Verteilte Benchmarks für Aktor-Systeme

Hauke Goldhammer

Bachelorarbeit

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Hauke Goldhammer

Verteilte Benchmarks für Aktor-Systeme

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Thomas C. Schmidt
Zweitgutachter: Prof. Franz-Josef Korf

Eingereicht am: 7. August 2020

Hauke Goldhammer

Thema der Arbeit

Verteilte Benchmarks für Aktor-Systeme

Stichworte

CAF, Erlang, Akka, Benchmarks, Aktor-Systeme

Kurzzusammenfassung

Software-Systeme müssen heutzutage immer mehr gleichzeitig leisten können, um dies zuerreichen werden immer öfter verteilte Systeme eingesetzt. Diese haben Eigenschaften, die das designen und programmieren komplieziert machen. Verteilte Systeme müssen sowohl robust und skalierbar sein, sowie eine gute Performanz mitbringen. Des Weiteren müssen die mit den folgenden Problemen klar kommen, sie müssen die Datenintegrität sicherstellen und mit dem Ausfall von Knoten zurechtkommen. Um dies einfach zuerreichen eigenet sich das Aktormodell. Welches den Anstatz der isolierten Aktoren verfolgt, diese teilen sich keinen Zustand und ihn nur selber als Reaktion auf eingehende Nachrichten ändern können. Um festellen zukönnen wie gut sich Aktor-Systeme für verteilte Systeme eignen, werden Benchmarks benötigt die diesen Aspekt untersuchen. In dieser Arbeit werden mehrere Benchmarks vorgestellt und verschiedene Aktor-Systeme mit ihnen verglichen. Diese sind Erlang, welches z.B. von WhatsApp verwendet wird, Akka und das C++ Actor Framework.

Hauke Goldhammer

Title of Thesis

Distributed benchmarks for actor systems

Keywords

CAF, Erlang, Akka, benchmarks, actor systems

Abstract

Nowadays software systems have to be able to do more and more at the same time. To achieve this, distributed systems are being used more and more often. They have properties that make designing and programming them complicated. Distributed systems must be both robust and scalable, as well as have a good performance. Furthermore, they must deal with the following problems, ensure data integrity and deal with node failure. The actor model is ideal for achieving this. It follows the approach of isolated actors, thay do not share state and can only change it themselves in response to incoming messages. In order to be able to determine how well actor systems are suitable for distributed systems, benchmarks are required to examine this aspect. In this thesis, several benchmarks are presented and various actor systems are compared with them. These are Erlang, which for example is used by WhatsApp, Akka and the C ++ Actor Framework.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

BASP Binary Actor System Protocol.

BEAM Bodgan's Erlang Abstract Machine.

CAF C++ Actor-Framework.

CLI Command-Line-Interface.

FIFO First In - First out.

HPC Hochleistungsrechnen (engl. high-performance computing) .

JIT-Compiler Just-in-time-Compiler.

JVM Java Virtual Machine.

VM Virtual Machine.

Listings

1 Einleitung

Software-Systeme müssen heutzutage immer mehr gleichzeitig leisten können. Gerade die Internetangebote der großen Unternehmen, wie Amazon, Google und Facebook, müssen viele Anfragen und Aufgaben gleichzeitig bearbeiten. Deshalb werden verteilte Systeme immer beliebter, denn diese können diese Last stemmen. Software für verteilte Systeme zu designen und zu programmieren ist durch die Eigenschaften dieser sehr kompliziert. Sie müssen sowohl robust und skalierbar sein, aber auch gleichzeitig eine gute Performanz mitbringen. Des Weiteren müssen sie mit Ausfällen von Knoten zurechtkommen, die Datenintegrität sicher stellen können und Race Conditions vermeiden.

Das Aktormodell bringt vieles von dem mit, denn Aktoren teilen keinen Zustand und können diesen nur selbst ändern, wenn sie eine Nachricht bekommen und sind dadurch automatisch Threadsicher. Durch dieses Konzept ist die Verteilung einfach möglich, denn der Ort, wo die Aktoren ausgeführt werden, ist für sie transparent.

Deshalb werden Aktor-Systeme für verteilte Anwendungen immer beliebter und die Möglichkeit diese vergleichen zu können immer wichtiger. Ein prominentes Beispiel dafür ist WhatsApp, sie nutzen Erlang im Backend. Ziel dieser Arbeit ist es daher Benchmarks für verteilte Aktor-System zu entwickeln. Da diese im Rahmen der Forschungsarbeit in der Arbeitsgruppe INET[1] der HAW Hamburg entstanden ist, wird auch ein nicht verteilter Benchmark vorgestellt. Dies geschieht, weil dieser Benchmark für ein Publikation während dessen entwickelt wurde.

1.1 Aufbau der Arbeit

Im folgenden Kapitel 2 werden die Grundlagen für diese Arbeit erläutert, dabei werden Aktor-Systeme im allgemeinen vorgestellt und danach wird auf spezifische eingegangen.

[1] https://inet.haw-hamburg.de/

Weiter werden die relevanten wissenschaftlichen Arbeiten aus dem Bereich des Benchmarkings in Kapitel 3 beleuchtet. In Kapitel 4 werden die Benchmarks definiert, diese bauen auf Benchmarks aus dem vorherigen Kapitel 3 auf. Im Kapitel 5 werden Punkte der Implementation erläutert. Das Kapitel 6 behandelt die Ergebnisse der Benchmarkauswertung.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für diese Arbeit erläutert, beginnend mit dem Aktormodell, in dem verschiedene Aktor-Systeme vorgestellt werden. Darauf folgend wird kurz beschrieben, was Benchmarks sind und die dafür benötigten Werkzeuge vorgestellt.

2.1 Das Aktormodell

Das Aktormodell ist ein Modell für nebenläufige und verteilte Programme in der Informatik und wurde erstmals 1973 von Carl Hewitt vorgestellt [14] und von Gul Agha [1] formalisiert.

Ein Aktor ist eine isolierte Einheit, die nur über Nachrichten kommunizieren kann. Jeder Aktor besitzt einen eigenen Speicherbereich und kapselt daher seinen Zustand gegenüber anderen Aktoren ab. Aktoren reagieren auf Nachrichten in drei verschiedenen Formen.

1. Zustand oder Verhalten (Behavior) verändern
2. Eine Nachricht an sich selbst bzw. einen anderen Aktor schicken
3. Einen neuen Aktor erstellen

Die Nachrichtenverarbeitung ist dabei nicht deterministisch, weil eine verlässliche Nachrichtenreihenfolge nicht gewährleistet wird. Das bedeutet, dass Nachrichten sich überholen können, werden aber vom Aktor in der Eingangsreihenfolge (First In - First out (FIFO)) abgearbeitet.

Durch die parallele Ausführbarkeit von Aktoren eignet sie sich besonders gut für die heute übliche Hardware mit mehreren Kernen, genauso wie eine Verteilung über mehrere Knoten möglich ist. Der logische Programmaufbau ist von der Verteilung unabhängig, weil ein Aktor nicht wissen muss, wo sich der Kommunikationspartner befindet, sondern

nur seine ID kennen. Die Zustellung der Nachricht wird vom Aktor-System übernommen. Des Weiteren wird die Verteilung durch das Fehlermodell unterstützt, dieses erlaubt uni- und bidirektionale Überwachung mittels Fehlermeldungen [23]. Diese erhält der überwachende Aktor als Nachricht und kann darauf durch vorher definiertes Verhalten reagieren. Dadurch kann auch auf einen Ausfall eines ganzen Knotens leicht reagiert werden, in dem alle Aktoren des Knotens einfach durch den überwachenden Aktor auf einem anderen Knoten neu gestartet werden. Man kann einen Aktor auch als leichtgewichtigen Thread mit eigenem Speicherbereich beschreiben.

2.1.1 C++ Actor-Framework (CAF)

Das C++ Actor-Framework (CAF) [8] [9] wird seit 2011 an der HAW Hamburg entwickelt und ist eine Implementierung des Aktormodells für C++, aktuell in der Version 0.17.5[1].

Ein Aktor in CAF hat einen getrennten Speicherbereich, dabei ist er nicht echt isoliert, weil dies in C++ nicht garantiert werden kann. Nachrichten empfängt ein Aktor über seine Mailbox. Diese ist als „Single-Reader-Many-Writer-Queue“ implementiert, dies erlaubt paralleles Schreiben in die Mailbox ohne Synchronisierungsmaßnahmen. Die Nachrichten werden vom Aktor in FIFO-Reihenfolge gelesen und abgearbeitet [23]. Es gibt noch eine Mailbox für Nachrichten mit hoher Priorität, welche vor allen anderen Nachrichten, ebenfalls in FIFO-Reihenfolge, abgearbeitet werden.

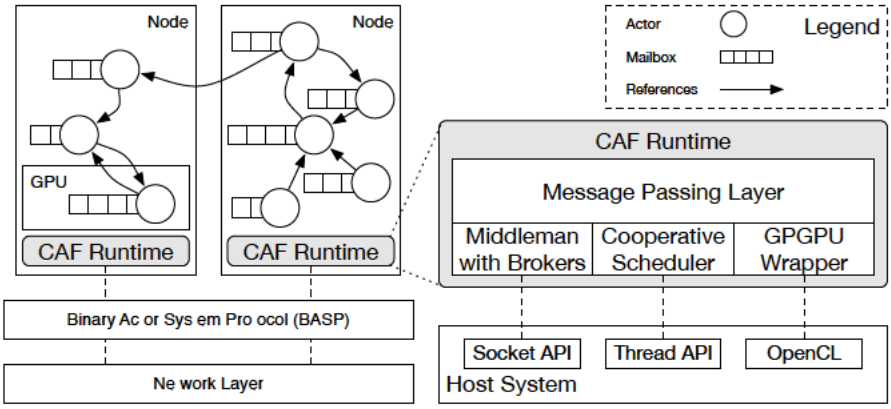


Abbildung 2.1: Die CAF Laufzeitumgebung [9].

[1] https://github.com/actor-framework

CAF stellt zudem eine Laufzeitumgebung (Abb. 2.1) zur Verfügung, die sich um den Nachrichtenversand und das lokale Scheduling kümmert. Der Scheduler organisiert das faire wie parallele Arbeiten der Aktoren. Dabei werden die Aktoren über die Workerthreads verteilt, deren Anzahl beim Start festgelegt wird, meistens einer pro logischem Kern [4]. Dadurch das viele Aktoren von einem Workerthread ausgeführt werden, ist ein sehr schneller Wechsel zwischen ihnen möglich ohne das der Mehraufwand des Threadwechsels dazukommt. Der Standardscheduler ist der „work stealing"-Scheduler, dieser „klaut" Aktoren die auf eine Ausführung warten aus anderen Workerthread-Warteschlangen, wenn er mit seiner fertig ist. Welcher wird dabei zufällig entschieden. Erst dann muss ein Aktor den Thread wechseln und wird von nun an in dem Workerthread ausgeführt. Der andere von, CAF zur Verfügung gestellte, Scheduler ist der „work sharing"-Scheduler, dieser hat nur eine globale Warteschlangen, aus der jeder Workerthread die Aktoren die ausgeführt werden müssen holt. Der Nachrichtenversand wird von einem Multiplexer, dem sogenannten Middleman, organisiert und durchgeführt. Dieser leitet die Nachrichten entweder an die lokalen Aktoren oder über das Netzwerk an die entfernten Aktoren, auf anderen Knoten, weiter. Wenn eine Nachricht an einen anderen Knoten gesendet wird, werden sie mit Hilfe des Binary Actor System Protocol (BASP) serialisiert und deserialisiert. Die OpenCL Schnittstelle, die es ermöglicht Aktoren auf Grafikkarten auszuführen, wird mit der CAF Version 0.18.0 entfernt[2].

2.1.2 Erlang/OTP

Beider nebenläufigen funktionalen Programmiersprache Erlang [2] [3] ist das Aktormodell Teil des Sprachkonzeptes. Die Entwicklung startete 1986 bei Ericsson Telecom AB und wurde im Jahre 2000 als Open Source veröffentlicht [2]. Die Programmiersprache würde ursprünglich für Telekommunikationstechnik entwickelt und die daraus entstehende Software sollte für „immer laufen" [2]. Erlang ist aktuell in der Version 23.0.3[3] veröffentlicht, zusammen mit einer umfangreichen Bibliothekssammlung, der OTP (ursprünglich ein Akronym für Open Telecom Platform).

In Erlang heißen Aktoren „Prozesse", sie teilen keinen Speicher und kommunizieren nur über Nachrichten. Diese werden aktiv, wenn eine Nachricht in der Mailbox passend zu einem Schema gefunden wurde. Ein Aktor kann sich mit einem Namen global registrieren (siehe Listing 2.1 1) und dann über diesen Nachrichten empfangen. Dafür hält Erlang eine

[2] https://github.com/actor-framework/actor-framework/blob/master/CHANGELOG.md
[3] https://github.com/erlang/

```
1       register(name).
2       global:whereis(name).
```

Listing 2.1: Regristieren eines Aktors und die Abfrage danach, wo er zu finden ist.

Datenstruktur, mit dem Namen als Schlüssel für die dazu gehörige Prozess ID. Diese kann über die Funktion, siehe Listing 2.1 2 abgefragt werden [10]. Falls der zu einem Namen gehörige Aktor ausfällt, kann der Name neu zugeordnet werden. Wenn man Erlang verteilt nutzt, baut dieses zwischen allen Knoten ein vollvermaschtes Netzwerk auf [10]. In verteilten Systemen führt der global Namensraum zu hohen Synchronisierungs- und Kommunikationskosten.

Anwendungen laufen in einer virtuellen Maschine, der Bodgan's Erlang Abstract Machine (BEAM) [7]. Sie können aber auch für eine bessere Performanz vom Compiler direkt in Maschinencode übersetzt werden.

2.1.3 Akka

Die Aktor-Bibliothek Akka [24] ist eine Implementierung des Aktormodells für Java und Scala und wird in der Java Virtual Machine (JVM) ausgeführt. Das Projekt wurde 2009 gestartet und ist mittlerweile Teil der Scala Standardbibliothek. Akka ist aktuell in Version 2.6.7[4] und Scala in der Version 2.13.3[5] auf GitHub verfügbar. Scala ist eine Multi-Paradigmen-Sprache, die die Features von objektorientierten und funktionalen Programmiersprachen kombiniert[7]. Objektorientierte Features sind zum Beispiel jeder Wert ist ein Objekt und jede Operation ein Methodenaufruf. Dazu kommen einige funktionale Features, wie z.B. Currying und Objekte können als unveränderlich deklariert werden. Des Weiteren hat Scala ein starkes statisches Typsystem und daher werden alle Nachrichten in Akka zur Laufzeit auf ihre Typen überprüft. Auf Grund dessen, dass Scala zu Byte Code für die JVM gebaut wird, können in Scala bzw. Akka Klassen aus Java benutzt werden und man hat somit Zugriff auf alle Bibliotheken, die für Java geschrieben wurden [26]. In Akka sind Aktoren ebenfalls Einheiten, die durch Nachrichten kommunizieren. Wie in CAF können auch sie Referenzen auf gemeinsamen Speicher per Nachricht bekommen und somit das Aktormodell verletzen. Dies wird hier ebenfalls als

[4] https://github.com/akka/
[5] https://github.com/scala/

schlechter Stil betrachtet, lässt sich aber nicht verbieten [26]. Das Fehlermodell nutzt den „lass-es-abstürzen“ Ansatz, unterstützt aber auch Überwachungshierarchie mit Aktoren [5]. Akka bietet mehrere Module an, die den Aktorsprachkern um verschiedene Funktionen erweitert, zum Beispiel ein Module um Clustering möglich zumachen [25]. Genau wie Erlang und CAF garantiert Akka keine Nachrichtenzustellung.

2.2 Benchmarking

In der Informatik wird Benchmarking als Akt der Messung und Bewertung der Leistung von Hardware, Software oder Netzwerken unter vergleichbaren Konditionen definiert. Das Ziel dabei ist ein fairer Vergleich von verschiedenen Lösungen unter festgelegten Vergleichsmetriken [6]. Ein Benchmark beinhaltet eine Reihe von Spezifikationen, die eine Bewertung möglich machen. Dazu gehört ein definiertes Szenario, die Leitungskriterien, die Leistungsmetrik und ein Ergebnis. Dabei sollte ein Benchmark die folgenden Eigenschaften [11] erfüllen:

Linearität bedeutet, wenn z.B die Leistung eines CPU's 3 mal höher als die einer anderen CPU ist, dann sollte sich dies im Ergebnis des Benchmark widerspiegeln.

Verlässlichkeit, wenn das Ergebnis für eine CPU höher ist als das einer anderen sollte die Leistung in dieser Metrik immer höher sein.

Reproduzierbarkeit, das Benchmarkergebnis sollte das gleiche sein, egal wie oft der Benchmark wiederholt wird.

Einfachheit der Messung, ist dies nicht gegeben ist eine höhere Wahrscheinlichkeit für Fehler zu erwarten.

Konsistenz, die Definition der Leistungsmetrik ist auf unterschiedlichen Maschinen valide.

Um die Vergleichbarkeit der Ergebnisse zu gewährleisten muss neben diesem auch die genaue Versuchsumgebung veröffentlicht werden. Dazu gehören zum Beispiel bei einem Softwarebenchmark die genauen Spezifikationen des Computers auf dem er ausgeführt wurde [15].

