Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Ein Testwerkzeug für das Internet der Dinge

Daniel Krüger

Studiengang:	Informatik
Prüfer/in:	PD Dr. rer. nat. habil. Holger Schwarz
Betreuer/in:	Dipl.-Inf. Pascal Hirmer, Ana Cristina Franco da Silva, M.Sc.
Beginn am:	8. Mai 2017
Beendet am:	8. November 2017
CR-Nummer:	C.3, C.4, D.1.3, D.2.3, D.3.3, D.4.1, D.4.7, H.5.2, H.5.3

Kurzfassung

Wegen der stetig fallenden Preise für Hardware sind in der heutigen Zeit immer mehr Geräte miteinander vernetzt. Dabei kommunizieren Sensoren, Aktoren und Steuergeräte miteinander. Diesen Wandel nennt man das Internet der Dinge (IoT). Ein Ziel des Internet der Dinge ist es, Situationen automatisch zu erkennen und zu steuern. Dies kann durch sogenannte Complex Event Processing (CEP)-Systemen ermöglicht werden. Diese lesen Datenströme ein und erkennen vorher definierte Muster, die Situationen.

Das Testen von IoT-Umgebungen ist jedoch teuer, da Hardware beschafft werden muss. Deswegen ist die Simulation von IoT-Umgebungen erstrebenswert. In dieser Arbeit wird ein web-basiertes Werkzeug vorgestellt, welches die Simulation von Sensoren ermöglicht. Es ist möglich, mehrere Sensoren mit unterschiedlichen Datentypen, Startwerten und Abweichungen zu simulieren.

Ein weiteres, im Rahmen dieser Arbeit behandeltes, Problem ist, dass noch keine Benchmarks für CEP-Systeme existieren. Für darauf aufbauende Arbeiten wird hier untersucht, wie eine Datengenerierung für solche Benchmarks umgesetzt werden kann und welche Anforderungen an die Benchmarks gestellt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACA Autonomous Component Architecture. 20

CEP Complex Event Processing. 3

CoAP Constrained Application Protocol. 18

DPWS Devices Profile for Web Services. 18

GUI Graphical User Interface. 20

HDFS Hadoop Distributed File System. 17

HTTP Hypertext Transfer Protocol. 18

IoT Internet der Dinge. 3

JNI Java Native Interface. 32

MQTT Message Queue Telemetry Transport. 15

MQTT-SN MQTT for Sensor Networks. 18

MVC Model-View-Controller. 24

NS2 Network Simulator. 20

OTcl Object-Oriented Tool Command Language. 20

SOAP Simple Object Access Protocol. 18

TCP Transmission Control Protocol. 18

UDP User Datagram Protocol. 34

WSDL Web Service Description Language. 18

1 Motivation

In den letzten Jahren sind die Preise für Hardware, wie zum Beispiel Netzwerkkomponenten und Prozessoren, stetig gefallen [15]. Wegen diesen Preistrends gibt es immer mehr mit dem Internet verbundene Geräte. Die Verbindung dieser Geräte bezeichnet man als IoT. Im IoT kommunizieren die Geräte meist selbstständig, d.h. ohne menschliches Zutun [SBH+17]. In den letzten Jahren hat sich die Forschung rund um IoT stark beschleunigt. Firmen wie Bosch [17c], Nest [17a] und andere bieten bereits Geräte an, welche IoT-Technologien benutzen. Zum Beispiel Feuermelder von Bosch [17b] und Überwachungskameras von Nest [17d], welche an das Internet angeschloßen sind. Zusätzlich ist ein Trend hin zur Automatisierung in vielen Bereichen des Alltags zu beobachten. Dies betrifft unter anderem Wohnumgebungen (Smart Home), industrielle Produktionsabläufe (Smart Factory) oder selbst fahrende Autos (Smart Cars). Ein Beispiel für diese Art der Automatisierung könnte ein Haus sein, welches an einem sonnigen Tag die Jalousien schließt, sobald die Helligkeit im Raum einen gewissen Wert überschreitet. Ein anderes Beispiel wäre eine Fertigungsmaschine in einer Fabrik, welche automatisch neue Teile bestellt, sobald sie feststellt, dass der Bestand unter einen gewissen Wert sinkt. Bei der Entwicklung von IoT-Anwendungen gibt es viele Plattformen, Sprachen und Systeme. Diese bilden eine komplexe Umgebung, in der Fehler teuer sind, da sie zu Produktionsausfällen oder gefährlichen Situationen führen.

Die Erkennung derartiger Situationen kann mit sogenannten CEP-Systemen ermöglicht werden [SHWM16]. Diese konsumieren Datenströme basierend auf einzelnen Events und sind in der Lage vordefinierte Muster echtzeitnah zu erkennen [SHB+17]. Dabei können mit ihnen Informationen gewonnen werden, die keiner Einzelkomponente zur Verfügung stehen. Außerdem bieten diese Systeme Anfragesprachen ähnlich zu SQL an, um die Situationen zu definieren.

Auch um derartige CEP-Systeme testen bzw. vermessen zu können sind realitätsnahe Testdaten notwendig. Aufgrund der Dynamik und Flexibilität realer Daten – besonders von Sensordatenströmen – stellt dies eine große Herausforderung dar [HBS+16] [HWBM16]. Werte können zum Beispiel stark variieren, d.h., es kann Ausreißer geben, sie können ungenau sein oder sogar falsch-positiv. Da Realdaten nur basierend auf kostspieligen, echten Szenarien (bspw. im Fabrikumfeld) erzeugt werden können, müssen diese aus Zeit- und Kostengründen für die Validierung von Anwendungen möglichst realitätsnah simuliert werden.

Auch gibt es noch keine Benchmarks für CEP-Systeme, um diese vergleichen zu können. Für CEP-Systeme gibt es bislang noch keine Benchmarks. Es ist somit bisher noch nicht

möglich, diese detailliert miteinander zu vergleichen. Eines der Ziele dieser Arbeit war es daher, eine Datengenerierung für Benchmarks zu entwickeln. Zusätzlich wurde untersucht, wie viele Threads synchron Daten erzeugen können und wie groß die maximale zeitliche Abweichung bei asynchronen Threads ist.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 1 – Motivation: motiviert die Ziele der Arbeit.

Kapitel 2 – Grundlagen: beschreibt die Grundlagen dieser Arbeit.

Kapitel 3 – Verwandte Arbeiten: stellt verwandte Arbeiten vor.

Kapitel 4 – Datengenerierung für Simulation: erläutert, wie das Simulationswerkzeug implementiert wurde.

Kapitel 5 – Datengenerierung zum Benchmarking von CEP-Systemen: erläutert, wie Wertegenerierung für Benchmarks implementiert wurde.

Kapitel 6 – Evaluation: beschreibt die Evaluation dieser Arbeit.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen

In diesem Kapitel werden wesentliche Grundlagen dieser Arbeit beschrieben.

2.1 Internet der Dinge

Das IoT bezeichnet die Verbindung von Alltagsgeräten mittels einem Netzwerk. Durch das Ermöglichen von Interaktion mit eingebetteten Systemen wird sich die Allgegenwärtigkeit des Internets noch weiter erhöhen, was zu einem hoch verteilten Netzwerk von Geräten führt. Diese Geräte können dabei sowohl mit Menschen als auch mit anderen Geräten kommunizieren. Xia et al. [XYWV12] beschreiben die Konzepte des IoT ausführlich.

Zum besseren Steuern und Reagieren auf Situationen in einer IoT-Umgebung bedient man sich der Verarbeitung von komplexen Ereignissen (Englisch: Complex Event Processing) [Rob10]. Dabei wird ein Ereignis (Englisch: event) als ein Objekt bezeichnet, welches eine Aufzeichnung einer Aktivität ist. Ereignisse stehen zeitlich, kausal und strukturell miteinander in Verbindung. Zeit ist dabei eine Eigenschaft eines Ereignis in Form von einem Zeitstempel. Ein höherwertiges Ereignis A kann aus einer Menge von Ereignissen $\{B_i\}$ erstellt werden und repräsentiert ein komplexes Ereignis, welches aus allen Aktivitäten besteht, die die aggregierten Ereignisse widerspiegeln. Es können sich daraus komplexe Hierarchien von Ereignissen ergeben, wobei diese weiter zu immer abstrakteren Ereignissen aggregiert werden können. So entstehen Ereignisse, welche näher an Geschäftsprozessen liegen und weiter von der primitiven technologischen Ebene entfernt sind [JMM11]. Ein primitives Ereignis ist zum Beispiel ein Wert, erzeugt durch einen Temperatursensor. Die Verarbeitung von komplexen Ereignissen involviert Regeln zum Aggregieren, Filtern und Abgleichen von primitiven Ereignissen, gekoppelt mit Aktionen um neue abstraktere Ereignisse daraus zu erzeugen. Diese Regeln werden meist mit speziellen Query-Sprachen der CEP-Systeme definiert.

