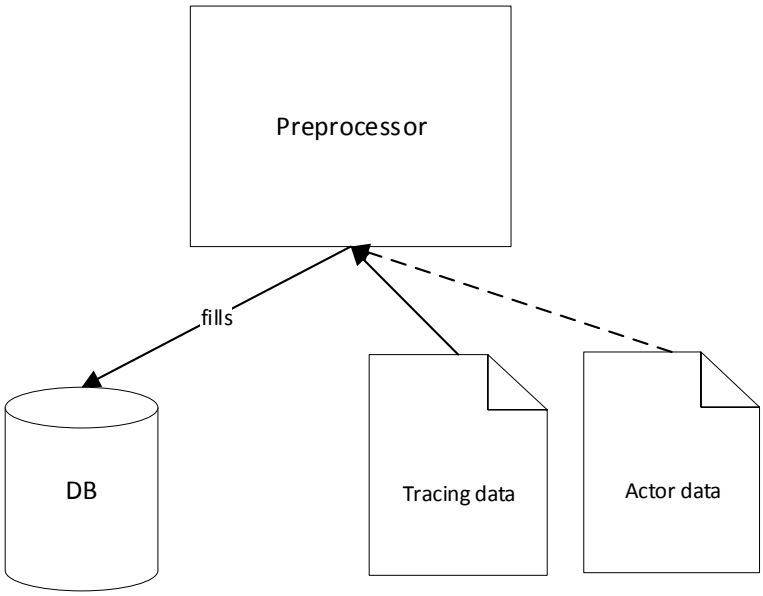


Abbildung 3.8: Einlesen der ermittelten Daten

Das Preprocessing hat die Aufgabe, die gesammelten Daten so aufzubereiten, als dass effizient auf diese Daten zugegriffen werden kann, damit diese bei der Visualisierung genutzt werden können. Hierbei werden die Daten aus der Tracingdatei mit den gesammelten Aktormetriken zusammengefasst und in eine Datenbank abgelegt, die von der Visualisierung genutzt werden kann. Zusätzlich werden gezielt Metadaten aus den gesammelten Informationen heraus

extrahiert. Hierbei handelt es sich um Informationen ,wie der Name eines Aktors oder dessen Implementierung.

Vorgang

Um die Aktormetriken mit den Tracingdaten zusammenzuführen und die Metadaten aus den ermittelten Daten heraus zu extrahieren, werden bestimmte Strategien angewendet. Diese sollen in diesem Abschnitt vorgestellt werden. Zur einfacheren Kommunikation werden im Folgenden alle Aktoren als Sender bezeichnet, die eine Nachricht absenden, einen Aktor starten oder einen beenden. Alle Aktoren, die Nachrichten empfangen, gestartet oder beendet werden, werden als Empfänger Bezeichnet.

Damit die Aktormetriken mit den Tracingdaten zusammengeführt werden können, wird ausgenutzt, dass sämtliche gesammelte Aktormetriken eindeutig über die Art der aufgezeichneten Information und der Referenz einer Aufzeichnung der Aktorkommunikation zugeordnet werden kann. Deshalb werden zunächst sämtliche gesammelte Aktormetriken in einer temporären Tabelle abgelegt und erst bei Einlesen der zugehörigen Aufzeichnung dieser direkt zugeordnet. Da die Zuordnung über die Art der aufgezeichneten Information und der Referenz stattfindet, wird ein kombinierter Index über diese Spalten genutzt, damit effizient auf die Informationen zugegriffen werden kann.

Um die Tracingtabelle aufzubauen, wird über die Information innerhalb der Tracingdatei iteriert. Sollte die Lamportuhr [Lam78] noch nicht bereits während des Sammelns der Daten erstellt worden sein, so wird diese nun nachträglich zusammengestellt. Ein nachträgliches Aufbauen der Lamportuhr ist jedoch nur möglich, wenn die beobachtete Umgebung eine Garantie bereitstellt, dass die Kommunikation zwischen Aktoren in der selben Reihenfolge eintrifft, wie sie versendet worden sind, wie es zum Beispiel in Erlang der Fall ist. Mittels der Lamportuhr kann auf diese Weise eine totale Ordnung der Traces auf Ebene der einzelnen Aktoren hergestellt werden. Für jeden Eintrag in der Tracingdatei wird ein Eintrag in der Tracingtabelle erstellt. Zusätzlich wird für jeden Eintrag in die temporäre Tabelle mit den Aktormetriken geschaut, ob dort ein zugehöriger Eintrag vorliegt. Ist dies der Fall, so wird der Eintrag in der Tracingtabelle um die entsprechenden Informationen angereichert. Zeitgleich wird beim Einlesen der Tracingdaten die Metadatentabelle für die einzelnen Aktoren aufgebaut. Diese enthält allgemeine Informationen über die Aktoren, wie deren Namen oder der Knoten, auf denen diese sich befinden.

Sollte die Lamportuhr, während des Preprocessings aufgebaut werden, so wird intern für jeden Aktor dessen Lamportuhr geführt. Zudem wird bei jeder Aufzeichnung, die eine versendende Nachricht beschreibt, festgehalten, wie die Lamportuhr des versendenden Aktors zu diesem Zeitpunkt war. Diese Information kann mittels der Referenz, mit der die Nachricht angereichert wurde, abgerufen werden. So kann auf die Uhr des Senders zugegriffen werden, wenn die empfange Nachricht eingelesen wird. Zudem kann so jede empfangene Nachricht mit dem zugehörigen Empfänger in Verbindung gebracht werden und auch diese Information in der Datenbank hinterlegt werden. Sollten bestimmte Informationen über den Empfänger beim Versenden der Nachricht nicht vorgelegen haben, so können auch diese zu diesem Zeitpunkt nachgepflegt werden. Dies könnte zum Beispiel der Fall sein, wenn der Aktor speziell Adressiert wurde, wie eventuell über einen speziellen Namen und so die genaue Adresse für den Absender unbekannt ist. In einem solchen Fall kann die genaue Adresse zu diesem Zeitpunkt nachgepflegt werden.

In Ausnahmefällen kann es vorkommen, dass das Empfangen einer Nachricht vor dem Senden einer Nachricht in der Tracingdatei vorliegt. Dies kann daran erkannt werden, dass zu einer empfangenen Nachricht keine versendete Nachricht gefunden werden kann. Sollte dies stattfinden, so wird das Ereignis festgehalten, sodass mittels der Referenz, mit der die Nachricht angereichert wurde, auf diese Daten zugegriffen werden kann. Deshalb wird bei jedem Einlesen einer versendeten Nachricht zusätzlich in diese Daten geschaut, ob bereits eine zugehörige empfangene Nachricht eingelesen wurde. Sollte das Empfangen einer Nachricht vor dessen Versenden eingelesen werden, so kann für den Empfänger die Lamportuhr nicht weiter geführt werden. Dies ist erst möglich sobald die zugehörige versendete Nachricht aufgefunden wurde. Dies gilt auch für alle Aktoren, die Nachrichten von dem Aktor erhalten, der die Nachricht empfangen hat, so wie alle von diesen Aktoren aufgerufenen Aktoren usw.. Erst sobald die versendete Nachricht aufgefunden wurde, können die Lamportuhren nachträglich erstellt werden. Sollte das Versenden der Nachricht nie aufgefunden werden, so wird die Lamportuhr vom Empfänger weitergeführt und der Absender ignoriert.

Nachdem alle Tracingdaten verarbeitet wurden, kann die temporäre Tabelle, die die gesammelten Aktormetriken speichert, wieder entfernt werden.

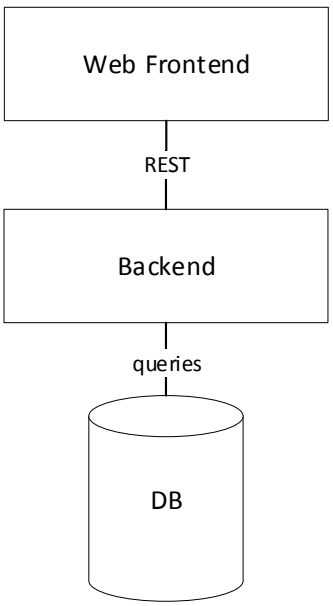


Abbildung 3.9: Architektur ErlViz-Visualizer

3.4 Visualisierung

Die Visualisierung erfolgt webbasiert. Hierbei kommt eine klassische Client Server Architektur zum Einsatz (Abb. 3.9). Das hat den Vorteil, dass die Daten backendseitig vorbereitet werden können, bevor diese frontendseitig visualisiert werden. Rechenaufwendige Operationen können so eventuell durch einen Leistungsstärkeren Server durchgeführt werden und der Client so entlastet werden.

Die Visualisierung besteht aus zwei Bereichen. Zum einen einem Bereich in dem ausgewählt wird, welche Aufzeichnung betrachtet werden soll und zum anderen, aus der konkreten Darstellung der Aufzeichnung, die untersucht wird. Im Folgenden werden die einzelnen Bereiche sowie deren Funktionalitäten näher erläutert.

3.4.1 Auswahl der Aufzeichnung

Auf der ersten sichtbaren Seite kann zunächst ausgewählt werden, welche Aufzeichnung betrachtet werden soll (Abb. 3.10). Die Auswahl der Aufzeichnung erfolgt in zwei Schritten. Zunächst wird die konkrete Aufzeichnung ausgewählt in einem zweiten Schritt kann eingeschränkt werden, was genau betrachtet werden soll. Hierbei kann es sich z.B. um einen genauen Zeitraum handeln der betrachtet werden soll. Dadurch können mehrere Dinge erreicht werden.

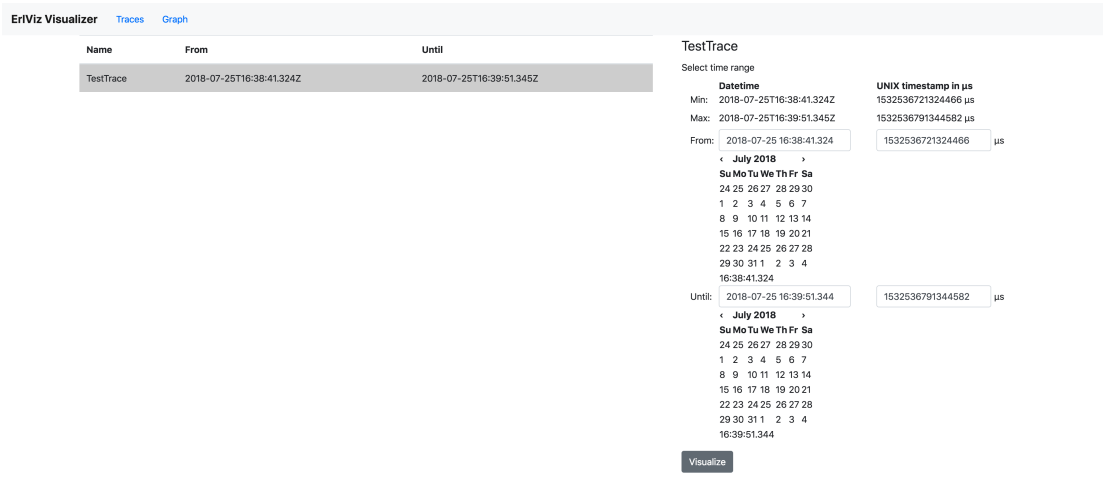


Abbildung 3.10: Übersicht der aufgezeichneten Traces

So kann zum einen das Frontend entlastet werden, weil es weniger verarbeiten muss aber es kann auch die Ladezeit verringert werden, weil weniger Daten geladen werden müssen. Aus Nutzersicht kann dies auch zu mehr Übersichtlichkeit beitragen, da weniger Daten zum Betrachten vorliegen. Insbesondere, wenn bereits sehr genau bekannt ist, welcher Zeitraum betrachtet werden soll.

3.4.2 Darstellung der Aktorkommunikation

Nachdem der Nutzer ausgewählt hat, was er betrachten möchte, gelangt dieser in die Visualisierung der Kommunikation in Form eines Graphen (Abb. 3.11). Die Oberfläche kann in drei Bereiche eingeteilt werden: die History im linken Bereich, die den Ablauf der Ereignisse sequentiell darstellt, die Konfiguration im oberen Bereich, in dem zusätzliche Funktionen zur Visualisierung zu Verfügung stehen sowie den Graphen, der die Kommunikation grafisch darstellt im Zentrum. Zum Untersuchen der Aufzeichnung stehen dem Nutzer eine Vielzahl von Funktionen zur Verfügung. Im Folgenden sollen zunächst die einzelnen Elemente der Visualsierung näher erläutert werden, anschließend werden die verschiedenen Funktionen die dem Nutzer zur Verfügung stehen dargestellt.

Elemente der Visualisierung

Damit ein besseres Verständnis für den Graphen aufgebaut werden kann, sollen im Folgenden die einzelnen Elemente der Visualisierung beschrieben werden. Dazu wird zunächst die History

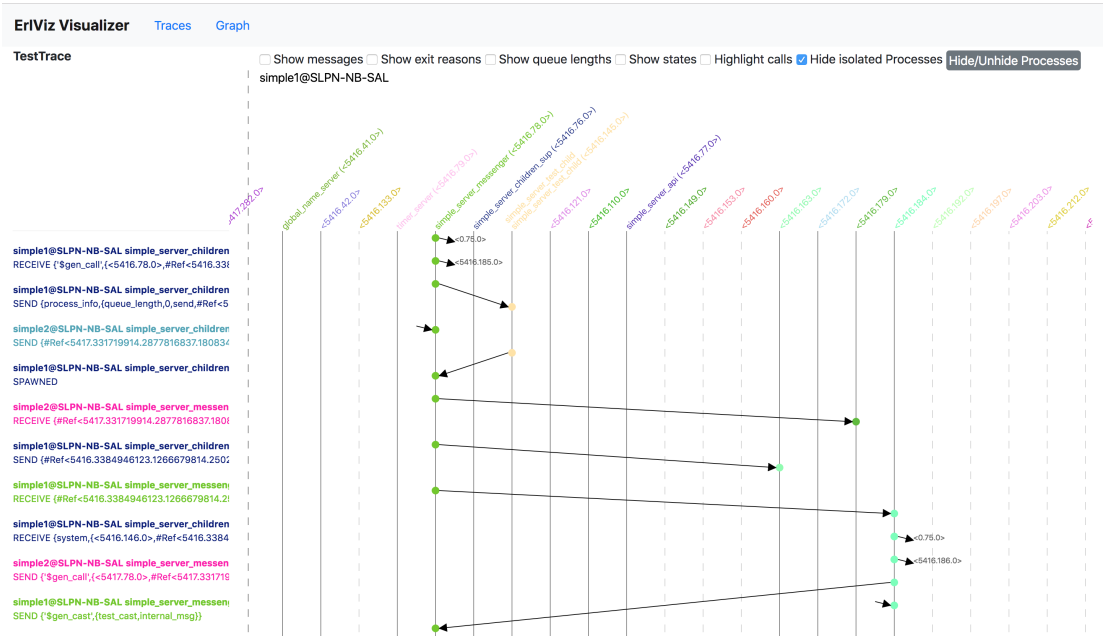


Abbildung 3.11: Ansicht des Kommunikatiosgraphen in ErlViz

und anschließend der Graph selber betrachtet. Die Konfiguration wird im anschließenden Abschnitt zusammen mit den Funktionen näher betrachtet.

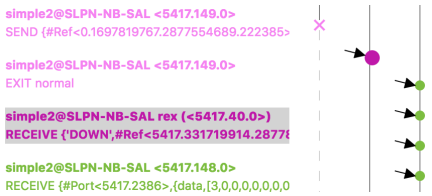


Abbildung 3.12: History mit hervorgehobenen Eintrag

History

Die History stellt die aufgezeichneten Ereignisse in einer sequentiellen Reihenfolge dar. Hierbei wird eine Kurzfassung des Ereignisses dargestellt. Diese enthält Informationen, wie den Knotennamen, den Namen des Aktors, wenn dieser bekannt ist, die Adresse des Aktors sowie die Art des Ereignisses und deren Inhalt. Zur besseren Unterscheidung werden die Einträge der einzelnen Aktoren in jeweils eigenen Farben dargestellt. So kann direkt gesehen werden zu welchem Aktor ein Eintrag gehört. Die Farbe entspricht zudem der Farbe, wie sie im Graphen für den Aktor verwendet wird. Zusätzlich wird der Ereignispunkt im Graphen hervorgehoben,

der zu einem Eintrag gehört, wenn mit der Maus über einen Eintrag gefahren wird (Abb. 3.12). Sollte der Ereignispunkt aktuell außerhalb des Sichtbereiches sein, so wird dieser automatisch in den Sichtbereich gefahren.

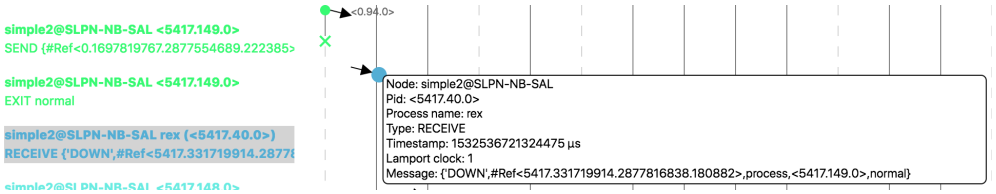


Abbildung 3.13: Hervorgehobener Eintrag mit zusätzlichen Informationen

Graph

Der Graph ist das zentrale Element zur Visualisierung der Kommunikation. Eingeteilt ist der Graph horizontal in die verschiedenen Knoten sowie den Aktoren, die sich auf den Knoten befinden. Ganz oben ist zunächst der Name des Knotens beschrieben. Darunter werden die Namen bzw. die Adressen der Aktoren gelistet, die sich auf dem Knoten befinden. Unterhalb der Bezeichnung für den Aktor befindet sich dessen Lebenslinie. Um die verschiedenen Knoten optisch voneinander zu trennen, befindet sich zwischen diesen zusätzlich eine gestrichelte Linie. Die Bezeichnung der Aktoren bleiben immer sichtbar, auch wenn der Graph bewegt wird. Dadurch wird die Orientierung im Graphen erleichtert.

Zudem hat jeder Aktor eine individuelle Farbgebung in der sowohl die Bezeichnung als auch die einzelnen Ereignisse eingefärbt sind. Diese entspricht auch den eingesetzten Farben in der History. Wird die Maus über ein Ereignis bewegt, so wird auch der zugehörige Eintrag in der History hervorgehoben. Ist der Eintrag innerhalb der History aktuell nicht sichtbar, so wird das Element automatisch in den Sichtbereich gefahren. Zusätzlich werden alle aufgezeichneten Werte in einem Tooltip angezeigt (Abb. 3.13), wenn sich die Maus über einem Ereignis befindet. Dieses enthält zusätzliche Informationen, wie die Art des Ereignisses, der Zeitpunkt an dem die Nachricht übermittelt wurde oder welche Informationen konkret übermittelt wurde. Zudem werden hier die zusätzlich gesammelten Aktormetriken, wie die Länge des Posteingangs oder der Zustand des Aktors angezeigt.

Lebenslinie

Die Lebenslinie stellt den Lebensverlauf eines Aktors dar. Um ersichtlich zu machen wann ein Aktor tatsächlich existiert hat, gibt es die Lebenslinie in zwei Ausführungen. So ist die

Linie hellgrau und gestrichelt, wenn der zugehörige Aktor zu dem Zeitpunkt nicht existiert und durchgezogen wenn der Aktor existiert. Dadurch ist jederzeit ersichtlich, wo sich die Linie eines Aktors befindet. Dies hilft bei der Orientierung im Graphen, vor allem wenn ein Aktor aktuell nicht existiert. So wird, sobald ein Aktor startet, dessen Linie Schwarz und durchgezogen (Abb. 3.14). Die Linie wird dann wieder hellgrau gestrichelt, sobald ein Aktor beendet wird (Abb. 3.15).

Ereignispunkt

Ereignispunkte beschreiben ein konkretes Ereignis, dass bei der Aufzeichnung stattgefunden hat. Diese Punkte sind immer in den Farben des Aktors zu dem sie gehören. Zudem ist es möglich zusätzliche Informationen anzeigen zu lassen, wenn mit der Maus über einen solchen Punkt gefahren wird (Abb. 3.13). Die einfachen Ereignispunkte (rund) beschreiben gesendete Nachrichten, empfangene Nachrichten oder dass ein Aktor einen anderen Aktor gestartet hat. Pfeile zwischen den Ereignissen helfen dabei den Kausalzusammenhang zwischen Ereignissen zu erkennen. So tritt beim Senden einer Nachricht und beim Starten eines Aktors der Pfeil aus dem Punkt aus und beim Empfangen einer Nachricht in den Punkt hinein. Neben den einfachen Ereignispunkten, gibt es noch zwei Zusatzpunkte, die den Lebensbeginn sowie das Lebensende eines Aktors anzeigen:

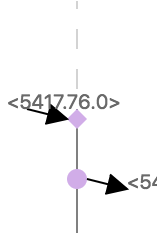


Abbildung 3.14: Beispiel für eine Lebenslinie: Hellgrau: Aktor lebt nicht; Schwarz: Aktor lebt

Spawnpunkt Zeigt an, dass ein Aktor gestartet wurde. Dies wird in Form einer Raute dargestellt (Abb. 3.14). Der eintretende Pfeil zeigt an, von welchem Aktor der Aktor gestartet wurde. Dieser beginnt immer bei dem Punkt, der den Aktor gestartet hat. Ist dieser der betrachteten Aufzeichnung nicht sichtbar, so wird die Adresse des Aktors an dem Pfeil angezeigt.

Abbildung 3.15: Lebensende eines Aktors

Terminierung Zeigt an, dass ein Aktor beendet wurde und wird in Form eines Kreuzes dargestellt (Abb. 3.15). Sollte bekannt sein, welcher Aktor den Aktor beendet hat, so wird dies auch hier über einen Pfeil dargestellt. Dieser kommt dann vom auslösenden

Aktor und führt zu dem Kreuz des beendeten Aktors. Wie beim Spawnpunkt wird auch hier die Adresse des Aktors an dem Pfeil dargestellt, wenn der auslösende Aktor in dem betrachteten Bereich nicht sichtbar ist.

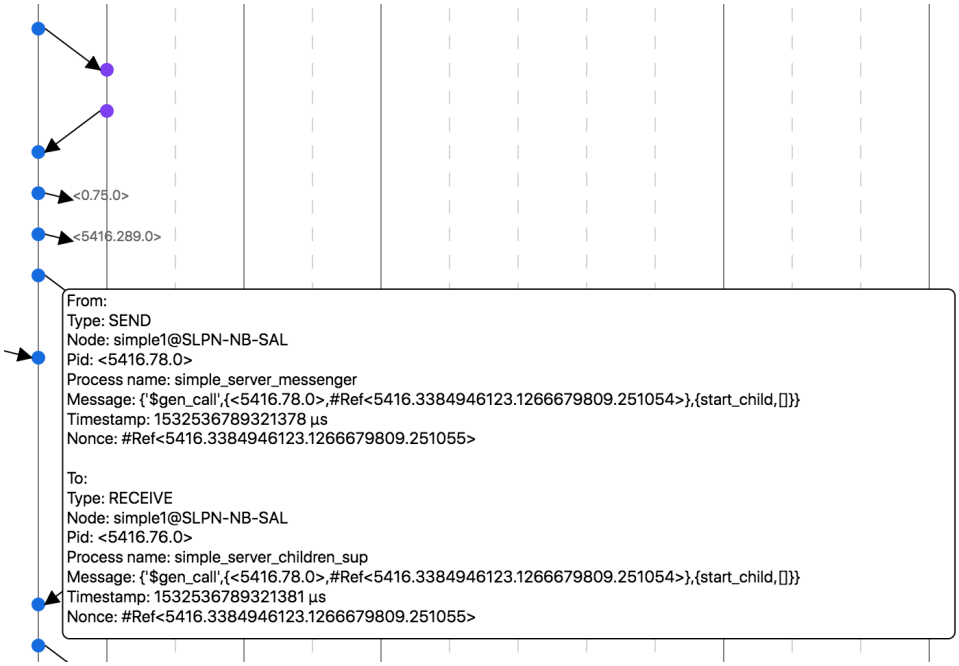


Abbildung 3.16: Verschiedene Ereignispfeile mit Tooltip

Ereignispfeil

Ereignispfeile zeigen an, wie zwischen den Aktoren während der Aufzeichnung kommuniziert wurde. Hierbei zeigt der Pfeil immer von dem Ereignis der eine Aktion ausgelöst hat (z.B. senden einer Nachricht), zu dem Ereignis, der die Aktion entgegen genommen hat (z.B. empfangen einer Nachricht). Eine direkte Verbindung von Ereignissen kann nur stattfinden, wenn beide Aktoren im betrachteten Bereich der Aufzeichnung bekannt sind. Ist einer der Teilnehmer nicht bekannt, so wird am anderen Ende die Adresse des anderen Aktors angezeigt. Sollte auch die Adresse des anderen Teilnehmers nicht bekannt sein, so wird nur der Pfeil ohne zusätzliche Informationen angezeigt. Durch zeigen der Maus auf den Pfeil, können zudem zusätzliche Informationen über die übermittelte Nachricht erhalten werden. Hierbei wird ein Tooltip eingeblendet, dass das auslösende Ereignis sowie das empfangene Ereignis anzeigt (Abb. 3.16).

Funktionalitäten

Nachdem die Elemente des Graphen beschrieben wurden, soll nun im Folgenden, die bereitgestellten Funktionen näher beschrieben werden. Die meisten Funktionen sind im oberen Bereich der Graphansicht an- und abwählbar und können generell miteinander kombiniert werden. Für alle folgenden Funktionalitäten gilt, dass sie über eine Checkbox im Konfigurationsbereich sowohl aktiviert als auch deaktiviert werden können, insofern es nicht anders angegeben wurde.

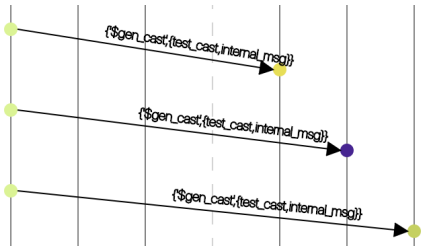


Abbildung 3.17: Anzeige der Nachrichten verschiedener Ereignispfeile

Nachrichten anzeigen Reichert die Ereignispfeile mit der versendeten Nachricht an (Abb. 3.17).

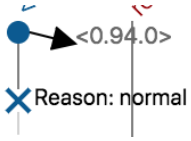


Abbildung 3.18: Anzeige der Begründung, warum ein Aktor beendet wurde

Terminierungsgründe anzeigen Reichert die Ereignispunkte für die Terminierung von Aktoren mit der Information an, warum dieser beendet wurde (Abb. 3.18), insofern dieser bekannt ist.

Ermittelte Aktormetriken anzeigen Auch die zusätzlich ermittelten Aktormetriken können angezeigt werden. Diese werden an den Ereignispunkten angezeigt, bei denen diese Werte ermittelt wurden. Da zum Sammeln der Daten mit Sampling gearbeitet werden kann, kann es vorkommen, dass nicht zu allen Ereignissen diese Informationen gesammelt wurden. An Ereignispunkten bei denen kein Wert ermittelt wurde, wird deshalb nichts angezeigt. Für jedes ermittelbare Datum gibt es eine Checkbox, mit der diese jeweils angezeigt werden kann.