In der Informatik wird häufig zwischen Mikro- und Makrobenchmarks (im englischen real-world-benchmark) unterschieden. Bei Mikrobenchmarks wird nur ein spezifischer

Aspekt der Hardware, Software oder des Netzwerks untersucht. Ein Beispiel hierfür im Aktorumfeld wäre das Messen des Nachrichtendurchsatzes zwischen Aktoren. Unter Makrobenchmark versteht man dagegen ein Programm, dass viele verschiedene Aspekte auf einmal testet bzw. eines welches eine reale Anwendung aus dem Umfeld des Testobjekts simuliert.

2.3 Verwendete Werkzeuge

In diesem Abschnitt werden alle in dieser Arbeit für das Benchmarking verwendete Werkzeuge vorgestellt.

2.3.1 Mininet

Mit dem Netzwerk-Emulator Mininet [20] kann man sich auf jedem Computer ein virtuelles Netzwerk erzeugen und mit diesem netzwerkbasierte Tests durchführen, ohne ein reales Testnetz bereitstellen zu müssen.

Mininet emuliert virtuelle Hosts, Switches, Links. Ein „Mininet“ wird von einem Controller instanziiert und gesteuert. Diese nutzen den „echten“ Kernel-, Switch- und Anwendungscode, dadurch ist die Virtualisierung sehr leichtgewichtig [18].

Ein Netzwerk wird über eine Python-API definiert und über das Command-Line-Interface (CLI) getestet oder verändert. Für das Testen hat man einige Standardwerkzeuge zur Verfügung, wie zum Beispiel ein Werkzeug, dass die Erreichbarkeit von allen Hosts überprüft. Des Weiteren kann man über das CLI eigene Programme auf jedem Hosts starten, z.B. Benchmarks. Netzwerktopologien gibt es vordefinierte wie eine einfache Host-Switch-Host-Topologie aber auch komplexere Baum- und Ringtopologien. Man kann sich über die Python-API auch eine eigene definieren. Für das Testen von verteilten Anwendungen hilfreich ist, dass man den Paketverlust und die Latenz einstellen kann.

3 Verwandte Arbeiten

In diesem Kapitel werden für diese Arbeit relevante wissenschaftliche Arbeiten vorgestellt. Beginnend mit Benchmarking im Allgemeinen. Darauf folgen Beispiele für Benchmarks, die nur auf einem Knoten ausgeführt werden und zum Schluss wird auf verteilte Benchmarks eingegangen.

3.1 Kriterien für Benchmarks

In der Veröffentlichung „How to Build a Benchmark“ [17] wird ein Benchmark als ein Standartwerkzeug für die Bewertung und den Vergleich von konkurrierenden Systemen oder Komponenten nach bestimmten Merkmalen wie z.B. Leistung, Zuverlässigkeit oder Sicherheit definiert. Hierfür reicht nicht nur ein Benchmark aus, um alle Arbeitsbereiche abzudecken, weil jede Designentscheidung Einfluss auf die Stärken oder Schwächen des Systems hat und somit jedes Arbeitsumfeld einen eigenen Benchmark benötigt. Diese Benchmarks sollten die folgenden fünf Kriterien erfüllen.

Relevanz, warum ist der Benchmark für dieses Umfeld relevant und was sagen die Ergebnisse aus.

Reproduzierbarkeit, der Benchmark sollte so gestaltet werden, dass auf der selben Maschine immer das gleiche Ergebnis resultiert.

Fairness, der Benchmark sollte so definiert und implementiert werden, dass verschiedene Konfigurationen nicht benachteiligt werden.

Überprüfbarkeit, Benchmarks sollten Validierungsmöglichkeiten beinhalten, zum Beispiel alle Konfigurationen sind im Ergebnis enthalten.

Benutzerfreundlichkeit, dies ist vor allem wichtig, um eine selbst Validierung zu ermöglichen.

3.2 Vergleichbarkeit von Benchmarks

Die Studie „Scientific Benchmarking of Parallel Computing Systems“ [15] der ETH Zürich hat 120 wissenschaftliche Arbeiten, die sich mit Hochleistungsrechnen (engl. high-performance computing) (HPC) beschäftigen, auf die Reproduzierbarkeit der Messergebnisse untersucht. Hierbei stellten die Autoren fest, dass ein großer Teil dieser Arbeiten nicht genug Details beinhalten, um die Ergebnisse zu interpretieren und erst recht nicht genug, um sie zu reproduzieren. Um die Ergebnisse der Studien reproduzieren zu können, benötigt man die Spezifikationen des Computers auf dem die Benchmarks ausgeführt wurden. Ebenfalls sollte angeben werden, welcher Benchmark wie oft und unter welchen Voraussetzungen ausgeführt wurde. Um dies bei zukünftigen Studien zu erreichen wurden zwölf Regeln erarbeitet, welche zu einer deutlich besseren Vergleichbarkeit der Ergebnisse im Bereich des HPCs führen sollen. Diese beginnen mit Regeln für Mittelwertbildung, gehen über zu den erforderlichen Versuchsdetails und regeln, was in einem Ergebnisgraph enthalten sein sollte. Besonders die Gesetze zur Mittelwertbildung sind für diese Arbeit wichtig, diese lauten:

3. Verwenden sie das arithmetische Mittel nur zur Zusammenfassung der Kosten.

4. Vermeiden sie es, Verhältnisse zusammenzufassen. Fassen sie stattdessen die Kosten oder Sätze zusammen, auf denen die Kennzahlen basieren. Nur wenn diese nicht verfügbar sind, verwenden Sie das geometrische Mittel zur Zusammenfassung der Verhältnisse.

5. Geben sie an, ob die Messwerte deterministisch sind. Geben Sie für nicht deterministische Daten die Konfidenzintervalle der Messung an.

Viele der Regeln zur Mittelwertbildung wurden schon in der Veröffentlichung „How not to lie with Statistics: The correct way to summarize benchmark results“ [12], welches sich nur mit Benchmarkstatistiks beschäftigt, aufgestellt.

In „Cross-Language Compiler Benchmarking“ [19] werden verschiedene Java Compiler verglichen. Der für diese Arbeit interessante Teil sind die in der Veröffentlichung vorgestellten Ziele für das Cross-Language Benchmarking. Benchmarks, die mehrere Programmiersprachen vergleichen wollen, sollten mit diesem Ziel entwickelt werden. Hierfür empfiehlt sich relevante Abstraktionen zu verwenden, welche in allen verglichenen Programmiersprachen vorkommen, Beispiele dafür sind Arrays und Listen. Dies ist auch für die Portabilität wichtig, denn alle im Benchmark verwendeten Funktionen, primitive

wie komplexe, müssen in allen Sprachen vorhanden sein. Man definiert jeweils für einen Benchmark eine sogenannte Kernsprache, welche alle Funktionen und Abstraktionen enthält, die für diesen wichtig sind. Dabei sollte die Balance zwischen Portabilität und den für das Arbeitsumfeld relevanten Funktionen gehalten werden. Ein weiteres Ziel ist, dass alle Benchmarks einfach auszuführen sind und die benötigte zusätzliche Software gering gehalten wird. Die Benchmarks selbst sollten so identisch wie möglich implementiert werden. Dies wird durch die vorher definierte Kernsprache unterstützt. Folglich unterstützt dies alles die Fairness und Verständlichkeit des Benchmarks. Alle hier besprochenen Ziele sind im Einklang zu den Kriterien aus [15] und sorgen für bessere Vergleichbarkeit von Benchmarks in verschiedenen Sprachen.

„The Computer Language Benchmark Game“ [13] ist ein Webprojekt, welches versucht für möglichst viele Programmiersprachen Implementierungen zu ihren Benchmarks zu sammeln. Derzeit sind es über 25 Sprachen für vierzehn verschiedene Benchmarks. Für diese Cross-Language Benchmarks gilt es soll der selbe Algorithmus verwendet werden und das Ergebnis muss übereinstimmen, sonst sind keine weiteren Einschränkungen festgelegt. So gibt es für einige Sprachen mehr als eine Implementation, z.B. eine sequenzielle und eine parallele, was aber die Vergleichbarkeit reduziert.

3.3 Beispiele für Benchmarks

Im folgenden Abschnitt werden Beispiele für Benchmarks vorgestellt, die mit dem Aktormodell arbeiten oder im Bezug dazu stehen.

3.3.1 Knotenlokale Benchmarks

Zu erst werden Benchmarks vorgestellt, die auf nur einer Maschine laufen und nicht mit anderen über ein Netzwerk kommunizieren.

Die Savina Benchmark Suite [16] umfasst 30 verschiedene Benchmarks, die das Ziel hat, die Performanz von aktororientierten Bibliotheken und Programmiersprachen in verschiedenen Bereichen zu vergleichen. Sie umfasst Benchmarks, die sich mit Nachrichtenaustausch sowie Synchronisierungsproblemen und Nebenläufigkeitsproblemen befassen. Die Benchmarks wurden von ihnen für 10 Java/Scala Aktor-Frameworks implementiert und getestet, von diesen hat kein Framework durchgehend überzeugt. Inzwischen gibt

es auch Implementierungen in anderen Programmiersprachen wie Pony [5] und in C++ von CAF [27]. Aus den 30 Benchmarks kann man die Benchmarks benutzen, die sich mit den Nachrichtenaufwand beschäftigen, um den Unterschied zwischen verteilter und lokaler Kommunikation zu analysieren.

Das Pony-Team schaut sich in ihrer Veröffentlichung „Run, Actor, Run“ [5] die Savina Benchmark Suite an und implementiert einige der Benchmarks in Pony. Dabei stellen sie fest, dass ein großer Teil der Benchmarks aus dem Umfeld der Threads kommen. Diese wurden so implementiert, dass sich die Aktoren eher wie Threads verhalten und nicht wie Aktoren. Des Weiteren stellen sie fest, dass einige der Benchmarks das Aktormodell verletzen, indem sie zum Beispiel mit geteiltem Speicher oder Synchronisierungsmethoden arbeiten. Deswegen haben sie sechs Regeln für das sprachübergreifende Benchmarking, mit Schwerpunkt auf Aktoren, erstellt.

Kein Verletzen des Aktormodells Eine wichtige Eigenschaft des Aktormodells ist die Isolation eines Aktors. Die Kommunikation findet nur über Nachrichten statt und nicht über z.B. gemeinsam genutzte Datenstrukturen oder Synchronisierungsmethoden wie ein Mutex.

Aufräumen außerhalb der Messzeit Aktorprogramme sind sehr langlebige Programme und werden daher selten hoch- bzw. heruntergefahren, deshalb würde das Messen dieser nur zu Ungenauigkeiten führen.

Kein selektiver Empfang Es sollte für einen Benchmark nicht notwendig sein gegen das Aktormodell zu verstoßen, in den man Nachrichten nicht in der Eingangreihenfolge abarbeitet, sondern sie z.B. zurückstellt.

Kein Fokus auf hoch optimierte Bibliotheken Gemessen werden soll das Framework oder die Programmiersprache und nicht eine hoch optimierte Bibliothek wie z.B. BigDicimal aus Java.

Kein Fokus auf parallele Algorithmen Paralleles Verarbeiten von großen geteilten Daten kommt im Aktorumfeld nicht so häufig vor und unterzieht dem asynchronen Verhalten von Aktoren keinem Stresstest.

Daher schlagen sie in ihrer Publikation einen neuen Benchmark vor, der diese Regeln beachten soll, die ChatApp. Dieser Benchmark soll durch viele Stellschrauben möglichst viele verschiedene Aspekte vergleichbar machen. Das erste Konzept sieht drei verschiedene Aktortypen vor, das Verzeichnis, den Client und den Chat. Clients führen pro Runde

eine Aktion aus, welche reale Eingaben simulieren soll. Der Benchmark ist dabei so aufgebaut, dass N Clients einem Verzeichnis zugeordnet sind und sich in M Chat unterhalten. Die ChatApp soll für ihre nächste Publikation ausgearbeitet und implementiert werden. Das CAF-Team hat das überarbeitete Konzept erhalten und wurde gebeten den Benchmark in CAF zu implementieren, dies wird ebenfalls in dieser Bachelorarbeit geschehen, obwohl der Benchmark in ihrem Konzept nicht verteilt ist.

In „Many Spiders Make a Better Web - A Unified Web-Based Actor Framework“ [21] wird das JavaSkript Aktor Framework Spider.js vorgestellt, welches Parallelität und Verteilung mitbringt. JavaSkript ist eigentlich eine Ein-Thread-Umgebung, daher ist Nebenläufigkeit nur mit der später hinzugekommenen „web worker“ API auf der Clientseite und dem „child processes“ Module auf der Serverseite möglich. Um die Leistungsfähigkeit von Spider.js zu überprüfen nutzen sie die Savina Benchmark Suite [16]. Dabei stellen sie fest, dass ihr Aktor Framework bei allem Benchmarks mindesten halb so schnell ist wie die „web worker“ API von JavaSkript ist. Dies kommt vor allem durch den Mehraufwand der Aktorerstellung und des Nachrichtenversands.

Die wissenschaftliche Arbeit „Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka“ [26] vergleicht die drei Programmiersprachen, von denen zwei das Aktormodell beinhalten, in den für eine Webanwendung wichtigen Eigenschaften. Diese sind eine schnelle Interprozesskommunikation, kurze Prozesserstellungszeit sowie eine hohe Anzahl an unterstützten Prozessen. Um diese Eigenschaften zu vergleichen haben sie zwei Mikrobenchmarks erstellt. Der erste schaut sich die Nachrichtenlaufzeit zwischen zwei Prozessen an und der zweite die Zeit, die benötigt wird, um einen Prozess zu starten, dabei wird die Anzahl der zu startenden Prozesse immer weiter bis auf 100.000 erhöht. Der letzte Benchmark soll ein Teil einer typischen Webanwendung simulieren, die die Kommunikation zwischen einem Client der Anfragen stellt und einem Worker, der diese bearbeitet. Dieser letzte Benchmark zusammen mit der vorgestellten simplen Architektur einer Webanwendung lässt sich gut in einen verteilten Benchmark portieren, indem man die Clients von einer variablen Anzahl von Knoten mit einem Server reden lässt, der ihre Anfragen bearbeitet.

3.3.2 Verteilte Benchmarks

Die hier vorgestellten Benchmarks laufen auf mehreren Maschinen gleichzeitig und kommunizieren dabei über Netzwerk miteinander.

Die Erlang Benchmarks Suite [3] enthält parallele und verteilte Benchmarks, die sie in zwei Kategorien aufteilen, die synthetischen und die realitätsbezogenen Benchmarks. Erstere untersuchen einen spezifischen Skalierbarkeitsaspekt, daher gehören sie zu den Mikrobenchmarks. Darunter fallen Benchmarks wie z.B. „bang“ und „big“, die sich mit Nachrichtenversand bzw. -durchsatz beschäftigen. Die zweiten sind Makrobenchmarks. Ein Beispiel hierfür wäre der Benchmark „serialmsg“, er beschäftigt sich mit der Nachrichtenverteilung, ähnlich zu der Webanwendung aus [26]. In ihren Benchmarks geht es, wie der Name der Arbeit vermuten lässt, darum, wie gut Erlang in verschiedenen Aspekten skaliert, sei es auf einer Maschine oder auf mehreren. Dabei nutzen sie die Parameter:

Anzahl der Nodes, auf wie vielen Virtual Machine (VM) läuft der Benchmark.

Anzahl der Kerne, die auf einer Maschine dem Benchmark zur Verfügung stehen.

Anzahl der Scheduler, dies ist vergleichbar mit der Anzahl der Workerthread die CAF startet, üblich ist einer pro logischem Kern.

Sie kommen zu dem Ergebnis, dass einige ihrer Benchmarks gut skalieren, andere aber nicht. Unter diesen sind vor allem die Benchmarks zu finden, die auf Funktionen zugreifen die Synchronisierungsmethoden beinhalten, zum Beispiel im Benchmark „parallel“ die Abfrage eines Zeitstempels.

Aus dieser Benchmark Suite eignen sich die Benchmarks „bang“, „big“ und „serialmsg“ als verteilte Benchmarks, weil alle drei nur wenige Abstraktionen verwenden und daher einfach zu portieren sind. Des Weiteren sind sie schon als zum Teil verteilte Anwendung konzeptioniert. Aus dem „serialmsg“-Benchmark zusammen mit der typischen Webanwendung aus der Veröffentlichung [26] lässt sich ein einfacher Makrobenchmark erstellen.

4 Design der Benchmarks

In diesem Kapitel wird das Design der in dieser Arbeit verwendeten Benchmarks vorgestellt, sie basieren auf Benchmarks die im letzten Kapitel 3 vorgestellt wurden.

4.1 Mikrobenchmarks

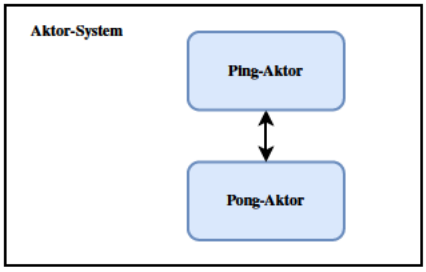
Begonnen wird mit den Mikrobenchmarks, die nur einen bestimmten Aspekt untersuchen sollen.