Message Queue Telemetry Transport (MQTT) [Loc10] ist ein Netzwerkprotokoll, welches das „Publish-Subscribe“-Muster implementiert [Loc10] und bestimmte „Quality Of Service“-Levels garantiert [Loc10]. Clients verbinden sich zum Broker (welcher als MQTT-Server fungiert) und veröffentlichen Nachrichten auf einen bestimmten Kanal (Englisch: Topic). Außerdem ist es möglich einen Kanal zu abonnieren. Dadurch wird der Abonnent beim Eintreffen einer Nachricht benachrichtigt.

2.2 Echtzeitprogrammierung

Ein Echtzeitsystem wird als ein System definiert, welches auf einen extern generierten Eingabestimulus innerhalb einer endlichen, vorher festgelegten Zeitperiode antworten muss. Die Korrektheit eines solchen Systems hängt nicht nur vom logischen Resultat ab, sondern auch von der Zeit, zu der es geliefert worden ist. Die Unfähigkeit zu antworten ist dabei genauso schlimm, wie eine falsche Antwort. Der Computer ist dabei oft Teil eines größeren Systems; er ist ein eingebettetes System. Als hartes Echtzeitsystem wird ein System definiert, welches beim Verpassen einer Deadline in einen abnormalen Zustand gerät (z.B. ein Flugzeug). Als weiches (Englisch: soft) Echtzeitsystem wird ein System definiert, bei dem Deadlines wichtig sind, welches jedoch trotzdem noch funktioniert, falls Deadlines verpasst werden (z.B. Software zur Datenakquise). Weitere Informationen zur Echtzeitprogrammierung werden von Burns et al. [BW01] beschrieben.

2.3 Benchmarking

Ein Benchmark [DM02] ist ein Versuchsaufbau, welcher es ermöglicht, empirische Messungen über ein System durchzuführen. Diese Messungen sollen es erlauben, die Leistungsfähigkeit verschiedener Systeme anhand bestimmer Metriken miteinander zu vergleichen. Meistens sind Benchmarks Tabellen, welche die Performanz eines jeden Lösungsansatzes für bestimmte Metriken, wie CPU-Zeit, Anzahl der Funktionsauswertungen oder ähnliches, zeigen. Ein Problem bei Benchmarks ist die Interpretation der Ergebnisse aus diesen Tabellen, da diese unterschiedlich ausfallen kann. Deswegen sind die Werkzeuge zum Analysieren dieser Daten wichtig. In [DM02] wird deswegen empfohlen, den Durchschnitt oder das Kumulative der Performanz für eine bestimmte Metrik zu benutzen. Da hier jedoch Durchläufe nicht beachtet werden, in denen der Lösungsansatz fehlschlägt, werden damit robustere Lösungsansätze bevorzugt und es kommt somit zu einer Befangenheit (Englisch: bias) in den Ergebnissen. Um dies zu verhindern, kann eine Strafe für jeden fehlgeschlagenen Durchlauf berechnet werden. In einer späteren Arbeit sollen Benchmarks für CEP-Systeme erstellt werden, damit diese miteinander verglichen werden können.

3 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten vorgestellt. Es wird außerdem erläutert, warum diese nicht vollständig zur Problemstellung dieser Arbeit passen.

3.1 Telit IoT Random Number Generator Action Block

Diese Anwendung [Tel17] kann verwendet werden, um einen Sensor, bzw. dessen Werte, zu simulieren. Damit kann zum Beispiel ein Trigger getestet werden. Bei der Simulation kann der Datentyp ausgewählt werden (`INT4, UINT4, BINARY, STRING`) sowie die Anzahl der generierten Werte. Darüber hinaus erlaubt die Anwendung die Definition von Minimum- und Maximumwerten. Die Zahlen werden in Variablen gespeichert. Dieses Werkzeug ist Teil des *Telit IoT Portal* [Tel17], welches ein cloud-basierter Subscription-Service ist, bei dem Kunden ihre IoT Umgebung modellieren, steuern und überwachen können. Der *Telit IoT Random Number Generator Action Block* ist nicht als einzelnes Modul erhältlich. Zwar ermöglicht der Random Number Generator von *Telit IoT* die Generierung von Werten zur Simulation von Sensoren, jedoch erfüllt dieser nicht die Echtzeitanforderungen, die für die CEP Benchmarks benötigt werden, da es nicht möglich ist, dass mehrere Generatoren synchron Werte erzeugen.

3.2 IOTSim: A simulator for analysing IoT applications

IoTSim [ZGS+17], welches auf *CloudSim* basiert, ist ein Werkzeug zur Simulation der Verarbeitung von IoT-Applikationen erzeugten Daten in Cloud-Umgebungen, zur Modellierung und Simulation der parallelen Ausführung von mehreren, großen IoT-Applikationen in einer geteilten Cloud-Umgebung, sowie zur Evaluation der Performanz von IoT-Applikationen in Cloud-Umgebungen. *CloudSim* ist ein Simulationswerkzeug, welches die Modellierung, Simulation und Evaluation von Cloud-Umgebungen, ihrer Policies und ihrer Workload-Modelle erlaubt. *IoTSim* besteht aus den folgenden Ebenen: dem *Cloudsim Core Simulation Engine Layer*, dem *Cloudsim Simulation Layer*, dem *Storage Layer*, dem *Big Data Processing Layer* und dem *User Code Layer*. Das Cloudsim Core Simulation Engine Layer ist die unterste Ebene, welche Ereignisse der Cloud-Umgebung (Englisch: Events) einreiht und verarbeit , Cloud-Entitäten erstellen (Services, Hosts, Rechenzentren, Broker und virtuelle Maschinen), zwischen Komponenten kommunizieren

und die Simulations-Clock steuern kann. Darüber gibt es das Cloudsim Simulation Layer, welches die Modellierung und Simulation von virtualisierten Rechenzentren unterstützt. Dies beinhaltet dedizierte Steuerschnittstellen für virtuelle Maschinen, Arbeitsspeicher, persistenter Speicher und Bandbreite. Diese Ebene simuliert folgendes Verhalten: Provisionierung von Hosts auf VMs, das Steuern der Ausführung und das Überwachen des Systemzustands. Das Cloudsim Simulation Layer besteht auch noch aus mehreren Unterebenen, welche Kernelemente der Cloud modellieren. Die untersten dieser Unterebenen steuern Netzwerktopologie, Datenzentren und Cloud-Koordinatoren. Das Storage Layer modelliert die Speicherung durch *Amazon S3*, *Azure Blob Storage* oder Hadoop Distributed File System (HDFS). IoT-Applikationen können mit dem Storage Layer interagieren wie gewohnt: sie schreiben und lesen Daten. Wie in der Realität, gibt es eine messbare Verzögerung durch diese Ebene. Das Big Data Processing Layer übernimmt die Verarbeitung der aus IoT-Applikationen entstehenden Daten. Es besteht aus zwei Unterebenen. Die *MapReduce*-Unterebene [DG08] unterstützt Applikationen mit einem batch-orientierten Datenverarbeitungsparadigma. Die Streaming-Computing-Unterebene unterstützt Applikationen, welche Echtzeitanforderungen haben. Das User Code Layer ist die oberste Ebene. Auf ihr werden Entitäten der Hosts (Anzahl der Maschinen, deren Spezifikation), die Konfigurationen von IoT-Applikationen, VMs, Anzahl von Benutzern und deren Applikationstypen zur Verfügung gestellt. Diese Ebene hilft Benutzern, ihre eigenen Simulationsszenarien festzulegen beziehungsweise zu konfigurieren, um ihre Algorithmen zu validieren. IotSim ermöglicht somit nur die Simulation der Cloud-Umgebung; das Simulieren des Verhalten von Teilen einer IoT-Applikation, ein Ziel dieser Arbeit, wird aber nicht von *IotSim* übernommen.

3.3 SimpleIoTSimulator

SimpleIoTSimulator [Sim17] ist ein Werkzeug zur Netzwerksimulation. Damit kann der Paketfluss zwischen Sensoren und Gateways aufgenommen werden. Diese gelernten Daten können dann als Template benutzt werden, um Testumgebungen mit tausenden von Sensoren zu generieren. Darin können dann Skripte ausgeführt werden, um Fehlerszenarien und Notifikationen zu erzeugen. Die Eigenschaften dieser Skripte können dynamisch geändert werden. Bei den erstellten Sensoren kann auch noch eingestellt werden, dass diese Zufallswerte senden. Dabei wird eine Reihe von Werten sowie ein Intervall in Sekunden angegeben und nach jedem Intervall wird einer dieser Werte ausgegeben. Unterstützte Protokolle zur Kommunikation sind Modbus über Transmission Control Protocol (TCP), MQTT, MQTT for Sensor Networks (MQTT-SN), Constrained Application Protocol (CoAP) und Hypertext Transfer Protocol (HTTP). Da die Generierung von Zufallswerten sehr begrenzt ist, die Werteausgabe nicht im sub-Sekunden-Bereich möglich ist und es auch nicht möglich ist, die Sensoren miteinander zu synchronisieren, ist der *SimpleIoTSimulator* nicht passend für diese Arbeit.