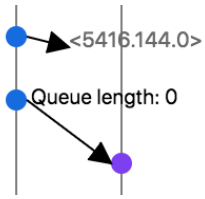


Abbildung 3.19: Länge des Posteingangs, der zu einem Ereignis ermittelt wurde

Länge Posteingang Zeigt an, wie lang die Warteschlange im Posteingang von einem Aktor zum Zeitpunkt des Ereignisses ist (Abb. 3.19).

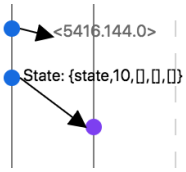


Abbildung 3.20: Zustand eines Aktors, der zu einem Ereignis ermittelt wurde

Zustand des Aktors Zeigt den Zustand des Aktors an, der während des Ereignisses ermittelt wurde (Abb. 3.20).

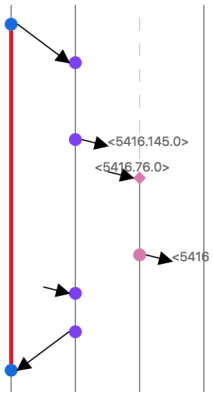
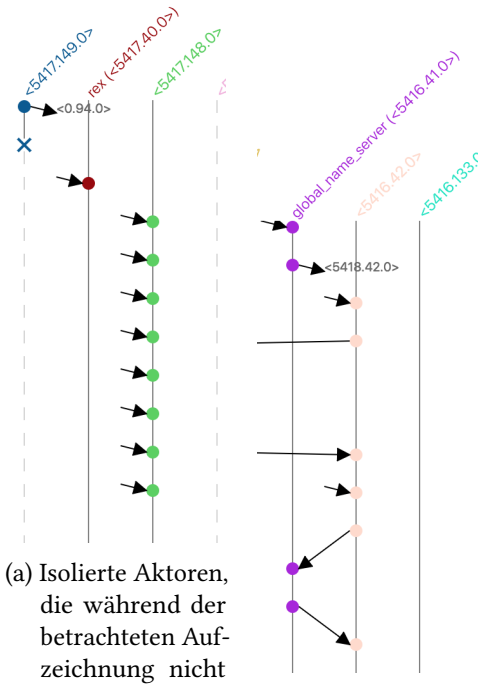


Abbildung 3.21: Hervorheben von einem blockierenden Aufruf

Blockierende Nachrichten hervorheben Hebt den Bereich hervor, während ein Aktor auf die Antwort eines anderen Aktors wartet, weil dieser einen blockierenden Aufruf getätigt hat (Abb. 3.21). Während dieser Zeit können nämlich keine weiteren Nachrichten empfangen werden. So ist es möglich zu ermitteln, ob dieses Pattern übermäßig benutzt wurde und so zu

einem Performanceproblem wird, da sich Nachrichten aufstauen. Insbesondere in Kombination mit der Anzeige der Länge des Posteingangs lässt sich so ein Verhalten gut ermitteln.



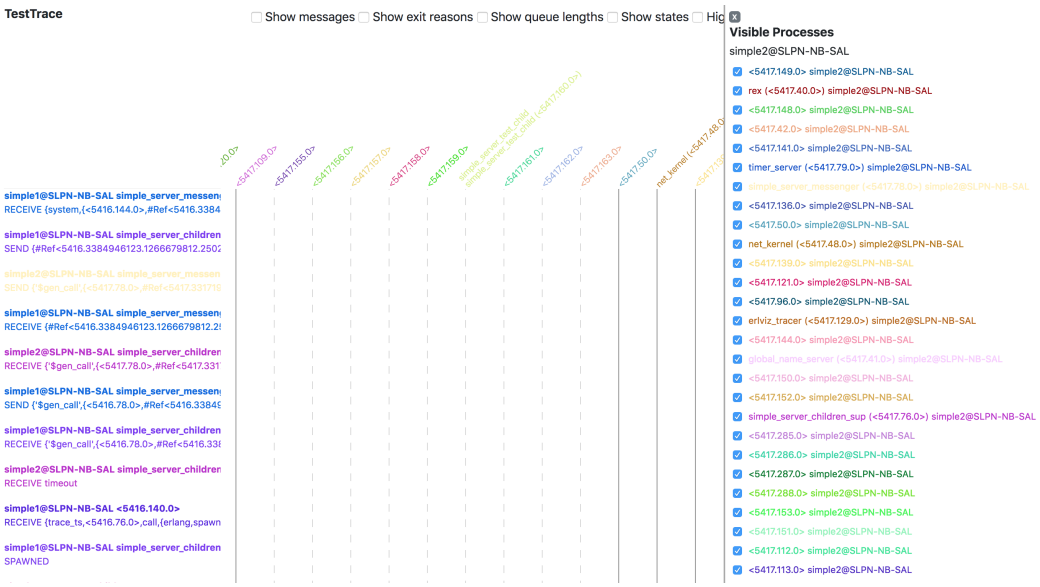
(a) Isolierte Aktoren, die während der betrachteten Aufzeichnung nicht mit anderen aufgezeichneten Aktoren interagieren

(b) Beispiel ohne isolierte Aktoren

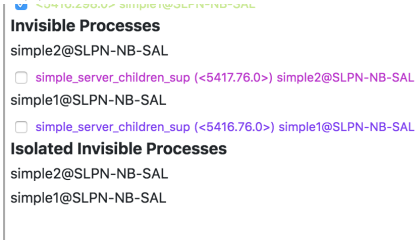
Abbildung 3.22: Ausblenden von isolierten Aktoren: a) vor dem Ausblenden, b) nach dem Ausblenden

Isolierte Aktoren ausblenden Blendet alle Aktoren aus, die während des betrachteten Aufzeichnungszeitraums, nicht mit anderen Aktoren, die aufgezeichnet wurden, interagiert haben. Durch das Ausblenden dieser isolierten Aktoren, entsteht eine bessere Übersichtlichkeit, denn die nicht vorhandene Interaktion ist ein Indikator dafür, dass diese Aktoren für den beobachteten Programmverlauf keine Rolle spielen. So ist es möglich sich auf die vorhandene Kommunikation der Aktoren zu konzentrieren (Abb. 3.22).

Aktoren ein-/ausblenden Das Ausblenden isolierter Aktoren entfernt bereits alle Aktoren aus dem Sichtbereich, die nicht mit den anderen Aktoren interagieren. Es gibt allerdings auch



(a) Übersicht der sichtbaren Aktoren

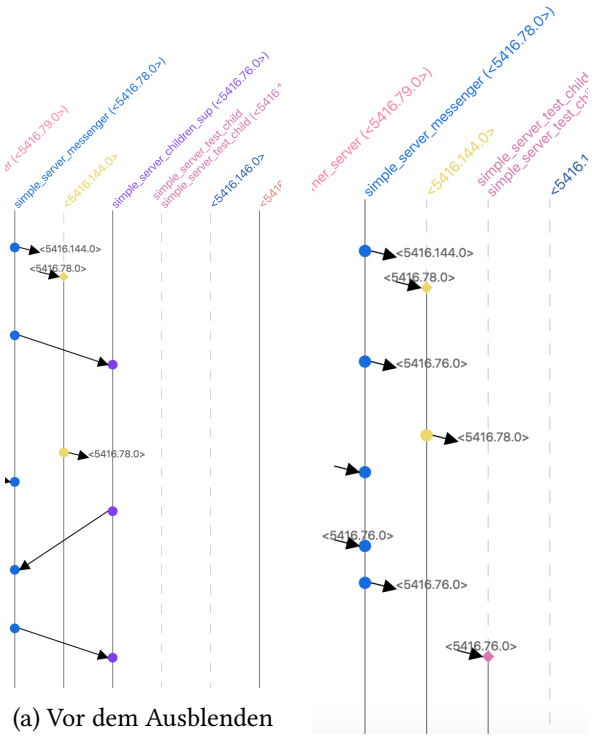


(b) Übersicht der nicht sichtbaren Aktoren: Oben: Manuell ausgeblendete Aktoren; Unten: Isolierte Aktoren die nicht sichtbar sind.

Abbildung 3.23: Ansicht zum ein- und ausblenden einzelner Aktoren.

Situationen, in denen die Beobachtung noch stärker verfeinert werden soll. Deshalb gibt es die Möglichkeit, einzelne Aktoren gezielt ein- und ausblenden zu können. In die Ansicht mit der die An- und Abwahl einzelner Aktoren möglich ist, kann über einen Button oben rechts gelangt werden. Diese Ansicht ist in drei Kategorien unterteilt (Abb. 3.23):

- sichtbare Aktoren
- nicht sichtbare Aktoren
- nicht sichtbare Aktoren die isoliert sind



(a) Vor dem Ausblenden

(b) Nach dem Ausblenden

Abbildung 3.24: Aktor, der mit einem anderen Aktor kommuniziert, vor und nach dem Ausblenden (*simple_server_children_sup (<5416.76.0>)*)

Zudem sind alle Aktoren innerhalb der einzelnen Kategorien nach den Knoten sortiert, auf denen sie sich befinden. Auch in dieser Seitenleiste zieht sich die Farbgebung der Aktoren fort. Jeder einzelne angezeigte Aktor hat dieselbe Farbe, wie innerhalb des Graphen. Wie zu sehen ist, sind die nicht sichtbaren Aktoren in zwei Kategorien eingeteilt. So gibt es eine zusätzliche Kategorie für nicht sichtbare Aktoren, die isoliert sind. Dadurch wird eine größere Übersicht erzielt, wenn zusätzlich isolierte Aktoren ausgeblendet wurden. Denn so gibt es eine Kategorie, in der die vom Nutzer ausgeblendeten Aktoren separat sichtbar sind. Diese können somit vom Nutzer leichter wiedergefunden werden. Da auch die nicht sichtbaren isolierten Aktoren gelistet sind, können diese auch wieder sichtbar gemacht werden. Sollte dies geschehen, so wird automatisch die Auswahl, dass isolierte Aktoren ausgeblendet sind, entfernt.

Beim Öffnen der Seitenleiste sind alle Aktoren zunächst angewählt. Um die Sichtbarkeit eines Aktors zu verändern, muss der Hacken bei diesem entfernt werden. Durch entfernen des

Hackens ändert sich die Sichtbarkeit des Aktors. D.h. wenn dieser zuvor sichtbar war, wird der Aktor unsichtbar und wenn dieser zuvor unsichtbar war, wird der Aktor sichtbar. Nachdem die Seitenleiste geschlossen wurde, werden die angestrebten Änderungen durchgeführt.

Das Ausblenden eines Aktor kann dazu führen, dass ein Aktor ausgeblendet wird, der über einen Ereignispfeil mit einem anderen Aktor verbunden ist. Ist dies der Fall, so kann der Ereignispfeil in dieser Form nicht mehr dargestellt werden. Deshalb wird der ausgeblendete Aktor bei der Darstellung behandelt, wie ein nicht aufgezeichneter Aktor. So wird ein Pfeil erzeugt, der mit keinem Punkt verbunden ist und am anderen Ende mit der Adresse des Aktors versehen wird (Abb. 3.24).

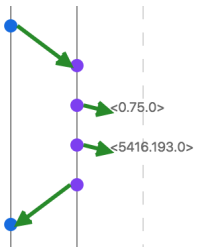


Abbildung 3.25: Anzeige eines Aufrufpfades einer Nachricht

Aufrufpfade anzeigen Durch anklicken eines Ereignispfeils ist es möglich zurück zu verfolgen, welche Folgen der jeweilige Aufruf hatte. So werden durch die Aktion der Aufruf sowie alle Folgeaufrufe grün eingefärbt (Abb. 3.25). Dies betrifft nicht nur die Folgeaufrufe auf den direkt aufgerufenen Aktor, sondern alle Aufrufe auf allen folgenden Aktoren. Die Einfärbung der Pfeile hört an dem Punkt auf, an dem wieder ein Aufruf des ursprünglich auslösenden Aktors stattfindet oder wenn ein aufgerufener Aktor zwischendurch durch einen anderen Aktor aufgerufen wird. Denn dann sind alle weiteren Aufrufe nicht mehr die Folge des ursprünglichen Aufrufes, sondern des folgenden Aufrufes.

Wird ein Pfeil angeklickt dessen Quelle nicht sichtbar ist, so ist entscheidend, ob die Quelle bekannt ist. Ist sie bekannt, so ist auch hier der Aufruf des initiierenden Aktors die Bedingung um die Einfärbung zu beenden. Ist die Quelle nicht bekannt, so übernimmt der zuerst aufgerufene Aktor, die Rolle des Aktors, bei dessen Aufruf die Einfärbung beendet wird.

3.5 Zusammenfassung

Es wurde ein System vorgestellt mit dem es möglich ist die Kommunikation von Aktoren untereinander aufzuzeichnen. Dies wird erreicht, in dem der Bytecode innerhalb der laufenden Anwendung manipuliert und so die Nachrichten mit zusätzlichen Informationen angereichert wird. Zudem ist es möglich, weitere Informationen über die Aktoren, wie die Länge der Warteschlange im Posteingang oder den Zustand der Aktoren, zu ermitteln. Alle aufgezeichneten Informationen werden von einem weiteren System verarbeitet und dauerhaft in einer Datenbank persistiert. Dadurch ist es möglich auf die Daten über ein Serversystem zuzugreifen. Dies ermöglicht es, dass auch größere Datenmengen verarbeitet werden können, da diese vom Backend vorverarbeitet werden können. Die aufgezeichneten Daten werden genutzt, um die Kommunikation der Aktoren zu visualisieren. Die Visualisierung ist hierbei stark angelehnt an ShiViz [BWBE16]. Der Kernunterschied zwischen den beiden Systemen besteht in der Backendseitigen Aufbereitung der Daten. Während ShiViz die Daten komplett Frontendseitig verarbeitet, können sie in ErlViz dauerhaft persistiert und durch ein Backend verarbeitet werden. Dadurch ist dieses System auch für größere Datenmengen geeignet, wie sie bei der Kommunikation von Aktoren entstehen.

4 ErlViz Experimentierplattform

Neben der konzeptionellen Entwicklung wurde ErlViz auch prototypisch umgesetzt, damit das Konzept auch validiert werden kann. Hierbei soll das Verhalten des Load Balancers [All17b] beobachtet werden, um die Fragestellung aus Abschnitt 2.1 zu klären. Deshalb wurde eine Lösung implementiert, die darauf ausgelegt ist Aktoren zu tracken, die mittels Erlang/OTP [Erlc] implementiert wurden. Diese wurde in eine simulierte Timadorusumgebung integriert, um dort getestet zu werden.

Im Folgenden wird zunächst die konkrete Umsetzung von ErlViz vorgestellt. Da die Implementierung in Timadorus integriert wurde, werden anschließend Timadorus und die genutzten Teilsysteme vorgestellt. Danach wird die konkrete Integration von ErlViz in Timadorus und der Testumgebung beschrieben. Zum Abschluss gibt es ein kurzes Fazit über die geschaffene Umgebung.

4.1 Umsetzung von ErlViz

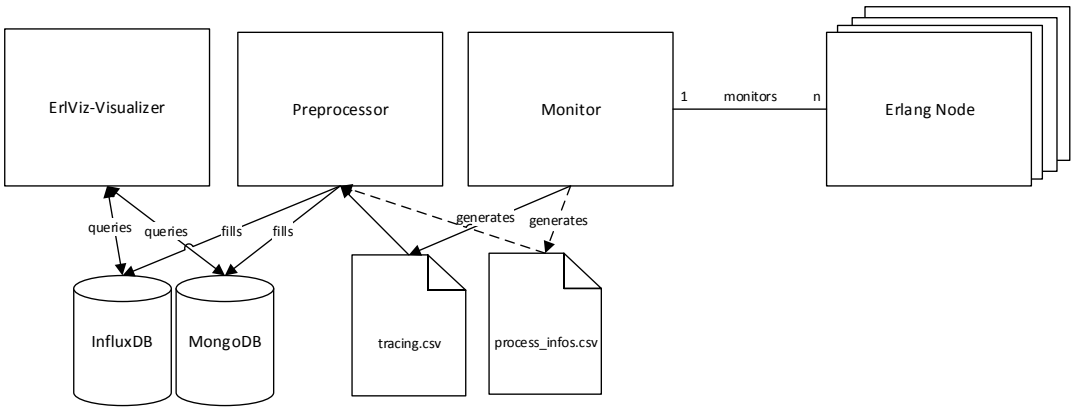


Abbildung 4.1: Überblick über das ErlViz System

Die Umsetzung von ErlViz erfolgte über drei unabhängig voneinander agierenden Teilsystemen. Das ist der Tracer, bestehend aus dem Monitor, dem Coordinator und dem Tracer, der Preprocessor und der Visualizer. Hierbei erfolgt das Übermitteln der Daten zwischen dem Tracer und dem Preprocessor mittels CSV-Dateien. Der Preprocessor synchronisiert die Daten mit dem Visualizer über eine MongoDB sowie eine InfluxDB (Abb. 4.1). Im Folgenden sollen diese Systeme vorgestellt werden. Neben der konkreten Umsetzung wird auch die Motivation dargestellt, warum bestimmte Entscheidungen getroffen wurden.

Da die Umsetzung von ErlViz darauf abzielt die Kommunikation von Aktoren in Systemen aufzuzeichnen, die in Erlang geschrieben sind, werden stellenweise auch erlangspezifische Begrifflichkeiten verwendet. Diese sollen zunächst kontextualisiert werden. In Erlang werden Aktoren auch als Prozesse bezeichnet. Deshalb kann es im Folgenden vorkommen, dass der Begriff Prozess anstatt Aktor benutzt wird. Dieser kann in diesem Kontext als Synonym für Aktor verstanden werden. In dem Zusammenhang wird im Folgenden auch des Öfteren der Begriff Prozessnummer fallen. Hierbei handelt es sich um dasselbe Konzept wie die Adresse des Aktors, was als Begriff in Abschnit 3 verwendet wurde.

4.1.1 Tracing

Das Tracing muss in der Lage sein alle Komponenten zu analysieren in denen der Load Balancer involviert ist. Dabei handelt es sich um alle Komponenten in denen der Load Balancing Monitor integriert ist sowie dem Load Balancing Server. Innerhalb der Testumgebung ist der Load Balancing Monitor innerhalb der Spielserver integriert [All17a]. Da all diese Komponenten in Erlang entwickelt worden, wurde auch das Tracing in ErlViz mittels Erlang realisiert. Dies beinhaltet den Monitor, den Coordinator sowie das Tracing der Anwendung. Hierbei ist es vor allem wichtig, dass die Kommunikation mittels Erlang/OTP [Erlc] aufgezeichnet werden kann, da dies von allen Komponenten zur Kommunikation genutzt wird.

Zum Aufzeichnen der Nachrichten können zwei Techniken angewendet werden. Zum einen den von Erlang zur Verfügung gestellten Tracing Tool Builder (TTB) [Erlb] und zum anderen eine in ErlViz integrierte selbst entwickelte Technik. TTB bietet die Möglichkeit sämtliche Aktionen der Aktoren innerhalb der Anwendungen aufzuzeichnen. Dadurch können auch die ausgetauschten Nachrichten der Aktoren betrachtet und ausgewertet werden. Auch hier ist es erforderlich, dass die Nachrichten wie in Abschnitt 3.2.4 beschrieben angereichert werden. Denn der Sender einer Nachricht ist ansonsten in den meisten Fällen nicht ermittelbar, wie im Folgenden noch aufgezeigt wird. Da über TTB eine sehr große Datenbasis erhoben wird,

die auch Hintergrundprozesse betrachtet, gibt es zusätzlich die Möglichkeit eine in ErlViz direkt integrierte Technik zu nutzen um die Kommunikationsdaten zu erheben. Dadurch kann die Menge der aufgezeichneten Daten deutlich eingeschränkt werden. Bei beiden Techniken werden nur Nachrichten angereichert die mittels Erlang/OTP versendet wurden. Damit ist das Tracing vollständig auf Systeme ausgelegt, die damit umgesetzt wurden. Während in der Aufzeichnung mittels TTB jedoch die Kommunikation aller Aktoren aufgezeichnet werden und somit auch nicht angereicherte Nachrichten die außerhalb von OTP versendet wurden, beschränkt sich die direkt in ErlViz integrierte Lösung auf die Kommunikation mittels OTP.

Im Folgenden soll zunächst die Aufzeichnung mittels TTB vorgestellt werden. Anschließend wird die ErlViz interne Lösung betrachtet.

Aufzeichnung mittels TTB

Wird TTB zur Aufzeichnung genutzt, so wird die Aufzeichnung von Kommunikation, Starten und Beenden von Aktoren vollständig von TTB übernommen. TTB erzeugt zunächst lokal auf den Knoten der beobachtet wird eine Binärdatei, die nach dem Tracing an den Knoten übermittelt wird, durch den das Tracing gestartet und verwaltet wird. Im Falle von ErlViz handelt es sich um den Monitor. TTB bietet zudem die Möglichkeit diese Dateien auszulesen. Dies wird auch von ErlViz genutzt um aus dem Ergebnis eine CSV-Datei zu erzeugen, die vom Preprocessor genutzt werden kann. Im Zuge der Entwicklung wurde TTB untersucht, um zu ermitteln ob TTB als Lösung zur Umsetzung geeignet ist. Deshalb soll im Folgenden zunächst diese Analyse mit einem anschließenden Fazit vorgestellt werden.

Analyse der TTB Nachrichten

Um zu verstehen welche Daten mittels TTB bezogen werden können, werden die bereitgestellten Informationen im Folgenden aufgezeigt und analysiert. Es wird zu sehen sein, dass vor allem Informationen die Spuren des OTP-Systems enthalten viele Informationen über das Vorgehen im System preis geben. Deswegen wird verstärkt auf diese Nachrichten eingegangen. Zur Vereinfachung wird innerhalb dieser Arbeit davon ausgegangen, dass OTP Nachrichten, die über das eigene System versendet werden, auch korrekt verarbeitet werden. Daraus folgt, dass Fehler innerhalb von OTP nicht oder nur schwer mithilfe der Tracing Informationen zu erkennen sind. Um die benötigten Informationen für ErlViz zu ermitteln, werden nicht alle von TTB bereitgestellten Traces benötigt. Entsprechend werden nur folgende Informationstypen verwendet und untersucht:

- Senden einer Nachricht

- Empfangen einer Nachricht
- Starten eines Aktors
- Beenden eines Aktors

Allgemeine Struktur

Die von TTB ermittelten Daten können mittels der bereitgestellten Bibliothek codeseitig ausgelesen werden. Alle so ausgelesenen Ereignisse folgen derselben Struktur. Wobei nur die ersten drei Felder fest definiert sind. Alle Folgenden sind abhängig davon was genau aufgezeichnet wurde:

```
{trace_ts, Pid, Type, ...}
```

Bei Pid handelt es sich um die Prozessnummer des Aktors, von dem das Ereignis aufgezeichnet wurde. $Type$ beschreibt was genau aufgezeichnet wurde, wobei im Kontext von ErlViz nur vier Arten von Ereignissen von Interesse sind: *send*, *receive*, *spawned* und *exit*. Diese werden im Folgenden näher vorgestellt.

Neben dem Aktor, bei dem das Ereignis aufgezeichnet wurde, können in einzelnen Ereignissen auch weitere beteiligte Aktoren mit dargestellt werden. Diese können anstatt der einfachen Prozessnummer in verschiedenen anderen Formen durch TTB dargestellt werden. Folgende Darstellungsformen können auftreten:

1. Prozessnummer: *<7238.76.0>*
2. Registrierter Name: *simple_server_messenger*
3. Registrierter Name mit Knoten: *{simple_server_api,'simple2@erlang-node'}*
4. Prozessnummer mit Startfunktion und Knoten:
   {<7598.117.0>, {proc_lib, init_p, 5}, 'simple1@erlang-node'}
5. Prozessnummer mit globalem Bezeichner und Knoten:
   {<7238.76.0>,simple_server_messenger,'simple1@erlang-node'}

Bei den registrierten Namen handelt es sich um Bezeichner über die ein Aktor benannt sein kann. So kann in Erlang über einen Namen auf einzelne Aktoren zugegriffen werden. Standardmäßig reichert TTB die Darstellung von Aktoren mit zusätzlichen Informationen an. So wird zusätzlich der Knoten angezeigt auf dem sich der Aktor befindet. Sollte der Aktor mit Namen registriert sein, so wird auch der Name des Aktors angezeigt. Ist dies nicht der Fall, so wird die Funktion

angezeigt mit der der Aktor gestartet wurde. Dadurch entsteht eine Struktur wie sie in 4. oder 5. zu sehen ist. Allerdings gibt es auch Ausnahmen, in denen diese Anreicherung nicht stattfindet. Auf diese wird in den folgenden Abschnitten näher eingegangen.

Senden einer Nachricht

Das Senden einer Nachricht ist anhand des Schlüsselwortes „send“ erkennbar. Das hinzu gehörige Ereignis hat dabei folgendes Format [Erla]:

```
{trace_ts, Pid, send, Msg, To, Timestamp}
```

Dieses Ereignis zeigt an, dass der Prozess Pid eine Nachricht mit dem Inhalt Msg an den Prozess To sendet. Zusätzlich gibt es einen Timestamp der anzeigt, wann eine Nachricht gesendet wurde. Eine solche Nachricht könnte Beispielsweise wie folgt aussehen:

```
{trace_ts, <7598.76.0>, send, {'$gen_cast', {test_cast, internal_msg
   }}, {<7598.117.0>, {proc_lib, init_p, 5}, 'simple1@erlang-node'},
    {1517, 777711, 393027}}
```

Es ist zu sehen, dass der Aktor mit der Prozessnummer *<7598.76.0>* eine Nachricht mit dem Inhalt *{'$gen_cast', {test_cast, internal_msg}}* an den Aktor *{<7598.117.0>, {proc_lib, init_p, 5}, 'simple1@erlang-node'}* sendet. Hier ist unter anderem erkennbar, wie die Nachricht versendet wurde. Der Nachrichteninhalt startet mit dem Schlüsselwort *'$gen_cast'*. Dies bedeutet, dass die Nachricht mittels OTP versendet wurde und zwar als *cast*. Hierbei handelt es sich um ein nicht blockierendes, asynchrones Senden einer Nachricht. Als Gegenstück, hierzu gibt es innerhalb von OTP, auch einen sogenannten *call*. Dieser ist am Schlüsselwort *'$gen_call'* erkennbar und ist synchron sowie blockierend.

Beim Empfänger handelt es sich um einen Prozess, deshalb sollte der von TTB gestartete Tracer den Prozess mit zusätzlichen Informationen angereichert haben [Erlb]. Tatsächlich hängt die Art der Darstellung des Prozesses aber davon ab, wie die Funktion aufgerufen wird. Die korrekte Darstellung in TTB ist nämlich für Nachrichten, die mittels OTP versendet wurden nur für den Typ Call garantiert. Bei Nachrichten vom Typ Cast findet die Anreicherung mit den Prozessinformationen nur statt, wenn der Aufruf mittels der Prozessnummer stattfindet. Bei einem Aufruf über den registrierten Namen, werden die Empfängerdaten nicht mit den Prozessinformationen angereichert. Stattdessen wird nur der registrierte Name oder, wenn der Aufruf mit Angabe des Knotens erfolgt, der registrierte Name mit Knoten bereitgestellt.