4.1.1 Nachrichtendurchsatz zwischen Aktoren

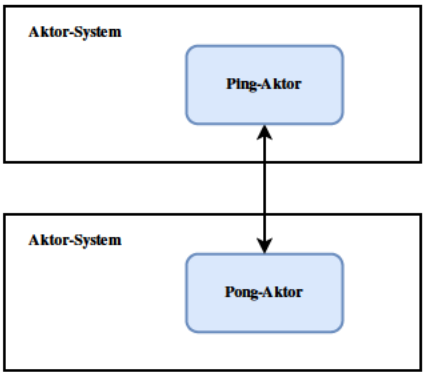
In dem nachfolgenden Abschnitt wird beschrieben, wie sich der Nachrichtendurchsatz zwischen zwei lokalen Aktoren zu dem von zwei verteilten Aktoren unterscheidet.

4.1.1.1 Pingpong

Um dies herauszufinden eignet sich der Pingpong Benchmark aus der Savina Benchmark Suite [16], dieser misst die Laufzeit des Aktor-Systems für n Nachrichten. Um den Nachrichtendurchsatz zu messen wird dieser so verändert, dass er so viele Ping-Pong-Nachrichten wie möglich zwischen zwei Aktoren austauscht. Dabei werden alle Nachrichten in einem festen Zeitintervall von einer Sekunde gezählt und dieser Wert gespeichert. Die Zählung findet im Ping-Aktor statt und wird nach m Wiederholungen beendet. Um den Nachrichtendurchsatz vergleichen zu können, wird der Benchmark lokal und verteilt ausgeführt. Die Abbildung 4.1a zeigt die lokale Variante des Benchmarks, bei der beide Aktoren im gleichen Aktor-System laufen und Nachrichten direkt zugestellt werden. Bei der verteilten Variante wird je einer der Aktoren in einem eigenen Aktor-System gestartet, wie in Abbildung 4.1b zusehen ist. Bei diesem Benchmark wird erwartet, dass



(a) Nachrichtenaustausch lokal zwischen zwei Aktoren.



(b) Nachrichtenaustausch zwischen Aktoren in zwei Aktor-Systemen.

Abbildung 4.1: Der Pingpong-Benchmark.

der verteilte Nachrichtendurchsatz kleiner ist als der lokale, weil der Serialisierungs- und Deserialisierungsauswand dazukommt. Des Weiteren soll auch der Einfluss von Netzwerklatenzen untersucht werden. Diese werden durch den Einsatz von Mininet festgelegt. Bei dem durch dieses Werkzeug erzeugten virtuellen Netzwerk lässt sich dieser, genau wie der Paketverlust, genau festlegen. Dafür wird je ein Aktor-System auf einem virtuellen Knoten gestartet. Dadurch entsteht eine Metrik von Nachrichten pro Sekunde abhängig von der Netzwerklatenz.

4.1.2 Nachrichtenverhalten in einem verteilten System

Wie skalieren Nachrichtenszenarios in einem verteiltem Aktor-System mit N Knoten? Dies zu untersuchen ist die Aufgabe der folgenden beiden Benchmarks.

4.1.2.1 Big

Der erste Benchmark basiert auf dem big Benchmark aus der Erlang Benchmarks Suite [3], der n Aktoren in einen System startet, welche jeweils einmal eine Ping-Nachricht an alle senden und eine Pong-Nachricht von jedem darauf erwarten. In einen verteilten Szenario wird auf jedem Knoten ein Aktor-System mit einen Aktor gestartet. Wie in Abbildung 4.2 dargestellt tauschen in diesem Benchmark alle Aktoren auf allen Knoten Ping-Pong-Nachrichten aus. Um das Skalierungsverhalten in diesem homogenen System zu untersuchen, wird dies mit unterschiedlich vielen Knoten durchgeführt. Jeder neue

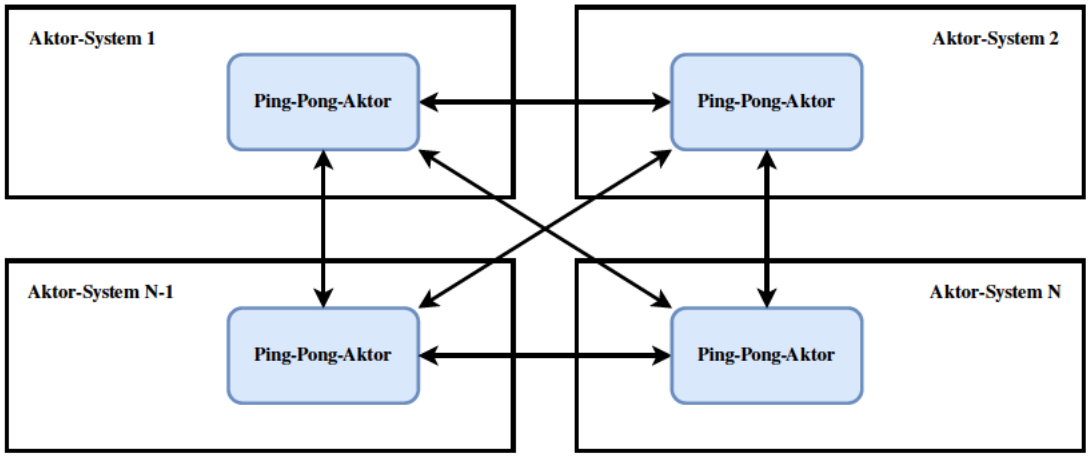


Abbildung 4.2: Ping-Pong-Nachrichtenaustausch zwischen n Aktor-Systemen in beide Richtungen.

hinzugefügte Knoten erhöht die Anzahl der Nachrichten um den Faktor n^2, daher wird die Gesamtlaufzeit ebenfalls exponentiell ansteigen. Die Metrik dieses Benchmarks ist „Nachrichten pro Sekunde“ abhängig von der Anzahl der Knoten und wie im vorherigen Benchmark auch der Netzwerklatenz.

4.1.2.2 Bang

Der zweite Benchmark zu dieser Frage basiert auf dem bang Benchmark ebenfalls aus der Erlang Benchmarks Suite [3], welcher von n Aktoren Ping-Nachrichten an genau einen Aktor sendet, auf diese wird mit einer Pong-Nachricht geantwortet. Abbildung 4.3 zeigt, dass in einen verteilten Szenario n Aktor-Systeme mit je einem Aktor gestartet werden. Diese Aktoren führen den Nachrichtenaustausch mit genau einem Aktor auf einem weiteren Knoten durch. Auch bei diesem wird das Skalierungsverhalten durch eine verschiedene Anzahl vom Knoten untersucht. Die Metrik dieses Benchmarks ist Nachrichten pro Sekunde abhängig von der Anzahl der Knoten und wie im vorherigen Benchmark auch von der Netzwerklatenz. Bei diesen beiden Benchmarks steigt der Nachrichtendurchsatz bis zu einem gewissen Level und könnte sinken sobald die Anzahl der zu verwaltenden Knoten das Aktor-System ausbremst.

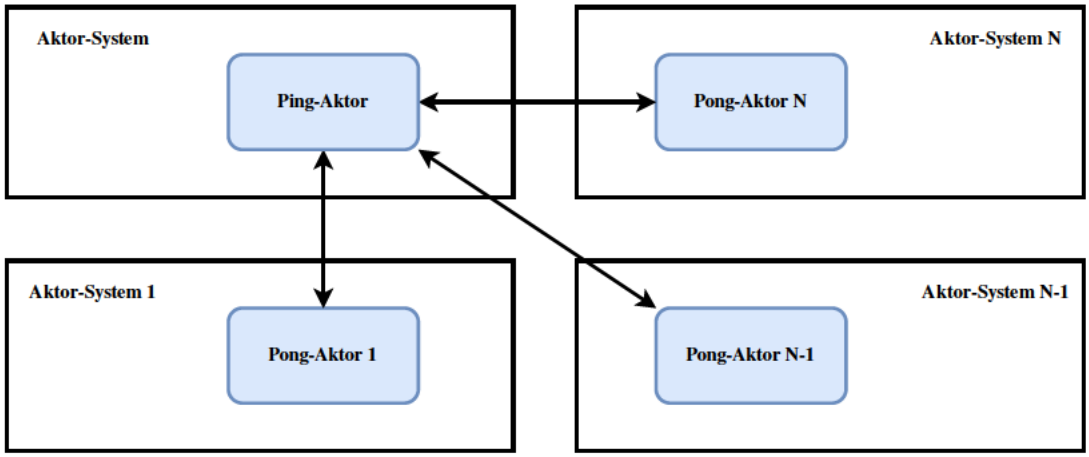


Abbildung 4.3: Bang-Benchmark mit n Knoten.

4.2 Makrobenchmarks

Laufzeitunterschiede zwischen Aktor-Systemen in realen Anwendungen? Dies zu untersuchen ist Aufgabe von Makrobenchmarks, also die Benchmarks, die mehr als ein Aspekt von Systemen oder Anwendungen auf einmal testen. Deshalb werden zwei vereinfachte Anwendungen vorgeschlagen, die auch in der Realität Aktor-Systeme verwenden.

4.2.1 Webanwendung

Eine typische *Webanwendung* bekommt Anfragen von Clients, die dann verarbeitet und beantwortet werden müssen [26]. Dies lässt sich mit dieser vereinfachten Architektur beschreiben. Wie in der Abbildung 4.4 zu sehen ist besteht diese aus Clients, welche die Anfragen (Requests) an den Server schicken. Einem Dispatcher, der die Anfrage an einen freien Worker weiterleitet, der diese dann verarbeitet und den Client das Ergebnis zurückschickt. Dadurch entsteht ein Nachrichtenverlauf, der vom Client über den Dispatcher zum Worker und wieder zum Client geht. In diesem Benchmark erzeugt jeder Client n Anfragen an den Server und wartet auf die Bearbeitung, bevor er die nächste verschickt. Die Kommunikation zwischen Clients und dem Server findet über das Netzwerk statt und um dessen Einfluss auf die Laufzeit zu minimieren wird wieder ein virtuelles Netzwerk durch das Werkzeug Mininet verwendet. Die Parameter diese Benchmarks sind die Anzahl der Worker im Server, die Anzahl der Clients und die Anzahl der

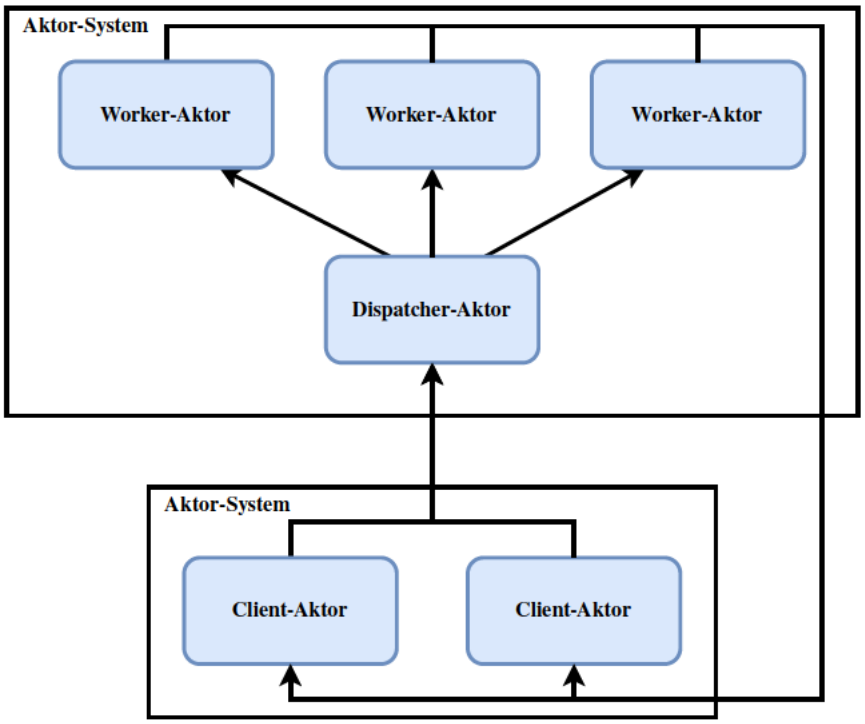


Abbildung 4.4: Architektur des Web-Benchmarks.

Anfragen pro Client. Bei diesem Benchmark kommt es auf schnelle Kommunikation in einem Knoten und auf den Support vieler Worker in einem Knoten an. Dadurch soll eine kurze Antwortzeit für den Client entstehen. Bei diesem Benchmark sind verschiedene Metriken möglich. Zum einem kann man die Dauer jeder Anfrage an der Server messen, also die Dauer vom absenden der Anfrage bis die Antwort beim Client ankommt. Des Weiteren kann man auch die Gesamtlaufzeit bei n Anfragen in verschiedenen Client Worker Kombinationen messen.

4.2.2 Chatanwendung

Das Benchmark *Chatanwendung* zielt darauf ab, typische Aspekte eines Aktor-Systems wie Senden und Erhalten von Nachrichten, Streit um die Mailbox, aber auch die Erstellung und Zerstörung von Aktoren, egal ob manuell oder vollautomatisch. Dabei soll man einstellen können, welchen der Aspekte man wie stark testet. Das Konzept des Chatanwendungsbenchmarks vom Pony-Team [5] sieht wie folgt aus. Die Anwendung wird in drei Aktortypen aufgeteilt, den Client, das Verzeichnis und den Chat. Zusätzlich gibt es noch den Poker-Aktor und den Aktortyp Accumulator, diese beiden gehören nicht zur eigentlichen Anwendung, sondern stimulieren bzw. messen diese. Der Chat-Aktor führt eine Liste über alle teilnehmenden Clients und leitet alle Textnachrichten an seine Mitglieder weiter. Der Verzeichnis-Aktor führt eine Liste über alle seine Clients und stößt jede Runde alle Clients per act-Nachricht an. Eine Runde wird durch den Poker-Aktor gestartet, dieser erstellt erst den für diese Runde zuständigen Accumulator-Aktor und diese Referenz wird allen Clients über das Verzeichnis mitgeteilt. Der Client-Aktor simuliert einen Teilnehmer im Netzwerk und wird einem Verzeichnis (Directory) zugeordnet. Er besitzt eine Liste von Freunden, andere Clients und eine Liste von Chats denen er angehört. Die Liste der Freunde wird beim Start der Anwendung durch das Verzeichnis befüllt und hat mindestens einen Eintrag. Der Client führt jede Runde eine Aktion durch, diese wird zufällig durch einen einfachen Pseudozufallsgenerator entschieden. Die möglichen Aktionen sind:

Einen neuen Chat erstellen: Der Client erstellt einen neuen Chat und lädt einige seiner Freunde zu diesem ein, dafür iteriert er über seine Freundesliste und entscheidet per Zufall ob dieser eingeladen wird oder nicht. Wie in Abbildung 4.5 zu sehen ist, wird der neue Chat zuerst erstellt, dann werden die zu einladenden Freunde ausgewählt und den Accumulator über die bump-Nachricht mitgeteilt. Nun wird der Chat über alle Teilnehmer informiert und fügt sie seiner Teilnehmerliste hinzu.

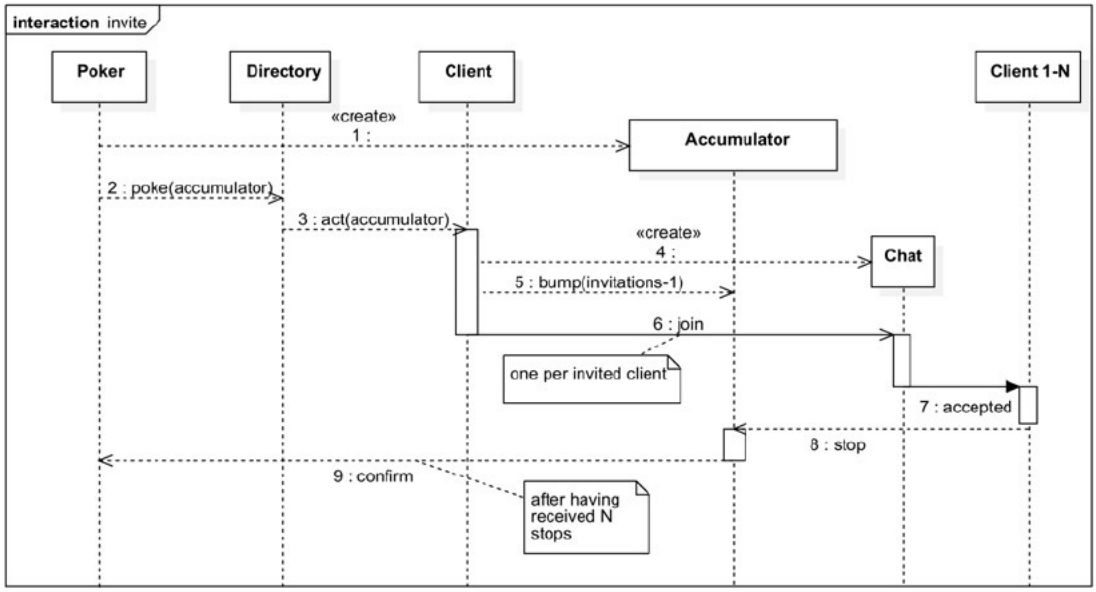


Abbildung 4.5: Ablauf der Aktion neuen Chat erstellen[22].