3.4 DPWSim

DPWSim [HLC+14] basiert auf dem Devices Profile for Web Services (DPWS)-Standard, der von *Microsoft* entwickelt wurde. In DPWSim können DPWS-Geräte simuliert werden, welche im Netzwerk entdeckt werden können und welche mit anderen Geräten oder Clients über das DPWS-Protokoll kommunizieren können. Zusätzlich kann DPWSim die Umgebung, in der sich die Geräte befinden, simulieren und ermöglicht es Benutzern, Simulationen mit hoher Flexibilität zu erstellen, zu speichern und zu laden. In DPWSim gibt es vier Komponenten: Räume, Geräte, Operationen und Ereignisse. Ein Raum besitzt Geräte und für jedes Gerät können Operationen und Ereignisse definiert werden. Operationen sind Funktionen des Geräts, wie zum Beispiel das Einschalten oder das Ausschalten. Die Operationen werden in Web Service Description Language (WSDL) [CCM+01] beschrieben und können durch Service-Endpunkte unter Benutzung von Simple Object Access Protocol (SOAP) [MPD+02] durchgeführt werden. Das Ergebnis einer Operation wird in der grafischen Benutzeroberfläche angezeigt. Ereignisse sind Veränderungen des Gerätezustands. Wenn sich der Gerätezustand ändert, werden die Klientenanwendugen des Geräts benachrichtigt. Ein Ereignis kann periodisch auftreten, zum Beispiel jede Millisekunde. Geräte werden dadurch simuliert, indem sie auf Basis von Sensordaten operieren. Diese Daten müssen vom User oder anderen Clients erzeugt werden. Insofern scheidet diese Arbeit aus, da DPWSim nicht selbst Werte generieren kann.

3.5 icclab IoT Simulator

Der IoT Simulator von icclab [icc17] simuliert Sensoren in einer Umgebung. Diese generieren Zufallszahlen vom Typ Integer und geben ihren Ladestand sowie die verbrauchte Leistung an. Dies passiert periodisch. Zusätzlich wird zufällig entschieden, ob eine Nachricht erfolgreich versendet wurde. Die Nachricht wird an einen RabbitMQ-Broker [VW12] geschickt. RabbitMQ ist ein Open-Source Message Broker. Mit dem icclab IoT Simulator kann eine ähnliche Funktionalität erreicht werden, wie in dieser Arbeit angestrebt. Jedoch kann der Benutzer nicht die Parameter der Simulation wählen. Des Weiteren ist keine Benchmarkgenerierung möglich, da Simulationen nicht wiederholbar sind und die Werteausgabe nicht synchron erfolgt.

3.6 Netzwerksimulatoren

NS2 [IH11], J-SIM [SHK+06], OMNet++ [VH08], Cooja [ODE+06] und TOSSIM [LLWC03] sind Simulatoren für Rechnernetze. Sie sind dazu gedacht, den Paketfluss in einem Netzwerk zu simulieren und sorgen so für eine bessere Implementierung von Rechnernetzen. Dabei gibt es Szenarien, in denen Hosts dazu kommen und wegfallen. Sie werden auch

zum Simulieren von Sensornetzwerken benutzt. Sie sind nicht dazu gedacht, Sensoren zu simulieren und deswegen unbrauchbar für uns.

Der Network Simulator (NS2) [IH11] ist ein event-basiertes Werkzeug zur Simulation von dynamischen Rechnernetzen gedacht. Mit NS2 können zum Beispiel Routing-Algorithmen und verschiedene Protokolle simuliert werden. NS2 liefert ein ausführbares Kommando: „ns“, welches eine Object-Oriented Tool Command Language (OTcl) Datei als Eingabeargument annimmt. NS2 besteht aus zwei Sprachen: C++ und der OTcl. C++ definiert die internen Mechanismen (das Backend) der Simulation, während OTcl die Simulation aufsetzt, indem es Objekte assembliert und konfiguriert, sowie diskrete Ereignisse scheduled. Variablen in der OTcl-Domäne können auf C++-Objekte abgebildet werden. Diese Variablen werden „Handles“ genannt. Die Funktionalität ist definiert im C++-Objekt. Der Handle agiert als ein „Frontend“, welches mit Benutzern und anderen OTcl-Objekten interagiert. Ein Handle kann seine eigenen Prozeduren und Variablen definieren um diese Interaktion zu vereinfachen. Simulationen werden durch den Aufruf „ns [<file>] [<args>]“ gestartet. Als erste Phase der Simulation wird die Simulation entworfen. Hier entscheidet der Benutzer, welchem Zweck die Simulation dienen soll, die Netzwerkkonfiguration aussieht, welche Annahmen gelten sollen, wie Performanzkriterien aussehen und wie die erwarteten Ergebnisse aussehen. In der zweiten Phase wird das zu simulierende Netzwerk konfiguriert und in der dritten Phase wird die Simulation durchgeführt. Nach der Simulation können die Ergebnisse noch weiter verarbeitet werden.

J-SIM [SHK+06] ist ähnlich zu NS2, es ermöglicht ebenfalls das Modellieren, Simulieren und Emulieren von Netzwerken und ist ebenfalls quelloffen. Es baut auf einer komponentenweisen Architektur namens Autonomous Component Architecture (ACA) auf. Die ACA schliesst die Lücke zwischen Hardware- und Software-ICs und erlaubt somit, dass neue Komponenten via plug-and-play in J-Sim integriert werden können. Ausserdem wurde aufbauend auf ACA ein generalisierendes Paketaustausch-Internetworking-Framework namens INET gebaut, welches zahlreiche gemeinsame Eigenschaften des Protokoll-Stacks implementiert. ACA und INET wurden beide in Java implementiert. Zusammen mit einem Scripting-Framework und einer Graphical User Interface (GUI) bilden sie J-Sim. Die grundlegenen Entitäten sind Komponenten, welche miteinander kommunizieren durch das Senden beziehungsweise Empfangen von Daten via ihrer Ports. Wie Komponenten auf Daten reagieren und diese handhaben wird während der Design-Phase des Systems in Verträgen (Englisch: Contracts) festgelegt. Das Binding findet jedoch erst in der Systemintegration-Phase statt. Diese Trennung des Vertrag-Binding und des Komponenten-binding erlaubt J-Sim Loose-Coupling-Eigenschaften zu besitzen, das heisst Komponenten können individuell entworfen, implementiert und getestet werden, ohne Annahmen voneinander zu machen. Die Tatsache, dass J-Sim in Java implementiert ist, zusammen mit ACA, macht es zu einer platformunabhängigen, erweiterbaren und hoch wiederverwendbaren Umgebung. J-Sim bietet eine Scripting-Schnittstelle, welche Scripting in Sprachen wie Perl und Python unterstützt.

OMNet++ ist in C++ geschrieben, basiert auf dem INET-Framework und bietet mehr Infrastruktur als NS2. Die Simulation muss aber auch noch selbst geschrieben werden.

Experimente in Omnet++ sind wie folgt aufgebaut: es gibt das Model (das zu testende Objekt, welches bestimmte Parameter hat), die Study (eine Reihe von Experimenten auf einem oder mehreren Modellen), das Experiment (Durchsuchen des Parameterraum des Modells), das Measurement (Seed für das Experiment). Experimente sind reproduzierbar (gleicher Seed). Es gibt außerdem eine Resultatsanalyse, wobei Daten mit Resultaten produziert werden. Diese können mit Regeln oder Mustererkennung noch feiner gefiltert werden, was die Analyse vereinfacht und zu besseren Einsichten führen kann.

Cooja [ODE+06] ist ein Netzwerksimulator zur Simulation von Sensoren, welche das Contiki IoT-Betriebssystem benutzen. Sensoren können nicht nur unterschiedliche Software besitzen, sie können auch auf unterschiedlichen Ebenen simuliert werden: Netzwerk-, Betriebs- und Maschinensystemebene sind möglich. Ein Sensor besitzt drei grundlegende Eigenschaften: einen Arbeitsspeicher, einen Knotentypen und seine Hardwareperipherie. Sensoren vom selben Typ werden mit dem selben Arbeitsspeicher initialisiert und lassen auch den selben Code laufen. Ihre Arbeitsspeicher können sich später jedoch unterscheiden, da sie eventuell unterschiedliche Eingaben durch ihre Hardwareperipherie empfangen. Viele Teile des Simulators können einfach ausgetauscht oder mit weiterer Funktionalität ausgestattet werden. Sensoren in einer COOJA-Simulation können auf den verschiedenen simulierten Ebenen existieren und miteinander interagieren. COOJA besitzt auch ein Modell, welches die Verbreitung von Radiowellen simuliert um auch diesen Aspekt in kabellosen Sensornetzwerken darzustellen. Dieses Modell kann auch selbst noch erweitert werden.