Das Senden der Cast-Nachrichten erfolgt in Erlang/OTP 21 innerhalb des Moduls *gen_server*. Hier wird das Senden der Nachricht direkt mit den an gen_server übergebenen Parametern

ausgelöst. Im Gegensatz dazu wird das Ausführen eines Calls an das Modul *gen* weitergeleitet. Dieses Modul löst den registrierten Namen des Prozesses in dessen Prozessnummer auf, bevor das Senden der Nachricht stattfindet. Bei Aufrufen an eine Statemachine (über *gen_statem*) gilt dassselbe. Auch hier werden Cast Nachrichten versendet, indem die übergebenen Attribute verwendet werden. Ein Call erfolgt auch hier über das Modul *gen*. Aufgrund der präziseren Information, die dem Tracer bei einem Call vorliegen, reichert dieser den Prozess um die zusätzlichen Prozessinformationen an. Eine Ausnahme bilden Aufrufe, die über den registrierten Namen an einen anderen Knoten gesendet werden. Hier wird auch bei einem Call, die Prozessnummer nicht zuvor aufgelöst. Dementsprechend stellt TTB den Prozess in solchen Fällen als registrierten Namen mit Knoten dar. Dadurch, dass die Prozessnummer nicht immer bereitgestellt wird, bildet sich implizit eine weitere Anforderung an den Präprozessor beim Verarbeiten der Daten. Denn dieser muss trotzdem in der Lage sein, die Prozessnamen den korrekten Prozessnumern zuzuorden. Dies kann nur gelingen, wenn die Prozessnummer mindestens einmal in Kombination mit dem zugehörigen Namen in den Aufzeichnungen gelistet ist. Das die Prozessnummer nicht immer aufgelöst wird ist eine überraschende Erkenntnis, denn innerhalb der Dokumentation von TTB wird die Prozessnummer immer mit aufgeführt. Dies ist eine wichtige Erkenntnis, die nicht eindeutig aus der Dokumentation hervorgeht [Erlb].

Empfangen einer Nachricht

Das Empfangen von Nachrichten wird von TTB in folgendem Format zur Verfügung gestellt [Erla]:

```
{trace_ts, Pid, 'receive', Msg, Timestamp}
```

Schon hier ist sichtbar, dass eine Zuordnung zum Sender nicht möglich ist. Weder ist der Sender aufgeführt, noch gibt es eine Id die eine Zuordnung ermöglichen würde. Bei Aufrufen durch OTP kann dies jedoch unter bestimmten Umständen möglich sein. Im Folgenden ist zum Beispiel sichtbar, dass der Prozess *<7598.104.0>* eine Nachricht vom Prozess *<7598.76.0>* empfangen hat:

```
{trace_ts, {<7598.104.0>, {proc_lib, init_p, 5}, 'simple1@erlang-
   node'}, 'receive', {'$gen_call', {<7598.76.0>, #Ref
   <7598.3619084660.2335965188.183166>}, {test_call, internal_msg}},
    {1517, 777711, 393030}}
```

Dies ist jedoch nur möglich, weil OTP die Nachricht mit dem Sender angereichert hat. So wurde die Nachricht *{test_call, internal_msg}* in einem *'$gen_call'*-Wrapper verpackt. Dieser enthält neben der Art des Aufrufes (*'$gen_call'*) auch die versendende Prozessnummer sowie eine Referenz, die die Nachricht eindeutig identifiziert. Allerdings geschieht diese Anreicherung

nur bei Call-Nachrichten. Bei Cast-Nachrichten ist diese Information nicht vorhanden, wie am folgenden Beispiel zu sehen ist:

```
{trace_ts, {<7598.117.0>, {proc_lib, init_p, 5}, 'simple1@erlang-
   node'}, 'receive', {'$gen_cast', {test_cast, internal_msg}},
   {1517, 777711, 393028}}
```

Hier wird die Nachricht zwar in einem eigenem *'$gen_cast'*-Wrapper verpackt, jedoch gibt es keine Hinweise auf den Sender und es ist auch nichts zur eindeutigen Identifizierung der Nachricht vorhanden. Dies bedeutet, dass auch wenn das Tracing auf OTP-Nachrichten beschränkt wird, nicht immer eine eindeutige Zuordnung möglich ist.

Starten eines Aktors

Das Starten eines neuen Aktors wird von TTB wie folgt dargestellt [Erla]:

```
{trace_ts, Pid, spawned, Pid2, {M, F, Args}, Timestamp}
```

Diese Nachricht sagt aus, dass der Aktor mit der Prozessnummer Pid erfolgreich vom Aktor mir der Prozessnummer $Pid2$ gestartet wurde. Darüber hinaus gibt es auch Nachrichten, dass ein Aktor gestartet wird, wenn ein Prozessname registriert wird oder wenn zwei Aktoren mit einander verlinkt werden. In diesem Kontext genügt es jedoch zu betrachten, wenn ein neuer Aktor erfolgreich gestartet wurde.

Neben dem eigentlichen Start eines Aktors kann in dem Ereignis auch gesehen werden, wie der Aktor gestartet wurde. Das Tupel *{M, F, Args}* zeigt an, dass der Aktor über das Modul M und der Funktion F mit den Argumenten $Args$ gestartet wurde. Werden durch OTP gestartete Aktoren betrachtet, so lassen sich zudem noch weitere Informationen aus dem Trace entnehmen, wie an folgendem Beispiel zu sehen ist:

```
{trace_ts, {<7598.118.0>, {proc_lib, init_p, 5}, 'simple1@erlang-
   node'}, spawned, <7598.74.0>, {proc_lib, init_p, [
   simple_server_children_sup, [simple_server_sup, <7598.72.0>], gen
   , init_it, [gen_server, <7598.74.0>, <7598.74.0>,
   simple_server_test_child,[],[]]]}, {1517, 777713, 393013}}
```

Hier ist zu erkennen, dass der Aktor über das Modul *proc_lib* mittels der Funktion *init_p* gestartet wurde. Der Mehrwert dieses Aufrufes kann vor allem aus den Argumenten gezogen werden. Hierbei sind die ersten beiden Parameter Informationen, die durch das Starten über *proc_lib* bezogen wurden. So ist der startende Aktor sowie dessen Elternaktoren in einer Supervisor-Hierarchie zu erkennen. Es ist zu sehen, dass der Aktor mir dem Namen *simple_server_children_sup* den Aktor gestartet hat und dass sein Elternaktor wiederum

simple_server_sup ist. Die Folgenden zwei Argumente beschreiben über welches Modul und welcher Funktion der Aktor in OTP gestartet wurden. Die Argumente der Funktion geben darüber Auskunft, dass es sich bei dem Aktor um einen *gen_server* handelt und dass dieser im Modul *simple_server_test_child* definiert wurde. Wäre der Prozess mit einem Namen im System registriert worden, so wäre auch das vor dem Modul aufgeführt worden. Diese Information gibt beim Tracing einen wichtigen Einblick darüber, welche Art von Prozess gestartet wurde und ermöglicht so eine leichtere Interpretation über das Verhalten des Systems.

Beenden eines Aktors

Wurde ein Aktor beendet, so ist das über ein Ereignis in folgendem Format zu erkennen [Erla]:

```
{trace_ts, Pid, exit, Reason, Timestamp}
```

Dies bedeutet dass der Prozess Pid mit dem Grund $Reason$ beendet wurde. Ein Beispiel für solch ein Ereignis sieht wie folgt aus::

```
{trace_ts, {<7598.104.0>, {proc_lib, init_p, 5}, 'simple1@erlang-
   node'}, exit, killed, {1517, 777717, 408020}}
```

Hier wurde der Prozess mit der Prozessnummer *<7598.104.0>* mit der Begründung *killed* beendet.

Fazit

Die Tracinginformationen des Tracing Tool Builders bieten bereits eine Vielzahl an Informationen. So lässt sich ermitteln welche Nachrichten in welcher Reihenfolge im System versendet wurden. Auch ist erkennbar wann und durch welchen Aktor ein anderer Aktor gestartet bzw. beendet wurde. Allerdings reichen diese Informationen für viele wichtige Aussagen nicht aus. So ist bei keiner Nachricht, die eintrifft, erkennbar wer der Absender einer Nachricht ist. Der Einsatz von OTP als Bibliothek zur Verwendung von Aktoren bringt jedoch eine ganze Reihe von zusätzlichen Informationen mittels denen sich auf die Umstände schließen lässt. So ist z.B. zum Zeitpunkt des Starts eines Aktors erkennbar, in welchem Modul dieser definiert wurde. Dies ist bei der Ermittlung welche Logik auf Codeebene zu einem Aktor gehört sehr wertvoll. Mit OTP sind auch synchrone Aufrufe in Erlang möglich, auch diese lassen sich über die Tracinginformationen unterscheiden. Allerdings lässt sich auch mit OTP nicht immer eindeutig auf einen Absender schließen. Zwar wird bei synchronen Nachrichten der Absender zusätzlich übertragen, doch gibt es bei asynchronen Nachrichten kein Indiz auf den Absender. Um eine Zuordnung zum Absender zu erreichen müssen die Informationen mit zusätzlichen Informationen angereichert werden.

Aufzeichnung mittels integrierter ErlViz Lösung

Neben TTB wurde auch eine eigene Lösung entwickelt, die in der Lage ist, die verschiedenen Ereignisse aufzuzeichnen. Die so ermittelten Daten werden zunächst auf dem aufgezeichneten Knoten in eine Binärdatei abgelegt, bevor sie am Ende der Aufzeichnung an den Monitor übermittelt werden, um dort in eine CSV-Datei überführt zu werden. Bei den Einträgen in der Binärdatei handelt es sich um Erlang Datenstrukturen, die zuvor in eine binäre Repräsentation überführt wurden. Dadurch lassen sich die Einträge zur Überführung in die CSV-Datei relativ simpel verarbeiten. Das Schreiben der Ereignisse in die Binärdatei wird von einem dedizierten Aktor übernommen. Eingebunden wird die Aufzeichnung über die Manipulation des Bytecodes. Im Folgenden werden die einzelnen Ereignisse näher dargestellt. Dies umfasst den Zeitpunkt der Aufzeichnung und wie die Daten in der Binärdatei abgelegt werde. Zuvor wird noch auf die Repräsentation von Daten innerhalb der Binärdatei eingegangen.

Allgemeine Darstellung der Daten

Alle Daten, die in die Binärdatei geschrieben werden, haben dieselbe Grundstruktur:

```
{Type, Process, ..., Timestamp}
```

$Type$ beschreibt um welche Art der Aufzeichnung es sich hier handelt. Dieser kann einen von vier Werten haben: *send*, *'receive'*, *spawned* oder *exit*. $Process$ ist der Aktor, in dem die Aufzeichnung stattgefunden hat. Als letzten Wert hat jeder Eintrag einen Zeitstempel ($Timestamp$). Hierbei handelt es sich um einen UNIX-Timestamp in Mikrosekunden. Alle Werte, die sich dazwischen befinden, sind Abhängig von der Art der Aufzeichnung.

Darstellung von Aktoren

Eine Besonderheit der integrierten Lösung ist, dass jeder Aktor in all seinen Adressierungen aufgelöst wird, insofern dieser sich auf demselben Knoten befindet, wie die Aufzeichnung stattfindet. Im Gegensatz zu TTB geschieht dies auch für den Aktor, in dem die Aufzeichnung stattgefunden hat. Sollte sich der Aktor auf einem anderen Knoten befinden, so wird nicht explizit der registrierte Name oder dessen Prozessnummer aufgelöst. Innerhalb der Binärdatei kann ein Aktor folgende Darstellungen haben:

- Mit Namen registrierte Aktoren: *{Node, {PId, Name}}*
- Alle anderen Aktoren: *{Node, PId}*

Wobei $Node$ der Knoten ist auf dem sich ein Aktor befindet und PId dessen Prozessnummer. Handelt es sich bei dem Aktor um einen Aktor, der sich auf einem anderen Knoten befindet

und per Name adressiert wird, so wird wie zuvor bereits beschrieben dessen Prozessnummer nicht aufgelöst. In diesem Fall wird der Aktor als mit Namen registrierter Aktor dargestellt. Jedoch wird anstelle der Prozessnummer, der Wert *unknown* hinterlegt.

Senden von Nachrichten

Die Aufzeichnung von versendeten Nachrichten erfolgt innerhalb derselben Funktion mit der diese auch angereichert werden, sodass zur Manipulation nur ein Funktionsaufruf eingebunden wird. Der Eintrag in die Binärdatei erfolgt hierbei in folgender Form:

```
{send, Process, Destination, Msg, Timestamp}
```

Bei $Destination$ handelt es sich um den Aktor an den die Nachricht versendet wurde. Die konkrete Nachricht verbirgt sich hinter Msg.

Empfangen von Nachrichten

Ähnlich wie beim versenden der Nachricht, findet die Aufzeichnung der empfangenen Nachricht innerhalb derselben Funktion statt, wie die Nachrichten aufgelöst werden. Dadurch muss auch hier nur ein Funktionsaufruf beim Empfangen einer Nachricht eingebunden werden. Der Eintrag in der Binärdatei erfolgt hierbei in der folgenden Form:

```
{'receive', Process, Msg, Timestamp}
```

Beim Empfangen einer Nachricht wird neben den Standardinformationen nur eine weitere Information gesammelt und das ist die Nachricht (Msg), die empfangen wurde.

Starten eines Aktors

Das Aufzeichnen des Starts eines Aktors erfolgt direkt nachdem ein Aktor erfolgreich gestartet wurde. Alle OTP basierten Aktoren einer Anwendung werden über das Modul *proc_lib* gestartet. Entsprechend wurden dort alle Funktionen manipuliert, über die potentiell ein Aktor gestartet werden kann. Innerhalb der Binärdatei sieht ein Starten eines Aktor wie folgt aus:

```
{spawned, Process, SpawnedProcess, Args, Timestamp}
```

Wie zu sehen ist, findet die Aufzeichnung innerhalb des Prozesses statt, der den Prozess $SpawnedProcess$ startet. Bei $Args$ handelt es sich um die Argumente, mit denen der Prozess über Erlang gestartet wurde. Wenn es sich um einen Aktor handelt der mittels OTP gestartet wurde sieht die Struktur von $Args$ in aller Regel wie folgt aus:

```
[proc_lib, init_p, [Parent, Ancestors, gen, init_it, [GenMod,
   Starter, Parent, Module, Args, Options]]
```

Soll der Aktor mit seinem Namen registriert werden, so sieht $Args$ folgendermaßen aus:

```
[proc_lib, init_p, [Parent, Ancestors, gen, init_it, [GenMod,
   Starter, Parent, Name, Module, Args, Options]]
```

Diese Struktur weist viele zusätzliche Informationen auf, die zur späteren Auswertung genutzt werden können. Im Kontext dieser Arbeit werden zusätzlich die Argumente $Name$ und $Module$ genutzt. Über diese Parameter kann der registrierte Name des Aktors sowie das Modul ermittelt werden, in dem dieser hauptsächlich implementiert wurde. Darüber hinaus werden auch OTP interne Aktoren aufgezeichnet. Diese werden als einfache Funktionen ohne Parameter gestartet. Entsprechend befindet sich in solchen Fällen eine leere Liste in $Args$.

Beenden eines Aktors

Das Beenden eines Aktors wird kurz bevor er sich beendet aufgezeichnet. Dazu wird gezielt Code in die *terminate()* Funktionen der OTP Module eingebunden. Alle Aktoren, die durch ein initiiertes Beenden oder durch einen internen Fehler beendet werden, können auf diesem Weg aufgezeichnet werden. Denn interne Fehler werden von OTP abgefangen und dadurch ein geregeltes Herunterfahren initialisiert. Aktoren, die durch Fehler innerhalb von OTP beendet werden, können allerdings auf diesem Weg nicht aufgezeichnet werden. Es werden allerdings nicht nur Aktoren aufgezeichnet, die in OTP implementiert wurden, auch OTP interne Aktoren werden aufgezeichnet. In diesem Kontext werden die Aktoren als Funktion gestartet. Um das Ende dieses Aktors aufzuzeichnen, wurden diese Funktionen von einer weiter Funktion umhüllt, in der die Funktion gestartet wird. Ist die Funktion beendet, so beendet sich auch der Aktor und das Ende kann aufgezeichnet werden. Zusätzlich werden alle Fehler, die von der Funktion geschmissen werden abgefangen. Dadurch kann auch aufgezeichnet werden, wenn sich ein Aktor irregulär beendet. Das Beenden eines Aktors sieht innerhalb der Binärdatei wie folgt aus:

```
{exit, Process, From, Reason, Timestamp}
```

Hinter $Reason$ verbirgt sich der Grund, warum ein Aktor beendet wurde. Handelt es sich um ein reguläres Ende, so enthält $Reason$ den Wert *undefined*. Ist bekannt durch wen ein Prozesses beendet wurde, so wird dies in $From$ abgelegt. Dies ist immer dann der Fall, wenn ein Aktor durch einen *Call*-Aufruf beendet wurde. Entsprechend enthält $From$ nicht nur die Prozessnummer, sondern auch eine Referenz, die den Call identifiziert und sieht folgendermaßen aus:

```
{PId, Tag}
```

Wobei Tag die Referenz des Calls ist. Sollte unbekannt sein durch welchen Prozess der Prozess beendet wurde, so enthält $From$ den Wert *undefined*.

Aufzeichnung zusätzlicher Aktormetriken

Das Aufzeichnen zusätzlicher Aktormetriken kann sowohl mit TTB als auch mit dem ErlViz internen Tracing genutzt werden. Gesammelt werden diese Daten nur beim Versenden und Empfangen von Nachrichten, die mit Tracinginformationen angereichert wurden. Diese enthalten nämlich Referenzen, die später genutzt werden, um eine Zuordnung zwischen den Aktormetriken und der Aufzeichnung zu machen. Damit möglichst wenig Code manipuliert werden muss, findet die Aufzeichnung innerhalb der Funktionen statt, die für die integrierte Aufzeichnung über ErlViz eingebunden wurden. Wie bei der Aufzeichnung mittels der integrierten Lösung, werden auch hier die Daten zunächst in eine Binärdatei abgelegt und am Ende der Aufzeichnung an den Monitor übermittelt, um in eine CSV-Datei überführt zu werden. Auch hier gibt es einen dedizierten Aktor, der die Daten in die Datei schreibt. Bei den einzelnen Datensätzen handelt es sich um binäre Repräsentationen von Erlang Datenstrukturen. Ein Datensatz hat hierbei immer die gleiche Struktur:

```
{Key, Value, TraceType, Nonce}
```

Key beschreibt was Aufgezeichnet wurde und $Value$ ist der ermittelte Wert. Aktuell werden zwei verschiedene Werte für Key unterstützt:

queue_length Beschreibt die aktuelle Länge des Posteingangs.

state Der aktuelle Zustand des Aktors. Hierbei handelt es sich um den Wert, der in mittels OTP implementierten Aktoren als Zustand übergeben wird.

Der $TraceType$ beschreibt das Ereignis zu dem die Aufzeichnung der Metrik stattgefunden hat. Dieser kann einen von zwei Werten einnehmen. Nämlich $send$ für Aufzeichnungen beim Versenden und $'receive'$ für Aufzeichnungen beim Empfangen von Nachrichten. $Nonce$ ist die Referenz, mit der die Daten eindeutig mit einer Aufzeichnung verknüpft werden können. Es kann zudem konfiguriert werden, wie oft die Aktormetriken ermittelt werden sollen. Hierbei sagt der Konfigurationswert aus, in welchen Abständen die Daten ermittelt werden sollen. Ein Wert von Zehn hieße in diesem Fall, dass die Daten bei jeder zehnten Aufzeichnung ermittelt werden sollen.

Der Tracer

Der Tracer ist die Komponente, die die Verwaltung des Tracings auf dem Knoten übernimmt. Dazu verwaltet dieser die Knoten, die vom Coordinator übermittelt werden und reichert versendete Nachrichten mit Tracinginformationen an. Zur Realisierung des gemeinsamen Speichers, auf den alle Prozesse zugreifen können, werden ETS Tabellen aus Erlang eingesetzt. Diese enthalten alle Knoten auf denen gerade ein Tracing stattfindet, um so ermitteln zu können an welche Prozesse angereicherte Nachrichten versendet werden können. Das Anreichern der Nachrichten wird im folgenden Abschnitt näher beschrieben.

Anreichern der Nachrichten

Um den Verlauf von Nachrichten zwischen den Aktoren ermitteln zu können, werden alle über OTP versendeten Nachrichten mit einer zusätzlichen Referenz angereichert, die die Nachricht eindeutig referenziert. Um diese Nachrichten identifizieren zu können, folgen alle aufgezeichneten Nachrichten dem selben Schema:

```
{$tracing_msg, Nonce, Msg}
```

Wie zu sehen ist, besteht eine angereicherte Nachricht aus drei Elementen. Bei dem ersten Element handelt es sich um den Wert *$tracing_msg*. Dieser dient als Schlüsselwort, um zu erkennen, dass es sich um eine angereicherte Nachricht handelt. Hinter $Nonce$ verbirgt sich eine Referenz, um die Nachricht eindeutig zu identifizieren. Ähnlich wie beim Call in OTP, ist es durch diese möglich, zusammenhängende gesendete und empfangene Nachrichten miteinander zu verknüpfen. Die $Nonce$ wird mittels der von Erlang bereitgestellten Funktion *make_ref()* erstellt. Dies garantiert, dass die $Nonce$ zwischen allen verbundenen Knoten eindeutig ist [Erla]. Eine einfache Referenz genügt in diesem Fall, da in Erlang alle Nachrichten zwischen zwei Aktoren in derselbe Reihenfolge beim Empfänger eintreffen, wie sie versendet wurden. Das Anreichern der Nachrichten geschieht durch die Manipulation des Codes von OTP. Dies ist in Erlang durch Manipulation des AST möglich. Im folgenden Abschnitt wird dieser Vorgang näher erläutert.

Manipulation des AST

Damit die Aufzeichnung durch die in ErlViz integrierte Lösung stattfinden oder die Nachrichten zwischen den Aktoren angereichert werden können, wird der Code von OTP manipuliert. Dies geschieht durch Manipulation des Abstract Syntax Tree (AST) der verschiedenen Erlang Module in denen OTP implementiert wurde. Hierzu wird zunächst das Kompilat des betroffenen Moduls geladen und aus diesem der zugehörige AST ermittelt. Innerhalb des ASTs werden nun

alle Passagen herausgesucht, die manipuliert werden müssen um die benötigten Informationen zu ermitteln. D.h. beim Versenden sowie Empfangen von Nachrichten wird Code eingebunden, der diese mit Tracinginformationen anreichert bzw. diese wieder entfernt, bevor sie an den ursprünglichen Code weitergereicht wird. Findet die Aufzeichnung der Nachrichten über die integrierte Lösung statt, so wird auch das Starten und Beenden eines Aktors manipuliert. Zusätzlich werden die für die Aufzeichnung benötigten Funktionen direkt in die manipulierten Module eingebunden. Dies ist Notwendig, weil diese Module ansonsten nicht die Möglichkeit haben Code des Tracers zu nutzen. Nach Abschluss der Manipulation wird der neu erstellte AST kompiliert und eingebunden. Dazu wird zunächst die vorliegende Version entfernt und anschließend die neue Version geladen. Nicht immer ist ein Aktor in der Lage den neuen Code direkt einzubinden, denn dazu muss dieser sein eigenes Modul erneut aufrufen. Das kann jedoch forciert werden, indem eine Nachricht an den Aktor gesendet wird. Beim Beenden der Aufzeichnung wird die manipulierte Version entfernt und die ursprüngliche Version wieder eingebunden.

Um das Verhalten einer Anwendung zu analysieren, die auf Erlang/OTP basiert, müssen eine ganze Reihe von Modulen manipuliert werden, die in der Interaktion der Aktoren involviert sind. Deshalb werden in ErlViz die folgenden Module manipuliert:

- gen
- gen_server
- gen_fsm
- gen_statem
- gen_event
- global
- proc_lib
- sys

Bei gen, global, proc_lib und sys handelt es sich um Hilfsmodule, die vor allem Funktionalitäten anbieten, die durch die anderen Module genutzt werden. Dagegen sind gen_server, gen_fsm, gen_statem und gen_event Module, die dazu dienen, Aktoren zu implementieren. Hierbei unterstützt jedes Modul ein bestimmtes Konzept zu realisieren. Dies können z.B. einfache Aktoren aber auch komplexere Konzepte wie Statemachines sein. Da vor allem die Interaktion

innerhalb einer Anwendung betrachtet werden soll, fokussiert sich die Manipulation der Module vor allem darauf die Kommunikation zwischen den Aktoren aufzuzeichnen die mittels OTP umgesetzt wurden. Zusätzlich werden allerdings auch die Aktoren aufgezeichnet, die innerhalb von OTP gestartet werden und vor allem unterstützende Aufgaben haben. Einzige Ausnahme hiervon bildet das Modul *global*. Hier werden interne Aktoren nicht aufgezeichnet. Das hängt vor allem damit zusammen, dass alle Aktoren, die mit einem neuen Code laufen sollen, potentiell dazu forciert werden müssen den neuen Code zu laden. Dies funktioniert jedoch nur, wenn ein Aktor eine Funktion aus seinem Modul erneut aufruft. Dies wird in ErlViz durch das Senden einer *system* Nachricht gemacht. Die internen Aktoren von global können diese Nachricht allerdings nicht verarbeiten, sodass die internen Aktoren potentiell nicht den veränderten Code nachgeladen haben und so mögliche angereicherte Nachrichten auch nicht verarbeiten können. Deshalb wird für diese internen Aktoren keine Manipulation und somit auch keine Aufzeichnung durchgeführt.

Versenden von Nachrichten

Nachrichten werden in OTP in nahezu allen involvierten Modulen versendet. Das einzige Modul in dem keine Nachricht versendet wird, ist das Modul *sys*. Synchrone Nachrichten (*Call*) stehen in OTP für alle Aktoren zur Verfügung, die mit den Modulen gen_server, gen_fsm, gen_statem und gen_event umgesetzt wurden. All diese Module nutzen zum Ausführen der synchronen Nachrichten das Modul gen. Neben der synchronen Kommunikation implementieren all diese Module auch alle eine Form zur asynchronen Benachrichtigung. So gibt es bei gen_server und gen_statem eine *cast* Funktion, bei gen_fsm die Möglichkeit Events zu senden und bei gen_event die Möglichkeit eventuelle Abonnenten zu benachrichtigen. In all diesen Modulen ist zudem eine *reply*-Funktion implementiert. Diese wird nach jedem Call Aufruf genutzt, um eine Antwort an den sendenden Prozess zu übermitteln.

Auch wenn Nachrichten an einer Vielzahl von Stellen versendet werden, so ist der Eingriff immer der Gleiche. Um eine Nachricht mit den benötigten Tracinginformationen anzureichern, wird der Code so angepasst, dass die gesendete Nachricht zunächst der Funktion *wrap_msg()* übergeben wird, welche als Ergebnis die angereicherte Nachricht zurückliefert. Um eine Nachricht zu versenden gibt es in Erlang zwei Möglichkeiten. Zum einen das Nutzen der Funktion *erlang:send()* und zum anderen das Nutzen des Bang-Operators (*!*). Beide Varianten werden im Folgenden vorgestellt.