Zuletzt teilt der Chat allen neuen Teilnehmern mit, dass sie eingeladen wurden. Diese informieren den Accumulator mit einer stop-Nachricht darüber, dass die Aktion abgeschlossen ist. Für den Accumulator ist die Aktion erst zu Ende, wenn er von allen teilnehmenden Clients ein stop bekommen hat, erst dann teilt er dem Poker-Aktor mit das diese Runde beendet ist.

Eine Berechnung durchführen: Es werden mehrere Fibonaccizahlen berechnet. Die Abbildung 4.6 zeigt den einfachen Ablauf der Rechenaktion. Wenn ein Client diese Aktion zufällig wählt, fängt er an zurechnen und teilt dem Accumulator mit wenn dies abgeschlossen ist.

Einen Chat verlassen: Der Client verlässt einen der Chats in dem er Mitglied ist. Der Ablauf dieser Aktion zeigt Abbildung 4.7. Zu beginn der Aktion wird per Pseudozufallsgenerator ein Chat ausgewählt, der verlassen werden soll. Der Chat wird dann durch eine leave-Nachricht darüber informiert und bestätigt den Client das verlassen mit einer left-Nachricht. Sollte dies der letzte Teilnehmer gewesen sein beendet sich der Chat. Der Client meldet dem Accumulator am Ende der Runde, dass diese Aktion erfolgreich beendet wurde.

In einem Chat schreiben: Der Client verfasst eine Nachricht in einem der Chats, den

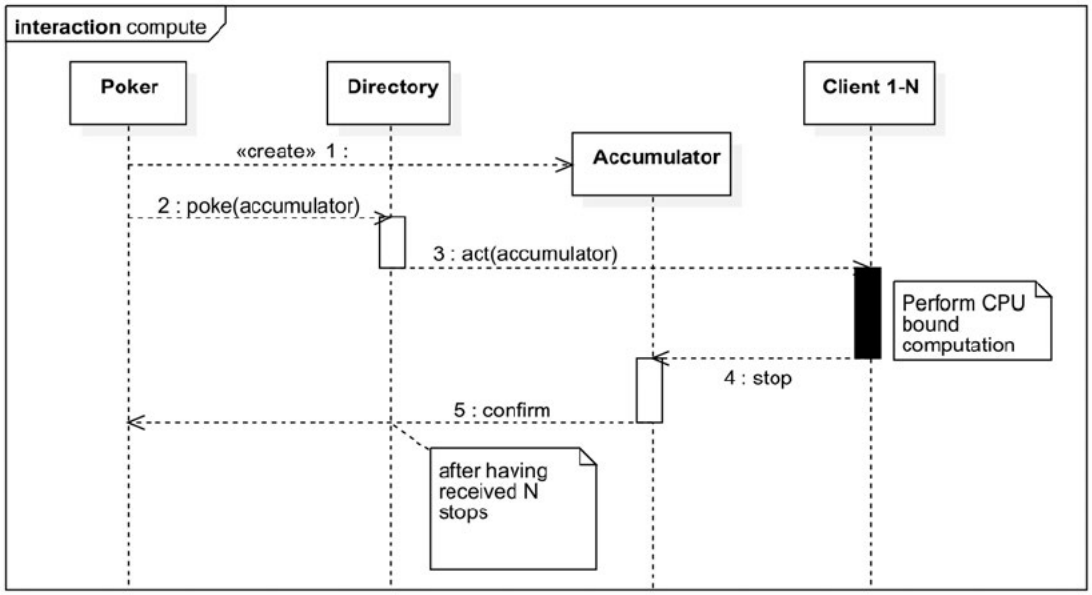


Abbildung 4.6: Ablauf der Aktion eine Berechnung durchführen[22].

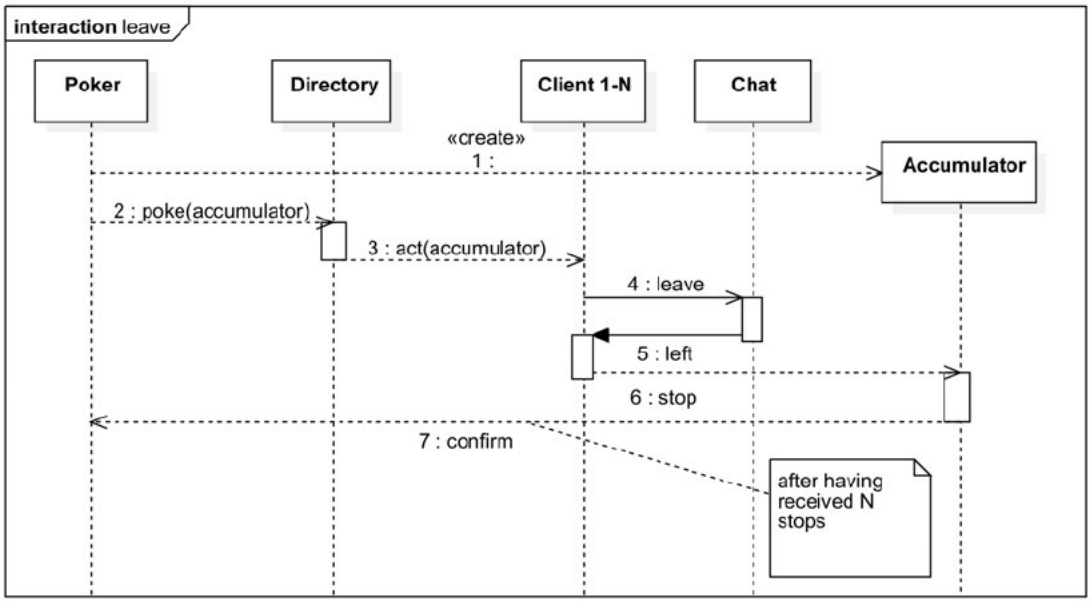


Abbildung 4.7: Ablauf der Aktion einen Chat verlassen[22].

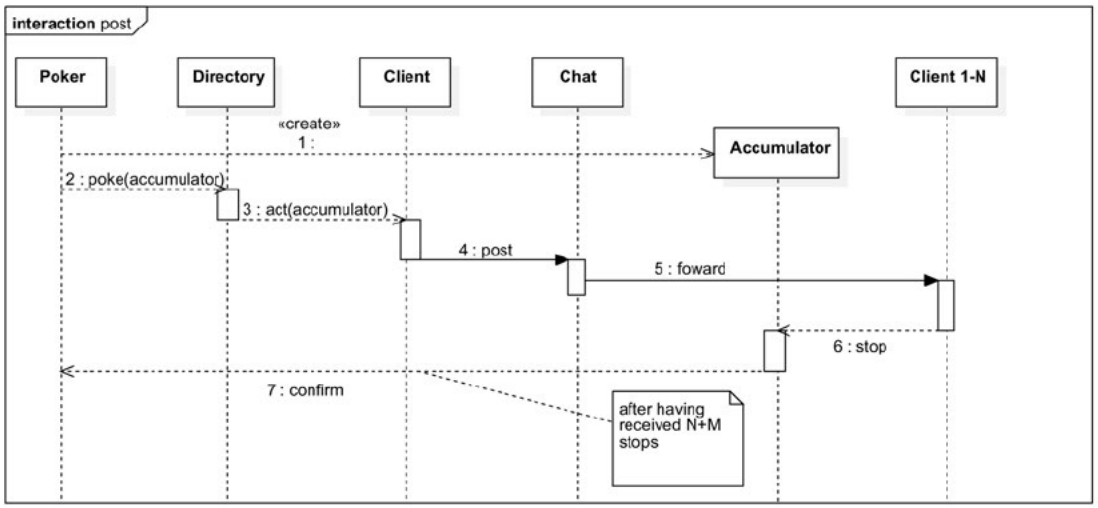


Abbildung 4.8: Ablauf der Aktion in einen Chat schreiben[22].

er angehört. Soll in einen Chat geschrieben werden läuft es wie folgt ab, siehe Abbildung 4.8. Der Client wählt zufällig einen Chat in den er schreiben möchte aus und informiert diesen mit einer post-Nachricht. Der Chat schickt dem Accumulator eine bump-Nachricht mit der Anzahl an Teilnehmer an diesem Chat. Danach schickt er allen seinen Teilnehmern mit der forward-Nachricht die Nachricht, dieser beenden diese Runde mit der stop-Nachricht an den Accumulator.

Runde aussetzen: Dies tritt nur auf wenn die eigentliche Aktion nicht durchführbar war. Dies kann auftreten, wenn ein Chat verlassen werden soll, aber keine vorhanden sind oder in einen Chat geschrieben werden soll, aber dieser Client in keinem Mitglied ist.

Die Parameter in diesem Benchmark sind die Anzahl der Verzeichnisse, die Anzahl an Clients pro Verzeichnis und die Anzahl an Runden die jeder Client ausgeführt. Er ist beendet, wenn alle Runden und die dadurch ausgelosten Nachrichten abgearbeitet sind.

5 Implementation der Benchmarks

In diesem Kapitel werden Kernpunkte der Implementierung der in Kapitel 4 vorgestellten Benchmarks erläutert. Der Quellcode und die Skripte zum Ausführen der Benchmarks befinden sich auf der beigefügten CD als Archiv. Die Abbildung 5.1 zeigt die dafür verwendetet Verzeichnisstruktur.

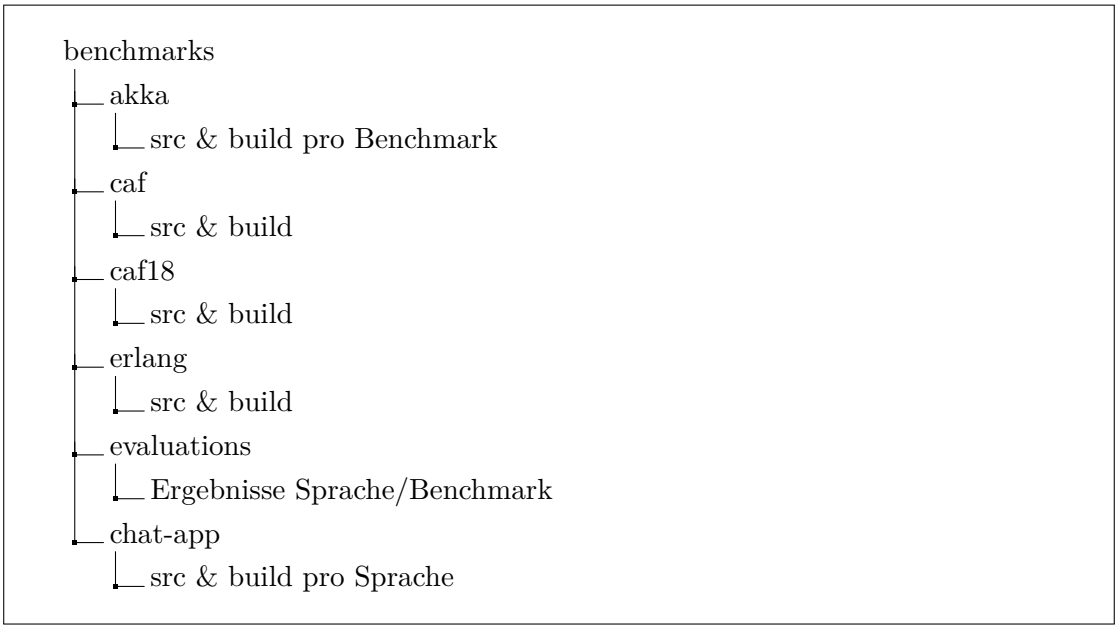


Abbildung 5.1: Verzeichnisstruktur der Benchmarks mit Quellcode und Skripten.

In den Verzeichnissen der verwendeten Programmiersprachen befinden sich neben den Quelldateien auch alle Dateien für die Konfiguration und zum Bauen der Benchmarks. Im Verzeichnis <benchmarks/evaluations> befinden sich die Skripte zum Ausführen aller Benchmarks im Werkzeug Mininet. Die Ergebnisse des jeweiligen Durchlaufs werden in Unterordner mit der Struktur <Sprache/Benchmark> abgelegt. Der Chatanwendungsbenchmark, hat sein eigenes Verzeichnis, weil er ihm Rahmen der Arbeit an der folge

```
1    akka.cluster.seed-nodes = [ "akka://ClusterSystem@IP:Port", ...]
```

Listing 5.1: Knoten in Akka über die Konfigurationsdatei bekannt geben.

Publikation von „Run, Actor, Run“ [5] entstanden ist und daher eigene Skripte zum Bauen und Auswerten hat.

5.1 Kommunikation

In diesem Abschnitt werden die verschiedenen Art und Weisen der einzelnen Sprachen, eine Kommunikation zwischen Knoten aufzubauen, erläutert.

5.1.1 Akka

In Akka verwendetet man das Modul Cluster um die Kommunikation zwischen Knoten herzustellen. Wenn alle Knoten vorher bekannt sind, dann gibt es die Möglichkeit diese in der Konfigurationsdatei *application.conf* bekannt zugeben, diese muss im Ordner <src/main/resources> liegen. Dies geschieht über den Parameter *seed-node*, siehe Listing 5.1. Dann kümmert sich die Laufzeitumgebung darum, dass eine Verbindung zu allen Knoten hergestellt wird und überwacht deren Status.

Wenn sie vorher nicht bekannt sind oder zur Laufzeit weitere hinzugefügt werden sollen, dann ist das über eine Nachricht an den Cluster Manager möglich, siehe Listing 5.2. Danach kümmert kümmert sich die Laufzeitumgebung darum diese aufrecht zuhalten und meldet wenn ein Knoten nicht mehr zu erreichen ist. Der Einfachheit halber wurde die erste Variante in allen Akka Benchmarks verwendet.

5.1.2 CAF

In CAF läuft der Kommunikationsaufbau zwischen Knoten über einen Aktor der explizit bekannt gemacht werden muss, das folgende Beispiel 5.3 zeigt wie.

Im Codebeispiel 5.4 wird gezeigt, wie man die Verbindung zu einen Knoten über den besagten Aktor aufbaut. Danach bleibt die Verbindung zwischen den Knoten bestehen.

```
1       import akka.actor.Address
2       import akka.actor.AddressFromURIString
3       import akka.cluster.typed.JoinSeedNodes
4
5       val seedNodes: List[Address] =
6           List("akka://ClusterSystem@IP:Port", ...).map(
                AddressFromURIString.parse)
7       Cluster(system).manager ! JoinSeedNodes(seedNodes)
```

Listing 5.2: In Akka einen Knoten zur Laufzeit bekannt geben.

```
1       auto my_actor = spawn(myActor);
2       system.middleman().publish(my_actor, Port);
```

Listing 5.3: Einem CAF Aktor einen Port zuweisen.

5.1.3 Erlang

In Erlang wird die Verbindung zu einem Knoten über einen Funktionsaufruf (siehe Listing 5.5) gestartet, dafür wird der beim Start des Erlangknoten festgelegte Name benötigt. Dieser Name ist atomare Zeichenkette, die mit einer beliebigen Zeichenkette beginnt und mit der IP-Adresse bzw. der URL der ausführenden Maschine verknüpft ist.

5.2 Nachrichten

Im folgenden Abschnitt wird erläutert, wie Nachrichten in den einzelnen Sprachen definiert und versendet werden.

```
1       auto remote = system.middleman().remote_actor(IP, Port);
```

Listing 5.4: Verbinden mit einem Aktor auf einem anderen CAF-Knoten

```
net_kernel:connect_node('name@IP').
```

Listing 5.5: Verbinden mit einem anderen Erlang-Knoten

```
sealed trait Messages
final class Start(actor: ActorRef[Messages]) extends Messages
final object Stop extends Messages

object MyActor {
    def apply(): Behavior[Messages] = Behavoirs.receive {
        (context, message) => {
            message match {
                case Start(actor) =>
                    actor ! Stop
                    Behavoirs.same
                case Stop =>
                    println(Stop erhalten)
                    Behavoirs.stopped
            }
        }
    }
}
```

Listing 5.6: Nachrichtendefinition und Verarbeitung in Akka

5.2.1 Akka

Nachrichten werden in Akka seit der neuen API typsicher über Klassen und Objekte definiert und dabei über Vererbung gruppiert. Diese Elternklasse wird dann einem Verhalten zugeordnet und der Aktor kann nur diese Nachrichtentypen verarbeiteten. In dem Beispiel 5.6 werden zwei Nachrichtentypen definiert und von einem Aktor behandelt.

5.2.2 CAF

In CAF legt man eine atomare Zeichenketten, sogenannte Atoms, als Identitiver für Nachrichten fest. Mit diesem Atom kann man beliebig viele andere Daten bzw. Datentypen verschicken, ob die Nachricht vom empfangenden Aktor verarbeiteten werden kann, wird nur zur Laufzeit überprüft. Das folgende Codebeispiel 5.7 zeigt die Definition von

```
1   using ping_atom = caf::atom_constant<caf::atom("ping")>;
2
3   caf::behavior my_actor(caf::event_based_actor* self) {
4       return {
5           [=](ping_atom, caf::actor& reply_to, int n) {
6               double m = n * 42.42;
7               self->send(reply_to, ping_atom::value, m);
8           },
9           [=](ping_atom, double m) {
10              std::cout << m << std::endl;
11              self->quit()
12          },
13      };
14  }
```

Listing 5.7: Nachrichtendefinition und Verarbeitung in CAF 0.17.5.

einem Atom und einen verarbeiteten Aktor, dabei werden mit der Kennung zwei verschiedenen Nachrichten verschickt. In der noch nicht veröffentlichten CAF Version 0.18.0 verändert sich die Definition von Atoms und zusätzlich muss jedem Datentypen eine ID zugewiesen werden.