Alle hier vorgestellten Netzwerksimulatoren eignen sich für die Simulation des Verhalten von Sensornetzwerken, die Simulation des Verhalten der Sensoren selbst ist jedoch nicht möglich, was diese Anwendungen unbrauchbar für die Ziele dieser Arbeit macht.

3.7 Streaming Engines

Neben den vorgestellten Simulatoren wurden Streaming Engines untersucht, um zu überprüfen, ob diese die Generierung von Werten unterstützen.

Zuerst wurde Apache Kafka [Gar13] untersucht. Apache Kafka ist ein verteilter Message-Broker und wird eher als Schnittstelle oder als verteilte Datenpipeline benutzt. Zum Beispiel kann eine Datei mit Datenwerten an ein Eingabe-Topic gesendet werden, woraufhin Kafka diesen einliest und verarbeitet. Danach wird der Stream an ein Ausgabe-Topic ausgegeben. Unser Ziel ist es jedoch, dass man die zu generierenden Werte als Datei eingibt, auf dessen Basis ein reproduzierbarer Stream erzeugt ein. Diese Funktionalität müsste mit Kafka selbst implementiert werden.

Apache Storm [IS15] ist eine verteilte Echtzeit-Berechnungsplattform. Daten werden aus einer Datei eingelesen, eine Quelle („Spout“) erzeugt daraus einen Stream, welcher an Verarbeitungseinheiten (Bolts) weitergeleitet wird. Die benutzte Datenstruktur sind Tupel (geordnete Liste von Elementen). Die Berechnung läuft verteilt in einem Cluster.

Dabei gibt es einen Master-Knoten und mehrere Worker-Knoten. Storm hat jedoch eine Mindeslatenz im Bereich von 10 ms (Kosten für Transfer von Daten zwischen Bolts, Garbage Collection), was für die Benchmarkgenerierung inakzeptabel ist.

Apache Flink [CKE+15] dient ebenfalls dem Verarbeiten von Streams. Es bietet jedoch mehr High-Level-Funktionalität, welche man in Storm von Hand implementieren müsste. Flink ist für zyklische, iterative Transformationen auf Collections optimiert. Außerdem bietet es Batch-Processing.

Apache Spark [SS15] ist ähnlich. Es ist ein Batch-Processing Framework, welches Streams verarbeiten kann. Spark ist optimiert auf das Verarbeiten von verteilten Datensätzen (resilient distributed datasets).

Zusammenfassend bieten Flink und Spark also nur die Verarbeitung von Streams an. Storm bietet von allen Frameworks die beste Realtime-Performanz, welche jedoch für die Ziele dieser Arbeit nicht ausreicht. Dadurch ist klar, dass sich die anderen Frameworks sich nicht für die Benchmarkgenerierung eignen.

4 Datengenerierung für Simulation

Eines der wesentlichen Ziele dieser Arbeit ist die Simulation von Sensoren. Das Simulationswerkzeug soll dabei an ein Überwachungswerkzeug oder an ein CEP-System angeschlossen werden. Es soll außerdem möglich sein, Live-Änderungen der Simulationsparameter vorzunehmen, wobei diese sofort in der Simulation reflektiert werden sollen. Durch das veränderte Verhalten der Umgebung zu beobachten in den angeschlossenen Werkzeugen können dann eventuell neue Schlüsse gezogen werden. Als Parameter für die Simulation wurden festgelegt:

- Name - der Name des Sensors
- Datentyp - der Datentyp (Integer, Float, Boolean) der zu generierenden Werte
- Startwert - der Wert, mit dem die Simulation beginnt
- Ausreißerwahrscheinlichkeit - Wahrscheinlichkeit für Anomalien, Float-Wert zwischen 0.0 % und 100.0 %
- Änderungsrate - gibt die Änderung pro Zeitschritt an

Falls der Datentyp als Boolean festgelegt wird, werden Startwert, minimale und maximale Abweichung, Ausreißerwahrscheinlichkeit und die normale Änderungsrate nicht benötigt. Außerdem soll das Werkzeug mit einer Wahrscheinlichkeit, die dem Quadrat der Ausreißerwahrscheinlichkeit entspricht, Werte erzeugen, die nicht dem Datentyp entsprechen. Jeder neu erzeugte zu simulierende Sensor soll in einem eigenen Thread Werte mittels Publish-Subscribe (bspw. realisiert durch MQTT) schicken, wobei der Topic-Name dem Sensornamen entspricht.

4.1 Systemübersicht und Implementierung

Zu Beginn der Simulation wurden noch keine Sensoren angelegt. Um dies zu tun, passt der Benutzer die Simulationsparameter nach seinen Bedürfnissen an und erstellt dadurch die Simulation eines Sensors. Die erstellte Simulation kann anschließend in einer Liste ausgewählt werden. Sobald diese ausgewählt ist, werden die generierten Werte in einer MQTT-Konsole angezeigt. Dies ist möglich, da in der Anwendung ein MQTT-Client benutzt wird, welcher die simulierten Werte empfängt und anzeigt. Währen der Simulation können die Parameter der ausgewählten Simulation dynamisch geändert werden. Nach dem Übernehmen dieser Änderungen wird die Simulation „live“ angepasst, d.h.,

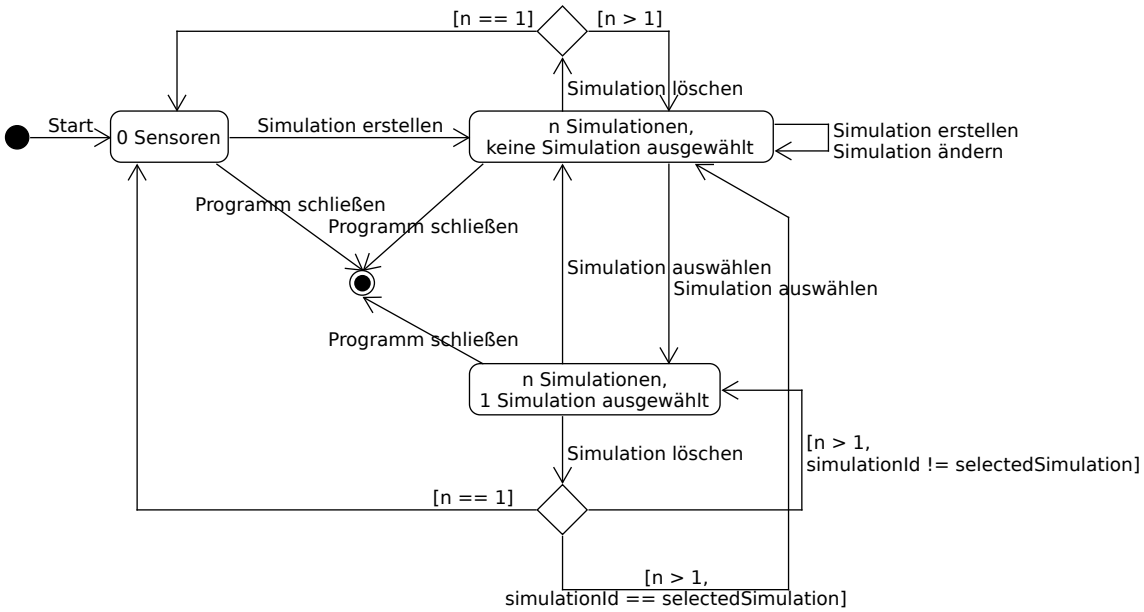

Abbildung 4.1: Zustandsdiagramm des Simulationswerkzeuges

die Änderungen sind sofort sichtbar. Wird die Simulation nicht mehr benötigt, kann diese angehalten und entfernt werden. Die Simulation mehrerer Sensoren parallel ist möglich.

Um die vorgestellte Funktionalität zu realisieren, wurde ein Prototyp erstellt. In der ersten Version dieses Prototyps war es lediglich möglich, eine feste vorgegebene Anzahl an Simulationen zu erstellen. Dies wurde in der Weiterentwicklung, zusammen mit den weiteren oben beschriebenen Funktionen, ermöglicht. Basierend auf diesem Prototypen, sollte die Benchmarkgenerierung integriert werden.

Bei der Weiterentwicklung des Prototyps fiel jedoch auf, dass die Benchmark-Generierung gesondert betrachtet werden muss. Daher ist diese Teil eines separaten Prototyps.