Wird eine Nachricht mittels *erlang:send()* versendet, so sieht das wie folgt aus:

```
erlang:send(Dest, Msg)
```

Die Manipulation ersetzt dies durch:

```
erlang:send(Dest, wrap_msg(Msg, Dest, TracedNodes, Config))
```

Das gleiche gilt bei Nachrichten, die mittels des Bang-Operators versendet werden:

```
Dest ! Msg
```

Dies wird zu:

```
Dest ! wrap_msg(Msg, Dest, TracedNodes, Config))
```

Es ist zu sehen, dass nicht nur die Nachricht selber übergeben wird, sondern auch eine Reihe weiterer Parameter. Dies hängt mit zwei Umständen zusammen. Zum einen können innerhalb der Funktion weitere Aufzeichnungen stattfinden und zum anderen werden zusätzliche Informationen benötigt, um die Nachrichten anreichern zu können. So enthält *Config* die Konfigration der Aufzeichnung. Dadurch kann ermittelt werden, ob die integrierte Aufzeichnung genutzt werden soll und ob zusätzliche Metriken erfasst werden sollen. Mithilfe des Ziels der Nachricht ($Dest$) und der Tabelle, der getraceten Knoten ($TracedNodes$), kann festgestellt werden, ob die Nachricht angereichert werden soll. Ist der Knoten des Ziels in der Tabelle enthalten, so wird die Nachricht angereichert, ist der Knoten nicht enthalten, so wird die Nachricht einfach unverändert wieder zurückgegeben. Innerhalb des Moduls *global* gibt es allerdings ein paar Sonderfälle. Denn wie zuvor erwähnt, werden hier nicht alle inneren Aktoren aufgezeichnet. Um Nachrichten an diese Aktoren aber dennoch aufzuzeichnen, wenn die integrierten Lösung genutzt wird, wird hier die Funktion *trace_msg()* anstelle von *wrap_msg()* genutzt. Dies stellt sicher, dass diese Nachrichten nich mit Tracinginformationen angereichert werden.

Wird TTB eingesetzt, so kann es vorkommen, dass die Prozessnummer des Zielprozesses nicht immer aufgelöst wird. Um das zu verhindern kann konfiguriert werden, dass diese zusätzlich aufgelöst werden soll. Dies soll vor allem dem Preprocessor Arbeit abnehmen, weil dieser sonst nachträglich versuchen muss, eine Zuordnung zwischen den Namen des Aktors und der Prozessnummer zu machen. Die Art der Auflösung ist davon abhängig, ob der betroffene Aktor lokal oder einem entfernten Knoten aufgerufen wird. Bei einem lokalen Aufruf kann die von Erlang bereitgestellte Funktion *whereis()* eingesetzt werden. Bei Aktoren, die sich auf entfernten Knoten befinden ist dies nicht möglich. Das hängt damit zusammen, dass die Prozessnummern von entfernten Aktoren generell nicht lokal aufgelöst werden können. Deshalb kann zusätzlich konfiguriert werden, ob auch diese aufgelöst werden sollen. Sollen diese aufgelöst werden, so

wird eine Hilfsfunktion aufgerufen, die diese Übersetzung übernehmen kann. Dazu ruft diese im Hintergrund per RPC die *whereis()* Funktion auf dem Knoten auf, auf dem sich der Aktor befindet. Per Konfiguration kann zusätzlich angegeben werden, ob die Prozessnummer lokal abgespeichert werden soll, um zukünftige RPC-Aufrufe zu vermeiden.

Empfangen von Nachrichten

Damit die Manipulation der Nachrichten transparent für den Programmcode ist, muss die Nachricht beim Empfangen wieder entpackt werden. In allen manipulierten Modulen befinden sich auch *receive*-Blöcke. Hier werden die Nachrichten von anderen Aktoren empfangen. Entsprechend müssen an dieser Stelle die Nachrichten wieder entpackt werden. Dazu wird der Block um die Strukturen erweitert, mit der die Nachricht verpackt wurde:

```
receive
        Msg ->
                do_something(Msg)
end
```

wird somit zu:

```
receive
        Gcjksg = {$tracing_msg, _, Msg} ->
                collect_on_receive(Gcjksg, Config),
                do_something(Msg);
        Gcjksg = Msg ->
            collect_on_receive(Gcjksg, Config),
                do_something(Msg)
end
```

Diese Erweiterung um einen Patternmatch, für angereicherte Nachrichten, geschieht für jede Nachricht in dem *receive*-Block. Zusätzlich wird die empfange Nachricht in dem Paramter $Gcjksg$ festgehalten, damit diese referenziert werden kann wenn *collect_on_receive()* aufgerufen wird. Der Name der Variable ist eine kryptische Zeichenkette, damit die Wahrscheinlichkeit einer Kollision mit dem Namen eines bereits vorhandenen anderen Parameters möglichst gering ist. Als erste Anweisung nach dem Empfangen einer verpackten Nachricht wird die *collect_on_receive()* Funktion aufgerufen. Diese ist für das Aufzeichnen der empfangenen Nachricht sowie für das Erfassen von Aktormetriken zuständig, insofern diese konfiguriert sind. Deshalb wird wie beim Versenden der Nachricht auch hier die Konfiguration übergeben, um ermitteln zu können was aufgezeichnet wird. Eine Erfassung der Aktormetriken erfolgt nur, wenn es sich um eine angereicherte Nachricht handelt, denn nur dann liegt eine Referenz vor, mittels der die Daten der aufgezeichneten Nachricht zugeordnet werden können. Sollte

die integrierte Lösung zur Aufzeichnung genutzt werden, so findet innerhalb dieser Funktion die Aufzeichnung der empfangenen Nachricht statt. Daraus ergibt sich, dass die Funktion keine Aufgaben übernimmt, wenn die Aufzeichnung über TTB stattfindet und die empfangene Nachricht nicht angereichert ist. Deshalb findet dieser Aufruf bei nicht angereicherten Nachrichten nur statt, wenn die Kommunikation mittels der integrierten Lösung aufgezeichnet wird. Ist dies nicht der Fall, so wird der Aufruf von *collect_on_receive()* nur für angereicherte Nachrichten in den AST eingebunden. Alle Ausdrücke nach dem Aufruf dieser Funktion sind eine Kopie der Original-Verarbeitung. Dadurch, dass der receive-Block erweitert wurde und nicht die Original-Nachricht ersetzt wurde, ist es immer noch möglich, auch Nachrichten zu erhalten, die nicht angereichert sind. Dies muss auch weiterhin möglich sein, da jederzeit auch Nachrichten von Knoten eintreffen können, auf denen keine Aufzeichnung stattfindet. Durch das Lauschen auf Nachrichten einer bestimmten Struktur, müssen Anwendungen die an der Kommunikation mit dem beobachteten Knoten beteiligt sind, die Konvention einhalten, dass keine eigenen Nachrichten, der Struktur der Nachrichten des Tracers entsprechen dürfen.

Starten eines Aktors

Neben der Kommunikation von Aktoren wird auch das Starten eines Aktors aufgezeichnet. Sollte die integrierte Lösung zur Aufzeichnung eingesetzt werden, so wird auch hier Code zum Aufzeichnen eingebunden. Daraus folgt, dass alle Änderungen, die in diesem Abschnitt vorgestellt werden nur durchgeführt werden, wenn die integrierte Lösung zum Einsatz kommt. Wird TTB eingesetzt, so bleiben die entsprechenden Stellen unverändert.

Aktoren im manipulierten Kontext werden auf zwei Arten gestartet. Zum einen über das Modul *proc_lib* und zum anderen als Funktionen. Bei den Aktoren, die über das Modul *proc_lib* gestartet werden, handelt es sich in der Regel um Aktoren, die eines der OTP Konzepte implementieren. Dies sind meistens die Aktoren, mit denen eine Anwendung umgesetzt wurde. Bei den Aktoren die als Funktion gestartet werden, handelt es sich um Aktoren, die von OTP intern genutzt werden.

Wenn Aktoren über *proc_lib* gestartet werden, so geschieht das über eine vom Modul *erlang* bereitgestellte Funktion. Dazu nutzt *proc_lib* eine der Funktionen *spawn()*, *spawn_link()* oder *spawn_opt()*. Die Manipulation, die vorgenommen wird, ist in allen Fällen dieselbe. Deshalb wird im Folgenden die Manipulation des Startens eines Aktors beispielhaft an dem Aufruf der Funktion *spawn()* aufgezeigt:

```
erlang:spawn(?MODULE, init_p, [Parent, Ancestors, M, F, A])
```

wird durch die Manipulation zu:

```
trace_spawn(erlang:spawn(?MODULE, init_p, [Parent, Ancestors, M, F,
   A]), [?MODULE, init_p, [Parent, Ancestors, M, F, A]], Config)
```

Hier wird das Ergebnis des Startens eines Aktors an die Funktion *trace_spawn()* übergeben, sodass der neue Aktor aufgezeichnet werden kann. Zusätzlich werden die Argumente, mit denen der der Aktor gestartet wurde (*[?MODULE, init_p, [Parent, Ancestors, M, F, A]]*), nochmal separat übergeben. Dadurch können auch die Argumente aufgezeichnet werden, die zum Starten eines Aktors verwendet wurden. Mit diesen Daten kann beim Überführen der Daten in die CSV-Datei ermittelt werden, mit welchem Modul der gestartete Aktor umgesetzt wurde. Dieser steckt in diesem Fall in dem Argument M. *Config* enthält neben Konfigurationsparametern auch interne Informationen für die Aufzeichnung. So ermittelt *trace_spawn()* hierüber, den Aktor, der die Aufzeichnung der Daten übernimmt.

Interne Aktoren werden dagegen als Funktion gestartet. Dazu werden die integrierten Funktionen *spawn()*, *spawn_link()* oder die Funktion *spawn_monitor()* aus dem Modul *erlang* genutzt. Da auch hier die durchgeführte Manipulation immer dieselbe ist, soll auch dies am Beispiel der Funkion *spawn()* gezeigt werden. Hierzu wurde der Start einer deutlich vereinfachten Funktion als Beispiel genommen, da der Umfang der Funktionen, aus den OTP-Modulen, zum Veranschaulichen zu groß gewesen wäre :

```
spawn(fun() -> ok end)
```

wird hierbei zu:

```
trace_spawn(spawn(fun() -> trace_spawned_fun_termination(fun() -> ok
    end, Config) end), [], Config)
```

Die ursprüngliche Funktion befindet sich nun innerhalb von *trace_spawned_fun_termination()*. Diese Funktion wird verwendet, um feststellen zu können, wann die Funktion und somit auch der Aktor beendet ist. Dies wird allerdings im folgenden Abschnitt näher beschrieben. Die Aufzeichnung des Starts des Aktors erfolgt auch hier über die Funktion *trace_spawn()*. Dadurch erfolgt die Aufzeichnung identisch, wie beim Start der Aktoren über *proc_lib*. Als Argumente, mit denen der Aktor gestartet wurde, wird in diesem Fall allerdings immer eine leere Liste übergeben. Denn das Argument mit dem der Aktor gestartet wurde, ist nur die Funktion selber. Aus dieser Information lässt sich jeoch aktuell kein Mehrwert für die Generierung der CSV-Datei gewinnen.

Beenden eines Aktors

Da der Lebenszyklus eines Aktors nur explizit aufgezeichnet werden muss, wenn die integrierte Lösung zum Einsatz kommt, findet auch die Manipulation zur Aufzeichnung des Lebensendes nur in diesem Fall statt. Auch hier gilt, wird TTB eingesetzt, so bleiben die beschriebenen Stellen unverändert. Da Aktoren, die mittels OTP genutzt werden, und interne Aktoren unterschiedlich beendet werden, gibt es auch abhängig davon zwei verschiedene Methoden, um dies aufzuzeichnen. In beiden Fällen wird das Lebensende kurz vor dem eigentlichen Ende aufgezeichnet.

Zunächst soll beschrieben werden, wie das Lebensende von Aktoren aufgezeichnet wird, die mittels OTP umgesetzt wurden. Hierbei handelt es sich um Aktoren, die mittels *gen_server*, *gen_event*, *gen_statem* oder *gen_fsm* realisiert wurden. Alle diese Module haben eine Funktion die ausgeführt wird, wenn ein Aktor beendet wird. Mit Ausnahme von *gen_event* heißt diese Funktion bei allen Modulen *terminate()*. Bei *gen_event* heißt sie *terminate_server()*. Auch wenn die Implementierung bei den unterschiedlichen Modulen durchaus unterschiedlich ist, funktioniert die Aufzeichnung immer nach dem selben Prinzip. So wird der Code zum Aufzeichnen zwischen dem Aufruf des Codes, der den Aktor implementiert, und dem abschließenden OTP Code eingebunden. Dies soll am Beispiel der Manipulation von *gen_server* verdeutlicht werden:

```
1 terminate(Class, Reason, Stacktrace, ReportReason, Name, From, Msg,
     Mod, State, Debug) ->
2       Reply = try_terminate(Mod, terminate_reason(Class, Reason,
           Stacktrace), State),
3       case Reply of
4               [...]
5       end.
```

wird somit zu:

```
1 terminate(Class, Reason, Stacktrace, ReportReason, Name, From, Msg,
     Mod, State, Debug) ->
2       Reply = try_terminate(Mod, terminate_reason(Class, Reason,
           Stacktrace), State),
3       trace_termination(From, Reason, Config),
4       case Reply of
5               [...]
6       end.
```

Innerhalb von *trace_termination()* wird nun aufgezeichnet, dass der Aktor beendet wurde. Dies geschieht auch, wenn ein Fehler beim Aufruf des Moduls auftritt. Denn OTP fängt alle Fehler,

die dort auftreten können, ab. Unter der Annahme, dass sich kein Fehler innerhalb von OTP befindet, gelingt somit jede Aufzeichnung eines beendeten Aktors innerhalb von OTP. Wie zu sehen ist, gibt es unter anderem den Parameter $From$, der auch an *trace_termination()* übergeben wird. Dieser enthält Informationen darüber, durch welchen Aktor das Ende des Aktors ausgelöst wurde. Dieser enthält jedoch nur einen Wert, wenn das Ende durch einen Call ausgelöst wurde. Ansonsten enthält $From$ den Wert *undefined*. Ist der auslösende Aktor bekannt, so wird nach Beenden des Aktors eine Antwort an diesen gesendet. Dadurch kommt eines zu einer weiteren Aufzeichnung, in der eine Nachricht durch den Aktor versendet wurde, obwohl dessen Lebensende zuvor aufgezeichnet wurde. $Reason$ enthält den Grund, warum ein Aktor beendet wurde und *Config* wird, wie bei der Aufzeichnung zum Start eines Aktors, benötigt, um den Aktor zu ermitteln, der die Aufzeichnung übernimmt.

Interne Aktoren werden, wie im Abschnitt zuvor bereits beschrieben, als Funktionen realisiert. Um das Ende der Funktion aufzeichnen zu können, wird die Aktorfunktion der aufzeichnenden Funktion übergeben und innerhalb dieser die Funktion ausgeführt. Dadurch kann festgestellt werden wann die Funktion beendet ist und dies als Lebensende des Aktors aufgezeichnet werden. Damit das Lebensende auch festgestellt werden kann, wenn ein Fehler innerhalb der Funktion stattfindet, werden alle auftretenden Fehler innerhalb der Funktion abgefangen. Sollte ein Fehler auftreten, so wird der Aktor als beendet aufgezeichnet und der Fehler als Grund für die Terminierung angegeben. Anschließend wird der Fehler weiter eskaliert, sodass die Aufzeichnung für die Anwendung transparent bleibt. Die Manipulation sieht dann wie folgt aus:

```
spawn(fun() -> ok end)
```

wird durch die Manipulation zu:

```
trace_spawn(spawn(fun() -> trace_spawned_fun_termination(fun() -> ok
    end, Config) end), [], Config)
```

Artefakte

Als Ergebnis aus der Aufzeichnung können ein bis zwei verschiedene Dokumente entstehen. Hierbei handelt es sich zum einen um eine Datei, die die Aufzeichnung der Kommunikation der verschiedenen Aktoren und deren Lebenszyklus enthält (tracing.csv) und zum anderen um eine Datei, die die zusätzlich erfassten Aktormetriken enthält (process_infos.csv). Wobei letztere Datei nur erstellt wird, insofern diese zusätzlichen Metriken auch erfasst wurden. Erstellt werden diese Dateien aus den Binärdateien, die während der Aufzeichnung erstellt

wurden. Dazu werden diese nach der Aufzeichnung zunächst durch den Monitor, von den einzelnen Knoten geladen und anschließend verarbeitet. Während diese Übermittlung für die integrierte Lösung sowie den zusätzlich ermittelten Aktormetriken implementiert werden mussten, ist dieser Prozess bereits Bestandteil von TTB und musste dort somit nicht zusätzlich implementiert werden. Damit die Dateien ein von Erlang unabhängiges Format haben, werden die Daten in eine CSV-Datei überführt. Statt durch eine Kommaseparierung erfolgt die Trennung der Werte mittels Semikolon. Dadurch können Probleme mit bestimmten Darstellungen von Datenstrukturen in Erlang vermieden werden, die Kommata enthalten. Im Folgenden werden die Formate der verschiedenen Dateien vorgestellt.

Aufzeichnung der Kommunikation

In TTB werden neben den Daten von Interesse auch eine Reihe zusätzlicher Informationen bereitgestellt, die für ErlViz nicht benötigt werden. Zum Zeitpunkt der Überführung in eine CSV werden die nicht benötigten Daten ausgefiltert, sodass nur die Informationen in die CSV gelangen, die für spätere Schritte noch benötigt werden. Bei Anwenden der integrierten Lösung ist kein ausfiltern nötig, da hier bereits nur die benötigten Daten aufgezeichnet werden. Sollte es sich bei ermittelten Daten um angereicherte Nachrichten handeln, so wird dies automatisch erkannt und die Referenz sowie die ursprüngliche Nachricht herausgezogen und separat in der CSV aufgeführt.

Das genaue Format der Daten in der CSV-Datei hängt davon ab um welche Art von Nachricht es sich handelt. Allerdings beginnen alle Zeilen mit den selben Informationen. So sieht das allgemeine Format wie folgt aus:

```
Type; Node; Timestamp; ...
```

Hierbei ist *Type* die Art der Aufzeichnung und beschreibt, ob es sich um eine gesendete Nachricht, empfangene Nachricht, um einen neu gestarteten Aktor oder um einen beendeten Aktor handelt. *Node* ist der Knoten, auf dem die Daten ermittelt wurden und *Timestamp* ein Zeitstemmpel, von dem Zeitpunkt an dem die Aufzeichnung stattfand. Hierbei handelt es sich um den POSIX Systemzeitstempel in Mikrosekunden. Dahinter ist das Format abhängig vom der Art der aufgezeichneten Daten. Deshalb sollen diese im Folgenden näher dargestellt werden.

Senden einer Nachricht

```
send; FromNode; Timestamp; FromPid; ToPid; ToName; ToNode; Msg;
   Nonce
```

Das Senden einer Nachricht wird mit dem Typ *send* angegeben. Zudem wird mit den Parametern *FromNode*, *FromPid*, *ToPid*, *ToName* sowie *ToNode* angegeben, welcher Aktor mit welchem anderen Aktor kommuniziert hat. Hierbei stellen *FromPid* und *ToPid* die Prozessnummern, der beteiligten Aktoren, dar. *ToName* ist der Namen des Aktors, an den die Nachricht gesendet wurde. Da nicht immer alle Informationen vom Empfänger vorliegen, kann es vorkommen, dass entweder *ToPid* oder *ToName* nicht bekannt sind. Sollte dies der Fall sein, so ist dort der Wert *UNKNOWN* eingetragen. *Msg* beinhaltet die versendete Nachricht, wobei es sich hierbei immer um die nicht angereicherte Nachricht handelt. Sollte eine Nachricht angereichert worden sein, so wird hier der entpackte Inhalt angezeigt. *Nonce* ist ein optionales Feld und nur Bestandteil des Eintrags wenn es sich hierbei um eine angereicherte Nachricht handelt. In solchen Fällen wird die Referenz aus der verpackten Nachricht extrahiert und als *Nonce* in den Datensatz eingesetzt.

Empfangen einer Nachricht

```
receive; Node; Timestamp; PId; Name; Msg; Nonce
```

Erhaltene Nachrichten werden durch das Schlüsselwort *receive* gekennzeichnet. Der Prozess der die Nachricht empfangen hat, wird über die Parameter *PId* für die Prozessnummer sowie *Name* für den Namen des Aktors beschrieben. Sollte ein Prozess keinen Namen haben, wird der Wert *NO_NAME* als Wert hinterlegt. Unter *Msg* wird die empfangene Nachricht angezeigt. Wie auch beim Senden einer Nachricht, wird hier die original Nachricht angezeigt. D.h. wenn die Nachricht während der Aufzeichnung angereichert wurde, dann wird nur die unverpackte Nachricht angezeigt. Ebenfalls wie beim Senden einer Nachricht, wird zusätzlich die *Nonce* aufgeführt, wenn es sich um eine angereicherte Nachricht handelt.

Neuer Aktor

```
spawned; Node; Timestamp; SpawningPid; SpawningName; SpawnedPid;
   SpawnedModule; SpawnedName
```

Neu gestartete Aktoren werden durch das Schlüsselwort *spawned* dargestellt. Der Eintrag sagt aus das der Aktor *SpawningPid* den Aktor *SpawnedPid* gestartet hat. Neben er Prozessnummer wird mittels *SpawningName* und *SpawnedName* auch die Benennung der Aktoren angegeben. Da auch hier gilt, dass nicht alle Aktoreb benannt sind, wird der Wert *NO_NAME* verwendet, wenn kein Name vorliegt. Das Modul des gestarteten Prozesses kann nur ermittelt werden, wenn der Aktor über *proc_lib* gestartet wurde. Denn das Modul (*SpawnedModule*) lässt sich dann aus den Parametern ermitteln mit denen der Aktor gestartet wurde. Ist dies nicht der Fall, so kann auch das Modul in dem ein Aktor umgesetzt wurde nicht ermittelt werden. In

solchen Fällen wird der Wert *UNKNOWN* hinterlegt.

Entfernter Aktor

```
exit; Node; Timestamp; Pid; Name; TerminatedByPId; CallTag; Reason
```

Wenn sich Aktoren beenden oder beendet wurden, so ist das durch das Schlüsselwort *exit* erkennbar. Diese Auflistung sagt aus, dass der Prozess mit der Prozessnummer *Pid* und dem Namen *Name* mit der Begründung *Reason* beendet wurde. Da nicht jeder Prozess benannt ist, wird der Wert *NO_NAME* für den Namen hinterlegt, wenn keiner vorliegt. Sollte die Aufzeichnung mittels der integrierten Lösung stattgefunden haben, so kann es vorkommen, dass auch bekannt ist, von wem der Aktor beendet wurde. In solchen Fällen werden auch die Parameter *TerminatedByPId* und *CallTag* ausgefüllt. Diese sind jedoch optional und können weggelassen werden, wenn der beendende Aktor nicht bekannt ist. *TerminatedByPId* enthält hierbei die Prozessnummer des Aktors. Da der Beendende Aktor nur bekannt ist, wenn Call gemacht wurde, steht dort auch die Referenz des Calls zur Verfügung mit der der Aktor beendet wurde. Diese Referenz wird in *CallTag* hinterlegt.

Zusätzliche Aktormetriken

Auch die zusätzlich ermittelten Metriken, werden in einer CSV-Datei abgespeichert. Das Format in dieser Datei sieht dabei wie folgt aus:

```
Key; Value; TraceType; Nonce
```

Hierbei ist *Key* der Schlüssel für die Art der Metrik, die gesammelt wurde und *Value* der aufgezeichnete Wert. Bei *Nonce* handelt es sich um die Referenz, die bei der Aufzeichnung des Wertes übergeben wurde. Hierbei handelt es sich um dieselbe Referenz, wie sie beim anreichern der Daten genutzt wurde. Da es jedoch zu jeder Referenz zwei Aufzeichnungen gibt, nämlich das Versenden sowie das Empfangen der Nachricht, wird für die Zuordnung zusätzlich der *TraceType* benötigt. Dieser hat entsprechend die Werte *send* oder *receive*. Über die Referenz in Kombination mit der Art der Aufzeichnung, lässt sich nun eindeutig die Aufzeichnung herleiten, zu der der ermittelte Wert gehört. Folgende Schlüssel stehen aktuell für die ermittelten Werte zur Verfügung:

queue_length Der Wert entspricht der aktuellen Größe des Postfaches des Aktors.

state Der Wert ist der aufgezeichnete Zustand des Aktors. Hierzu wird der innerhalb von OTP festgehaltene Zustand genommen.

4.1.2 Preprocessing

Das Format, der durch das Tracing erzeugten Dateien, reicht nicht aus, um die Prozesskommunikation darzustellen. Deshalb werden im Preprocessing die Daten in ein Format überführt, mit dem die Prozesskommunikation dargestellt werden kann. Zusätzlich werden die Aufzeichnung der Aktorkommunikation und die zusätzlich ermittelten Metriken zusammengeführt. Alle verarbeiteten Daten werden von dem Preprocessor in eine der Datenbanken hinterlegt, die von der ErlViz-Visualisierung zum Auslesen der Daten genutzt werden. Ziel ist es, die Daten zum einen so zu hinterlegen, das auch nachträglich auf die Daten zugegriffen werden kann, aber auch einen effizienteren Zugriff auf die Daten zu haben, als es durch CSV-Dateien möglich wäre. Eine wichtige Anforderung ist es, dass die Daten der Aufzeichnung zeitbezogen abgerufen werden können, da für gewöhnlich ein zeitlicher Abschnitt der Aufzeichnung betrachtet werden soll. Deshalb wird die Aufzeichnung in eine InfluxDB [Inf] hinterlegt. Bei InfluxDB handelt es sich um eine Datenbank die auf das abspeichern von zeitbezogenen Daten spezialisiert ist [Inf]. Metadaten zu den einzelnen Aktoren werden in einer MongoDB [Mon] festgehalten. So können die Aktoren gezielt indexiert werden, um auf allgemeine Daten der Aktoren zugreifen zu können. Neben dem Bereitstellen der Daten, ist das Ziel des Preprocessings auch das Überführen in ein von ErlViz verarbeitbares Format. Dazu werden die bereitgestellten Aufzeichnungen mit Lamportuhren angereichert, die aus den vorliegenden Daten errechnet werden.

Vorgang

Um die Daten bereitstellen zu können, müssen die Artefakte, die bei der Aufzeichnung entstanden sind, aufbereitet und korrekt in die Datenbank eingepflegt werden. Dazu gehört auch das Zusammenführen der beiden CSV-Dateien mit den Tracingdaten sowie den zusätzlichen Aktormetriken, insofern diese aufgezeichnet wurden. Da der allgemeine Vorgang bereits in 3.3 beschrieben wurde, soll hier vor allem auf technische Besonderheiten eingegangen werden, die für die Umsetzung interessant sind. Wie zuvor bereits erwähnt, werden alle Tracingdaten in einer InfluxDB und alle Metadaten in einer MongoDB hinterlegt. Die temporäre Tabelle in der alle zusätzlich aufgezeichneten Aktormetriken hinterlegt sind, ist auch Bestandteil der MongoDB. Damit effizient auf die temporären Daten zugegriffen werden kann, wird dort ein Index auf die Referenz der Aufzeichnung gesetzt. Die zur Aufzeichnung zugehörige Lamportuhr wird aufgebaut, während die Tracingdatei abgearbeitet wird . Dies ist möglich, weil es wie zuvor beschrieben in Erlang eine Garantie gibt, dass Nachrichten zwischen Aktoren in der selben Reihenfolge eintreffen wie sie abgesendet wurden. Diese Reihenfolgegarantie auf Aktorebene

reicht aus, um eine Lamportuhr nachträglich aufzubauen. Bei Traces vom Typ *spawned* wird neben dem Standardprozedere auch ein Eintrag für den gestarteten Prozess getätigt. Hierzu wird ein zusätzlicher Eintrag vom Typ *spawnee* erstellt, dessen Lamportuhr abhängig vom erzeugenden Aktor erstellt wird. Der gestartete Aktor wird somit wie ein Empfänger einer Nachricht behandelt. Die Prozessnummer entspricht entsprechend dem *to*-Eintrag des startenden Aktor und als *from* wird die Prozessnummer des startendem Aktor eingetragen. In einigen Fällen kann es vorkommen, dass beim einlesen einer empfangenen Nachricht, der Sender nicht ermittelt werden kann. Ist dies der Fall so wird *UNKNOWN* für den Sender hinterlegt.