5.2.3 Erlang

Wie in CAF verwendet man auch in Erlang Atoms als Kennung für Nachrichten, diese müssen aber vorher nicht bekannt geben werden. Dabei können wieder beliebig viel Daten bzw. Datentypen an einen Aktor geschickt werden. Auch in dem Erlang Codebeispiel verarbeitet ein Aktor dasselbe Atom mit verschiedenen angehängten Daten, siehe Listing 5.8.

5.3 Zeitgeber (Timer)

Nachfolgend wird erläutert wie Zeitgeber in den einzelnen Sprachen verwendet werden. In dieser Arbeit wurden Zeitgeber vor allem dafür verwendet in den Mikrobenchmarks den Nachrichtenzähler einmal die Sekunde auszugeben. Dies wurde durch verzögerte Nachrichten an den zählenden Aktor realisiert, wie das in den einzelnen Sprachen funktioniert wird Nachfolgend gezeigt.

```
1       my_actor() ->
2           receive
3               {ping, ReplyTo,N} ->
4                   M = N * 42.42,
5                   ReplyTo ! {ping, M},
6                   my_actor();
7               {ping, M} ->
8                   io::format(M)
9           end.
```

Listing 5.8: Nachrichtendefinition und Verarbeitung in Erlang.

```
1       object myActor {
2           final case object Tick
3
4           apply(): Behavior[Tick] = Behavoirs.withTimers[Tick] {
5               timers =>
6                   timers.startSingleTimer(Tick, 1.second)
7                   Behavoirs.receiveMessage {
8                       case Tick =>
9                           println("Eine Sekunde vergangen!")
10                          Behavoirs.stopped
11                  }
12          }
13      }
```

Listing 5.9: Zugriff auf den Zeitgeber und senden bzw. empfangen einer verzögerten Nachricht in Akka.

5.3.1 Akka

Um in Akka Zeitgeber verwenden zu können muss das Verhalten (Behavior) explizit mit einem Zeitgeber gestartet werden und es können nur Nachrichten an sich selbst verschickt werden. Dabei wird dazwischen unterschieden ob die Nachricht nur einmal oder immer wieder mit dem selben Zeitabstand abgeschickt werden soll. Jede verzögerte Nachricht ist dabei an einen Nachrichtentyp gebunden und sollte ein weitere Nachricht mit diesem Typ verschickt werden, wird der alte Zeitgeber überschrieben. Des Weiteren kann man natürlich die periodischen Nachricht jeder Zeit stoppen. Im Listing 5.9 wird ein verzögerte Nachricht abgeschickt und dann vom Aktor verarbeitet.

```
1   using tick_atom = caf::atom_constant<caf::atom("tick")>;
2
3   caf::behavior my_actor(caf::event_based_actor* self) {
4       self->delayed_send(self, std::chrono::seconds(1), tick_atom
            ::value)
5       return {
6           [=](tick_atom) {
7               std::cout << "Eine Sekunde vergangen!" << std::endl;
8               self->quit;
9           },
10      };
11  }
```

Listing 5.10: Senden und empfangen einer verzögerten Nachricht in CAF.

```
1   my_actor() ->
2       erlang:send_after(1000, self(), tick)
3       receive
4           tick ->
5               io::format("Eine Sekunde vergangen!")
6       end.
```

Listing 5.11: Senden und empfangen einer verzögerten Nachricht in Erlang.

5.3.2 CAF

In CAF kann jeder Aktor eine verzögerte Nachricht abschicken, denn der Zeitgeber ist hier Teil der CAF Laufzeitumgebung und steht immer zur Verfügung. Anders als in Akka kann man zusätzlich auch Nachrichten an andere Aktoren verzögert abschicken. Das Codebeispiel 5.10 zeigt wie eine Nachricht an sich selbst mit einer Verzögerung von einer Sekunde abgeschickt wird und dann von diesem Aktor verarbeitet wird.

5.3.3 Erlang

In Erlang kann wie in CAF von jedem Aktor an jeden Aktor eine verzögerte Nachricht abgeschickt werden. Ein Unterschied zu Akka und CAF ist, dass das Zeitformat auf Millisekunden festgelegt ist und nicht wie bei den anderen beiden wählbar. Im Listing 5.11 wird eine Nachricht um 1000 Millisekunden verzögert abgeschickt und vom selben Aktor verarbeitet.

6 Auswertung

Alle Benchmarks wurden auf einem Ubuntu 18.04 Server mit 128 Kernen und 512GB RAM ausgeführt. Dabei wurde Mininet in der Version 2.3.0d6 und Python 2.7.18rc1 für die Virtualisierung des Netzwerks verwendet. Bei den Aktor-Systemen würde Akka 2.6.8 mit Scala 2.11.12 in der OpenJDK VM (Java 14.0.1), Erlang/OTP 22 mit Eshell V10.6.4 und CAF in den Versionen 0.17.5 und pre-release 0.18.0 (commit 8d35937[1]) verwendet.

6.1 Nachrichtendurchsatz lokal und verteilt

Im folgenden wird der Nachrichtendurchsatz verglichen, beginnend mit dem lokalen Nachrichtenaustausch.

In der Abbildung 6.1 ist das Ergebnis des Pingpong-Benchmarks zu sehen wenn beide Aktoren im selben Aktor-System betrieben werden. Dabei ist auf der Y-Achse die Anzahl der Pong-Nachrichten pro Sekunde aufgeführt und auf der X-Achse die einzelnen Version. Dabei ist zusehen, dass Erlang die meisten Ping-Pong-Nachrichtenaustausche schafft und Akka am wenigsten. Des Weiteren fällt ein deutlicher Unterschied zwischen den beiden CAF Versionen auf, die neue, die sich noch in der Entwicklung befindende Version 0.18.0 schafft deutlich mehr. Dies war einer der Ziele der Neugestaltung des Nachrichtenschicht von CAF.

Die Abbildung 6.2 zeigt den gleichen Benchmark wie Abbildung 6.1, aber bei diesem mal sind die kommunizierenden Aktoren auf zwei unterschiedlichen, durch ein Netzwerk verbunden, Aktor-Systemen. Dabei sind Netzwerklatenzen und -verluste durch Mininet unterbunden. Auch hier ist auf der Y-Achse die Anzahl an Pong-Nachrichten pro Sekunde aufgeführt und auf der X-Achse die verschiedenen Version. Wieder ist Erlang mit

[1] https://github.com/actor-framework/actor-framework/tree/8d359374ff69cda19b6c2e8096803f7cc1fa64e7

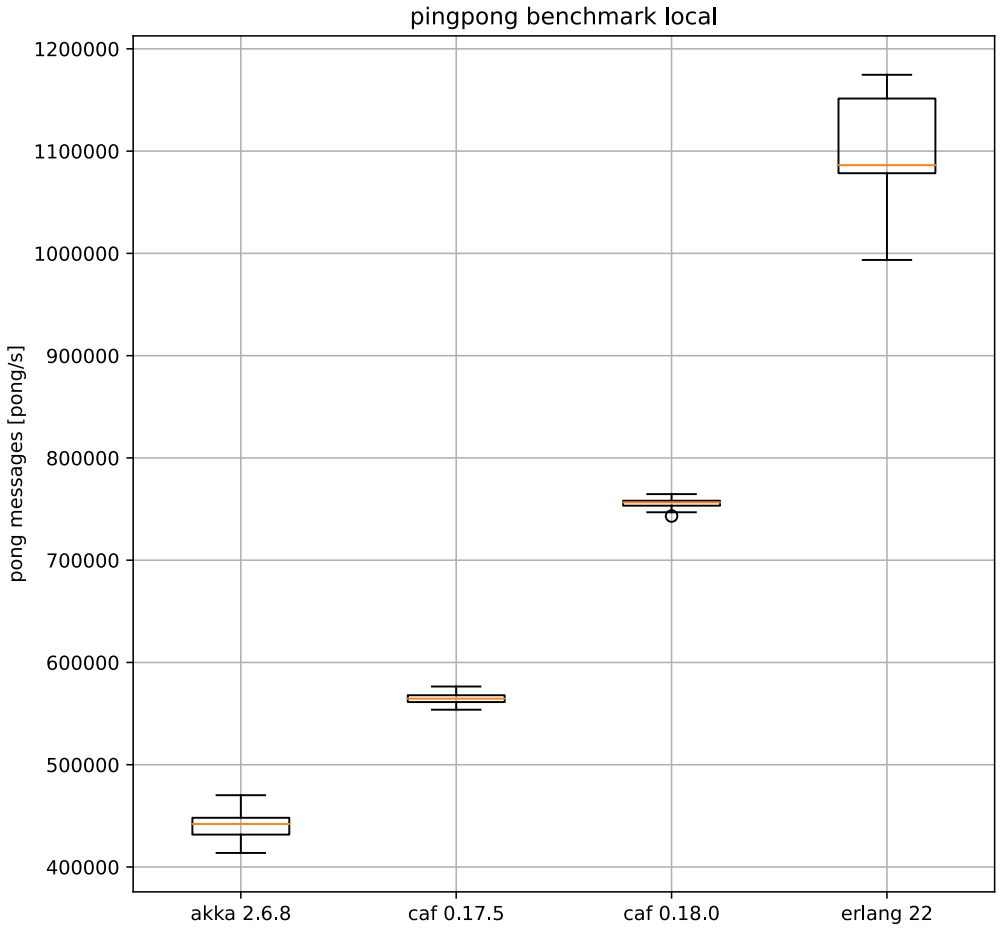


Abbildung 6.1: Ping-Pong-Nachrichten zwischen zwei Aktoren in einem Aktor-System.

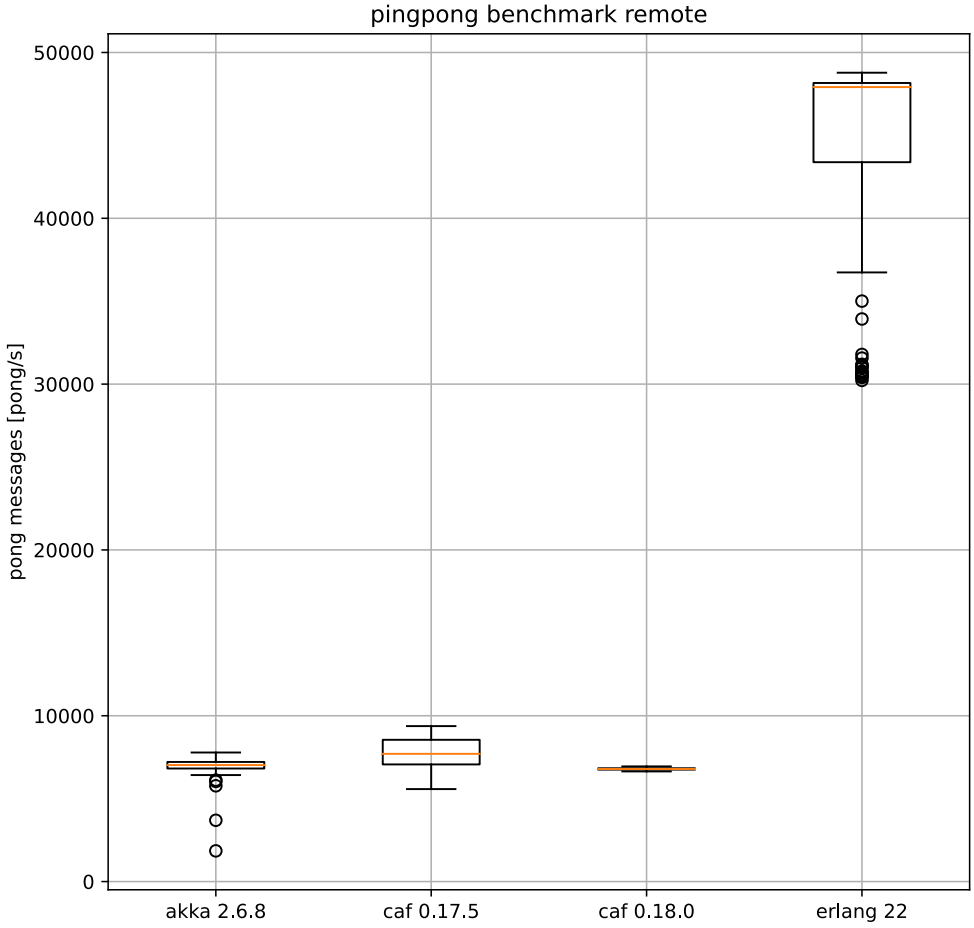


Abbildung 6.2: Ping-Pong-Nachrichten zwischen zwei Aktoren in unterschiedlichen Aktor-Systemen.

deutlichem Vorsprung dabei, die anderen drei halten sich in der Waage. Das die Anzahl der Nachrichten im Allgemeinen deutlich niedriger ist als in Abbildung 6.1 war zu erwarten, weil bei jedem Sendevorgang der Serialisierungsaufwand auf der Sendeseite und Deserialisierungsauswand auf der Empfängerseite dazukommt, erst dann kann die Nachricht beim eigentlichen Aktor zugestellt werden.

6.2 Nachrichtenverhalten in einem verteilten System

In diesen Abschnitt wird sich das Nachrichtenverhalten im einem verteilten Aktor-System angeschaut. Dabei kommen zwei verschiedene Kommunikationsbeziehungen zum Einsatz. Beim Big-Benchmark ist es eine *n-zu-n* Beziehung und der Bang-Benchmark verwendet eine *n-zu-eins* Beziehung. In beiden fällen beinhaltet jedes Aktor-System genau einen Aktor der mit den auf den anderen Knoten kommuniziert.

6.2.1 Skalierungsverhalten

Zu erst wird sich das Skalierungsverhalten angeschaut. In der Abbildung 6.3 wird der Big-Benchmark mit einer steigenden Anzahl von Knoten immer wieder ausgeführt. Hierbei ist auf der Y-Achse die Anzahl der Pong-Nachrichten pro Sekunde aufgeführt und auf der X-Achse die Anzahl der Knoten, mit je einem Aktor-System. Das Kommunikationsnetzwerk hat eine *n-zu-n* Beziehung. Es fällt auf, dass bei Erlang die Anzahl der ausgetauschten Nachrichten bis zum 17 Knoten stark ansteigt und dann stark einbricht. Dies ist damit zu erklären, dass Erlang standardmäßig ein vollvermaschtes Netzwerk aufbaut zusammen mit einem häufigen globalen Nachfrage wo ein Aktor zu finden ist, was auch schon von [3] gezeigt worden ist. Die beiden CAF Versionen unterscheiden sich nicht stark, aber die neue Version liegt immer etwas oberhalb der älteren Version 0.17.5. Akka wird nur bis zehn Knoten dargestellt, weil dann die Mininet Knoten keine Verbindung zu einander aufrecht erhalten können, warum dies nur mit Akka geschieht ist bisher unklar. Bis dahin steigt Akka kontinuierlich an, aber nicht so stark wie Erlang.

Die Abbildung 6.4 zeigt die Ergebnis Bang-Benchmarks mit dem selben Aufbau wie Abbildung 6.3, nur mit das Kommunikationsnetzwerk hat eine *eins-zu-n* Beziehung. Auch hier erreicht Erlang eine deutlich höhere Anzahl an Nachrichten als CAF, wobei diesmal der Einbruch nicht vorhanden ist, dies liegt daran, dass diesmal die Knoten die vielen Verbindungen nicht nutzen. In diesem Fall unterscheiden sich die beiden CAF

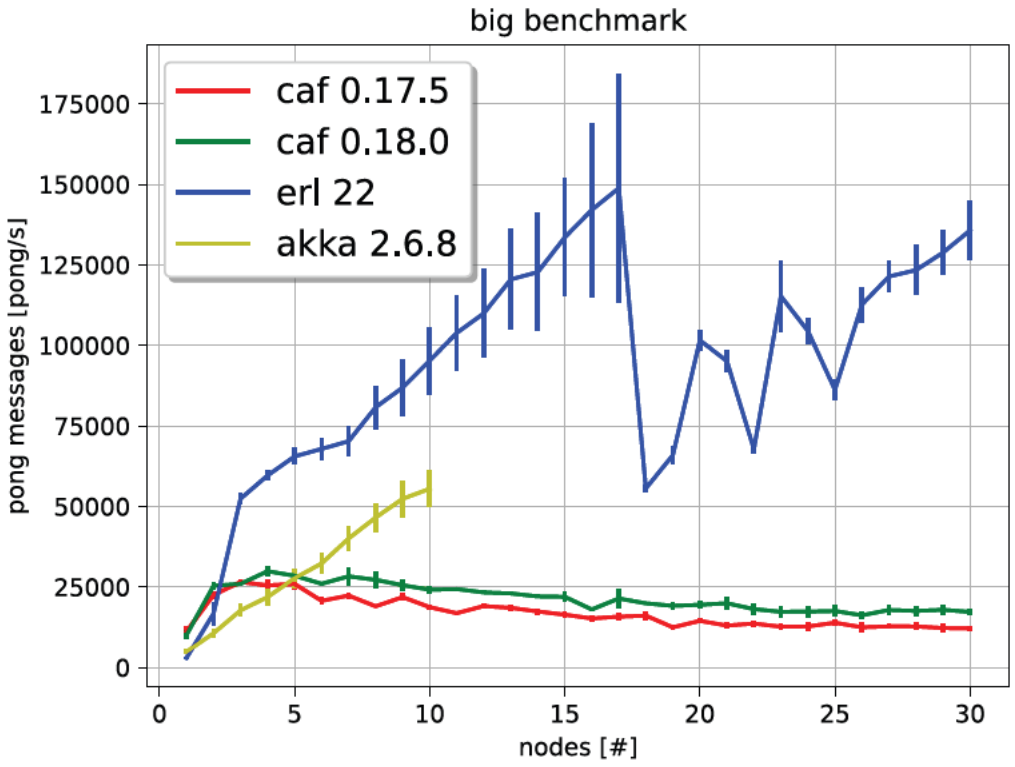


Abbildung 6.3: Big-Benchmark mit steigender Anzahl an Knoten.