4.2 Benutzerschnittstelle

Die Benutzerschnittstelle wurde mittels dem ReactJS-Framework [1] umgesetzt. React ist ein Framework, welches 2011 von Facebook entwickelt wurde. Es wurde 2013 Open Source freigegeben [Fac17b], 2017 unter der MIT-Lizenz lizensiert [Fac17a] und seitdem von namhaften browserbasierten Applikationen Webseiten / Web-Apps (Facebook, Instagram, Paypal, Netflix, Walmart) verwendet. In der Model-View-Controller (MVC)-Architektur bietet React lediglich die Realisierung des Views beziehungsweise der Ansicht an. Durch Hinzunahme von Frameworks wie Flux, Redux oder MobX kann jedoch eine vollständige

[1] https://reactjs.org/

MVC-Architektur realisiert werden. Der Code in React ist in Komponenten eingeteilt, welche logische beziehungsweise funktionale Einheiten der Benutzeroberfläche repräsentieren. Zum Beispiel kann eine derartige Komponente eine Liste oder ein Auswahlknopf sein.

Die Einteilung der View in Komponenten erleichtert die Entwicklung der Benutzeroberfläche, beispielsweise durch vereinfachtes Debugging. Komponenten in React besitzen *Props*, welche ähnlich zu den Parametern einer Funktion funktionieren und für die Komponente unveränderlich sind. Es können darüber hinaus Callback-Funktionen übergeben werden. Dies folgt dem Prinzip „properties flow down, actions flow up“. Der unidirektionale Fluss von Daten erleichtert die Fehlersuche. Komponenten besitzen neben Props auch einen Zustand, *state* genannt. Außerdem besitzt React ein Virtuelles Document Object Model (Virtual DOM). Sobald sich bei Komponenten der Zustand verändert, unterscheidet sich das Virtual Dom vom „echten Dom“, d.h. der visualisierten Benutzeroberfläche. Daraufhin werden lediglich die Komponenten neu gerendert, bei denen sich der Zustand geändert hat. Dies gibt React eine bessere Performance als manch andere Frameworks [2]. Außerdem muss der Programmierer sich nicht darum kümmern, Änderungen des Zustands auf DOM zu übertragen. Dies wird automatisch von React übernommen. React bietet eine Syntaxerweiterung von Javascript namens JSX an, mit der man Markup in Javascript schreiben kann. Dies erleichtert die Lesbarkeit des Codes und verhindert XSS-Schwachstellen. Durch das Benutzen von Source-Maps beim Build-Prozess mit Webpack und Babel lässt sich der Code auf Komponenten-Ebene in den Chrome Entwicklerwerkzeugen betrachten. Source-Maps bilden den kompilierten und minimierten Javascript-Code auf die Ursprungsdateien ab und erleichtern somit das Debuggen [3]. Außerdem lassen sich dort Haltepunkte setzen und der aktuelle Wert von Variablen beziehungsweise Attributen auslesen. Zusätzlich gibt es Entwicklungswerkzeuge speziell für React, welche man für die Webbrowser Chrome und Firefox herunterladen kann. Mit ihnen können der Zustand und die Parameter der React-Komponenten verfolgt werden.

Neben React wurde bei der Frontend-Entwicklung Flow eingesetzt. Flow erlaubt die statische Code-Analyse von Javascript-Programmen. Flow wurde von Facebook entwickelt und wird unter der MIT-Lizenz lizensiert. Flow erlaubt auch die Erstellung von Alternativnamen für Typen, um auftretende Fehler aussagekräftiger zu machen. Das Benutzen von Flow erhöht die Robustheit des Codes der Benutzeroberfläche. Dies wird durch eine hohe Typ-Abdeckung erreicht. Typ-Abdeckung bezeichnet den Prozentsatz des Codes, für den der Typ einer Operation bekannt ist. Webpack kompiliert die Flow-annotierten Dateien zu validem Javascript-Code. Gao et al. [GBB17] haben gezeigt, dass der Einsatz von Flow oder Typescript ungefähr 15 Prozent der Bugs in öffentlichen Github-Projekten verhindern könnten. In ihrer Arbeit wurde auch gezeigt, dass Fehler in Flow mit geringerem Zeichen- und Zeitaufwand erkennbar sind, als bei Typescript [GBB17]. Dies ist dadurch erklärbar, dass Flow schon einige Typen selbst inferiert. Bei Typescript müssen

[2] https://rawgit.com/krausest/js-framework-benchmark/master/webdriver-ts/table.html
[3] https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/

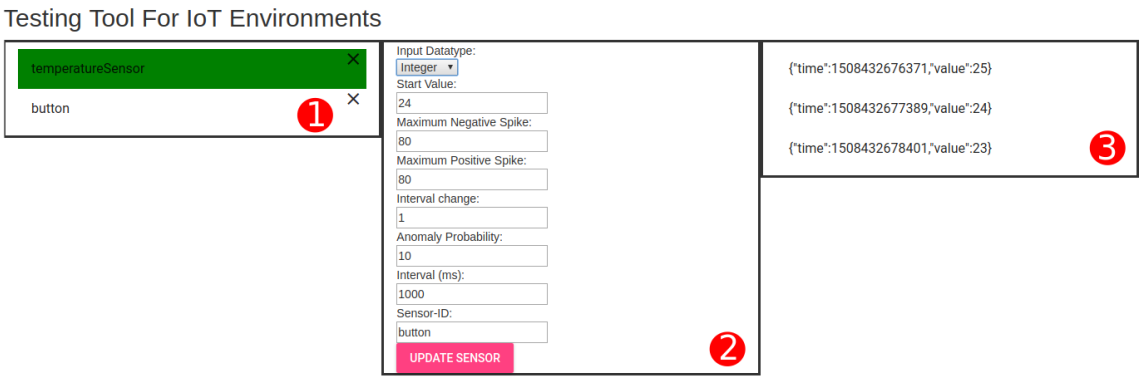

Abbildung 4.2: Frontend des Simulationswerkzeuges

diese meist vom Programmierer selbst annotiert werden. Diese Gründe haben dazu geführt, Flow für das Frontend zu benutzen. Darüber hinaus hatte ich in einem früheren Projekt die Erfahrung gemacht, dass die Entwicklung mit „reinem“ Javascript zu vielen vermeidbaren Laufzeitfehlern führt.

Bei dieser Arbeit wurde kein Werkzeug zum Handhaben des Zustands, wie zum Beispiel Flux oder Redux benutzt, da dies die Benutzeroberfläche unnötig kompliziert gemacht hätte. Der Zustand wird im Backend verwaltet und die Sensorenliste für die Benutzeroberfläche wird in der App-Komponente gespeichert und ihren Unterkomponenten als Argument übergeben.

Die Benutzeroberfläche ist in folgende Komponenten aufgeteilt: App, SensorList, InputForm, MqttConsole. App ist die Hauptkomponente, welche die restlichen Komponenten darstellt. In ihrem Zustand werden die vom Backend erhaltene Liste der Sensorsimulationen und der derzeit ausgewählte, zu simulierende Sensor gespeichert. App besitzt zwei Handler-Funktionen; eine, welche die Sensorliste aktualisiert und eine andere, welche den derzeit selektierten Sensor aktualisiert. Anstatt ein Werkzeug zum Handhaben des Zustands zu benutzen habe ich mich entschieden, Zustandsänderungen durch asynchrone *Callbacks* zu bewerkstelligen. *Callbacks* werden Funktionen genannt, welche anderen Funktionen als Argument übergeben werden und somit zu einem späteren Zeitpunkt ausgeführt werden können, meist um Werte in der aufrufenden Funktion zu verändern. Dazu bekommen Kindkomponenten diese Handler übergeben, um somit den rückfließenden Datenfluss zu ermöglichen.

SensorList (siehe Nummer 1 in Abbildung 4.2) zeichnet die Liste der aktuell existierenden Sensoren. SensorList verfügt selbst über keinen Zustand sondern bekommt die Liste der zu simulierenden Sensoren von App übergeben. SensorList besitzt zwei Handler-Funktionen. Eine zum Auswählen von Sensoren und eine zum Löschen von Sensoren. Falls ein Sensorsimulation zum Löschen markiert wird, wird eine GET-Anfrage an das Backend durchgeführt, um diese zu löschen. Sobald das Löschen erfolgreich durchgeführt wurde, wird die als Callback übergebene Funktion zur Aktualisierung der Sensorliste aufgerufen und bekommt die neu erhaltene Sensorliste übergeben.

InputForm (siehe Nummer 2 in Abbildung 4.2) stellt das Formular zum Eintragen der Simulationsparameter dar. Falls der ausgewählte Datentyp Boolean ist, werden bestimmte Eingabefelder wie Startwert, maximale und minimale Abweichung, Ausreißerwahrscheinlichkeit und Änderungsrate nicht dargestellt. Die Komponente InputForm besitzt die Werte der Eingabefelder der Simulationsparameter als Zustand und wird mit Beispielswerten initialisiert. Es verfügt über Handler-Funktionen zum Starten beziehungsweise Ändern der Simulation und zum Ändern der angezeigten Parameterwerte. Falls eine Sensorsimulation gerade selektiert ist, führt das Ändern der Werte und Auswählen von „Change Simulation“ zum sofortigen Ändern des Simulationsverhaltens.