Beim Einlesen der Tracingdaten werden zudem die mitgelieferten Aktorinformationen genutzt, um die Metadatentabelle zur Zuordnung von Prozessnummer, Name, Modul und Knoten zu befüllen. Immer wenn ein Eintrag aus der Aufzeichnung eingelesen wird, wird zusätzlich geschaut, ob es bereits einen entsprechenden Eintrag in der Metadatentabelle gibt und dieser ggf. angelegt bzw. um noch nicht vorhandene Informationen erweitert werden muss. Beim Senden von Nachrichten per Namen kann es vorkommen, dass die Prozessnummer des Empfängers nicht ermittelt wurde. Dies kann auftreten, wenn die Aufzeichnung mittels TTB stattfand oder die Nachricht an einen entfernten Knoten versendet wurde. In einem solchen Fall wird zunächst der Eintrag des Prozessnamens für den aufgerufenen Knoten in der Metadatentabelle getätigt. Beim *receive* ist die Prozessnummer des Empfängers bekannt und sie kann dem Namen des Prozesses zugeordnet werden. Um zu erkennen, dass diese Zuordnung noch gemacht werden muss, wird der Name des Empfängers beim Senden so hinterlegt, dass per angereicherter Referenz auf sie zugegriffen werden kann. Wird der Eintrag beim einlesen des *receive* gefunden, so ist bekannt, dass die Information entsprechend noch angereichert werden muss. Unter besonderen Umständen kann es vorkommen, dass der Eintrag für einen Prozess auf einem Knoten zweimal in der Metadatentabelle vorliegt, weil diese beiden Einträge nicht einander zugeordnet werden konnten. Dies ist der Fall, wenn für einen Aktor aus einem Eintrag nur dessen Prozessnummer bekannt ist und von einem anderen Eintrag vom Typ *send* nur dessen Name. Eine direkte Verbindung der beiden Einträge ist in einem solchen Fall nicht möglich. Erst wenn der Eintrag mit der Prozessnummer um einen Namen oder der mit dem Namen um die Prozessnummer erweitert wird, kann eine Zuordnung stattfinden. Damit in einem solchen Fall die Zusammenführung gelingt, wird bei jedem Erweitern überprüft, ob es den Eintrag schon einmal gibt. D.h. wird eine Prozessnummer um einen Namen erweitert, wird zunächst geschaut ob es den Eintrag für den Namen schon gibt. Und umgekehrt wird bei der Erweiterung des Namens, um die Prozessnummer zunächst geschaut, ob es die Prozessnummer bereits gibt. Dies geschieht jeweils unter Berücksichtigung des Knotens auf dem sich der Aktor befindet. Wird

ein bereits vorhandener Eintrag gefunden, so werden die beiden Einträge zusammengelegt.

Bei allen Aufzeichnungen vom Typ *send* wird zudem in der Datenbank als Empfänger die zugehörige Prozessnummer eingetragen. Ist jedoch beim Auslesen der Datei nur der Name bekannt, so kann der Empfänger nicht eingetragen werden. In einem solchen Fall wird der Sendevorgang festgehalten und sobald der zugehörige *receive*-Eintrag eingelesen wird, wird der Eintrag um die Prozessnummer erweitert. Sollten am Ende der Abarbeitung der Tracedatei noch Einträge von Sendevorgängen ohne Prozessnummer vorliegen, so wurde hier kein entsprechendes *receive* gefunden und konnten somit nicht aufgelöst werden. Ist dies der Fall, wird zunächst überprüft, ob es in der Metadatentabelle einen Eintrag gibt über den der Name einer Prozessnummer zugeordnet werden kann. Ist dies erfolglos, so kann der Name nicht in die zugehörige Prozessnummer aufgelöst werden. Damit der Eintrag dennoch stattfinden kann, wird eine künstliche Prozessnummer generiert und der Trace mit dieser Nummer in der Tracingtabelle abgelegt. Gleichzeitig wird diese Nummer auch in der Metadatentabelle hinterlegt. Sollte dies mehrere Aufrufe betreffen, die alle denselben Prozess mit dem selben Namen aufgerufen haben, so verwenden alle dieselbe künstliche Prozessnummer.

Aufbau der Tabellen

Nachdem der Vorgang der Vorverarbeitung näher gebracht wurde, soll an dieser Stelle definiert werden, in welches Format die Daten transformiert werden. Die Daten werden in zwei verschiedene Tabellen untergebracht. Zum einen das Tracing als zeitbasierter Datensatz und zum anderen in eine Tabelle, die Metadaten zu den verschiedenen Aktoren bereitstellt.

Metadaten

Die Metadaten enthalten allgemeine Informationen zu den einzelnen Aktoren und bestehen aus den folgenden Feldern:

pid Beschreibt die Prozessnummer eines Aktors.

node Der Knoten auf dem sich der Aktor befindet.

name *(Optional)* Der Name eines Aktors. Ist nur ausgefüllt wenn ein Aktor benannt ist.

module *(Optional)* Das Modul, in dem ein OTP basierter Aktor definiert wurde. Ist nur ausgefüllt, wenn das Modul ermittelt werden konnte.

Indexiert wird die Tabelle zusätzlich mit zwei kombinierten Indexen. Der eine Index enthält die Felder *pid* sowie *node* und der andere *name* sowie *node*. Wie zu sehen ist, wird in jedem Index

der Knoten auf dem sich ein Prozess befindet mit aufgeführt. Das hängt damit zusammen, dass sich sowohl die Prozessnummer als auch der Name eines Prozesses auf verschiedenen Knoten wiederholen können. Für das Auslesen in der Visualisierung wird nur der erste Index (*pid* und *node*) benötigt. Der zweite Index (*name* und *node*) wird beim Preprocessing als zusätzlicher Index eingesetzt. Denn hier kann es kurzfristig vorkommen, dass nur der Name eines Aktors aber nicht dessen Prozessnummer bekannt ist.

Tracingdaten

Die Tracingtabelle enthält die einzelnen Aufzeichnungen, die getätigt wurden in angereicherter Form. D.h. die Daten enthalten neben der Aufzeichnung auch potentiell ermittelte Aktormetriken. Die genaue Struktur der einzelnen Daten innerhalb der Tracingtabelle ist davon abhängig was aufgezeichnet wurde. Allerdings gibt es auch bestimmte Felder, die Bestandteil aller Datensätze sind:

time Der Zeitstempel ,der bei der Aufzeichnung ermittelt wurde.

type Die Art der Aufzeichnung. Kann die Werte *send*, *receive*, *spawned*, *spawnee* oder *exit* annehmen. Wobei es sich bei *spawnee* nicht direkt um einen aufgezeichneten Wert handelt, sondern um einen durch den Preprocessor generierten.

node Der Knoten auf dem die Aufzeichnung stattfand.

pid Die Prozessnummer von dem Aktor, bei dem die Aufzeichnung stattfand.

lamport_clock Die Lamportuhr des Datensatzes. Diese wird von ErlViz zum vorsortieren der Daten genutzt.

Bis auf *lamport_clock* sind all diese Felder indexiert. So handelt es sich bei *time* um den InfluxDB Zeitstempel. Die Felder *type*, *node* und *pid* sind als Tags abgespeichert. Da *type* die Art der Aufzeichnung beschreibt, definiert dieser auch, welche Felder ausgefüllt werden. Diese sollen im Folgenden näher beschrieben werden.

Aufzeichnungen vom Typ *send* oder *receive* können neben den oben genannten Felder auch die folgenden Felder enthalten:

msg Nachricht, die zwischen den Aktoren versendet wurde.

nonce *(Optional)* Die Referenz, die bei angereicherten Nachrichten mit übermittelt wird. Wird nur aufgeführt, wenn es sich um eine Nachricht zwischen zwei aufgezeichneten Aktoren handelt.

queue_length *(Optional)* Die Länge des Posteingangs eines Aktors. Ist nur vorhanden, wenn dieser aufgezeichnet wurde.

state *(Optional)* Der Zustand eines Aktors. Ist nur vorhanden, wenn dieser aufgezeichnet wurde

Bei Aufzeichnungen von dem Typ *send* gibt es noch ein zusätzliches Feld:

to Die Prozessnummer des Aktors, zu dem die Nachricht gesendet wurde.

Entsprechend gibt es bei Aufzeichnungen von dem Typ *receive* zusätzlich das Gegenstück:

from *(Optional)* Die Prozessnummer des Aktors, von dem die Nachricht erhalten wurde. Wird nur ausgefüllt wenn der sendende Aktor aufgezeichnet wurde.

Aufzeichnungen vom Typ *spawned* haben zusätzlich folgendes Feld:

to Prozessnummer des Aktors, der gestartet wurde.

Als Gegenstück dazu gibt es bei Einträgen vom Typ *spawnee* folgendes Feld:

from Prozessnummer des Aktors, der den Aktor gestartet hat.

Aufzeichnungen vom Typ *exit* haben zusätzlich folgendes Feld:

reason Die aufgezeichnete Begründung, warum ein Prozess beendet wurde.

4.2 Visualisierung

Die Visualisierung besteht aus zwei technischen Hauptkomponenten. Zum einen ein webbasiertes Frontend und zum anderem dem Backend, über welches das Frontend mittels REST die Daten bezieht. Die Daten werden hierbei aus zwei Quellen bezogen. So gibt es zum einen eine InfluxDB, in der die Tracingdaten gespeichert sind und zum anderen eine MongoDB, aus der die Metadaten für die Aufzeichnungen bezogen werden. Darunter fallen allgemeine Informationen über die Aktoren, wie z.B. der Name, das Modul mit dem ein Aktor implementiert wurde oder der Knoten, auf dem sich ein Aktor befindet. Nachdem in Abschnitt 3.4 auf die verschiedenen Visualisierungen konzeptionell eingegangen wurde, wird im Folgenden vor allem auf die technische Umsetzung der Visualisierung eingegangen. Neben der allgemeinen Generierung des Graphen, wird auch auf zwei Funktionalitäten näher eingegangen, da diese Besonderheiten aus dem Erlangkontext berücksichtigen um ihre Funktionalität zu erfüllen.

4.2.1 Darstellung der Aktorkommunikation

Zur Darstellung der Aktorkommunikation wird der Graph per REST aus dem Backend geladen und anschließend im Frontend visualisiert. Im Folgenden wird zunächst der Ladevorgang des Graphen beschrieben und anschließend näher auf die Visualisierung eingegangen.

Laden des Graphen

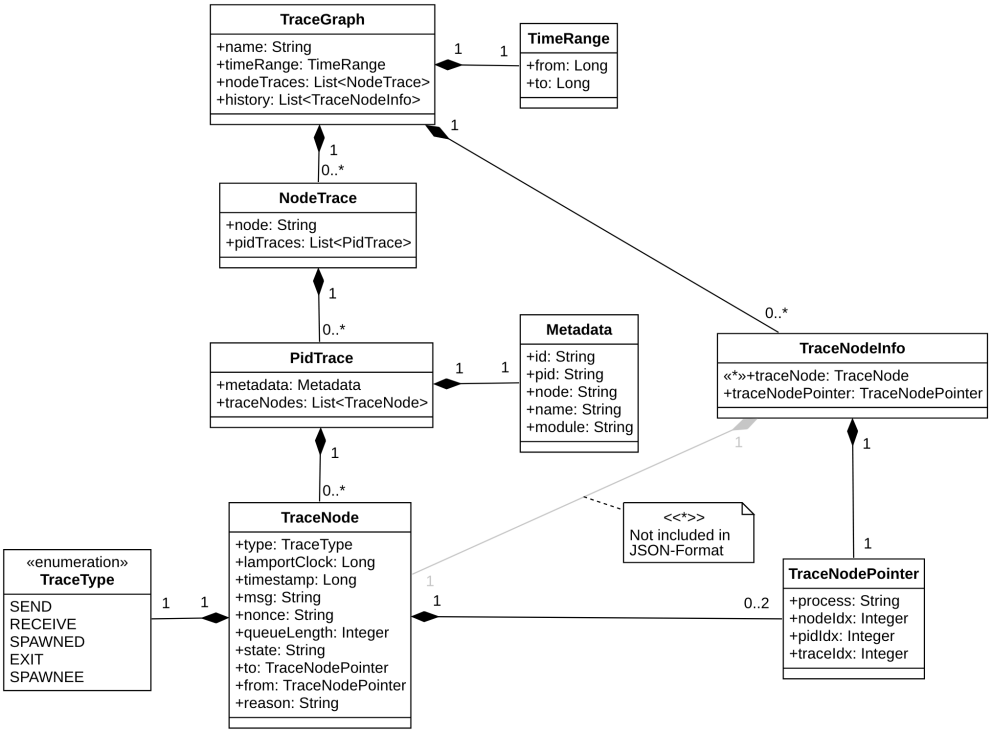


Abbildung 4.2: Datenstruktur zur Darstellung des Graphen

Beim Laden des Graphen wird Backendseitig der Graph aufgebaut. Dazu führt dieses die Traceinformationen mit den Metadaten zusammen. Hierbei werden die einzelnen Aufzeichnungen nach dem Knoten sowie dem Aktor bei dem sie aufgezeichnet wurden sortiert. Die einzelnen Aufzeichnungen werden zudem untereinander nach der vorliegenden Lamport Clock sortiert. Außerdem wird eine History der einzelnen Aufzeichnungen generiert. Diese enthält sämtliche Aufzeichnungen, in der Reihenfolge wie diese aufgetreten sind. Dies geschieht in diesem Fall primär über den Zeitstempel. Heraus kommt eine Datenstruktur wie sie in Abb. 4.2 zu sehen ist. Diese ist stark auf die Nutzung durch ein Frontend optimiert. Auf oberster Ebene der Datenstruktur liegt der *TraceGraph*. Hier gespeichert ist der Name der Aufzeichnung

sowie der Zeitbereich in dem der Graph liegt. Darunter befinden sich zwei weitere große Datenstrukturen. Zum einen eine Liste von *NodeTrace* sowie die History der Ereignisse als Liste vom Typ *TraceNodeInfo*.

In *NodeTrace* befinden sich die Aufzeichnungen zu einem Knoten. Dort befindet sich wiederum eine Liste der Aufzeichnungen der einzelnen Aktoren. Die Aufzeichnungen für Aktoren werden in *PidTrace* festgehalten. Hier werden auch die Metadaten zu einem Aktor hinterlegt. Diese enthalten Daten wie die Prozessnummer, den Namen oder das Modul in dem ein Aktor umgesetzt wurde. Außerdem enthält *PidTrace* eine Liste mit den Ereignissen, die für den Aktor aufgezeichnet wurden. Diese sind sortiert nach der vorliegenden Lamportuhr. Die konkrete Aufzeichnung ist vom Typ *TraceNode*. Da dies die konkrete Aufzeichnung ist, sind hier alle Daten enthalten, die aus der InfluxDB stammen. Ausgenommen hiervon sind redundante Informationen, wie der Knoten und der Prozess in dem aufgezeichnet wurde, da dies bereits aus den Datentypen *NodeTrace* und *PidTrace* ersichtlich ist. Welche Felder im Detail Daten enthalten ist hierbei vom Typ des *TraceNode* abhängig (siehe auch 4.1.2). Bei bestimmten Typen sind die Felder *from* oder *to* ausgefüllt. Dies bedeutet, dass die Ereignisse eintreffende oder ausgehende Nachrichten repräsentieren. Die Darstellung hiervon erfolgt über einen *TraceNodePointer*. Hierbei handelt es sich um einen erste Methode, um den Graphen effizienter im Frontend verarbeiten zu können. Denn diese zeigen mittels verschiedener Indexe auf die Knoten. Dadurch kann die Information auch ohne Vorhandensein von direkten Referenzen effizient mittels JSON übermittelt werden. So enthält der Pointer die Felder *nodeIdx*, *pidIdx* und *traceIdx*. Diese werden genutzt, um den referenzierten Knoten ermitteln zu können. Hierbei ist *nodeIdx* der Index des Erlangknotens auf dem das referenzierte Ereignis ermittelt wurde. Dieser entspricht dem Index innerhalb der *nodeTraces*-Liste in der *TraceGraph*-Klasse. *pidIdx* ist wiederum der Index in der *pidTraces*-Liste in *NodeTrace* und *traceIdx* der Index in der *traceNodes*-Liste in *PidTrace*. Sollte der referenzierte Knoten unbekannt sein, so haben alle Indexe den Wert -1. Dies kann der Fall sein, wenn eine Nachricht von einem Aktor eingetroffen oder an einen Aktor gesendet wurde, der nicht aufgezeichnet wurde oder nicht in dem betrachteten Bereich liegt.

TraceNodeInfo beschreibt ein Ereignis in der History des Graphen. Diese wird auch Backendseitig bereits zur Verfügung gestellt. Ein *TraceNodeInfo*-Objekt enthält eine Referenz zu dem entsprechenden Knoten im Graphen. Diese ist jedoch nur eine Hilfsstruktur, die zum Aufbau der History benötigt wird. Sie liegt nur im Backend vor und wird nicht an das Frontend übertragen. Da Referenzen in JSON nicht dargestellt werden können, hätte somit nämlich eine Kopie übertragen werden müssen. Damit das Frontend dennoch die Referenzierung der einzelnen

Knoten verarbeiten kann, wird ein *TraceNodePointer* bereitgestellt, der den konkreten Knoten referenziert. Darüber kann das Frontend die benötigten Informationen zur Darstellung der History beziehen.

Nachdem das Frontend, den Graphen vom Backend bezogen hat, nutzt dieses die Datenstruktur zum Aufbau des Graphen und der History. Die History nutzt den vorliegenden *TraceNodepointer*, um die benötigten Informationen heranzuziehen. Dieser wird allerdings nicht nur zum heranziehen der Informationen genutzt, sondern auch um festzustellen, ob ein Knoten angewählt oder ausgeblendet ist. Die History wird zudem herangezogen, um den Graphen aufzubauen. So wird zum Aufbau über die History iteriert und so die einzelnen Elemente hinzugefügt. Dadurch kann der Graph von oben nach unten aufgebaut werden. Dies macht es sehr einfach die Position auf der Y-Achse zu ermitteln, denn mit jedem weiteren Eintrag, kann der aktuelle Wert um einen konstanten Wert erhöht werden. Die Position auf X-Achse wird mittels der Position des Erlangknotens in der *nodeTraces*-Liste und der Position des Aktors innerhalb der *pidTraces*-Liste ermittelt. Die verschiedenen Werte in *TraceNode* werden genutzt, um die einzelnen Knoten im Graphen mit Informationen anzureichern, so sind die Informationen sichtbar wenn sich der Cursor über einem Knoten befindet. Der Graph wird hierbei Frontendseitig als SVG erzeugt. Dadurch können die einzelnen grafischen Bestandteile des Grafen sehr einfach generiert werden. Dies liegt zum einen daran, dass die Elemente die dargestellt werden sollen, deskriptiv beschrieben werden können und zum anderen an der nativen Unterstützung von SVG-Tags in HTML 5.

Blockierende Nachrichten hervorheben

Synchrone Nachrichten in OTP stellen eine Besonderheit dar. Denn sie sind nicht über ein einziges Ereignis nachvollziehbar. Stattdessen müssen mehrere Informationen kombiniert werden. Bei dieser Art der Nachrichtenübertragung handelt es sich um eine Besonderheit, die von OTP bereitgestellt wird. Denn diese blockieren den aufrufenden Aktor und warten auf eine Antwort von dem aufgerufenen Aktor.

Erlang bietet von sich aus nicht die Möglichkeit synchrone Nachrichten zwischen Aktoren zu verschicken. Deshalb sind synchrone Aufrufe in OTP mittels mehrerer asynchroner Nachrichten implementiert worden. Dies hat auch zur Folge, dass zur vollständigen Betrachtung eines synchronen Aufrufes in der Visualisierung mehrere asynchrone Nachrichten betrachtet werden müssen. Ein vollständiger synchroner Aufruf enthält die folgenden asynchronen Aufrufe:

1. Senden der Anfrage

2. Erhalt der Anfrage
3. Senden der Antwort
4. Erhalt der Antwort

Bei synchronen Nachrichten innerhalb von OTP handelt es sich um *Calls*. Diese sind innerhalb der Aufzeichnung daran zu erkennen, dass die Nachricht das Schlüsselwort *$gen_call* enthält. Die Zusammengehörigkeit von Nachrichten kann über eine in der Nachricht enthaltene Referenz ermittelt werden. Im Folgenden soll dies anhand von einem Beispiel verdeutlicht werden.

Nehmen wir an der Aktor mit der Prozessnummer *<7598.76.0>* sendet einen Call an den Aktor mit der Prozessnummer *<7598.117.0>*. Dann könnte eine versendete Nachricht wie folgt aussehen:

```
{'$gen_call', {<7598.76.0>, #Ref
   <7598.3619084660.2335965188.183187>}, {test_call,internal_msg}}
```

Hier ist auch direkt das Schlüsselwort *$gen_call* innerhalb der Nachricht zu sehen. Dahinter befinden sich Informationen für den Empfänger, die diesem helfen, eine Antwort an den Sender zu senden. Das ist zum einen der Aktor der die Nachricht versendet hat (*<7598.76.0>*), und zum anderen eine Referenz, die für zum Antworten verwendet wird (*#Ref<7598.3619084660.2335965188.183187>*). Diese wird vom Sender benötigt, um bei einer Antwort eindeutig erkennen zu können, dass diese von dieser Anfrage stammt. Hinter dieser Information befindet sich die eigentliche Nachricht, die versendet wurde (*{test_call,internal_msg}*).

Nachdem der Aktor *<7598.117.0>* die Anfrage verarbeitet hat, würde dieser eine Antwort an den Sender schicken. Dies könnte dann wie folgt aus sehen:

```
{#Ref<7598.3619084660.2335965188.183187>, {test_result, internal_msg
   }}
```

Auch hier lässt sich direkt die Referenz wiederfinden, die beim Versenden der initialen Nachricht übermittelt wurde. Auffällig ist allerdings, dass hier kein direktes Indiz aufgeführt ist, dass auf eine OTP Nachricht hindeuten würde. So ist hier nirgends direkt aufgeführt, dass es sich um die Antwort auf einen *$gen_call* handelt. Stattdessen lässt sich das nur aus der mitgelieferten Referenz innerhalb der Nachricht schließen.

Die Analyse und Suche nach der Referenz passiert innerhalb von ErlViz vollständig Frontendseitig. Dazu wird schon beim Rendern des Graphen für jeden Knoten überprüft, ob es sich

um das Versenden einer Call Nachricht oder das Empfangen einer Antwort handelt. Handelt es sich bei einem Ereignispunkt um das Senden einer Call Nachricht, so wird der Aufruf direkt mit seiner Referenz hinterlegt. Sobald auf eine empfangene Nachricht gestoßen wird, die potentiell die Antwort eines Calls sein könnte, so wird bei den hinterlegten Calls für den betroffenen Aktor überprüft, ob es einen entsprechenden Aufruf gab. Ist dies der Fall, so wird zwischen den beiden Punkten eine Linie gerendert, die auf die Konfiguration zur Visualisierung von blockierenden Aufrufen lauscht. Wird die Konfiguration angewählt so wird diese Linie direkt eingefärbt bzw. die Färbung wieder entfernt, wenn die Auswahl abgewählt wird.

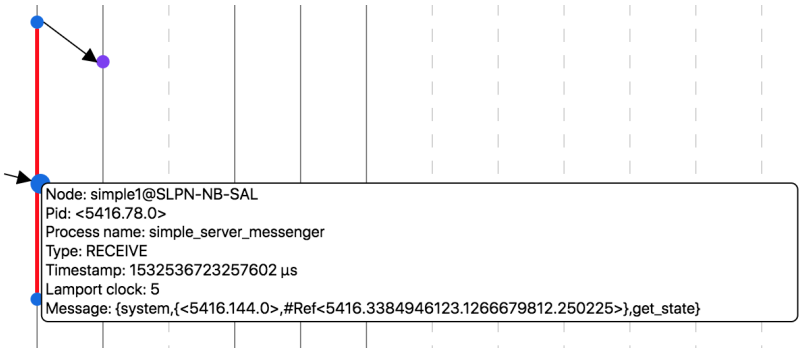


Abbildung 4.3: Hervorheben von Call Aufruf, mit Systemnachricht zwischendurch

Nun wurde bisher davon gesprochen, dass Aktoren, die einen synchronen Aufruf durchführen, blockiert sind. Hiervon gibt es jedoch eine Ausnahme. So gibt es in OTP spezielle Systemnachrichten. Diese werden von einem in OTP umgesetzten Aktor auch dann verarbeitet, wenn der Aktor eigentlich blockiert ist (siehe Abb. 4.3).

Isolierte Aktoren ausblenden

Sollte die Aufzeichnung mittels TTB stattgefunden haben, so wurden nicht nur Aktoren aus dem OTP-Kontext aufgezeichnet, sondern alle im System befindlichen Aktoren. Dadurch handelt es sich bei diesem Szenario bei isolierten Aktoren nicht nur um welche, mit denen nicht in dem Zeitraum kommuniziert wurde, sondern vor allem um Hintergrundprozesse, die innerhalb von Erlang genutzt werden. Denn die Nachrichten zwischen diesen Aktoren wurden nicht angereichert und so kann nicht ermittelt werden, wie eine Kommunikationslinie zwischen diesen Aktoren zu ziehen ist. Da das Ziel des Verfahrens aber vor allem darin liegt, das Verhalten der Anwedungslogik zu ermitteln, spielen diese in der Regel eine untergeordnete Rolle und können somit vernachlässigt werden.

4.3 Timadorus

Timadorus ist ein Projekt an der HAW Hamburg mit dem Ziel Technologien für MMORPGs zu entwickeln [Tim17]. In diesem Kontext ist auch eine Umgebung zur Simulation von Online Spielen entstanden. Da die Experimentierumgebung in dieser Umgebung eingebettet wurde, sollen die genutzten Aspekte im Folgenden vorgestellt werden. Innerhalb der Experimentierumgebung werden insbesondere das QuP-System zur Datenhaltung [BGL14] sowie der Load Balancer [BAW+16] als Untersuchungsgegenstand genutzt. Bei den restlichen Elementen handelt es sich um spezielle Testimplementierung zur Simulation einer Spielwelt. Diese sind in [All17a] bereits beschrieben und werden im Folgenden nicht näher betrachtet.

4.3.1 QuP

QuP dient vor allem zur Datenhaltung der Spielwelt. QuP nutzt zum verwalten der Daten ein Spatial Publish Subscribe, um die Area of Interest zu verwalten [BGL14]. Konkret bedeutet das, dass man mithilfe eines Clients einen räumlichen Bereich der Spielumgebung abonnieren kann und dann über jegliche Veränderungen in diesem Bereich informiert wird. Neben der Möglichkeit räumliche Bereiche zu abonnieren, können auch einzelne Objekte abonniert werden.