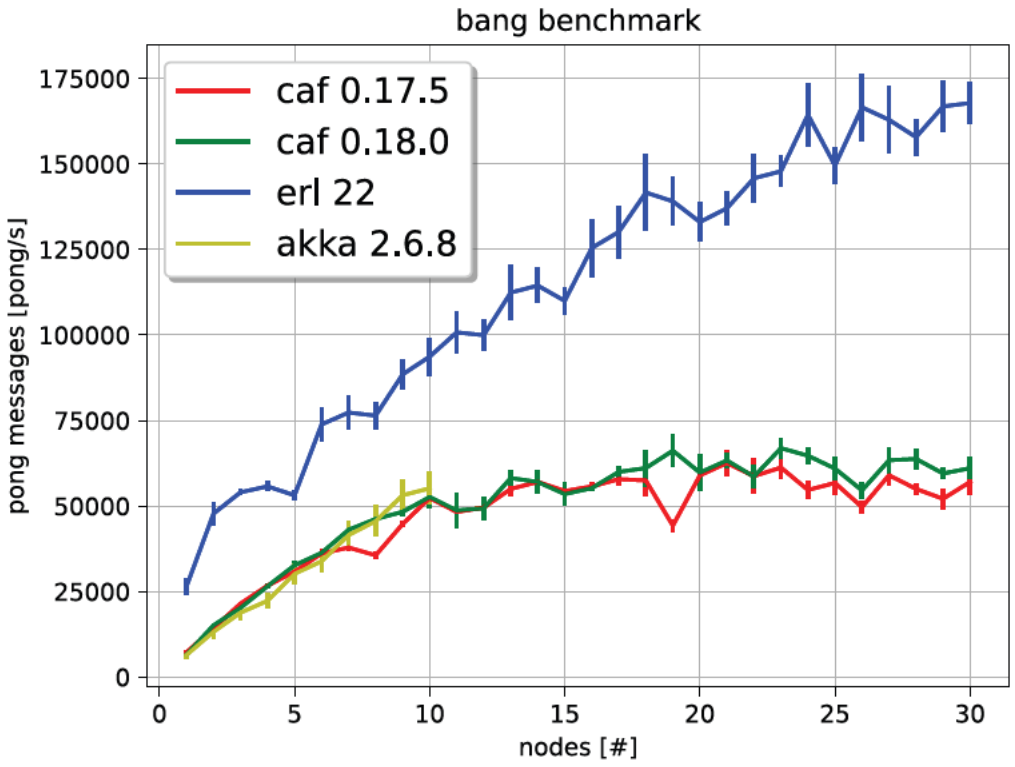


Abbildung 6.4: Bang-Benchmark mit steigender Anzahl an Knoten.

Versionen vernachlässigbar, weil es auch keinen gibt der immer mehr schafft, wie in Abb. 6.3. Akka wird aus dem eben schon genannten Grund wieder nur bis zehn Knoten aufgeführt, verhält sich diesmal aber wie CAF.

6.2.2 Latenz im Netzwerk

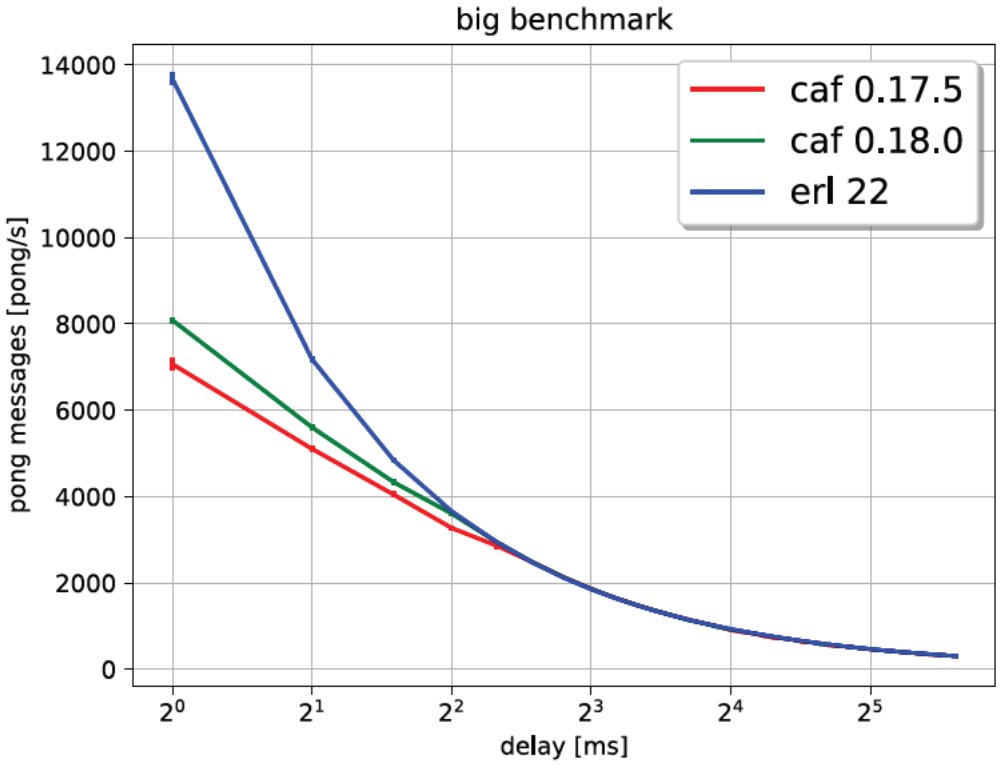


Abbildung 6.5: Der Big-Benchmark mit steigender Latenz im Netzwerk, bei konstanter Anzahl von 30 Knoten.

Im folgenden wird die Auswirkung der Netzwerklatenz auf den Nachrichtenaustausch untersucht.

Die erste Abbildung 6.5 zu diesem Thema verwendet wieder den Big-Benchmark. In diesem Fall ist die Anzahl der teilnehmenden Knoten auf 30 festgelegt und die Latenz steigt. Die Anzahl der Pong-Nachrichten ist wieder auf der Y-Achse zusehen und die X-Achse zeigt die Latenz logarithmisch skaliert. Die zu erwartende Halbierung der Anzahl der

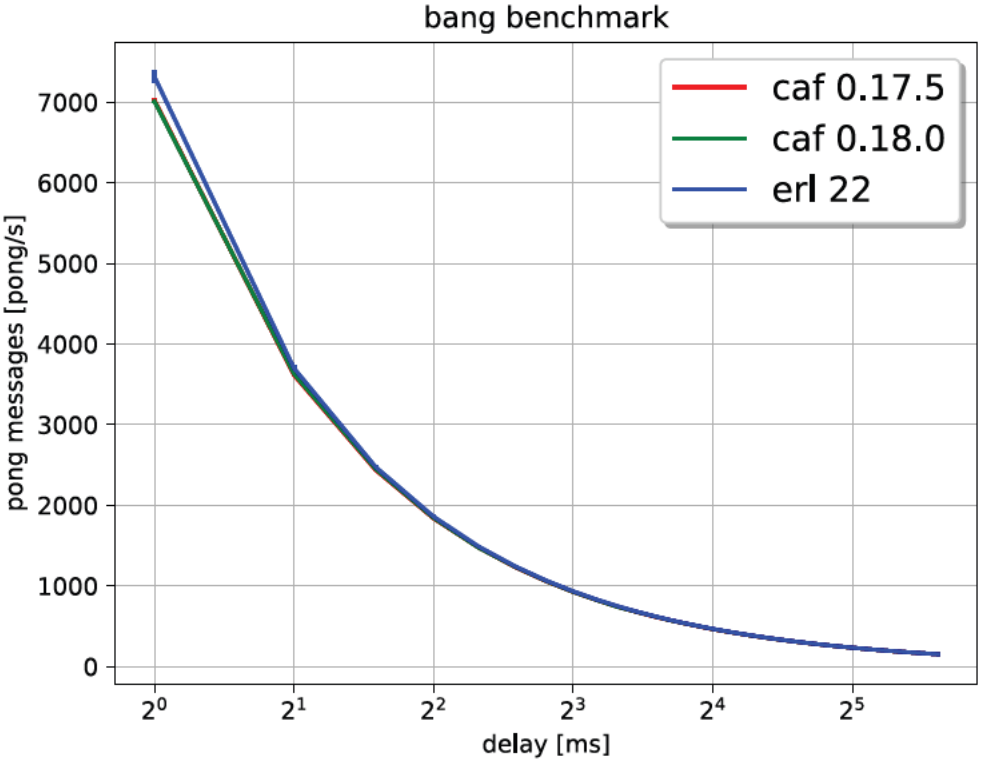


Abbildung 6.6: Der Bang-Benchmark mit steigender Latenz im Netzwerk, bei konstanten 30 Knoten.

ausgetauschten Nachrichten bei Verdoppelung der Latenz ist sehr gut bei Erlang beobachten. Wie nach dem gezeigten Skalierungsverhalten (siehe 6.2.1) zu erwarten startet Erlang mit deutlich mehr Pong-Nachrichten pro Sekunde als CAF. Bei etwa 5ms Latenz ist die Verarbeitungsgeschwindigkeit relativ zur Latenz so gering, dass dadurch kein Vorteil mehr entsteht. Die Abweichung vom Mittelwert ist so klein, dass sie nur bei 1ms Latenz zu erkennen ist.

Beim Bang-Benchmark hingegen, sieht man das zu erwartende Verhalten bei allen Dreien, wie Abbildung 6.6 zeigt. Der Startunterschied zwischen Erlang und CAF ist überraschend klein, wenn man das Skalierungsverhalten (siehe Abschnitt 6.2.1) bedenkt, wo allerdings ohne künstliche Latenz gemessen wurde. Das dieser Unterschied nicht so groß ist wie beim Big-Benchmark lässt sich dadurch erklären, dass im Allgemeinen viel weniger Nachrichten unterwegs sind. Kein Unterschied mehr zuerkennen ist hier ab etwa 3ms Latenz. Ein Unterschied zwischen den CAF Versionen gibt es hier nicht.

6.2.3 Verlust im Netzwerk

Nun werden die Auswirkungen von Paketverlust im Netzwerk untersucht. Dabei beginnt wieder der Big-Benchmark, Abbildung 6.7. Hier wurde auf der Y-Achse eine logarithmische Skala zur Basis Zehn verwendet, weil die Anzahl der Nachrichten zu beginn zu groß ist um beim Rest noch etwas aus dem Graphen lesen zu können. Aus der X-Achse steigt der Paketverlust von Null Prozent bis zu Zehn Prozent an. Die Anzahl der beteiligten Knoten ist in dieser Testreihe konstant. Überraschend ist der Anstieg von beiden CAF zwischen 0% und 2% Paketverlust. Ein Messfehler würde durch eine nochmalige Durchführung der Testreihe ausgeschlossen. Den Grund für diesen Anstieg konnte nicht ermittelt werden.

In Abbildung 6.8 findet sich dieser Anstieg im vorderen Bereich nicht. Die Abbildung 6.8 ist genau so Aufgebaut, wie Abb. 6.7. Auch hier ist der Startunterschied nicht so groß, wie es auch schon bei Abbildung 6.6 im Vergleich zu der Abbildung 6.5 war.

6.2.4 Die CAF Scheduler im Vergleich

Zu Letzt wird sich noch der Unterschied beim Nachrichtenverhalten zwischen den beiden CAF Schedulern angeschaut, die Unterschiede wurden in Kapitel 2.1.1 erläutert.

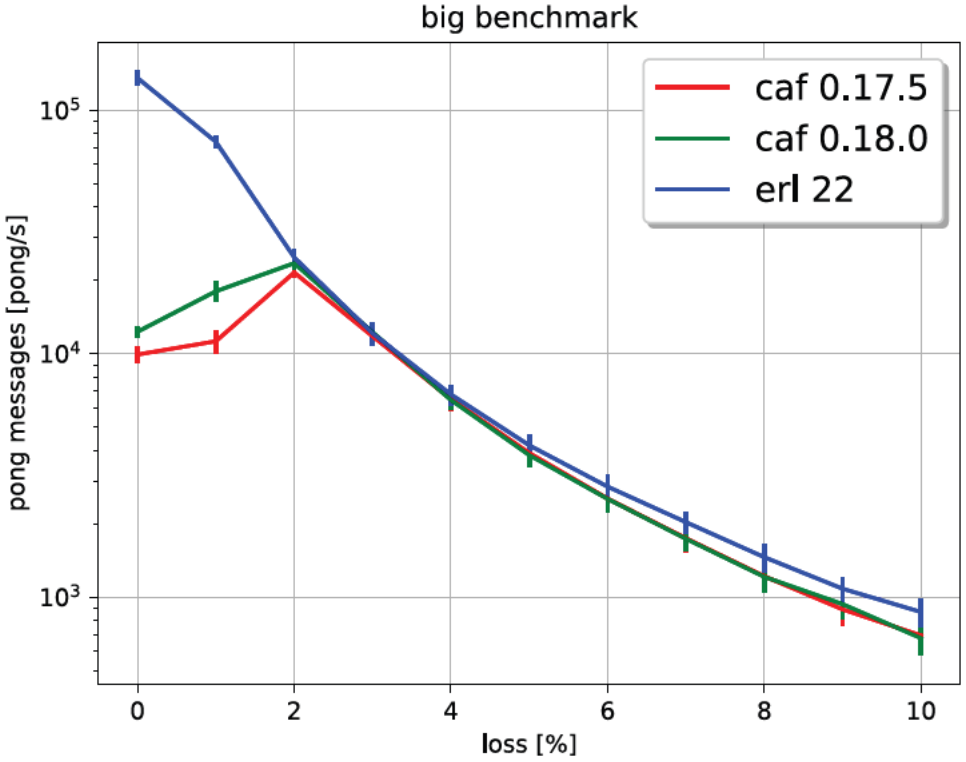


Abbildung 6.7: Der Big-Benchmark mit steigendem Verlust im Netzwerk, 30 Netzwerkknoten.

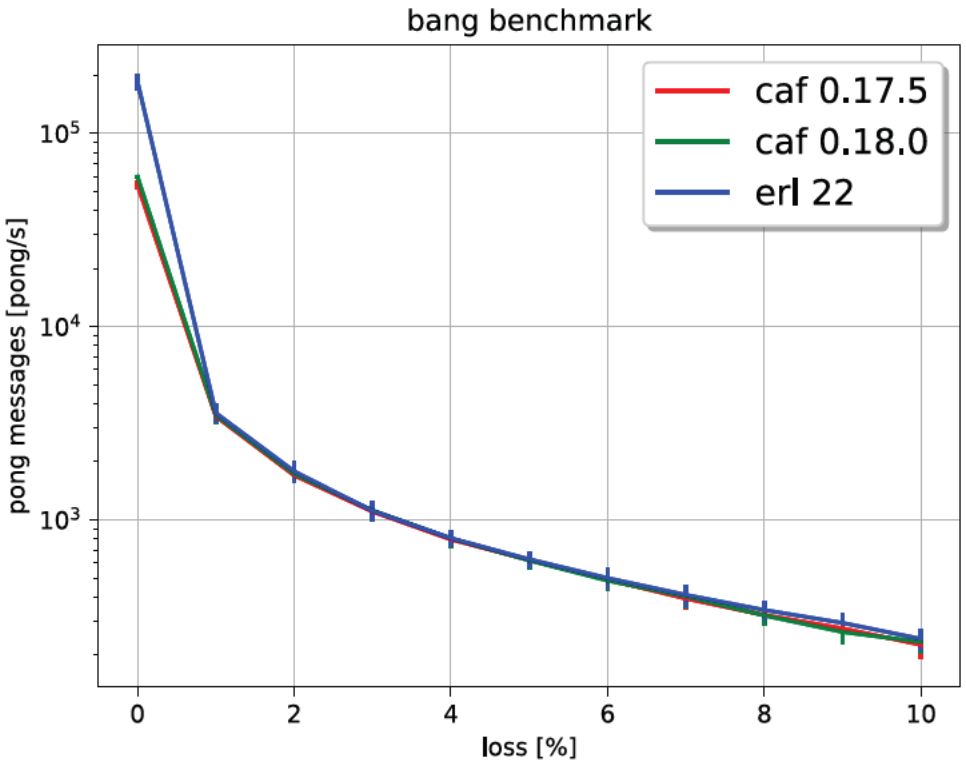


Abbildung 6.8: Der Bang-Benchmark mit steigendem Verlust, bei 30 Knoten.