MqttConsole (Nummer 3 in Abbildung 4.2) stellt die Konsole zum Anzeigen der generierten Zufallszahlen dar. Die Komponente speichert die ausgewählte Sensorsimulation sowie die letzten erhaltenen Nachrichten. Außerdem besitzt sie über einen eingebauten MQTT-Client. Falls ein Sensor ausgewählt wird, registriert der Client sich zu dem Topic mit dem selbigen Namen. Falls eine neue Simulation ausgewählt wird, registriert der Client sich zu einem neuen Topic. Sobald der Client Nachrichten empfängt, werden diese in das Nachrichten-Array des Zustands aufgenommen. Schlussendlich werden die Array-Elemente als Listenelemente gezeichnet.

4.3 Backend

Das Backend wurde mittels Spring realisiert [JHD+04]. Spring Boot ist eine Variante des Spring Frameworks, welche das „Convention-Over-Configuration“ Entwurfsmuster implementiert. „Convention-Over-Configuration“ bezeichnet dabei, dass der Benutzer eines Frameworks nur die Implementierungen konfigurieren muss, welche von den Konventionen abweichen. Spring Boot erlaubt es „stand-alone“ Spring-Anwendungen zu erstellen, welche einen eingebetteten Tomcat Webserver benutzen. Außerdem können Spring-Boot-Andwendungen einfach mit Maven [Sma+05] gebaut werden und durch Aufrufen von „java -jar <spring-boot-application>.jar“ ausgeführt werden, es muss also keine war-Datei in einem Tomcat-Server bereitgestellt werden. Zusätzlich werden vorgefertigte Project Object Models angeboten, um die Konfiguration zu vereinfachen. Ich hatte mich für Spring Boot entschieden, da ich schon aus einem vorherigen Projekt gute Erfahrungen gemacht hatte und den niedrigen Konfigurationsaufwand und die

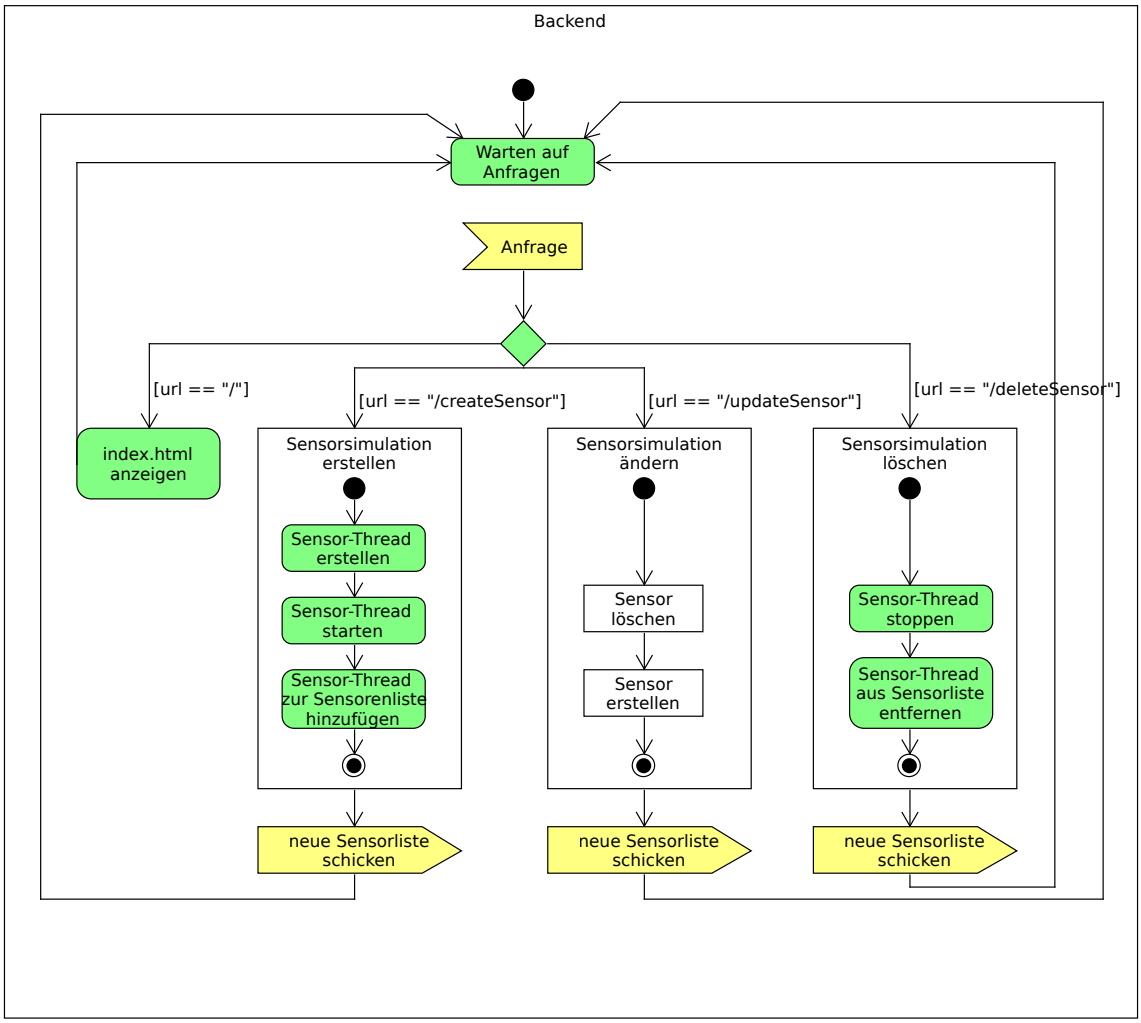

Abbildung 4.3: Aktivitätsdiagramm des Simulationswerkzeuges

einfache Aufstellung von Spring-Boot-Anwendungen als großen Vorteil sehe, was zu einem schnelleren Entwicklungszyklus führen kann.

Das Backend ist aus folgenden Klassen aufgebaut: Application, SimulatorController, ThreadManager und SensorSimulatorThread. Die Klasse Application ist Einstiegspunkt sowie Hauptklasse des Simulationswerkzeugs und startet die gesamte Spring-Boot-Anwendung.

Die Klasse SimulatorController bildet Anfragen auf Methoden ab. Anfragen an */createSensor* führen dazu, dass eine neue Sensorsimulation mit den übermittelten Parametern erstellt wird und dass die neue Liste von Simulationen dem Benutzerschnittstelle übermittelt wird. Anfragen an */updateSensor* führen dazu, dass ein neuer Sensor seine Simulation anhand der übermittelten Parameter ändert und dass die neue Liste von Sensorsimulationen dem Benutzerschnittstelle übermittelt wird. Anfragen an */deleteSensor* führen dazu, dass eine Sensorsimulation aus der Simulationsliste gelöscht wird und dass die neue Liste

von Simulationen dem Benutzerschnittstelle übermittelt wird. Anfragen an */setBroker* führen dazu, dass der zu verwendende MQTT-Broker geändert wird.

Die Klasse ThreadManager verwaltet die Liste von SensorSimulatorThreads, welche die Simulationsliste darstellt. Sie besitzt die Methoden: *createSensor*, welche einen neuen SensorSimulatorThread erzeugt und zur Sensorliste hinzufügt, *updateSensor*, welche „deleteSensor“ für den alten Sensor und *createSensor* mit den erhaltenen Parametern aufruft, *deleteSensor*, welches den per Sensor-Id gesuchten Sensor-Thread findet, ihn stoppt und ihn aus der Sensorenliste löscht und•sensorsToJSONArray, welche die Sensorliste zu einem JSONArray umwandelt, um diese der Benutzerschnittstelle zu schicken.

Die Klasse SensorSimulatorThread besitzt die Parameter der Simulation als Attribute. Die Klasse besitzt die folgenden Methoden: „toJSON“, welche ein JSON-Objekt aus dem Sensor erzeugt. Diese Methode wird von der Methode „sensorsToJSONArray“ der Klasse ThreadManager benutzt. „generateInt“ erzeugt eine Integer-Zufallszahl basierend auf den Simulationsparametern. „generateFloat“ erzeugt eine Float-Zufallszahl basierend auf den Simulationsparametern. „generateBoolean“ erzeugt einen Boolean-Zufallswert basierend auf den Simulationsparametern. Die Methode „run“ generiert einen Wert mithilfe einer der obigen Methoden, packt diesen zusammen mit einem Zeitstempel in ein JSON-Objekt und schickt anschließend das JSON-Objekt als String an den MQTT-Broker.