Auch in der Experimentierumgebung kommt QuP zum Einsatz. Neben dem Server werden jeweils spezielle Clients in die Spielserver sowie dem Load Balancer integriert. Hierbei wurde der Client innerhalb des Spielservers auf das nötigste beschränkt, um die Fähigkeiten von ErlViz nachweisen zu können. So wurde der Einsatz des Octree entfernt und es wird auch das Publish Subscribe von QuP nicht genutzt, um möglichst wenig Einfluss durch QuP während der Tests zu haben.

4.3.2 Load Balancer

Der Load Balancer ist dafür zuständig, die Spieler auf die verschiedenen Spielserver zu verteilen. Darüber hinaus sammelt der Load Balancer die ganze Zeit über Statistiken darüber, wie die Spieler in der Spielwelt verteilt und die einzelnen Spielserver ausgelastet sind. Auf Basis dieser Informationen werden nicht nur die Spieler über die Spielserver verteilt, sondern auch umverteilt wenn ein Ungleichgewicht in der Verteilung entsteht [BAW+16]. Um die Spieler über die Spielserver zu verteilen kommt die avatarbasierte Lastverteilung zum Einsatz. Um ein Ungleichgewicht zu erkennen und anschließend die Spieler umzuverteilen, können zwei Verfahren eingesetzt werden. Das eine ist eine periodische Neuberechnung für die einzelnen Spieler und dass andere ist die zellbasierte Migration (siehe auch Abschnitt 2.1.1 und 2.1.2).

4.4 Experimentierumgebung

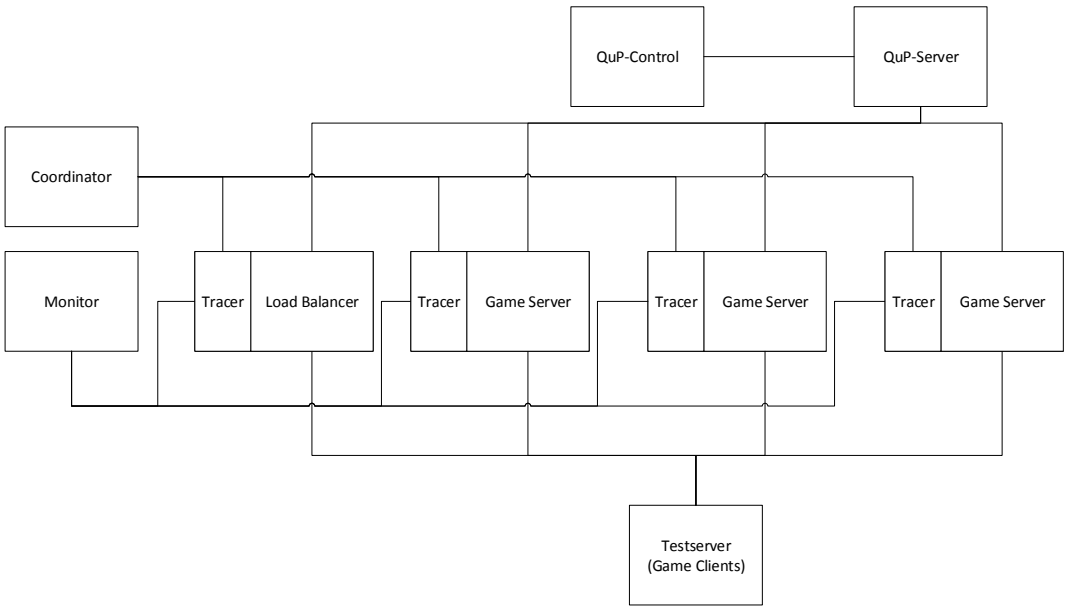


Abbildung 4.4: Architektur des Timadorus Systems, während der Experimente

Wie in Abschnitt 4.3 bereits erwähnt finden die Experimente innerhalb von Timadorus statt. Deshalb werden alle Bestandteile des Timadorus Backendsystems gestartet, die zur Simulation einer Spielwelt benötigt werden. Diese sind:

- Load Balancer
- QuP-Control
- QuP-Server
- Spielserver

Wobei es sich bei den Spielservern, nicht um voll funktionsfähige Spielserver handelt, sondern um Server die vor allem dazu dienen, das Verhalten der Avataren zu simulieren. Dieser wurde speziell zur Untersuchung von dynamischen und verteilten Spielumgebungen entwickelt [All17a]. Neben dem Timadorus Systemen wurden auch die von ErlViz benötigten Komponenten gestartet, die zur Aufzeichnung der Aktoren benötigt werde. Das sind als eigene Knoten:

- ErlViz Coordinator
- ErlViz Monitor

Neben den beiden Knoten muss auf den beobachteten Knoten auch jeweils der ErlViz Tracer gestartet werden. In dieser Umgebung werden die Tracer auf dem Load Balancer sowie auf den einzelnen Spielservern gestartet. Die Tests werden auf einem separatem Testserver gestartet. Dieser sammelt Statistiken über zugewiesene Spieler, die Zeiten, die zur Zuweisung der Spieler benötigt wurden sowie die Speicherauslastung. Zudem werden auf dem Testserver die Spielclients simuliert und die einzelnen Avatare von hier aus den Spielservern zugewiesen. In [All17a] wird dies näher beschrieben. Um einen geeigneten Spielserver für die einzelnen Avatare zu finden, kommuniziert der Testserver zunächst mit dem Load Balancer und weißt die Avatare dem berechneten Spielserver zu. Anschließend kommuniziert der Spielserver direkt mit dem Load Balancer, um diesem seinen Zustand in regelmäßigen Abständen mitzuteilen. Stellt der Load Balancer eine ungünstige Verteilung der Avatare fest, so weisst dieser den Spielserver an bestimmte Avatare anderen Spielservern zu übergeben. Hierzu muss der Spielserver wiederum mit dem Testserver kommunizieren. Denn dieser teilt im Grunde dem Spielclient nur mit, an welchem Spielserver sein Avatar umverteilt werden soll. Um die Umverteilung abzuschließen, teilt der Spielclient wiederum dem neu zugewiesenen Spielserver mit, dass er ihm zugewiesen sein möchte. Neben der Kommunikation wegen der Verteilung der Avatare bezieht der Testserver auch Statistiken von den Spielservern (Abb. 4.4).

4.5 Fazit

Mit der Implementierung konnte ein System realisiert werden, dass die Konzepte des ErlViz Systems umsetzt (Kapitel 3). Hiezu wurde ein Tracer für den Einsatz innerhalb von Erlang/OTP entwickelt. Zusätzlich wurde auch die Vorverarbeitung und die Visualisierung umgesetzt. Hierbei kamen bestimmte Datenbanktechnologien zum Einsatz, die auf zeitbasierte Daten ausgelegt sind, um so einen effizienten Zugriff zu gewährleisten. Dieses System wurde innerhalb von Timadorus eingebettet, um dort Versuche durchführen zu können. Somit konnte gezeigt werden, dass die ausgearbeiteten Konzepte realisierbar sind. Im Folgenden Kapitel soll die Effizienz des Konzepts anhand der Umsetzung evaluiert werden.

5 Untersuchen der Anwendung

Um die Effektivität des Verfahrens nachzuweisen, wurde dieses in verschiedenen Experimenten untersucht. Hierbei wurden vor allem zwei Aspekte betrachtet. Zum einen die Performance des Verfahrens und zum anderen die Effektivität für den Endnutzer. Zur Evaluierung der Performance, wurde betrachtet, wie effizient das avatarbasierte Lastverteilungsverfahren [BAW+16] mit und ohne Tracing arbeitet. Mit effizient ist in diesem Fall gemeint, wie sehr sich die Berechnungszeit zum ermitteln des Spielserver verändert. Zudem wurde betrachtet, wie sich das Verfahren auf die Speicherbelastung der Anwendung auswirkt. Die Evaluierung der Effektivität für den Endnutzer, wurde mithilfe der Problemstellung aus Abschnitt 2.1 ermittelt. Hier wurde betrachtet, ob es möglich ist mit der Visualisierung festzustellen, wie bestimmte Probleme entstanden sind und wie sich das System allgemein verhält.

5.1 Experimente

In den Experimenten werden vor allem zwei Aspekte betrachtet. Zum einen wie sich die Aufzeichnung auf die Performance des Systems auswirkt und zum anderen wie gut das System geeignet ist, um das Verhalten des Systems zu analysieren. Eine Analyse der Performance ist ein gängiges Verfahren, dass bereits aus anderen Arbeiten bekannt ist [Sai05]. Im Folgenden sollen die Experimente näher vorgestellt werden. Die Erhebung der Daten läuft bei allen Tests gleich. Diese kann in zwei Phasen unterteilt werden. Zum einen die Initialisierung und zum anderen dem eigentlichen Testlauf. Die Initialisierung beschreibt die Phase in der zunächst nur die initialen Avatare einem Spielserver zugeordnet werden, die Avatare sich aber noch nicht bewegen. Während des Testlaufs werden die Avatare in Bewegung versetzt und in regelmäßigen Abständen neue Avatare der Umgebung hinzugefügt. Die genaue Konfiguration sieht wie folgt aus:

- 1200 Avatare
- davon werden 900 bereits initial zugewiesen
- die restlichen 300 werden in einem Abstand von einer Sekunde einem Server zugewiesen

- Präzision der Heat Map: 10
- Dauer des Tests: 5 Minuten

Die Bewegung der Avatare erfolgt in einer Mischung aus Random Way Point [JM96] und Hot Points All [RMOV07]. Hierbei werden 3 Hot Spots definiert, die von jedem Avatar zwischendurch angesteuert werden. Zusätzlich werden für jeden Avatar 5 zufällige Ziele definiert, die diese individuell ansteuern können. Die Bewegung der Avatare erfolgt in festen Intervallen von 350 ms. Hierbei bewegt sich jeder Avatar um 0,001 innerhalb des Bereichs der Spielwelt. Diese Art der Bewegung wurde in [All17a] entwickelt. Dort ist auch beschrieben nach welchem Schema die Bewegung der Avatare erfolgt.

Bei allen Tests findet eine periodische Neuberechnung in Abständen von 15 Sekunden statt [BAW+16]. Bei der visuellen Auswertung des Systems wird neben der periodischen Neuberechnung auch die zellbasierte Migration durchgeführt [All17b]. In beiden Fällen findet die Neuberechnung in einem Abstand von 15 Sekunden statt.

5.1.1 Performanceanalyse

Zur Untersuchung wie sich die Aufzeichnung auf die Performance auswirkt, wurde vor allem auf die Tests aus [All17b] zurückgegriffen. Über diese kann ermittelt werden, wie lange die Berechnung für eines Spielservers für einen Avatar dauert. Dieser Wert soll als Indikator für die Performance des Systems dienen. Hierbei wird zum einen betrachtet wie sich die Zeiten bei einem Anstieg der Avatare im System verändern aber auch wie sich die Rechenzeit bei einem ruhigen System verhält, in dem sich die Avatare nicht bewegen. Das ruhige System liegt während der Initialisierungsphase vor. Während des eigentlichen Testlaufs bewegen sich die Avatare nach dem beschriebenen Muster und es werden neue Avatare im Laufe der Zeit hinzugefügt. Die Folgenden Testszenarien wurden durchgeführt:

1. Ohne Tracing
2. Tracing ohne Aktormetriken
3. Tracing mit Aktormetriken
    a) mit Länge des Posteingangs
    b) mit Zustandsinformationen
    c) mit Länge des Posteingangs und Zustandsinformationen

4. Tracing mittels TTB
    a) ohne Auflösen der Prozessnamen
    b) mit Auflösen der lokalen Prozessnamen

1. dient als Referenz, um zu ermitteln, wie sich das System ohne Tracing verhält. Bei allen Tests mit Tracing wurde die integrierte Lösung verwendet, insofern es nicht explizit anders angegeben wurde. Somit wurde in 4. als einzige Ausnahme TTB verwendet. Das sammeln von Prozessinformationen in 3. wurde mit verschiedenen Sampling Raten getestet. So wurde jedes Szenario mit einer Samplingrate von 1, 5, 10 und 15 durchgeführt. Damit soll ermittelt werden, wie sich verschiedene Raten auf das Verhalten auswirken. Die Experimente unter 4. liefen allesamt ohne sammeln von Prozessinformationen.

Neben der Betrachtung des Laufzeitverhaltens wurde auch die Speicherauslastung betrachtet. Dazu wurde bei allen Tests in regelmäßigen Abständen ermittelt, wie die Speicherauslastung der einzelnen Knoten ist. 1. dient auch hier als Referenz für das Verhalten ohne Tracing. Bei den restlichen soll betrachtet werden, ob sich die Speicherauslastung im Laufe der Zeit verändert. Das könnte z.B. daran liegen, dass die Aufzeichnung nicht in der Lage ist, die aufgezeichneten Daten schnell genug in die Datei zu schreiben.

5.1.2 Visualisierung

Beim Untersuchen der Visualisierung soll vor allem Ermittelt werden, ob das System geeignet ist, um das Verhalten von verteilten Aktorsystemen zu analysieren. Dazu wird das Tracing mit Aktormetriken durchgeführt, die sowohl die Länge der Posteingangs als auch die Zustände der Aktoren aufzeichnen. Hierbei wird eine Samplingrate von Fünf eingesetzt. Dies wird in zwei Testdurchläufen gemacht. Bei dem einen wird zur Migration der Avatare zur Laufzeit die periodische Neuberechnung und bei dem anderen die zellbasierte Migration genutzt. Hierbei soll die Problemstellung aus Abschnitt 2.1 untersucht werden. Als Ergebnisse werden die Erkenntnisse aus dieser Untersuchung präsentiert. Da in eine Beurteilung der visuellen Darstellung auch immer subjektive Eindrücke mit einfließen, wird weniger eine konkrete Beurteilung abgegeben, als vor allem eine Einschätzung wie groß der Mehrwert der entstandenen Erkenntnisse ist.

	Initialisierung (ms)	Testlauf (ms)	Testlauf 95er Perzentile (ms)	Testlauf 90er Perzentile (ms)	Geamt (ms)
Ohne Zusatzmetriken	263,7	1,7	0,1	-0,1	198,3
Queue 1	623	46,5	1,2	0,8	479,2
Queue 5	415	36,5	1,2	0,9	320,6
Queue 10	714,6	32,8	0,5	0,2	544,6
Queue 15	394,8	39,4	0,6	0,3	306,2
State 1	2661,4	23,9	1,6	1,3	1092,6
State 5	1431,1	73,6	6,1	0,6	1092,6
State 10	658,5	158,8	15,6	2,8	533,9
State 15	856,3	1,3	0,1	-0,1	643,1
Queue & State 1	4395,3	46	1,1	1	3310,7
Queue & State 5	697,1	85,3	25,7	11,5	544,9
Queue & State 10	1110,1	1,4	0	-0,2	833,6
Queue & State 15	536,4	83,5	14,9	4,4	423,5
TTB	201,7	1	0,1	-0,1	151,6
TTB Prozesse lokal aufgelöst	197,9	1,7	0,1	0	149

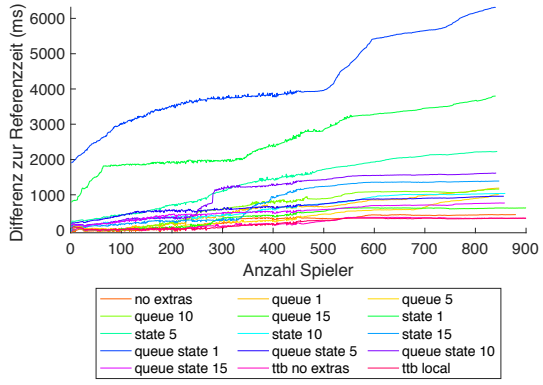
Tabelle 5.1: Durchschnittswerte der Differenz zur Zuweisung ohne Aufzeichnung, während der beiden Testphasen. Die Zahlen hinter den Namen stellen jeweils die Konfigurierte Aufzeichnungsrate dar (1 = jede Nachricht, 5 = jede 5. Nachricht etc.).

5.2 Auswertung der Ergebnisse

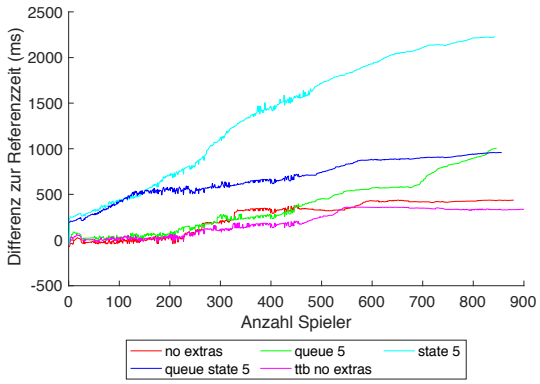
Im Folgenden werden die ermittelten Ergebnisse der Experimente analysiert. Dazu werden zunächst die Performancemetriken betrachtet und anschließend die Leistungsfähigkeit der Visualisierung beurteilt. Zur besseren Darstellung werden in Grafiken und Tabellen die Begriffe Queue, für die Länge des Posteingangs und State für den Zustand der Aktoren genutzt, wenn Aktormetriken aufgezeichnet wurden.

5.2.1 Performance

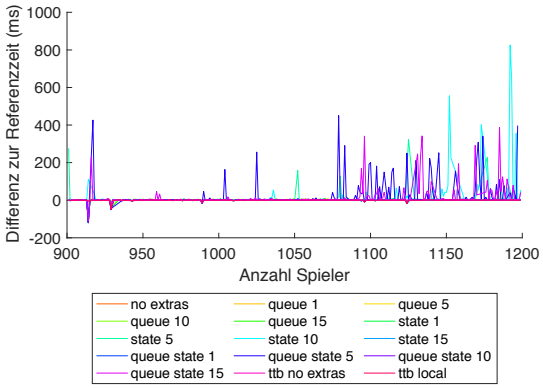
Die Beurteilung der Performance erfolgt in zwei Kategorien. So wird zum einen betrachtet, wie schnell der Load Balancer in der Lage ist, den Avataren einen Servern zuzuweisen. Dies wurde in Relation zur Zuweisung ohne Aufzeichnung gesetzt. Die zweite Kategorie die betrachtet wird, ist die zusätzliche Belastung des Arbeitsspeichers.



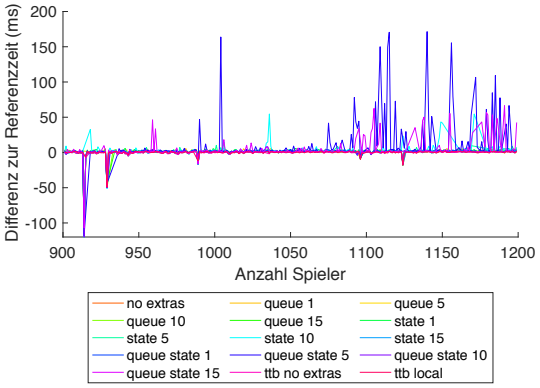
(a) Initiale Zuweisung (alle Verfahren)



(b) Initiale Zuweisung (ausgewählte Verfahren)



(c) Zuweisungen während des Testlaufs



(d) Zuweisungen während des Testlaufs (90er Perzentile)

Abbildung 5.1: Differenz der Zuweisungszeit mit verschiedenen Aufzeichnung zur Zuweisung ohne Aufzeichnung. Alle Darstellungen erfolgen mit einer 95er Perzentile insofern nicht anders angegeben.

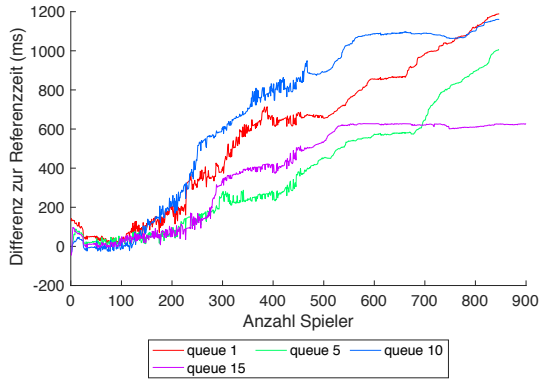
Geschwindigkeit

Die Aufzeichnung der Geschwindigkeit wurde zur besseren Übersicht in zwei Phasen betrachtet. Zum einen in die Initialisierung und zum anderen in den Testlauf. Während der Initialisierung wurde die ersten 900 Avatare den einzelnen Spielservern zugeordnet. Sie waren jedoch während dieser Phase noch nicht in Bewegung. Die Zuweisung der Avatare erfolgte hierbei nicht sequentiell, sondern es wurden alle Anfragen parallel gestellt. Erst mit Beginn des Testlaufs wurden die Avatare in Bewegung gesetzt. Hier wurden 300 weitere Avatare den Spielservern zugewiesen und die benötigte Zeit zur Berechnung aufgezeichnet. Dies geschah in einem Abstand von je einer Sekunde.

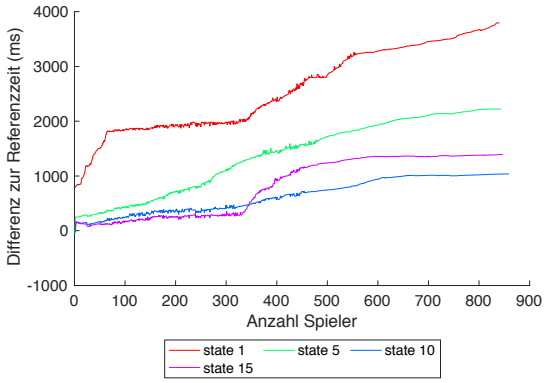
In Abb. 5.1 ist sichtbar, dass der Faktor, ob es sich um die Initialisierung oder den Testlauf handelt, einen deutlichen Unterschied macht. Bei allen Testszenarien ist zu sehen, dass die Berechnungsdauer während der initialen Phase länger ist als ohne Aufzeichnung. Allerdings ist der Grad der Ausprägung stark davon abhängig, in welchen Modus aufgezeichnet wurde. So hat die Zuweisung während der initialen Phase mit Aufzeichnung des Posteinganges und des Zustands bei jeder Nachricht im Durchschnitt ca. 4,4 Sekunden länger gedauert. Ohne Zusatzmetriken lag dieser Wert bei 263,7 Millisekunden (siehe Tabelle 5.1). In Relation zu der Dauer die Durchschnittlich ohne Aufzeichnung während der initialen Phase benötigt wurde (368,5 ms), ist das bei einer Aufzeichnung ohne Zusatzmetriken um den Faktor 1,72 länger und bei der Aufzeichnung mit zusätzlicher Aufzeichnung von Posteingang und Zustand der Faktor 12,9. Wird Abb. 5.1b betrachtet, so ist deutlich zu sehen, dass die Differenz vor allem mit den ausgewählten Zusatzmetriken zusammenhängt. Denn Aufzeichnungen mittels der integrierten Lösung, ohne Zusatzmetriken, benötigen während der Initialisierung im Schnitt 62 Millisekunden länger als mittels TTB. Deutlich stärkere Auswirkungen entstehen durch das Ermitteln von Zusatzmetriken, so benötigt das Berechnen des Servers mit diesen zwischen 415 und 4395,3 Millisekunden länger als ohne jegliche Aufzeichnung. Wohingegen bei Aufzeichnungen ohne Zusatzmetriken mit der internen Lösung 263,7 Millisekunden und mittels TTB 201,7 Millisekunden länger benötigt wurde. Dies hängt auch damit zusammen, wie diese Aufzeichnungen stattfinden. Zum ermitteln des Posteingangs kann eine Erlang interne Funktion aufgerufen werden. Zum Ermitteln des Zustands muss jedoch eine spezielle Nachricht an den Aktor gesendet werden, der den Zustand anschließend an den Fragesteller zurücksendet. Dies belastet den Aktor mit einer weiteren Nachricht und kann so die Effizienz des Aktors beeinflussen.

Im Gegensatz zur Initialisierung sind während des Testlaufs die Unterschiede nicht so deutlich ersichtlich. Dies hängt auch damit zusammen, dass es trotz genutzter 95er Perzentile einige deutliche Ausreißer gibt (Abb. 5.1c). Bei betrachten der Durchschnittswerte scheint zunächst auch in dieser Phase die Aufzeichnung deutliche Auswirkungen zu haben. So wird z.B. beim Nutzen der integrierten Lösung ohne zusätzliche Metriken im Schnitt 1,7 Millisekunden länger benötigt als ohne Aufzeichnungen (siehe Tabelle 5.1). Die durchschnittliche Zeit zur Berechnung ohne jegliche Aufzeichnung betrug in diesem Zeitraum 3,5 Millisekunden, damit würde die Berechnung hier immer noch um den Faktor 1,5 länger brauchen. Werden jedoch die Durchschnittswerte der Perzentilen betrachtet, so ist schnell ersichtlich, dass dieser Wert durch starke Ausreißer entstanden ist. So ist der Durchschnittswert bei der integrierten Lösung ohne zusätzliche Metriken innerhalb der 95er Perzentile nur noch 0,1 Millisekunden und bei betrachten der 90er Perzentile mit -0,1 Millisekunden sogar leicht negativ. Bei Betrachten von

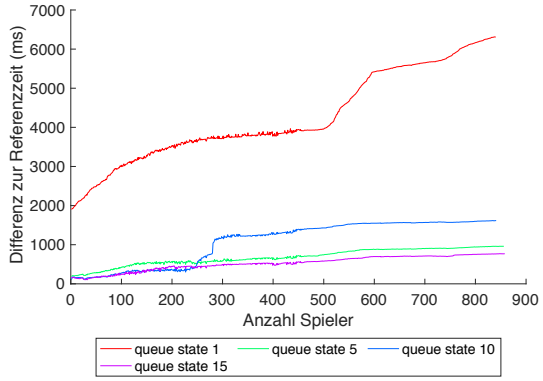
Abb. 5.1c und Abb. 5.1d ist deutlich zu sehen, dass die Häufung der Ausreißer gegen Ende der Tests deutlich zunimmt. Dies könnte mit dem starken Speicherverbrauch zusammenhängen, der mit der Aufzeichnung einhergeht (u.a. Abb. 5.4). Diese werden in einem späteren Abschnitt noch etwas genauer besprochen. Weil die Unterschiede während der Initialisierung deutlicher sichtbar sind, wird in den folgenden Abschnitten vor allem diese Phase herangezogen, um die verschiedenen Testszenarien zu vergleichen.



(a) Nur Posteingang



(b) Nur Zustand



(c) Posteingang und Zustand

Abbildung 5.2: Posteingang und Zustand

Zusatzmetriken

Ziel dieses Vergleiches ist vor allem zu ermitteln, wie sich die verschiedenen Samplingraten auf die Performance des Systems auswirkt. Es wurde mit der Hypothese in die Untersuchung gegangen, dass sich eine höhere Samplingrate eindeutig negativ auf die Performance auswirkt. Tendenziell ist diese Hypothese auch zu erkennen, jedoch ist der Trend nicht so eindeutig wie angenommen. So gilt zwar bei allen Szenarien, dass sich die schlechteste Zeit mit einer

Samplingrate von 1 ergeben hat, es gab aber durchaus auch Testläufe bei denen eine höhere Samplingrate zu einem schlechteren Endresultat geführt haben (Abb. 5.2). So ist der Endwert beim Ermitteln des Zustands mit einer Samplingrate von 15 höher als der mit einer Samplingrate von 10 (Abb. 5.2b). Dies wirkt sich auch auf den Durchschnittswert aus. So wurde in diesem Szenario für eine Samplingrate von 10 im Schnitt 658,5 Millisekunden und mit einer Samplingrate von 15 856,3 Millisekunden benötigt.