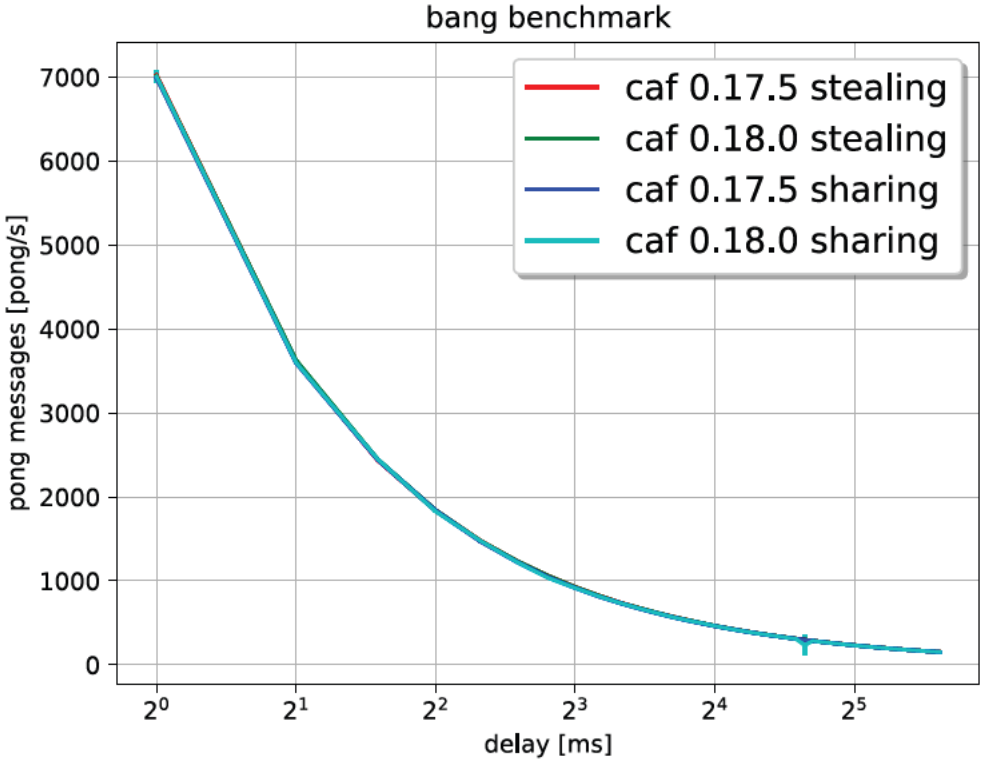


Abbildung 6.9: Der Bang-Benchmark mit steigender Latenz im Netzwerk, bei konstanten 30 Knoten und den zwei verschiedenen CAF Schedulern.

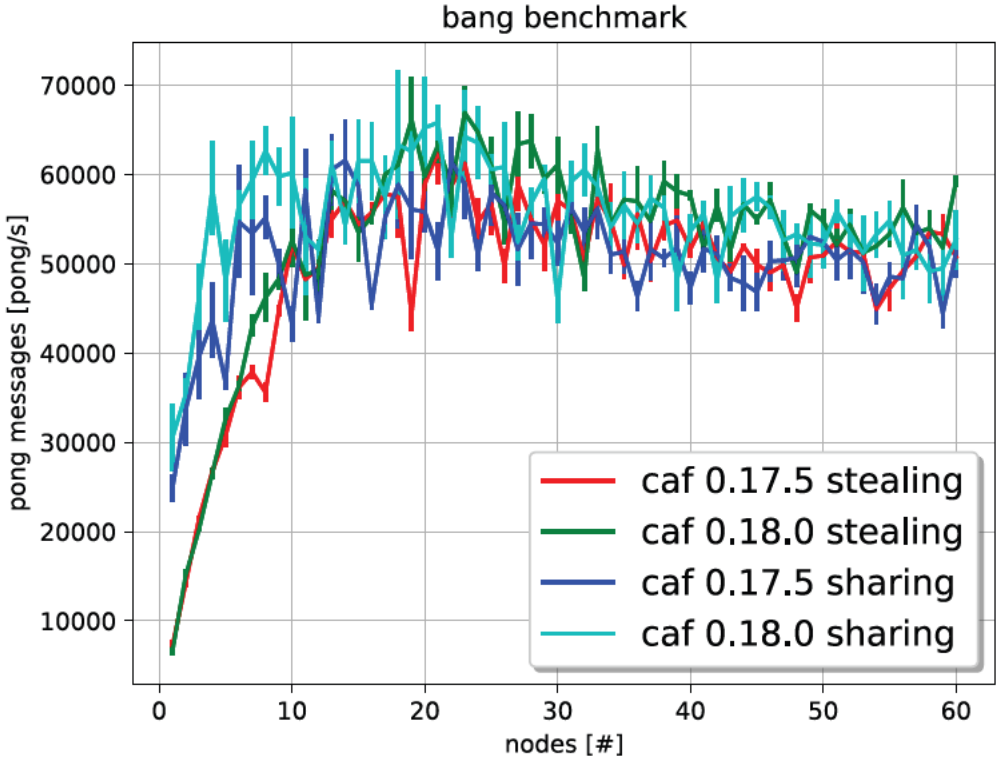


Abbildung 6.10: Der Bang-Benchmark mit steigender Anfang an Konten und den zwei verschiedenen CAF Schedulern.

Die Abbildung 6.9 die Ergebnisse des Bang-Benchmarks mit steigender Latenz bei konstanten 30 Knoten. Die Anzahl der Pong-Nachrichten pro Sekunde zeigt die Y-Achse und auf der X-Achse ist die Latenz im Netzwerk aufgeführt. Diese steigt logarithmische Funktion zur Basis zwei an. Bis auf einen kleinen Ausreißer kurz vor 32ms Latenz, verhalten sich beide Scheduler identisch.

In der Abbildung 6.10 wird der Bang-Benchmark mit beiden gerade vorgestellten Schedulern in beiden Versionen ausgeführt. Auf der X-Achse wird die Anzahl der Knoten aufgeführt und auf der Y-Achse die Anzahl an verarbeiteten Pong-Nachrichten in einem Aktor. Bei diesem Graphen fällt der deutliche Startunterschied von fast Faktor drei zwischen den beiden Schedulern auf, egal ob CAF Version 0.17.5 oder 0.18.0, der „work sharing“-Scheduler ist besser. Dies ist bis ungefähr zehn Knoten der Fall, danach unterscheiden sich die beiden kaum noch und schwanken stark um 55.000 Nachrichten die Sekunde.

6.3 Laufzeitvergleich mit realen Anwendungen

Im folgenden Abschnitt werden die Laufzeiten der Makrobenchmarks verglichen. Begonnen wird mit der Webanwendung.

6.3.1 Webanwendung

Die Abbildung 6.11 zeigt die Webanwendung. Auf der Y-Achse wird die Laufzeit im Millisekunden für einen gesamten Durchgang aufgeführt. Die X-Achse gibt an welches Aktor-System verwendet wird. Bei diesem Graphen sticht sofort ins Auge, dass Akka eine sehr viel kürzere Laufzeit hat, als die anderen. Während die CAF Version 0.17.5 wie erwartet langsamer als die pre-release Version 0.18.0 ist, denn in dieser ist vor allem daran gearbeitet worden, dass die Abarbeitung von Nachrichten schneller geht. Dies wurde dadurch erreicht, dass jeder Datentyp jetzt eine eigene ID hat und nicht die Typinformationen zur Laufzeit verarbeitetet und zugeordnet werden müssen.

Bei der Analyse warum Akka so deutlich schneller ist als alle anderen, wurden zuerst die Laufzeitunterschiede der Fibonacci-Berechnung, die jede Anfrage auslöst untersucht. Dies ergab, dass bei Akka der Just-in-time-Compiler (JIT-Compiler) das Ergebnis der Berechnung nach nur wenigen aufrufen gecached hat und in unregelmäßigen Abständen

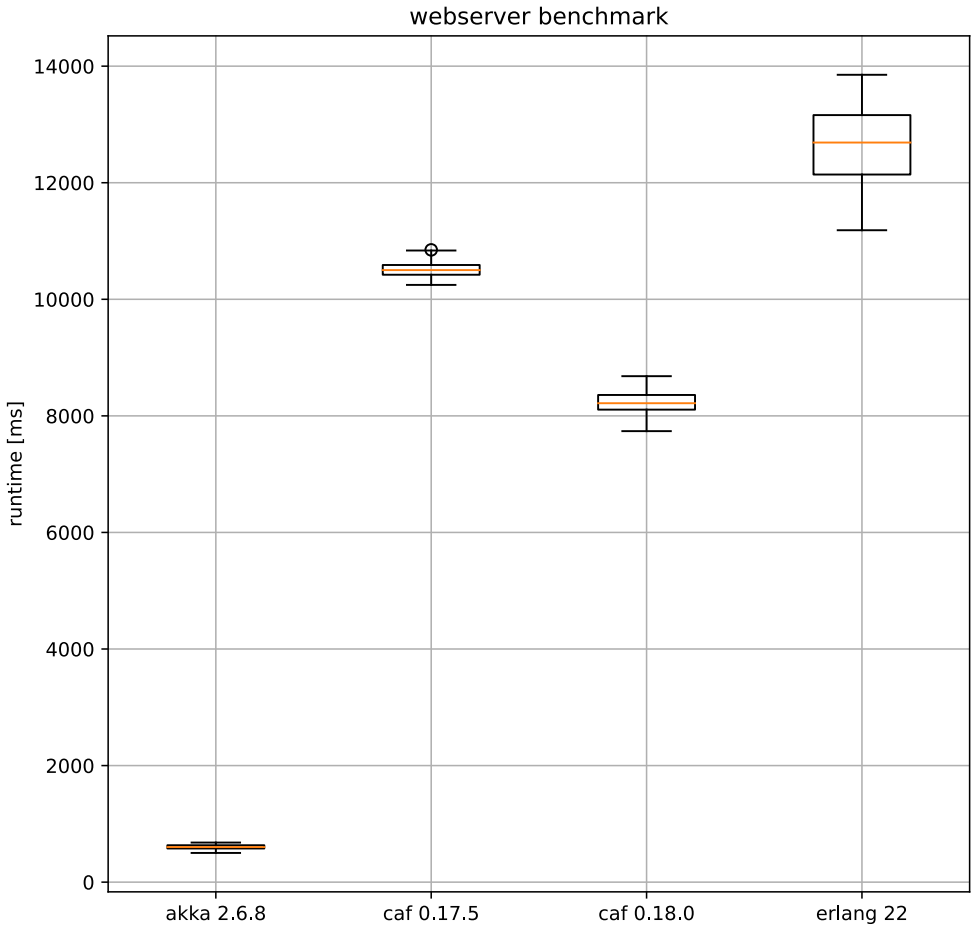


Abbildung 6.11: Die Webanwendung im Vergleich, Akka überraschend schnell.

neu berechnet. Wenn Akka die Berechnung durchführt dauert die bis zu 10 Millisekunden, während der Zugriff auf den gecachten Wert nur maximal eine Millisekunde dauert. CAF und Erlang berechnen es jedes Mal, CAF ist dabei etwas schneller als eine Millisekunde und Erlang benötigt mehrere Millisekunden.

Diese Unterschiede erklären, nicht die Geschwindigkeit von Akka. Daher wurde als nächstes bei Akka untersucht ob es genau das gleiche macht wie die anderen Aktor-Systeme. Hierbei viel auf, dass Akka nur einen Aktor vom selben Typ erstellt wenn dieser als Objekt definiert wurde, wie im Listing 5.6. Dies viel bisher nicht auf, weil es kein Problem ist, wenn man den Aktor einmal pro Aktor-System startet, wie es in den anderen Benchmarks der Fall ist. Um mehrere Aktoren von einem Typ bzw. Verhalten in einem Aktor-System zu starten, muss dieser als Klasse definiert werden und über ein Objekt-Aktor instanziiert werden. Zu sehen ist eine Beispiel Definition eines solchen Aktors in Listing 6.1. Dies war bisher unbekannt, weil alle Beispiele und Tutorials zu Akka, die ich kenne, Aktoren nur einmal instanziieren.

Nach dem dieser Fehler behoben wurde und der Benchmark neu ausgeführt wurde, sehen die Ergebnisse für Akka ganz anders aus wie die Abbildung 6.12 zeigt. Jetzt liegt Akka im Mittel zwischen den beiden CAF Versionen und unterhalb von Erlang. Des Weiteren wurden auch die anderen Implementationen nochmals überprüft und keine weiteren Fehler gefunden.

6.3.2 Chatanwendung

Bei der Chatanwendung hat die Erlang-Version einen noch nicht gefundenen Fehler und Akka ist nicht rechtzeitig fertig geworden, daher wurde hier kurzfristig zum Vergleich die Pony-Version hinzugezogen. Die Pony-Version wurde im Rahmen der Arbeit, zusammen mit dem Team von „Run, Actor, Run“ [5], für die nachfolge Publikation erstellt.

Pony ist eine Programmiersprache die das Aktormodell beinhaltet. Aktoren sind isoliert und Nachrichten typsicher. Das Aktor-System besitzt noch keine Netzwerkschnittstelle und ist daher nur lokal ausführbar. Die aktuelle Version ist 0.36.0[2].

Die Abbildung 6.13 zeigt das Skalierungsverhalten der Chatanwendung wenn ihr immer mehr Kerne zur Verfügung stehen. Für diesen Benchmark würden die Chatanwendung so konfiguriert, dass 1024 Clients auf 24 Verzeichnisse verteilt wurden und nur die Aktion

[2] https://github.com/ponylang/

```
1   object MyActor {
2     sealed trait Msg
3     final case class Do(replyTo: ActorRef[Msg], value: Int)
        extends Msg
4     final case object Quit extends Msg
5
6     def apply(): Behavior[Msg] =
7       Behaviors.setup { ctx =>
8         new MyActor(ctx)
9     }
10  }
11
12  class MyActor(ctx: ActorContext[MyActor.Msg])
13    extends AbstractBehavior[MyActor.Msg] (ctx) {
14  override def onMessage(msg: MyActor.Msg):
15    Behavior[MyActor.Msg] =
16      msg match {
17        case MyActor.Do(replyTo, value) =>
18          val fib = fibonacci(value)
19          replyTo ! MyOtherActor.Reply(fib)
20          Behaviors.same
21        case MyActor.Quit =>
22          Behaviors.stopped
23      }
```

Listing 6.1: Definition eines Aktors, von dem mehr als eine Instanz gelichzeitig laufen kann in Akka

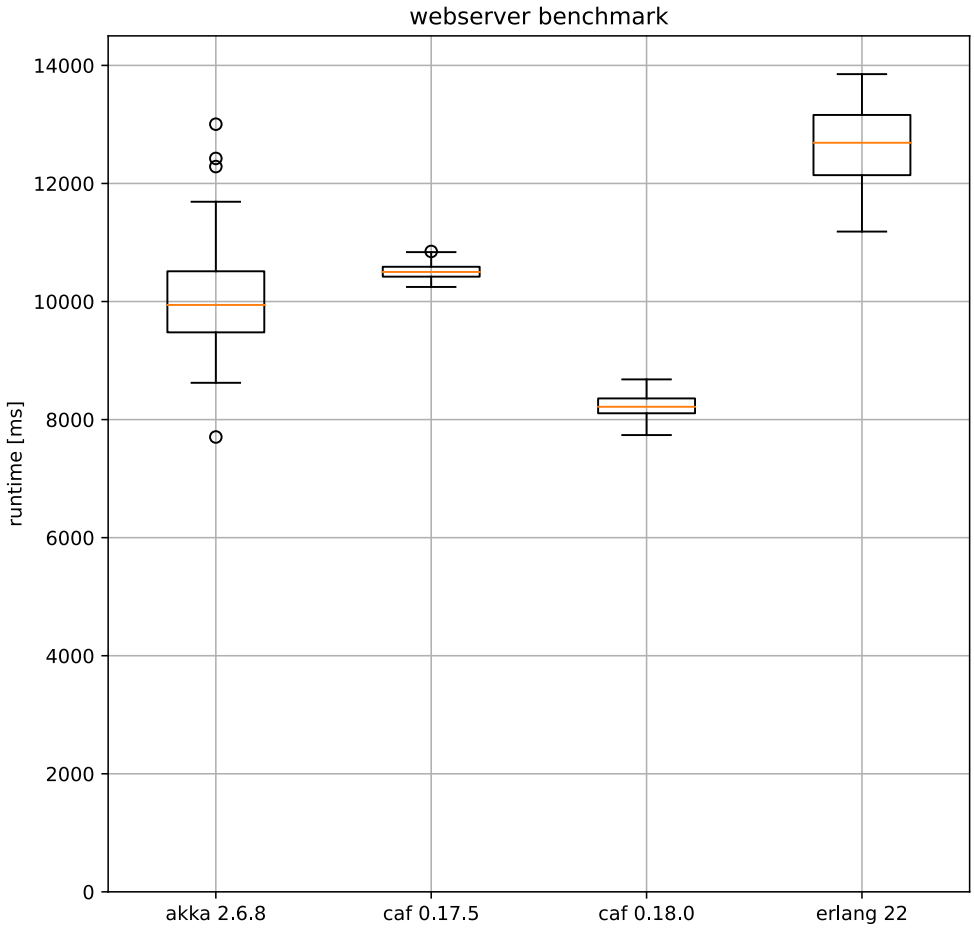


Abbildung 6.12: Die Webanwendung im Vergleich.