5 Datengenerierung zum Benchmarking von CEP-Systemen

Ein weiteres Ziel dieser Arbeit ist die Generierung von Daten, welche zur Erstellung von Benchmarks für CEP-Systeme benutzt werden können. Die Aufgabe ist wie folgt definiert: als Eingabe soll eine Datei mit Simulationswerten und Zeitstempeln dienen. Der Benutzer soll dann die Anzahl der Threads für die Werteausgabe angeben können. Daraufhin sollten alle Threads synchron (das heisst auf die Millisekunde gleiche Zeitstempel) ausgeührt werden. Als maximale Werte, welche von der Datengenerierung unterstützt werden sollen, wurden 1000 Threads mit 1 Millisekunde Intervalllänge angegeben. Dabei sollte die Reproduzierbarkeit der Wertegenerierung nicht verloren gehen. Um diese Performanz zu erreichen, wurden bei allen außer dem ersten Versuch Effizienzoptimierungen vorgenommen. Es wurde, zum Beispiel, kein Frontend verwendet und statt MQTT wurden die Daten direkt in Dateien oder Arrays geschrieben oder an das CEP-System weitergegeben.

5.1 Erste Implementierung in Java

Zuerst wurde versucht, die Wertegenerierung für die Benchmarks durch das Simulationswerkzeug zu realisieren. In dem Werkzeug generieren die Threads der Sensorsimulation ihre Zufallswerte zusammen mit dem momentanen Zeitstempel in die Ausgabe und warten dann, durch Benutzen von „Thread.sleep“, so viele Millisekunden, wie in der Eingabedatei angegeben. Die Ausgabe wurde per MQTT verschickt. Ein Abwandlung davon ist, dass die Threads sich durch *Busy Waiting* synchronisieren. Dies geschieht, indem der Thread Manager allen Threads der Sensorsimulation eine Startzeit, welche von der Anzahl der Threads abhängt, vorgibt. Alle Threads führen dann Busy Waiting durch via Überprüfung von „System.nanoTime“ bis die Startzeit erreicht ist. Sobald dies der Fall ist, generieren sie die Ausgabe und erhöhen die nächste Startzeit um die in der Eingabedatei definierte Intervalllänge. Anschließend wird erneut Busy Waiting durchgeführt. Hier gab es aber schon bei geringen Threadanzahlen, Verzögerungen von mehreren Millisekunden.

5.2 Implementierungen in C

Als nächstes wurde untersucht, wie die Referenzimplementierung in performanteren C-Code umgewandelt werden kann. Das C-Programm besteht im Wesentlichen aus zwei den Funktionen: „main“ und „wait“. Bei der Simulationsausführung werden die Threads erstellt und diese führen dann die Funktion „wait“ aus. Bei wait setzen sich die Threads eine Startzeit, welche von der Anzahl der Threads abhängt. Danach wird, wie oben beschrieben, bis zum Startzeitpunkt gewartet. Sobald die Startzeit erreicht ist, werden die aktuellen Millisekunden ausgegeben. Beim Messen der Zeiten wurde das „time value“-Struct von C benutzt, zusammen mit der Funktion „gettimeofday“. Beim Warten auf die nächste Ausgabezeit wurden verschiedene Ansätze versucht. Als erste Funktion zum Warten auf den nächsten Durchlauf wurde usleep benutzt mit verschiedenen Zeitwerten. Usleep erlaubt es, den Thread die angegebene Zahl an Mikrosekunden ruhen zu lassen. Als Werte wurden untersucht: 1 Mikrosekunde, 10 Mikrosekunden und 50 Mikrosekunden. Als zweite Funktion zum Warten auf den nächsten Durchlauf wurde Nanosleep benutzt mit verschiedenen Zeitwerten. Nanosleep erlaubt es, die Ausführung vom Thread eine angegebene Zahl an Nanosekunden ruhen zu lassen. Als Werte wurden untersucht: 1 Nanosekunde, 10 Nanosekunden und 100 Nanosekunden. Als letzter Ansatz wurde Busy-Waiting untersucht. Das Benutzen der Funktion usleep zum Warten von einer Mikrosekunde als Argument lieferte dabei die beste Performanz.

Bei der vorherigen Implementierung wurden die Zeitwerte durch das „printf“-Kommando ausgegeben. Somit war es nicht möglich, die Zeitstempel der einzelnen Threads miteinander zu vergleichen. Deswegen wurde eine weitere Implementierung angefertigt, bei der das C-Programm via Java Native Interface (JNI) aufgerufen wurde. Dann wurden in den Durchläufen Dateien mit den Zeitwerten erstellt und die Dateien wurden miteinander verglichen. Um C-Programme über JNI aufzurufen, müssen folgende Schritte durchgeführt werden. Zuerst muss aus dem Java-Programm eine Header-Datei erstellt werden, welche im C-Programm eingebunden wird. Anschließend muss das C-Programm als Shared Library kompiliert werden. Danach kann die Shared Library im Java-Programm importiert werden und dessen Funktionen aufgerufen werden. Das C-Programm wurde so abgeändert, dass in dieser Implementierung die Zeitstempel der Threads in Dateien geschrieben werden. Jeder Thread erstellt dabei seine eigene Datei und schreibt in diese, um Probleme beim Zugriff von gemeinsamen Ressourcen zu vermeiden und den Auslesezeitprozess zu vereinfachen. Nach Ausführung des C-Programms werden die Dateien vom Java-Programm ausgelesen und die Zeitstempel der jeweiligen Schleifendurchläufe aller Threads mit den Zeitstempeln des ersten Threads verglichen. Falls es Abweichungen gibt, werden die ID des Threads und die Schleifennummer, in der es zu einer Abweichung gekommen ist, ausgegeben. Hier war es möglich, Werte zu generieren mit bis zu 35 synchronen Threads.

Bei den vorherigen Implementierungen von synchronen Threads in C wurden noch keine Optimierungen beim Scheduling angewandt. Alle Threads liefen mit normaler Priorität. Zusätzlich lief die Implementierung auf einem normalen Ubuntu Linux-System und nicht

auf einem richtigen Echtzeit-Betriebssystem. Diese Punkte wurden als Verbesserungspotentiale gesehen. In der main-Methode der Implementierung wird zuerst die Verwendung von Arbeitsspeicher überprüft, da es leicht möglich ist, zu viel Arbeitsspeicher zu beanspruchen. Zuerst wird überprüft, wieviele Seitenfehler es gibt. Seitenfehler treten auf, falls ein Programm versucht auf eine Speicherseite zuzugreifen, wofür es keine Abbildung in der Memory Management Unit in dem virtuellen Speicher des Prozesses gibt. Die Speicherseite muss dann von der Festplatte geladen werden, was deutlich länger dauert als ein Zugriff auf den Arbeitsspeicher. Anschließend wird das Verhalten der dynamischen Speicherverwaltung konfiguriert. Auf den Speicherbereich des Prozesses wird *mlockall* angewandt, damit dieser nicht ausgelagert werden kann, was Zugriffszeiten erhöhen würde. Außerdem wird angegeben, dass der Prozess keinen Speicher freigibt, da dies die Ausführung des Prozesses verlangsamt. Danach wird wieder die Anzahl der Seitenfehler überprüft. Als nächstes werden der vom Prozess benötigte Speicher reserviert und es wird überprüft, ob die Anzahl der Seitenfehler 0 beträgt. Anschließend werden mehrere Variablen definiert zur Verwaltung der Simulation, wie zum Beispiel Anzahl der Threads, Anzahl der Schleifendurchläufe, Länge des Intervalls in Millisekunden sowie die anfängliche Verzögerung in Mikrosekunden. Zusätzlich werden 2 Strukturen erzeugt: eine für Zeitwerte und eine andere für die Argumente der Threads. Auch wird ein Array vom Typ pthread erzeugt, um die Zeiger auf die Threads abzuspeichern. Der Typ pthread bezieht sich auf POSIX Threads. POSIX Threads ist ein sprachenunabhängiger IEEE Standard, welcher eine API für die Erstellung von Threads in C definiert. Danach wird geprüft, ob die Anzahl der vergangenen Mikrosekunden seit der letzten Millisekunde kleiner als 500 sind. Andernfalls wird per Busy Waiting gewartet, bis das der Fall ist. Dies wird getan, um zu vermeiden, dass ein Thread mit seiner Ausgabe in die nächste Millisekunde schreitet, da „usleep“ eine gemessene Mindestlatenz von 500 Mikrosekunden hat, da hierbei auch Systemaufrufe erledigt werden müssen, zum Beispiel beim Auslesen der Uhrzeit des Systems. Als nächstes wird die Startzeit festgelegt, indem zwei long-Werte erstellt werden (einer für die Sekunden, einer für die Mikrosekunden) welche schliesslich inkrementiert werden abhängig davon wie groß die Verzögerung ist. Danach werden die Argumente für die Threads vorbereitet. Zu den Argumenten für die Threads zählen: die Thread-ID, die Startzeit in Sekunden, die Startzeit in Mikrosekunden, die Länge des Intervalls in Mikrosekunden und die Schleifenanzahl. Anschließend werden die Threads gestartet und sie führen die Funktion „rundeadline“ aus. Am Anfang der Funktion werden die Argumente entpackt und in lokale Variablen gespeichert. Außerdem werden eine Struktur für das Erfassen der Zeit angelegt und eine Datei zur Persistierung der Zeitstempel des Threads. Danach werden die Variablen initialisiert und die Scheduling-Policy des Threads wird festgelegt. Wie vorher erwähnt, wurde hier auch versucht, ein Echtzeit-Betriebssystem zu nutzen. Deswegen wurde auf dem Testsystem ein Echtzeit-Betriebssystemkern zu den verfügbaren Betriebssystemkernen hinzugefügt. Mit dem einem Echtzeit-Betriebssystemkern ist es möglich, Zugang zu Echtzeit-Schedulern zu bekommen. Ein Scheduler entscheidet, in welche Reihenfolge Prozesse und Threads auf dem Prozessor ausgeführt werden. Jeder Thread besitzt eine Policy und eine Priorität. Der Scheduler entscheidet aufgrund der Policy und der Priorität, welcher Thread als nächstes ausgeführt wird. Standardmäßig wird bei Linux der „Completely Fair Scheduler“ benutzt.