Auch wenn im allgemeinen die erwartete Tendenz erkennbar ist, so lässt sich noch kein allgemeingültiges Verhalten aus den Daten ableiten. Hierfür werden noch weitere Untersuchungen benötigt, die den Datenbestand vergrößern und somit eine zuverlässigere Aussage zulassen.

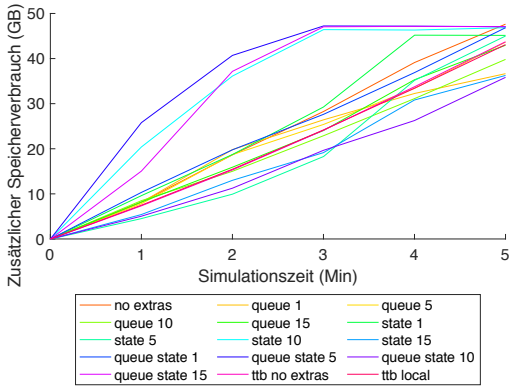
Abbildung 5.3: Differenz der Zuweisungszeit beim Einsatz von TTB während der Initialisierung

TTB

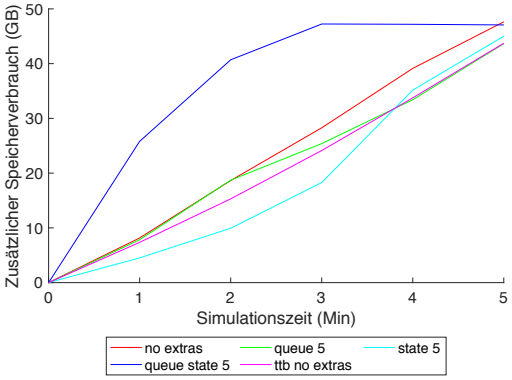
Beim Vergleich der mittels TTB ermittelten Datensätze, soll vor allem betrachtet werden, wie sich das Ermitteln der Prozessnummer auf die Performance auswirkt. Es lässt sich feststellen, dass sich das Auflösen von Prozessnummern kaum auf die Performance auswirkt (Abb. 5.3). Dies untermauern auch die ermittelten Durchschnittswerte. Während der Initialisierung benötigten Zuweisungen, die während einer Aufzeichnung mittels TTB mit lokal aufgelösten Prozessen stattfanden, im Durchschnitt sogar um 3,8 Millisekunden weniger Zeit als bei Aufzeichnungen, bei denen nur TTB eingesetzt wurde. Während des Testlaufs benötigte das Verfahren mit aufgelösten Prozessnummern allerdings im Durchschnitt 0,7 Millisekunden länger (siehe Tabelle 5.1).

Auch hier können die Ergebnisse nur als ein erster Trend betrachtet werden. Für zuverlässigere Aussagen werden weitere Untersuchungen in der Zukunft benötigt.

Speicherverbrauch



(a) Periodische Neuberechnung



(b) Periodische Neuberechnung

Abbildung 5.4: Zusätzlicher Speicherverbrauch der verschiedenen Tests im Laufe der Simulationszeit im Vergleich zu ohne Aufzeichnung

Eine der größten Herausforderungen beim Testen der Aufzeichnungen war dessen Speicherauslastung. Deshalb soll diese an dieser Stelle gesondert betrachtet werden. Wie in Abb. 5.4 zu sehen ist steigt die Speicherauslastung bei allen Verfahren deutlich an. Auf eine Andeutung der Tests ohne Aufzeichnung wurde bei all diesen Graphen verzichtet, da dessen Verbrauch im Vergleich zu den Aufzeichnungen verschwindend gering ist. Dies zeigt auch das Betrachten des zusätzlichen Verbrauchs der Verschiedenen Tests gegen deren Ende. So wurde ohne jegliche Aufzeichnung 0,25 GB zusätzlich benötigt. Bei den Aufzeichnungen liegen die Werte zwischen 36,1 GB und 47,89 GB (siehe Tabelle 5.2). Dies zeigt auch, dass dieses Problem nicht rein in der Implementierung von der in ErlViz integrierten Lösung liegt. Denn auch bei der bereits in Erlang integrierten Lösung (TTB) ist diese Problematik erkennbar. Dies könnte auch erklären, warum sich die Menge der Ausreißer bei der Berechnungszeit gegen Ende der Tests erhöht (Abb. 5.1c). Ein zusätzlicher Indikator für den Einfluss könnte sein, dass die Häufung von Ausreißern bei den Verfahren verstärkt auftritt, die einen besonders hohen Speicherbedarf hatten. So gibt es die stärksten Häufungen von Ausreißern bei den Aufzeichnungen mit Ermittlung der Länge des Posteingangs und des Zustands in den Samplingraten 1, 5 und 15 sowie bei der Aufzeichnung mit Ermittlung des Zustands mit einer Samplingrate von 10. Diese sind zugleich mit einem zusätzlichen Speicherbedarf von über 47 GB die vier Verfahren mit dem höchsten zusätzlichen Speicherbedarf (siehe Tabelle 5.2).

Der erhöhte Speicherbedarf könnte mit mehreren Faktoren zusammenhängen. So könnte

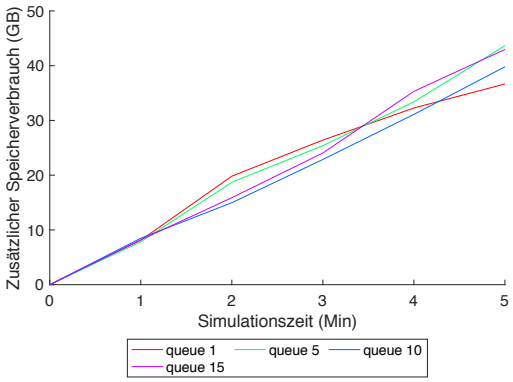
	Zusätzlicher Speicherbedarf (GB)
Keine Aufzeichnung	0,25
Ohne Zusatzmetriken	47,89
Queue 1	36,91
Queue 5	43,9
Queue 10	40,06
Queue 15	43,21
State 1	45,38
State 5	45,25
State 10	47,09
State 15	36,5
Queue & State 1	47,08
Queue & State 5	47,31
Queue & State 10	36,1
Queue & State 15	47,33
TTB	43,98
TTB Prozesse lokal aufgelöst	43,35

Tabelle 5.2: Zusätzliche benötigter Speicherbedarf der Verschiedenen Verfahren gegen Ende der Tests

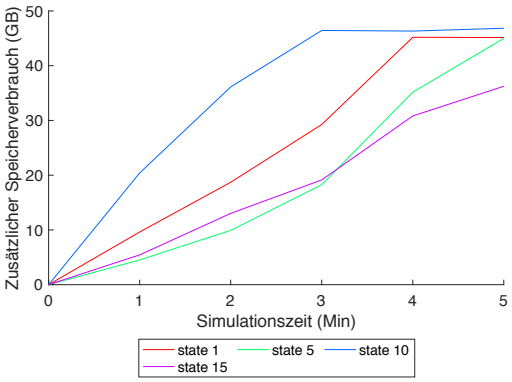
es zum einen sein, dass sich der Posteingang des Aktors zum Abspeichern der Daten schneller füllt als dieser abgearbeitet werden kann. Ein anderer Aspekt, der eine Rolle spielen kann ist, dass das Ablegen der Informationen in die Binärdatei nicht schnell genug geschieht, sich die Informationen so aufstauen und sich der Speicher dadurch immer weiter füllt. Hier sind jedoch weitere Untersuchungen nötig, um eine oder beide dieser Thesen zu stützen. Ein weiterer interessanter Aspekt zur weiteren Analyse ist die Untersuchung, ob es sich hierbei um eine rein technische Schwäche handelt oder ob diese konzeptionell bedingt ist.

Zusatzmetriken

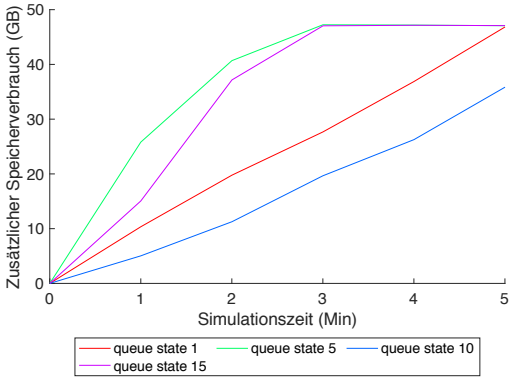
Ziel der Betrachtung der Zusatzmetriken ist es zu ermitteln, wie sich die Samplingrate auf die Speicherauslastung auswirkt. Betrachtet man das Verhalten der verschiedenen Szenarien, so ist auf den ersten Blick kein klares Muster erkennbar (Abb. 5.5). Es gibt zwar einige Tests bei denen ein stärkerer Anstieg des Speicherverbrauchs sichtbar ist, wie z.B. die Aufzeichnung des Zustands mit einer Samplingrate von 10 oder die Aufzeichnungen von Posteingang und Zustand mit Samplingraten von 15 sowie 5, doch diese sehen eher nach Zufall aus. Zumindest wenn nur die Samplingrate als Vergleichswert herangezogen wird. So ist keine klare Samplingrate sichtbar, bei denen dieses Verhalten auftritt. Auffällig ist jedoch, dass der starke



(a) Nur Posteingang



(b) Nur Zustand



(c) Posteingang und Zustand

Abbildung 5.5: Zusätzlich benötigter Speicherbedarf der verschiedenen Zusatzmetriken

Anstieg nur bei Tests auftritt, bei denen der Zustand aufgezeichnet wird (Abb. 5.5b & 5.5c). Dies kann damit zusammenhängen, dass der konkrete Zustand der Aktoren einem relativ großen Zufall unterliegt. So ist dieser stark davon abhängig, wie die einzelnen Avatare in der Spielwelt verteilt sind. Hinzu kommt, dass es sich bei dem Zustand eines Aktors um verhältnismäßig große Datensätze halten kann. Dadurch wirkt sich die Zufälligkeit der Daten noch einmal stärker aus, als dass es mit kleinen Datensätzen tun würde. Werden die Tests betrachtet bei denen nur der Posteingang aufgezeichnet wird, so ist der Anstieg bei allen ungefähr gleich stark (Abb. 5.5a). Ein Muster zwischen Samplingrate und Speicherverbrauch ist allerdings auch hier nicht erkennbar.

Eine klare Verbindung zwischen Samplingrate und Speicherverbrauch konnte nicht hergestellt werden. Dies kann allerdings auch daran liegen, dass die Auswirkungen durch die Samplingrate im Vergleich zu anderen Aspekten vernachlässigbar ist. Denn ein hoher Speicherverbrauch ist,

	Durchschnitt (ms)	Durchschnitt 95er Perzentile (ms)	Durchschnitt 90er Perzentile (ms)	Median (ms)
Periodische Neuberechnung	43,8	10,6	5,1	3,9
Zellbasierte Migration	5,5	4,3	4,1	4,1

Tabelle 5.3: Durchschnittswerte zur Berechnung des Spielservers während des Testlaufs

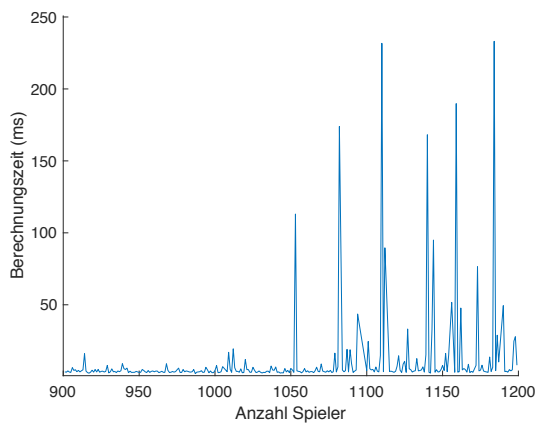
wie zuvor bereits beschrieben, in allen Testszenarien zu beobachten. Die einzige Auffälligkeit die festgestellt werden konnte ist dass es bei Aufzeichnungen des Zustands gelegentlich zu stärkeren Steigungen beim Speicherverbrauch gekommen ist. Die erste Deutung der Ergebnisse liegt eine Verbindung zwischen der Größe einzelner Datensätze und der Wahrscheinlichkeit einer stärkeren Steigung nah. In diesem Zusammenhang wäre eine interessante zukünftige Fragestellung, inwiefern sich die Größe einzelner Datensätze tatsächlich auf solche Ausreißer auswirkt.

5.2.2 Visualisierung

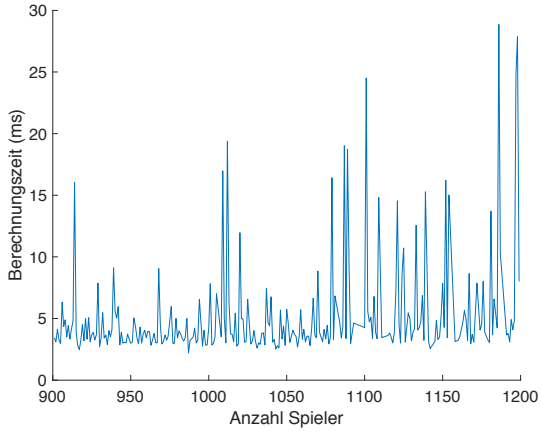
Um die Fähigkeiten des Verfahrens zu zeigen, soll mittels der Visualisierung das beschriebene Problem aus Abschnitt 2.1 untersucht werden. Deshalb soll zunächst überprüft werden, ob dieses Problem nach wie vor existiert. Anschließend wird das Verhalten des Systems mithilfe der Visualisierung von ErlViz auf Basis dieser Ergebnisse näher untersucht.

Überprüfen des aktuellen Verhaltens

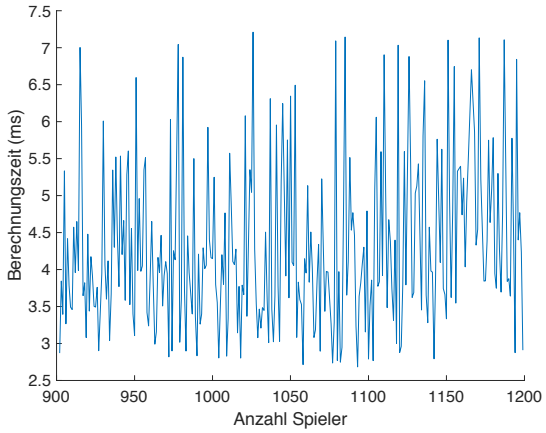
Werden die beiden Testläufe miteinander verglichen, so ist festzustellen, dass der große Unterschied nicht mehr ersichtlich ist (Abb. 5.6). Werden die Durchschnittswerte verglichen, so hat die Berechnung bei der periodischen Neuberechnung zunächst sogar länger benötigt. Werden jedoch die Durchschnittswerte der 90er und 95er Perzentile sowie der Median betrachtet, so ist schnell ersichtlich, dass der hohe Durchschnittswert vor allem mit starken Ausreißern zusammenhängt. So ist der Durchschnittswert für die periodische Neuberechnung bei 43,8 Millisekunden und der für die zellbasierte Migration bei 5,5 Millisekunden. Wird der Median jedoch betrachtet, so ist dieser für die periodische Neuberechnung bei 3,9 Millisekunden und der für die zellbasierte Migration bei 4,1 Millisekunden (siehe Tabelle 5.3). Dass der hohe Durchschnittswert vor allem mit starken Ausreißern zusammenhängt lässt sich sehr gut erkennen, wenn Abb. 5.6a betrachtet wird. Vor allem gegen Ende des Tests scheint es eine Häufung von Ausreißern zu geben. Daraus lässt sich schließen, dass es insgesamt keinen großen Unterschied zwischen der periodischen Neuberechnung und der zellbasierten Migration gab.



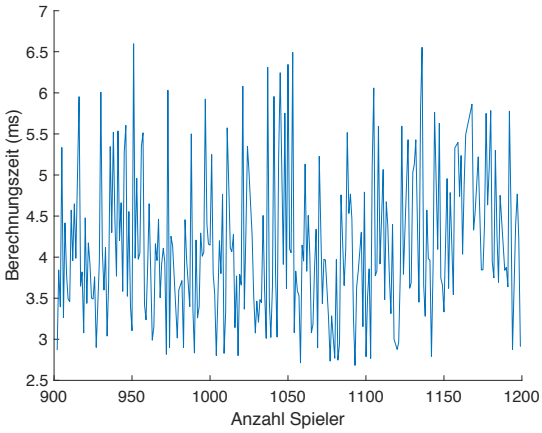
(a) Periodische Neuberechnung (95er Perzentile)



(b) Periodische Neuberechnung (90er Perzentile)



(c) Zellbasierte Migration (95er Perzentile)



(d) Zellbasierte Migration (90er Perzentile)

Abbildung 5.6: Vergleich der Dauer zur Berechnung des Spielservers bei periodischer Neuberechnung sowie zellbasierter Migration während des Testlaufs

Somit konnte das Verhalten, dass in Abschnitt 2.1 dargestellt wurde, nicht nachgestellt werden. Da in der Zwischenzeit keine weiteren Anpassungen am Load Balancer stattgefunden haben, liegt der Verdacht nahe, dass die Ursache im QuP lag. Deshalb wird im Folgenden vor allem die Interaktion zwischen dem Load Balancer sowie QuP betrachtet, um so festzustellen, wo die Unterschiede zwischen der periodischen Neuberechnung und der zellbasierten Migration in der Interaktion mit QuP liegen. Daraus sollen Rückschlüsse geführt werden, warum zuvor der Unterschied zwischen den beiden Methodiken bestand.

Untersuchen der Anwendung

Um das Verhalten des Systems zu analysieren wurden 3 verschiedene Abläufe betrachtet, die innerhalb der Anwendung stattfinden:

- die initiale Zuweisung eines Avatars
- die Neuberechnung eines Spielservers sowie die Migration des Avatars
- das Übermitteln von Zustandsinformationen am Beispiel eines neu zugewiesenen Avatars

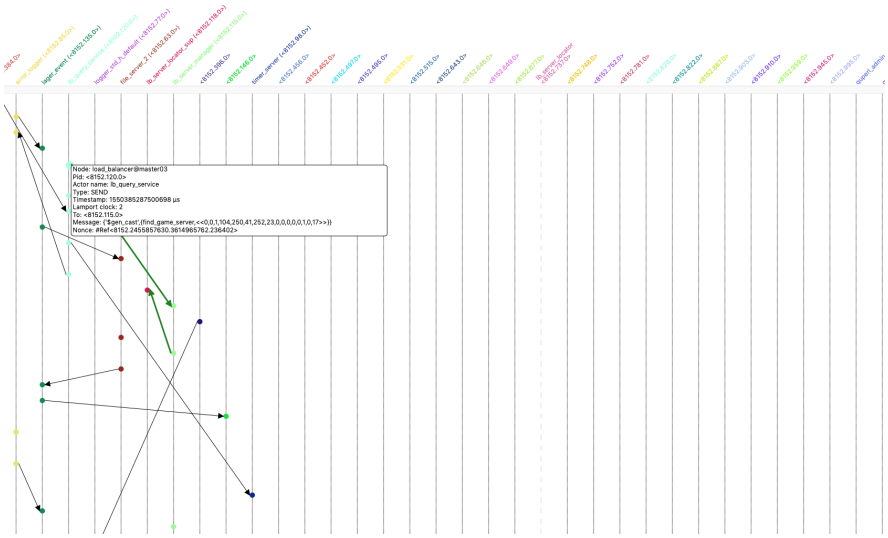
All diese Untersuchungen wurden jeweils mit der periodischen sowie mit der zellbasierten Neuberechnung durchgeführt. Wobei im Folgenden zur Vereinfachung, die Prozesse, die bei beiden Verfahren identisch, sind nur einmal und nicht für beide Verfahren betrachtet werden.

Initiale Zuweisung

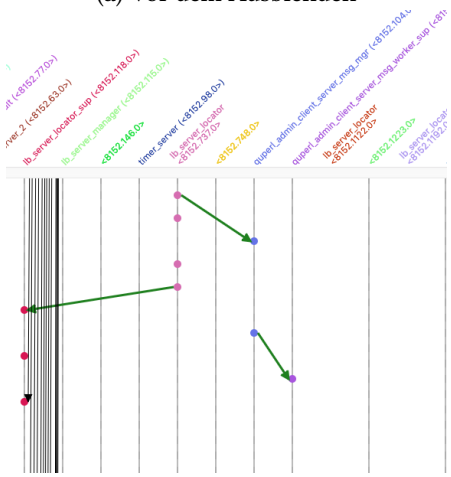
Die initiale Zuweisung erfolgt bei beiden Verfahren identisch, deswegen wird in dieser Analyse auch nur ein einzelnes Beispiel analysiert. In diesem Fall stammen die Bilder und Daten von dem Testlauf mit der periodischen Neuberechnung.

Abb. 5.7a zeigt den Beginn der Berechnung zur Zuweisung eines Avatars. Es ist zu erkennen, dass neben den eigentlichen Aktoren, die betrachtet werden sollen auch eine Vielzahl von weiteren Aktoren sichtbar sind, die zu diesem Zeitpunkt nicht betrachtet werden sollen. Deshalb wurde diese Aktoren ausgeblendet, um eine bessere Übersicht zu gewinnen (5.7b). Die Möglichkeit Aufrufpfade anzuzeigen wurde massiv genutzt, um besser zu erkennen, welche Aufrufe genau betrachtet werden (grüne Pfeile). Dies ist auch in allen Abbildung zur Untersuchung, mittels der Visualisierung, gut erkennbar. Mit diesen Werkzeugen konnte die initiale Zuweisung erfolgreich untersucht werden. Bis zum Ende der Berechnung des Spielservers konnten keine Auffälligkeiten festgestellt werden. So konnte eingesehen werden, wie die Daten des Avatars von QuP-Server bezogen und an den *lb_server_locator* übergeben wurden. Hierbei konnte erwartetes Verhalten ohne Auffälligkeiten festgestellt werden. Die einzige kleine Auffälligkeit war, dass sich beim *lb_server_locator_sup*, der Posteingang bereits leicht aufgestaut hatte. So konnte dort zwischendurch eine Länge von 12 festgestellt werden. Es konnte jedoch keine Auswirkung auf den Zuweisungsprozess dadurch festgestellt werden.

Eine deutliche Auffälligkeit konnte jedoch nach der Berechnung des Spielservers festgestellt werden. So wurde der berechnete Spielserver vom *lb_data_storage*-Aktor mehrfach nicht akzep-



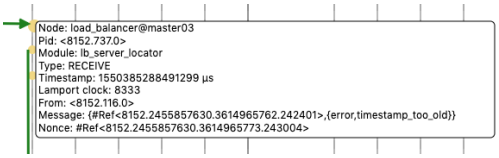
(a) Vor dem Ausblenden



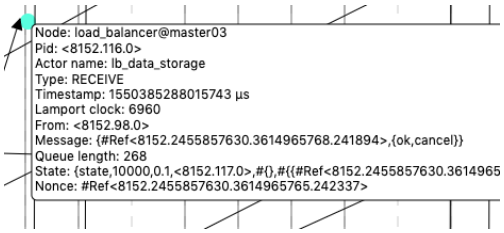
(b) Nach dem Ausblenden

Abbildung 5.7: Vergleich der Informationsmenge vor und nach dem ausblenden einzelner Aktoren

tiert, weil die genutzten Informationen zur Berechnung des Spielservers veraltet waren (Abb. 5.8a). So wurde der berechnete Spielserver erst nach dem 36. Versuch akzeptiert. Akzeptieren in diesem Fall heißt, dass es nicht möglich war, die Zuweisung des Avatars beim *lb_data_storage*-Aktor zu reservieren. Da die hohe Anzahl an Versuchen bei der Analyse verdächtig erschien, wurde als nächstes die Länge des Posteingangs herangezogen (Abb. 5.8b). Dadurch konnte festgestellt werden, dass die Fülle des Posteingangs relativ hoch war. So war die Länge beim 24. Reservierungsversuch bei 133 und beim 36. Versuch sogar bei 430 (siehe Tabelle 5.4). Motiviert



(a) Erhalt der Nachricht *timestamp_too_old*



(b) Informationen des *lb_data_storage*-Aktors, mit der Länge des Posteingangs

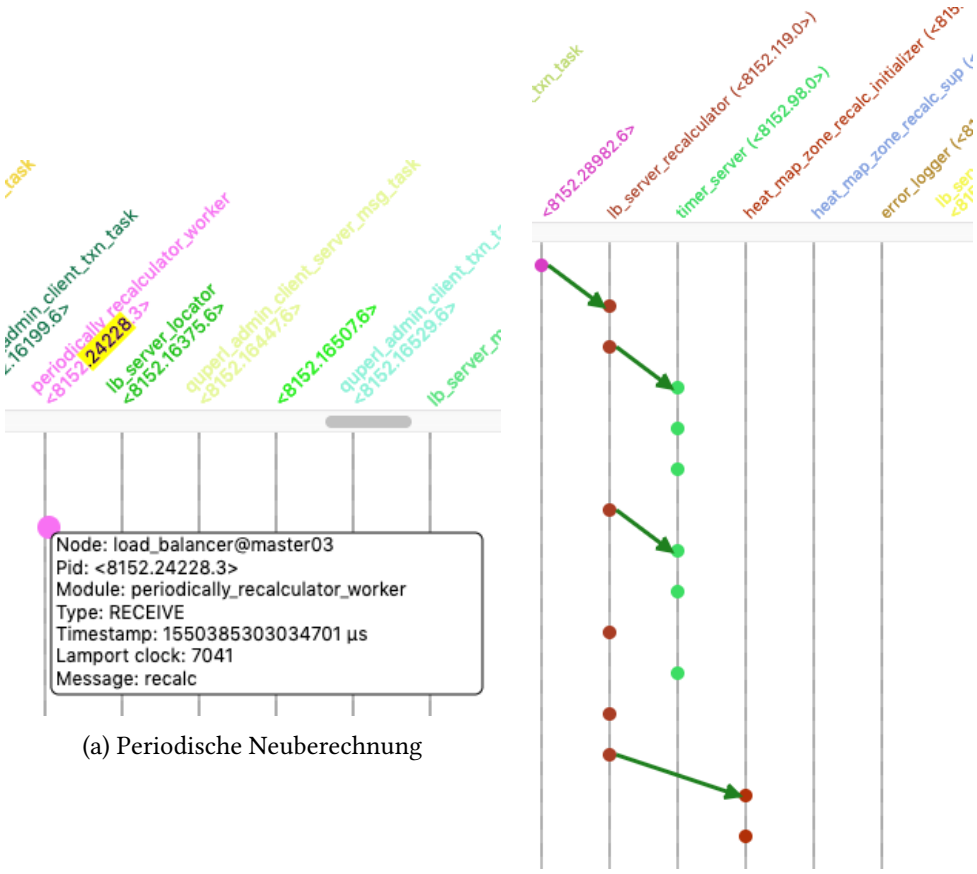
Abbildung 5.8: Anzeige der Informationen einzelner Ereignisse, zum evaluieren vom *timestamp_too_old* Nachrichren

Versuch	1	2	3	4	24	26	32	33	35	36
Dauer (ms)	8,28	17,48	4,56	0,9	-	-	65,88	228,09	344,95	291,17
Länge Posteingang	-	-	-	-	133	85	158	-	-	430

Tabelle 5.4: Dauer der verschiedenen Reservierungsversuche am *lb_data_storage*-Aktor und dessen Länge des Posteingangs

durch die Fülle des Posteingangs wurde anschließend auch die Dauer der Anfragen an den *lb_data_storage*-Aktor betrachtet. Denn bereits in Abschnitt 2.1.4 wurde der Verdacht geäußert, dass es sich hierbei um einen Flaschenhals handeln könnte. Während die ersten Anfragen an den Aktor zwischen ungefähr 8,3 und 17,5 Millisekunden benötigt haben, haben die letzten Anfragen deutlich länger gedauert. So benötigte die 35. Anfrage ganze 344,95 Millisekunden (siehe Tabelle 5.4). Das legt die Interpretation nahe, dass die Anfrage durch die starke Fülle des Posteingangs lange warten muss um tatsächlich verarbeitet zu werden. Die gesamte Verarbeitung zur Ermittlung des Spielservers hat dadurch in diesem Fall 1287,45 Millisekunden gedauert.