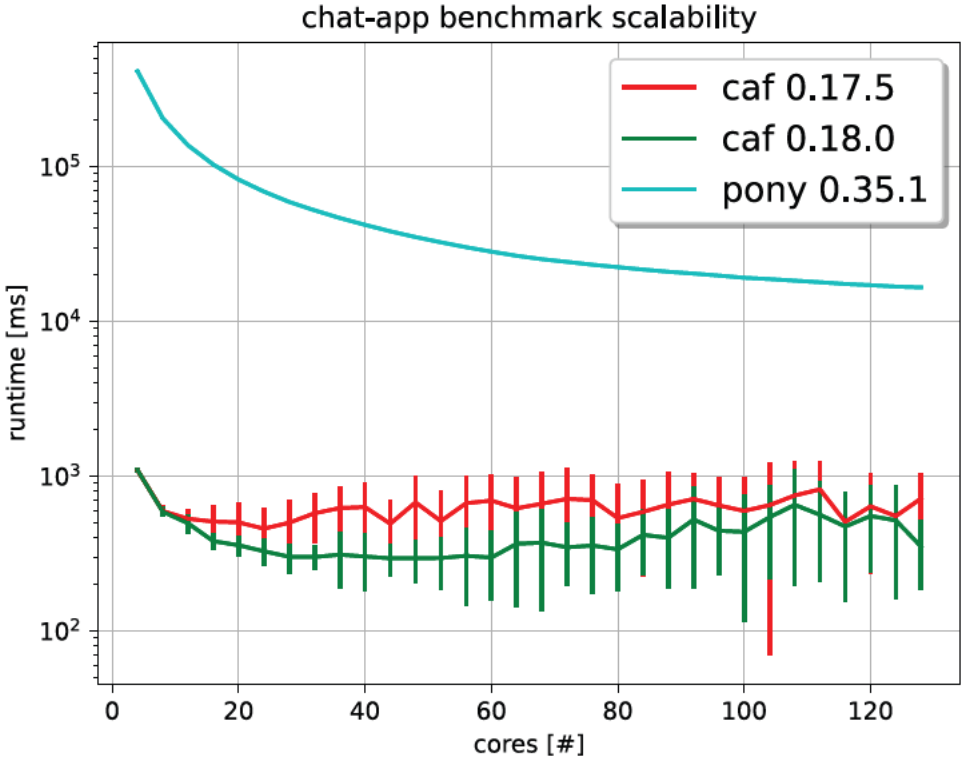


Abbildung 6.13: Die Chatanwendung mit steigender Anzahl an Kernen im Laufzeitvergleich.

„eine Berechnung durchführen“ im Client möglich ist, des Weiteren sind 10% der Clients befreundet. Die Y-Achse gibt die Laufzeit in Millisekunden an und die X-Achse die Anzahl der verwendeten Kerne. In diesem Szenario benötigt CAF immer deutlich weniger Zeit als Pony. Während CAF nahe zu immer unter einer Sekunde bleibt, benötigt Pony auch bei bei 128 Kernen noch deutlich über 10 Sekunden.

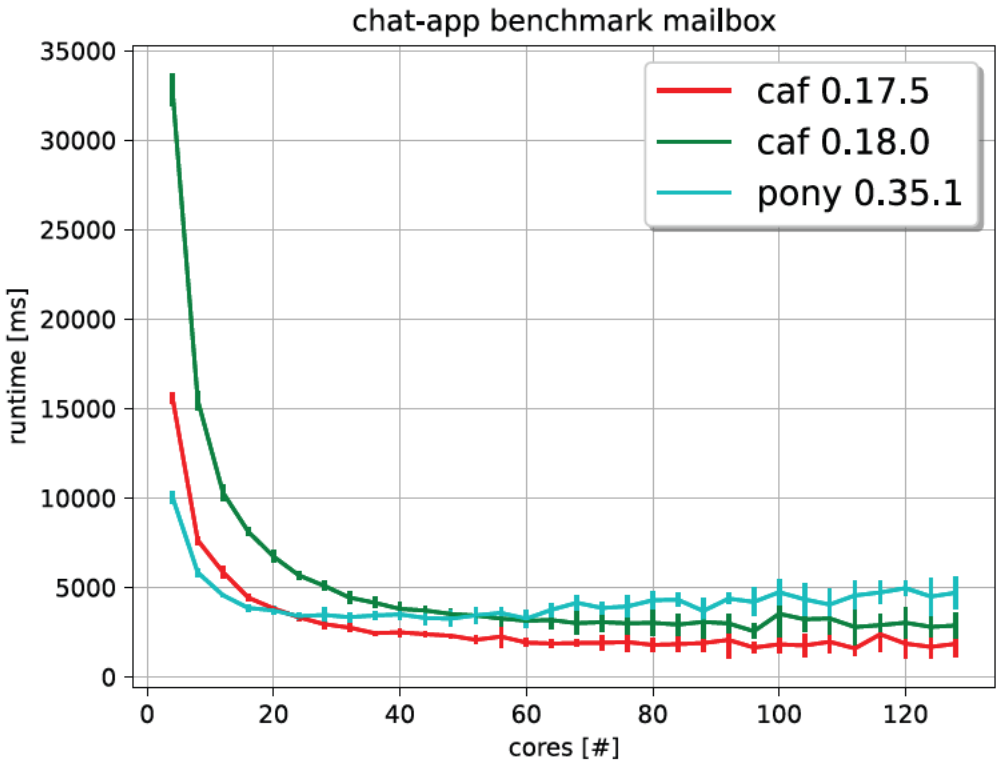


Abbildung 6.14: Die Chatanwendung mit steigender Anzahl an Kernen, Mailbox-Stresstest.

Das Szenario Mailbox, welches die Leistungsfähigkeit der Nachrichtenqueue eines Aktors testet, ist in Abbildung 6.14 zu sehen. Für diesen Fall würde die Chatanwendung so konfiguriert, dass 256 Clients auf 256 Verzeichnisse verteilt sind und alle Clients miteinander befreundet sind. Die möglichen Aktionen sind wie folgt eingestellt: Textnachricht schreiben 80% und neuen Chat erstellen 20%. Pony ist in diesem Szenario zu beginn schneller als CAF, aber ab 20 Kernen wird erst die Version 0.17.5 schneller und ab 50 Kernen auch die Version 0.18.0. Die ältere CAF Version bleibt dabei immer schneller als

die neue Version.

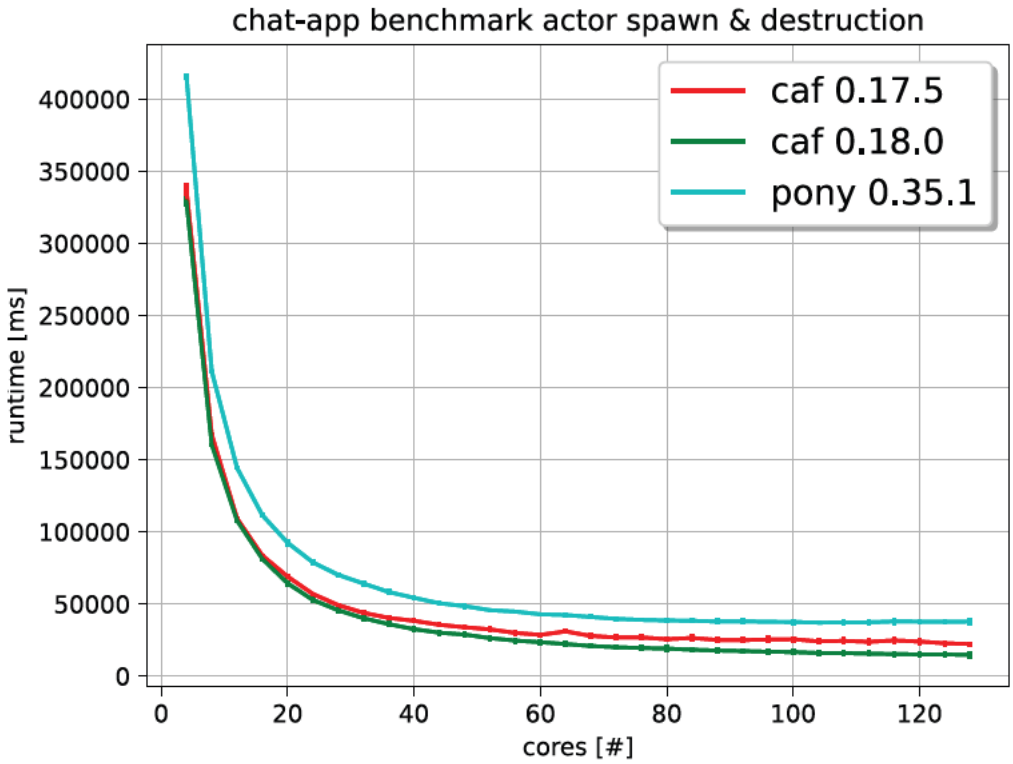


Abbildung 6.15: Die Chatanwendung, Aktoren starten und beenden.

Die Abbildung 6.15 zeigt das Szenario bei dem Aktoren gestartet und wieder beendet werden, dabei soll auch getestet werden wie gut die automatische Beendigung von Aktoren, die keinem mehr bekannt sind, funktioniert. Hierfür würde eine Konfiguration gewählt, in der alle Aktionen im Client möglich sind. Die Wahrscheinlichkeiten der einzelnen Aktionen sind: Rechnen 40%, Textnachricht schreiben 30%, Chat verlassen 5% und Chat erstellen 25%. Dabei wurden 2048 Clients auf 8 Verzeichnisse verteilt, wobei sich 10% der Clients kennen. In der Abbildung 6.15 ist die Laufzeitvergleich auf Y-Achse angegeben und die Anzahl der Kerne auf der X-Achse. Die diesem Szenario ist CAF durchgehend vor Pony.

7 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es Benchmarks für verteilte Aktor-Systeme zu entwickeln und damit verschiedene Systeme zu vergleichen. Es wurden mehrere Benchmarks vorgestellt und getestet. Des Weiteren wurden sie in verschiedenen Sprachen programmiert und die Ergebnisse verglichen.

Die Benchmarks haben sich mit dem Nachrichten- und dem Skalierungsverhalten in verteilten Systemen befasst. Von den verglichenen Aktor-Systemen konnte keins durchgehend überzeugen.

Im folgenden werde mögliche nächste Schnitte und notwendige Verbesserungen vorgestellt:

Akka, für dieses Aktor-System muss ein Weg gefunden werden, die Benchmarks mit mehr als 10 Knoten im Mininet, ohne Verbindungsabbrüche, auszuführen. Des Weiteren muss die Implementierung der Chatanwendung fertiggestellt werden.

Erlang, hier muss der Fehler in der Implementierung der Chatanwendung behoben werden.

Chatanwendung, dieser Benchmark ist bisher noch nicht verteilt worden. Hierfür muss eine Möglichkeit gefunden werden die Notwendigkeit der Nachrichtenreihenfolge loszuwerden.

Literatur

[1] Gul Agha. *Actors: A Model of Concurrent Computation In Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 978-0262511414.

[2] Joe Armstrong. „A History of Erlang“. In: *Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*. San Diego, California: ACM, 2007, S. 6–1–6–26.

[3] Stavros Aronis u. a. „A Scalability Benchmark Suite for Erlang/OTP“. In: *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*. Erlang '12. ACM, 2012, S. 33–42. ISBN: 978-1-4503-1575-3. DOI: 10.1145/2364489.2364495. URL: http://doi.acm.org/10.1145/2364489.2364495.

[4] S. Barghi und M. Karsten. „Work-Stealing, Locality-Aware Actor Scheduling“. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, S. 484–494. DOI: 10.1109/IPDPS.2018.00058.

[5] Sebastian Blessing u. a. „Run, Actor, Run: Towards Cross-Actor Language Benchmarking“. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2019. Association for Computing Machinery, 2019, S. 41–50. ISBN: 9781450369824. DOI: 10.1145/3358499.3361224. URL: https://doi.org/10.1145/3358499.3361224.

[6] Stefan Bouckaert u. a. „Benchmarking computers and computer networks“. In: *EU FIRE White Paper* (2010).

[7] Rafael C. Cardoso u. a. „Towards Benchmarking Actor- and Agent-Based Programming Languages“. In: *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE! 2013. Association for Computing Machinery, 2013, S. 115–126. ISBN: 9781450326025. DOI: 10.1145/2541329.2541339. URL: https://doi.org/10.1145/2541329.2541339.

[8] Dominik Charousset, Raphael Hiesgen und Thomas C. Schmidt. „CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications“. In: *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!* Portland, OR: ACM, 2014, S. 15–28.

[9] Dominik Charousset, Raphael Hiesgen und Thomas C. Schmidt. „Revisiting Actor Programming in C++“. In: *Computer Languages, Systems & Structures* 45 (2016), S. 105–131. URL: http://dx.doi.org/10.1016/j.cl.2016.01.002.

[10] N. Chechina u. a. „Evaluating Scalable Distributed Erlang for Scalability and Reliability“. In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2017), S. 1–1.

[11] Andreas Ehliar und Dake Liu. „Benchmarking network processors“. In: *Department of Electrical Engineering, Linkoping University, Sweden, Copyright* (2004).

[12] Philip J. Fleming und John J. Wallace. „How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results“. In: *Commun. ACM* 29.3 (März 1986), S. 218–221. ISSN: 0001-0782. DOI: 10.1145/5666.5673. URL: https://doi.org/10.1145/5666.5673.

[13] Gouy, Isaac. *The Computer Language Benchmarks Game*. 2016. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/ (besucht am 05. 05. 2020).

[14] Carl Hewitt, Peter Bishop und Richard Steiger. „A Universal Modular ACTOR Formalism for Artificial Intelligence“. In: *Proc. of the 3rd IJCAI*. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, S. 235–245.

[15] Torsten Hoefler und Roberto Belli. „Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results“. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2015, S. 1–12.

[16] Shams Imam und Vivek Sarkar. „Savina – An Actor Benchmark Suite“. In: *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!* Portland, OR: ACM, 2014, S. 67–80.

[17] Jóakim v. Kistowski u. a. „How to Build a Benchmark“. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Association for Computing Machinery, 2015, S. 333–336. ISBN: 9781450332484. DOI: 10.1145/2668930.2688819. URL: https://doi.org/10.1145/2668930.2688819.

[18] Bob Lantz, Brandon Heller und Nick McKeown. „A Network in a Laptop: Rapid Prototyping for Software-Defined Networks“. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Association for Computing Machinery, 2010. ISBN: 9781450304092. DOI: 10.1145/1868447.1868466. URL: https://doi.org/10.1145/1868447.1868466.

[19] Stefan Marr, Benoit Daloze und Hanspeter Mössenböck. „Cross-Language Compiler Benchmarking: Are We Fast Yet?“ In: *SIGPLAN Not.* 52.2 (Nov. 2016), S. 120–131. ISSN: 0362-1340. DOI: 10.1145/3093334.2989232. URL: https://doi.org/10.1145/3093334.2989232.

[20] Mininet Team. *Mininet*. 2018. URL: http://www.mininet.org/ (besucht am 12. 04. 2020).

[21] Florian Myter, Christophe Scholliers und Wolfgang De Meuter. „Many Spiders Make a Better Web: A Unified Web-Based Actor Framework“. In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2016. Association for Computing Machinery, 2016, S. 51–60. ISBN: 9781450346399. DOI: 10.1145/3001886.3001892. URL: https://doi.org/10.1145/3001886.3001892.

[22] S. Blesing. *Principled, tunable, cross-language actor benchmarking (no release)*. 2020. URL: https://github.com/sblessing/run-actor-run/tree/master/OOPSLA2020 (besucht am 06. 07. 2020).

[23] Marian Triebe u. a. „Das C++ Actor Framework im Leistungsvergleich“. In: *Report 302, 8. GI/ITG Workshop Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen (MMBnet15)*. Hamburg, Germany: Universität Hamburg, Dept. Informatik, 2015, S. 83–89.

[24] Typesafe Inc. *Akka Framework*. 2020. URL: http://akka.io (besucht am 12. 04. 2020).

[25] Typesafe Inc. *Akka Framework Documentation*. 2020. URL: http://akka.io/docs (besucht am 12. 04. 2020).

[26] Ivan Valkov, Natalia Chechina und Phil Trinder. „Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka“. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Association for Computing Machinery, 2018, S. 218–225. ISBN: 9781450351911. DOI: 10.1145/3167132.3167144. URL: https://doi.org/10.1145/3167132.3167144.

[27] Sebastian Wölke u. a. „Locality-Guided Scheduling in CAF“. In: *Proc. of the 8th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '17), Workshop AGERE!* Vancouver, CA: ACM, 2017, S. 11–20.

A Anhang

Der Anhang zu dieser Arbeit befindet sich auf der CD, diese enthält die Implementierungen der Benchmarks und die Rohdaten für die Graphen.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Verteilte Benchmarks für Aktor-Systeme

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	[redacted]

[redacted]