Das beobachtete Verhalten erklärt warum die Berechnung während der initialen Phase deutlich länger gedauert hat als während des Testlaufs. Denn durch die parallele Verarbeitung, wirken sehr viele Aktoren auf den *lb_data_storage*-Aktor ein und veränderten den Zustand der Umgebung stetig. Dadurch wurden Reservierungen abgelehnt, weil sich die Grundlage auf dem die Berechnung stattfand in der Zwischenzeit zu stark verändert hat. Hinzu kommt, dass der Posteingang des *lb_data_storage*-Aktor dadurch auch stark befüllt war und deshalb länger benötigte um Anfragen zu beantworten. Die lange Berechnungsdauer ist somit ein Wechselspiel aus der regelmäßigen Ablehnung von Reservierungen und der Länge des Posteingangs.



(a) Periodische Neuberechnung

(b) Zellbasierte Migration

Abbildung 5.9: Start der Neuberechnung innerhalb der verschiedenen Verfahren

Neuberechnung

Bei der Betrachtung der Neuberechnungen konnten keine Auffälligkeiten in Bezug auf den untersuchten Gegenstand festgestellt werden. Dennoch soll an dieser Stelle auf zwei Dinge eingegangen werden. Zum einen auf den Unterschied, wie die periodische und die zellbasierte Migration starten und zum anderen eine Auffälligkeit, die beim Beziehen der Daten eines Avatars festgestellt wurde. Zunächst soll dargestellt werden, wie sich das unterschiedliche Starten der Neuberechnung innerhalb von ErlViz ausdrückt. In Abb. 5.9 sind die beiden Verfahren dargestellt. Es ist dort zu sehen, dass der Start einer Neuberechnung bei der periodischen Neuberechnung durch eine einzelne Nachricht an den zuständigen Aktor geschieht (Abb. 5.9a). Bei der zellbasierten Migration ist dies ein wenig komplexer. Hier erhält der zuständige *lb_server_recalculator* eine Nachricht und startet anschließend die *heat_map_zone_recalc_initializer*, die für den eigentlichen Start der Neuberechnung zuständig

sind (Abb. 5.9b). Auch wenn in der Abbildung nur ein einzelner Start dargestellt ist, so ist in ErlViz deutlich sichtbar, dass mehrere Starts hintereinander stattfinden. Um die Darstellung prägnant zu halten, wurden diese jedoch außen vorgelassen. Dieser Unterschied rührt daher, dass bei der periodischen Neuberechnung diese individuell für jeden Avatar gestartet wird und der zugehörige Aktor auch den jeweiligen Zeitpunkt selbstständig koordiniert. Bei der zellbasierten Migration gibt es einen zentralen Aktor der das Starten der Neuberechnung verwaltet. Dazu werden zunächst gezielt Kandidaten ermittelt und für diese eine Neuberechnung gestartet.

Eine Auffälligkeit die während der Analyse festgestellt wurde, ist die tiefe Kopplung zwischen verschiedenen Aktoren innerhalb von QuP beim Beziehen der Daten eines Aktors. Damit ist gemeint, dass nach Aufruf des *quperl_client_cache*-Aktors weitere Aktoren gestartet werden, die während der Initialisierung wiederum weitere Aktoren starten. Dies erfolgt alles blockierend und synchron, womit der *quperl_client_cache*-Aktor in der Zeit keine Nachrichten annehmen kann. In Abb. 5.10 ist gut die Antwortkaskade zu sehen, die erfolgt nachdem der letzte Aktor gestartet wurde. Dieses blockierende Verhalten könnte zu Performanceproblemen führen, denn der *quperl_client_cache*-Aktor ist ein zentraler Aktor, der von vielen anderen Aktoren aufgerufen wird. Für alle weiteren beteiligten Aktoren stellt diese Verkettung vermutlich ein eher geringeres Problem dar, denn diese wurden speziell zum Beziehen der Daten gestartet und haben keine größere Interaktion mit anderen Aktoren im System. Durch ErlViz konnte somit eine potentielle Schwachstelle ausgemacht werden, die nun weiter untersucht werden kann.

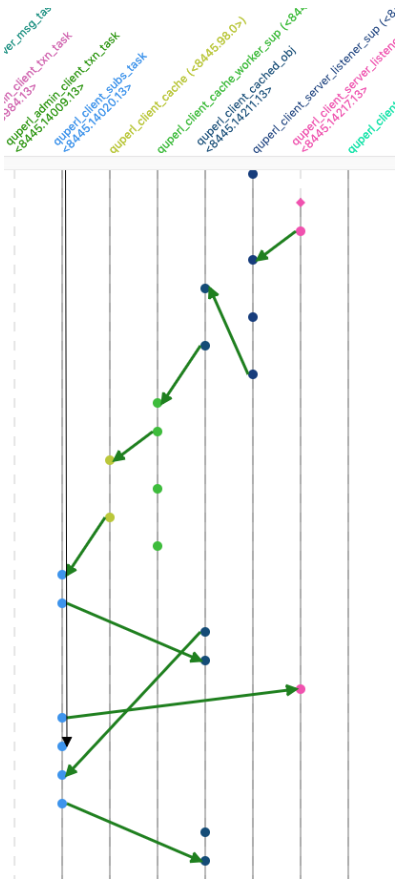


Abbildung 5.10: Kopplung zwischen den Aktoren, die beim Beziehen von Avatardaten involviert sind.

Übermitteln von Zustandsinformationen

Beim übermitteln der Zustandsinformationen konnten keinerlei Auffälligkeiten festgestellt werden. Auch hier wurde, wie bei der Neuberechnung sehr gut sichtbar, worin der Unterschied zwischen den beiden Verfahren liegt. Wenn die Information übermittelt wird, dass ein neuer Avatar einem Spielserver zugeordnet wurde, so wurde bei der periodischen

Neuberechnung direkt nach Eintreffen der Information beim *lb_server_recalculator*-Aktor, der für den Avatar zuständige Aktor gestartet. Dahingegen wurde bei der zellbasierten Migration die Information über den neuen Avatar durch den *lb_server_recalculator*-Aktor einfach entgegengenommen ohne danach weitere Interaktionen in dem Zusammenhang zu tätigen. Stattdessen hat dieser den neuen Umstand intern in seinen Zustand eingetragen.

Zusammenfassung

Die zu untersuchende Problemstellung konnte mittels der Visualisierung nicht erklärt werden. Jedoch konnten andere Aspekte des Systems entdeckt werden. So wurde eine potentielle Erklärung gefunden, warum die initiale Zuweisung bei paralleler Zuweisung von vielen Avataren deutlich länger dauert als während der Testphase. Auch konnte ein potentieller Flaschenhals innerhalb des QuP-Clients auf dem Spielserver aufgedeckt werden. Bei dieser Untersuchung haben auch zusätzliche Werkzeuge der Visualisierung stark unterstützend gewirkt. So wurden vor allem die Anzeige der Aufrufpfade sowie das Ausblenden von einzelnen Aktoren genutzt, um eine bessere Übersichtlichkeit zu gewinnen. Es ist auch sichtbar geworden, dass die Ordnung nach dem Zeitstempel nicht immer zuverlässig funktioniert. So kommt es ab und zu vor, dass der Pfeil scheinbar „rückwärts“ zeigt.

Auch wenn keine Erkenntnisse zum Untersuchungsgegenstand gewonnen werden konnten, so wurde durch das Aufdecken anderer Aspekte deutlich, dass ErlViz Potential besitzt.

5.2.3 Fazit

Es konnte gezeigt werden, dass das System geeignet ist, um das Verhalten von aktorbasierten Systemen zu beobachten und anschließend auszuwerten. Die Auswirkungen auf die Performance des Systems hängt stark von den ermittelten Metriken und der Belastung des Systems ab. So konnten zwar während der Initialisierung zum Teil deutliche Unterschiede zum Testlauf ohne Aufzeichnung festgestellt werden, während des Testlaufs waren sie jedoch kaum feststellbar. Ein gravierender Einfluss konnte jedoch in der Belastung des Arbeitsspeichers festgestellt werden. Dies legt nahe, dass der aktuelle Stand zwar generell für Auswertungen geeignet ist, dies jedoch nur in einer gezielten Testumgebung geschehen sollte. Für den produktiven Einsatz scheint das aktuelle Verfahren nicht geeignet zu sein. Für zukünftige Untersuchungen könnte es interessant sein zu beobachten, ob die starke Auslastung des Arbeitsspeichers auch in Zusammenhang mit dem Anwendungsfall liegt und ob diese unter geringeren Belastungen eventuell nicht so stark ins Gewicht fällt.

Auch bei der Visualisierung konnte ein Potential erkannt werden. So wurde die Problemstellung zwar nicht gelöst. Jedoch konnten eine ganze Reihe anderer Erkenntnisse gewonnen werden. Bestimmte Hilfsmittel erwiesen sich hierbei als enorm hilfreich. So halfen vor allem das Ausblenden einzelner Aktoren und das Einblenden der Aufrufpfade dabei eine bessere Übersicht zu gewinnen. Jedoch müssen für eine effizienterer Nutzung noch eine ganze Reihe weiterer Hilfsmittel entstehen. So wäre es z.B. wünschenswert andere Aktoren nach bestimmten Kriterien zeitgleich ausblenden zu können, anstatt jeden einzeln auszublenden. Denn in den meisten Fällen wurden vor allem Aktoren desselben Typs ausgeblendet.

Trotz der Erfolge gibt es einiges an Verbesserungspotential. So sollte vor allem der hohe Speicherverbrauch bekämpft werden. Ein Ansatz hier wäre es gezielte Filter bereits bei der Aufzeichnung der Nachrichten einzusetzen. Ein weiteres recht konkretes Problem liegt in der Umsetzung der Aufzeichnung, diese ist stark an die konkrete Erlang/OTP Version gebunden. Dadurch bricht diese direkt bei größeren Änderungen. Zudem wird durch die enge Bindung an Erlang/OTP nicht jede versendete Nachricht registriert, vor allem wenn diese nicht über Erlang/OTP versendet wurde. Dies ist zwar genug, um das Konzept im Kontext dieser Arbeit darzustellen und zu evaluieren, für einen reellen Einsatz wäre jedoch eine allgemeingültigere Lösung wünschenswert, die eventuell sogar in der Lage ist, jede versendete Nachricht aufzuzeichnen. Da dies voraussichtlich auch den Speicher stärker belasten würde, wäre es sinnvoll auch einen Filter zeitgleich nutzen zu können, damit nicht alle Aktoren aufgezeichnet werden.

6 Abschluss

Zum Abschluss soll ein Resümee über diese Arbeit stattfinden. Dazu gehören eine Zusammenfassung der Arbeit, ein Fazit darüber was erreicht wurde und ein Ausblick auf mögliche folgende Arbeiten.

6.1 Zusammenfassung

Innerhalb dieser Arbeit wurde mit ErlViz ein System zur Analyse von verteilten aktorbasierten Systemen entwickelt. Dazu wurde zunächst Bezug auf vorhergehende Arbeiten gezogen und eine allgemeine Problemstellung aus der Entwicklung eines Verfahrens zur Lastverteilung im Kontext von Timadorus hergeleitet. Aus dieser Problemstellung wurden Anforderungen herausgebildet und nach betrachten der bisherigen Arbeiten der Schluss gezogen, dass ein eigenes System entwickelt werden soll. Anschließend wurde die Architektur sowie die Visualisierung vorgestellt, aus der im Folgenden ErlViz realisiert wurde. Hierbei wurden mehrere Möglichkeiten zur Aufzeichnung der Daten umgesetzt. Zum einen eine Variante, bei der TTB eingesetzt wird sowie eine komplette Eigenentwicklung. Diese Umsetzung wurde in die Timadorusumgebung eingebunden und auf dieser Basis verschiedene Experimente durchgeführt. Hierbei stellte sich heraus, dass die Aufzeichnungen aktuell einen starken Speicherverbrauch haben und somit nur für den Testbetrieb geeignet sind. Allerdings konnten mithilfe der Visualisierung einige potentielle Probleme innerhalb von Timadorus aufgedeckt werden.

6.2 Fazit

Es konnte gezeigt werden, dass das Konzept von ErlViz tragbar ist. So hat sich zwar herausgestellt, dass der Speicherverbrauch aktuell noch zu hoch ist, allerdings konnten in der Visualisierung einige potentielle Schwachstellen innerhalb von Timadorus aufgedeckt werden. Es wurde ein System entwickelt, mit dem es möglich ist, den Programmfluss nachträglich nachzuvollziehen und so Rückschlüsse auf das Verhalten des Systems zu ziehen. Herbei konnte die lokale sowie die verteilte Nebenläufigkeit dargestellt werden. Geschafft wurde dies durch anreichern der Nachrichten zwischen den Aktoren mit zusätzlichen Informationen. Durch

einschleusen von eigenem Programmcode zur Laufzeit, konnte dies komplett transparent realisiert werden. Hierbei erfüllt es alle gestellten Anforderungen. So wird die lokale sowie verteilte Kommunikation der Aktoren aufgezeichnet und aus diesen Daten ein Tracing-Graph erzeugt. Dies impliziert auch, dass die Daten in einer Happens Before Order hinterlegt werden konnten. Durch die Transparente Umsetzung des Tracing ist auf Programmcode Ebene keine aktive Tätigkeit durch den Entwickler nötig. Auch ist es möglich zusätzliche Metriken für die Aktoren zu erfassen. Zudem wurde eine einfache Form von Queries umgesetzt, die aktuell nur in der Lage sind, den betrachteten Zeitraum zu verringern. Neben den Pflichtanforderungen wurden auch einige optionale Anforderungen realisiert. So kann zur Ermittlung der Metriken Sampling eingesetzt werden und das Tracing auch dynamisch zur Laufzeit eingebunden werden.

6.3 Ausblick

Auf Basis der erbrachten Arbeit lassen sich eine ganze Reihe von weiterführenden Arbeiten durchführen. Zunächst gibt auf Grundlage der Ergebnisse aus den Experimenten ein paar konkrete Fragestellungen. So könnte in folgenden Arbeiten die Methodik verfeinert werden, um die Probleme mit der starken Speicherbelastung in den Griff zu bekommen. Hierbei könnten mehrere Ansätze verfolgt werden. Zum einen eine konzeptionelle Anpassung bei der das Überführen der Daten in die Datei geändert wird. Ein anderer Ansatz wäre es, die Datenmenge generell zu verringern. Dies könnte zum Beispiel erreicht werden, indem es eine Möglichkeit gäbe, mit einem Filter zu bestimmen, welche Aktoren überhaupt aufgezeichnet oder nicht aufgezeichnet werden sollen. Auch denkbar wäre Sampling auf Basis von Aktoren. D.h. dass nicht alle Aktoren sondern nur einzelne aufgezeichnet werden. Diese beiden Ansätze könnten auch die Visualisierung unterstützen, da viele nicht benötigte Daten nicht dargestellt werden. Eine andere Fragestellung die sich aus den Experimenten ergibt, ist, ob die Ergebnisse auch auf andere verteilte Systeme übertragbar sind oder ob die Ergebnisse spezifisch für Timadorus sind. Es gibt aber auch mögliche weiterführende Arbeiten, die sich nicht direkt aus den Experimenten ergeben. So könnte auf Basis der Daten, die aufgezeichnet werden, auch versucht werden eine automatisierte Analyse zu erschaffen die Beispielsweise bestimmte Muster erkennt. Eventuell lässt sich das auch mit Machine Learning verbinden um typische Muster zu finden die auf Fehler hinweisen. Würde eine Möglichkeit geschaffen werden, die Daten automatisiert während der Aufzeichnung zu übertragen, so könnte auch betrachtet werden ob sich dass System zum Monitoring eignet. Wenn es dann auch eine direkte automatisierte Analyse gäbe so könnte diese auch einen Alarm auslösen, wenn ein auffälliges Verhalten auftritt. Neben der Aufzeichnung gibt es auch weiterführende Themen in der Visualisierung.

So könnte diese um eine hierarchische Darstellung der Lebenslinien erweitert werden. Damit ist gemeint, dass alle Lebenslienien eines Knoten zusammengefasst dargestellt werden und erst bei expliziter Auswahl des Knotens, dessen lokalen Abläufe dargestellt werden. Dadurch kann die Visualisierung deutlich übersichtlicher werden. Ein weiterer Aspekt der Ausgebaut werden sollte sind die Queries. Hier könnten z.B. komplexere Analysen abgefragt werden. Es wird gut sichtbar, dass das Potential von ErlViz noch nicht ausgeschöpft ist und eine Reihe von Folgearbeiten erfolgen kann.

Literaturverzeichnis

[All15] Allers, Sven: *Lastverteilung in einer clusterbasierten verteilten virtuellen Umgebung*, HAW Hamburg, Bachelor of Science, 2015

[All17a] Allers, Sven: TESTUMGEBUNG FÜR DIE AVATARBASIERTE LASTVERTEILUNG IN VERTEILTEN VIRTUELLEN SYSTEMEN / HAW Hamburg. Version: 2017. https://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2017-proj/allers.pdf. Hamburg, 2017. (Master - Grundprojekt). – Forschungsbericht

[All17b] Allers, Sven: Zellbasierte Migration für die avatarbasierte Lastverteilung / HAW Hamburg. Version: 2017. https://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2017-proj/allers2.pdf. Hamburg, 2017. (Master - Hauptprojekt). – Forschungsbericht

[BAW+16] Behnke, Lutz ; Allers, Sven ; Wang, Qi ; Grecos, Christos ; Luck, Kai von: Avatar Density Based Client Assignment. In: *Entertainment Computing - ICEC 2016* Bd. 1, 2016. – ISBN 9783319461007, S. 137–148

[BCPC11] Bovenzi, Antonio ; Cotroneo, Domenico ; Pietrantuono, Roberto ; Carrozza, Gabriella: Error detection framework for complex software systems. In: *Proceedings of the 13th European Workshop on Dependable Computing - EWDC '11* (2011), 61. http://dx.doi.org/10.1145/1978582.1978596. – DOI 10.1145/1978582.1978596. ISBN 9781450302845

[BGL14] Behnke, Lutz ; Grecos, Christos ; Luck, Kai von: QuP : Graceful Degradation in State Propagation for DVEs. In: *Proceedings of International Workshop on Massively Multiuser Virtual Environments*, ACM, 2014

[BIMN03] Barham, Paul ; Isaacs, Rebecca ; Mortier, Richard ; Narayanan, Dushyanth: Magpie : online modelling and performance-aware systems. In: *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, USENIX, 2003, 85–90

[BWBE16] Beschastnikh, Ivan ; Wang, Patty ; Brun, Yuriy ; Ernst, Michael D.: Debugging distributed systems: Challenges and options for validation and debugging. In: *Communications of the ACM* 59 (2016), Nr. 8, 32–37. http://dx.doi.org/10.1145/2909480. – DOI 10.1145/2909480. – ISSN 00010782

[CDG+08] Chang, F A Y. ; Dean, Jeffrey ; Ghemawat, Sanjay ; Hsieh, Wilson C. ; Wallach, Deborah A. ; Burrows, Mike ; Chandra, Tushar ; Fikes, Andrew ; Gruber, Robert E.: Bigtable : A Distributed Storage System for Structured Data. 26 (2008), Nr. 2. http://dx.doi.org/10.1145/1365815.1365816.. – DOI 10.1145/1365815.1365816.

[CMF+14] Chow, MIchael ; Meisner, David ; Flinn, Jason ; Peek, Daniel ; Wenisch, Thomas F.: The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014. – ISBN 978–1–931971–16–4, 217–231

[DMT+16] Deligiannis, Pantazis ; Mccutchen, Matt ; Thomson, Paul ; Chen, Shuo ; Donaldson, Alastair F. ; Erickson, John ; Huang, Cheng ; Lal, Akash ; Mudduluru, Rashmi ; Qadeer, Shaz ; Schulte, Wolfram: Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In: *File and Storage Technologies (FAST)* (2016), 1–26. https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis. ISBN 9781931971287

[Erla] *Erlang – erlang*. http://erlang.org/doc/man/erlang.html, Abruf: 2018-12-04

[Erlb] *Erlang – Trace Tool Builder*. http://erlang.org/doc/apps/observer/ttb_ug.html, Abruf: 2018-11-19

[Erlc] *Erlang/OTP 20.1*. http://erlang.org/doc/, Abruf: 2018-12-29

[GMS+14] Gunawi, Haryadi S. ; Martin, Vincentius ; Satria, Anang D. ; Hao, Mingzhe ; Leesatapornwongsa, Tanakorn ; Patana-anake, Tiratat ; Do, Thanh ; Adityatama, Jeffry ; Eliazar, Kurnia J. ; Laksono, Agung ; Lukman, Jeffrey F.: What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In: *Proceedings of the ACM Symposium on Cloud Computing - SOCC '14* (2014), 1–14. http://dx.doi.org/10.1145/2670979.2670986. – DOI 10.1145/2670979.2670986. ISBN 9781450332521

[HBS73] Hewitt, Carl ; Bishop, Peter ; Steiger, Richard: Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. In: *Advance Papers of the Conference*, 1973

[Inf] *InfluxData (InfluxDB) | Time Series Database Monitoring & Analytics*. https://www.influxdata.com/, Abruf: 2019-03-12

[JM96] Johnson, David B. ; Maltz, David A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: *Mobile Computing* 353 (1996), 153–181. http://dx.doi.org/10.1007/b102605. – DOI 10.1007/b102605. ISBN 978–0–7923–9697–0

[KDJ+12] Kavulya, Soila P. ; Daniels, Scott ; Joshi, Kaustubh ; Hiltunen, Matti ; Gandhi, Rajeev ; Narasimhan, Priya: Draco : Statistical Diagnosis of Chronic Problems in Large Distributed Systems. In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, IEEE, 2012, S. 1—-12

[Lam78] Lamport, Leslie: Time , Clocks , and the Ordering of Events in a Distributed System. In: *Commun. ACM* 21 (1978), Nr. 7, S. 558–565

[LGW+08] Liu, Xuezheng ; Guo, Zhenyu ; Wang, Xi ; Chen, Feibo ; Lian, Xiaochen ; Tang, Jian ; Wu, Ming ; Kaashoek, M. F. ; Zhang, Zheng: D3S: Debugging Deployed Distributed Systems. In: *NSDI* (2008), S. 423–437. ISBN 111–999–5555–22–1

[LHJ+14] Leesatapornwongsa, Tanakorn ; Hao, Mingzhe ; Joshi, Pallavi ; Lukman, Jeffrey F. ; Gunawi, Haryadi S.: SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), 399–414. http://dl.acm.org/citation.cfm?id=2685048.2685080. ISBN 978–1–931971–16–4

[LLL+17] Liu, Haopeng ; Li, Guangpu ; Lukman, Jeffrey F. ; Li, Jiaxin ; Lu, Shan ; Gunawi, Haryadi S. ; Tian, Chen: DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '17* (2017), 677–691. http://dx.doi.org/10.1145/3037697.3037735. – DOI 10.1145/3037697.3037735. – ISBN 9781450344654

[LLLG16] Leesatapornwongsa, Tanakorn ; Lukman, Jeffry F. ; Lu, Shan ; Gunawi, Haryadi S.: TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: *Asplos* (2016), 517–530. http://dx.doi.org/

10.1145/2872362.2872374. – DOI 10.1145/2872362.2872374. – ISBN 9781450340915

[LSZE11] Lee, Kyu H. ; Sumner, Nick ; Zhang, Xiangyu ; Eugster, Patrick: Unified debugging of distributed systems with Recon. In: *Proceedings of the International Conference on Dependable Systems and Networks* (2011), S. 85–96. http://dx.doi.org/10.1109/DSN.2011.5958209. – DOI 10.1109/DSN.2011.5958209. – ISBN 9781424492336

[MBB06] Meijer, Erik ; Beckman, Brian ; Bierman, Gavin: LINQ : Reconciling Objects , Relations and XML in the . NET Framework. (2006), S. 59593. ISBN 1595932569

[MMO14] *MMOData4.1.0.xlxx*. http://users.telenet.be/mmodata/%0DCharts/MMOData4.1.0.xlsx. Version: 2014, Abruf: 2019-03-05

[Mon] *Open Source Document Database | MongoDB*. https://www.mongodb.com/, Abruf: 2019-03-12

[MRF15] Mace, Jonathan ; Roelke, Ryan ; Fonseca, Rodrigo: Pivot Tracing: Dynamic causal monitoring for distributed systems. In: *Symposium on Operating Systems Principles (SOSP)*. Monterey, California : ACM, 2015. – ISBN 978–1–4503–3834–9, 378–393

[NKN12] Nagaraj, Karthik ; Killian, Charles ; Neville, Jennifer: Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. San Jose, CA : USENIX Association, 2012

[RMOV07] Rueda, Silvia ; Morillo, Pedro ; Orduña, Juan M. ; Valencia, Universidad D.: A Peer-To-Peer Platform for Simulating Distributed Virtual Environments. (2007). ISBN 9781424418909

[Sai05] Saito, Yasushi: Jockey : A User-space Library for Record-replay Debugging. In: *International Symposium on Automated Analysis-driven Debugging*. Monterey, California, USA : ACM, 2005, 69–76

[SBB+10] Sigelman, Benjamin H. ; Barrosso, Luiz A. ; Burrows, Mike ; Stephenson, Pat ; Plakal, Manoj ; Beaver, Donald ; Jaspan, Saul ; Shanbhag, Chandan: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. In: *Google Research* (2010),

Nr. April, 14. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf

[Tim] *The Timadorus Project | Project Timadorus.* http://timadorus.org/general_publications.html, Abruf: 2019-01-29

[Tim17] *Goals and Aims of the Project Timadorus | Project Timadorus.* http://timadorus.org/general_goals.html. Version: 2017, Abruf: 2019-01-22

[YCW+09] Yang, Junfeng ; Chen, Tisheng ; Wu, Ming ; Xu, Zhilei ; Liu, Xuezheng ; Lin, Haoxiang ; Yang, Mao ; Long, Fan ; Zhang, Lintao ; Zhou, Lidong: MODIST : Transparent Model Checking of Unmodified Distributed Systems. In: *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2009

[YLZ+14] Yuan, Ding ; Luo, Yu ; Zhuang, Xin ; Rodrigues, Guilherme R. ; Zhao, Xu ; Zhang, Yongle ; Jain, Pranay U. ; Stumm, Michael: Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: *Operating Systems Design and Implementation (OSDI)* (2014), 249—-265. http://dx.doi.org/10.1002/anie.201107947. – DOI 10.1002/anie.201107947. – ISBN 9781931971164

[Zip] *OpenZipkin - A distributed tracing system.* https://zipkin.io/, Abruf: 2019-03-10

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 13. März 2019 Sven Allers