Die normalen Scheduling-Policies sind: `SCHED_OTHER`, `SCHED_IDLE` und `SCHED_BATCH`. Mit dem Echtzeit-Betriebssystemkern hat man Zugriff auf weitere Scheduling-Policies: First-In-First-Out, Round-Robin und dem Deadline-Scheduler. Bei First-In-First-Out wird, wie der Name schon sagt, der erste Thread in der Warteschlange als nächstes ausgeführt. Beim FIFO-Scheduling lagen die Verzögerungen im Bereich von 30 Millisekunden, was sich durch Unterbrechungen von anderen Threads erklären lässt. Bei Round-Robin bekommt jeder Thread ein Zeitquantum zugeteilt, wovon seine Ausführungsdauer abhängt. Bei Round-Robin-Policy ist es nicht möglich, das Zeitquantum festzulegen. Das vorgegebene Zeitquantum ist beim Testsystem 25 Millisekunden, was es unmöglich macht, alle Threads auf die Millisekunde genau zu synchronisieren. Bei der Deadline-Policy bekommt jeder Thread eine Periode und eine Laufzeit zugeteilt. Der Scheduler lässt jeden Thread dann jede Periode so lange laufen, wie in der Laufzeit angegeben ist. Bei dieser Implementierung wurde eine Deadline-Scheduling-Policy mit einer Laufzeit von 50 Mikrosekunden und einer Periode von 1 Millisekunde gewählt, damit jeder Thread möglichst genau einmal pro Millisekunde eine Ausgabe tätigt. Nach der Festlegung der Scheduling-Policy für den Thread läuft die Zeitmessung. Jeder Thread wartet, bis die Startzeit beginnt, indem er *usleep(1);* aufruft und dann den jetzigen Zeitwert erfasst. Sobald die Startzeit erreicht ist, schreibt er seine ID, die jetzige Schleifennummer und seinen jetzigen Zeitstempel in eine Datei. Danach erhöht er die nächste Startzeit um die Intervalllänge und inkrementiert die Schleifennummer. Falls er die maximale Anzahl von Durchläufen erreicht hat, bricht die Ausführung ab. Nach der Ausführung werden die Zeitstempeln in den Dateien miteinander verglichen und es wird überprüft, ob alle Threads den gleichen Zeitstempel in jedem Schleifendurchlauf haben.

Bei der reinen C-Implementierung mit verbessertem Scheduling ließen sich schon bessere Ergebnisse erzielen. Die Ein- und Ausgabe von Dateien ist jedoch mit einem hohen Zeitaufwand verbunden, da hier in persistentem Speicher während der Ausführung geschrieben wird. Deswegen wurde eine weitere Implementierung vorgenommen, wobei die Ein- und Ausgabe vermieden wurde, indem man die Zeitstempel stattdessen via TCP verschickt und per Serversocket in Java einliest. Zusätzlich werden die empfangenen Zeitwerte in ein CEP-System eingegeben. In diesem Fall wurde Esper verwendet. Die Implementierung auf C-Seite ist hier nahezu identisch. Der Ablauf der Simulation ist hier wie folgt. Zuerst werden Werte wie die Thread- und Schleifenanzahl sowie die Startzeit initialisiert. Danach werden Threads erstellt, welche Server-Sockets aufbauen und auf ankommende Verbindungen warten. Es gibt dabei eine 1:1 Beziehung zwischen Threads auf C- und Java-Seite. Sobald eine Verbindung aufgebaut wird, wird ein Socket erstellt und die empfangenen Werte in Ereignisse verpackt und an die Esper-Engine geschickt. Ein Ereignis besteht aus einer Thread-ID, einer Schleifendurchlaufsnummer und einem Integer-Wert. Während die Threads auf Java-Seite auf ankommende Verbindungen warten, wird die C-Seite via JNI gestartet. Jeder Thread auf C-Seite generiert einen Wert und schickt diesen per TCP an die Java-Seite. Es wurde TCP genommen, da der teure Verbindungsaufbau nur einmal beim Start des Threads passiert und somit keine Nachteile gegenüber anderen Protokollen, wie zum Beispiel dem User Datagram Protocol (UDP), entstehen. Als nächstes wird eine simple Anfrage an die Esper CEP-Engine gestellt,

welche alle empfangenen Ereignisse auswählt. Außerdem wird ein Listener hinzugefügt, welcher die empfangenen Zeitstempel in ein zweidimensionales Array schreibt, wobei der Zeilenindex gleich der Threadnummer und der Spaltenindex gleich der Schleifennummer ist. Am Ende der Ausführung werden die Zeitstempel des Array verglichen und die maximale Verzögerung zwischen zwei Threads ausgegeben.

Bei der gemischten Implementierung von C und Java geht jedoch die Echtzeit verloren, da auf Java-Seite Just-In-Time-Kompilierung vorgenommen wird und die Garbage-Collection zu beliebigen Zeitpunkten aktiv werden kann laufen kann, wann sie will. Deswegen wurde auch noch eine reine Implementierung der Zeitmessung in C vorgenommen. Diese ist nahezu identisch zu der C-Implementierung mit Echtzeit-Schedulern, nur dass die Zeitwerte in ein Array geschrieben werden und dass am Schluss der main-Funktion die Zeitwerte verglichen werden und die maximale zeitliche Verzögerung zwischen den Threads gemessen wird.

5.3 Timer, Monitor und Threads in Java

Da nach den anfänglichen Versuchen klar war, dass harte Echtzeit mit „normalem“ Java nicht möglich ist, wurde die Zielsetzung aufgelockert. Diese Umstellung ist möglich, da Esper es erlaubt, dass man ein Zeitfenster definiert, in dem Werte berücksichtigt werden. Die folgenden Implementierungen wurden vorgenommen, um zu untersuchen, wie groß die maximale Verzögerung zwischen den Threads bei der Wertegenerierung ist.

Bei dieser Implementierung läuft die Simulation wie folgt ab. Zuerst werden mehrere Variablen deklariert, darunter: die Anzahl der Threads, die Anzahl der Schleifendurchläufe, die Länge des Intervalls und die Startzeit. Anschließend wird ein Timer erstellt mit der Startzeit, dem Intervall und einem Monitor als Argumenten. Jedes Mal wenn der Timer dann feuert, benachrichtigt der Monitor alle Objekte, die auf ihn warten. Außerdem werden Threads gestartet, welche die Schleifenanzahl und den Monitor als Argumente haben. Die Threads warten auf den Monitor. Sobald sie von ihm benachrichtigt werden, schreiben sie einen Zeitstempel in ein eindimensionales Array mit der aktuellen Schleifendurchlaufnummer als Index und warten dann auf die nächste Benachrichtigung vom Monitor. Dies wird solange durchgeführt, wie in der Schleifenanzahl angegeben. Nachdem alle Threads beendet sind, werden ihre Arrays zu einem zweidimensionalem Array fusioniert, nach Thread-ID und Schleifendurchlaufnummer als Indizes. Zum Schluss werden die minimalen und maximalen Thread-Zeiten von jedem Schleifendurchlauf aufgenommen und deren Differenz berechnet. Das Maximum dieser Differenzen wird berechnet und dann als die maximale zeitliche Verzögerung zwischen zwei Threads in allen Schleifendurchläufen ausgegeben.

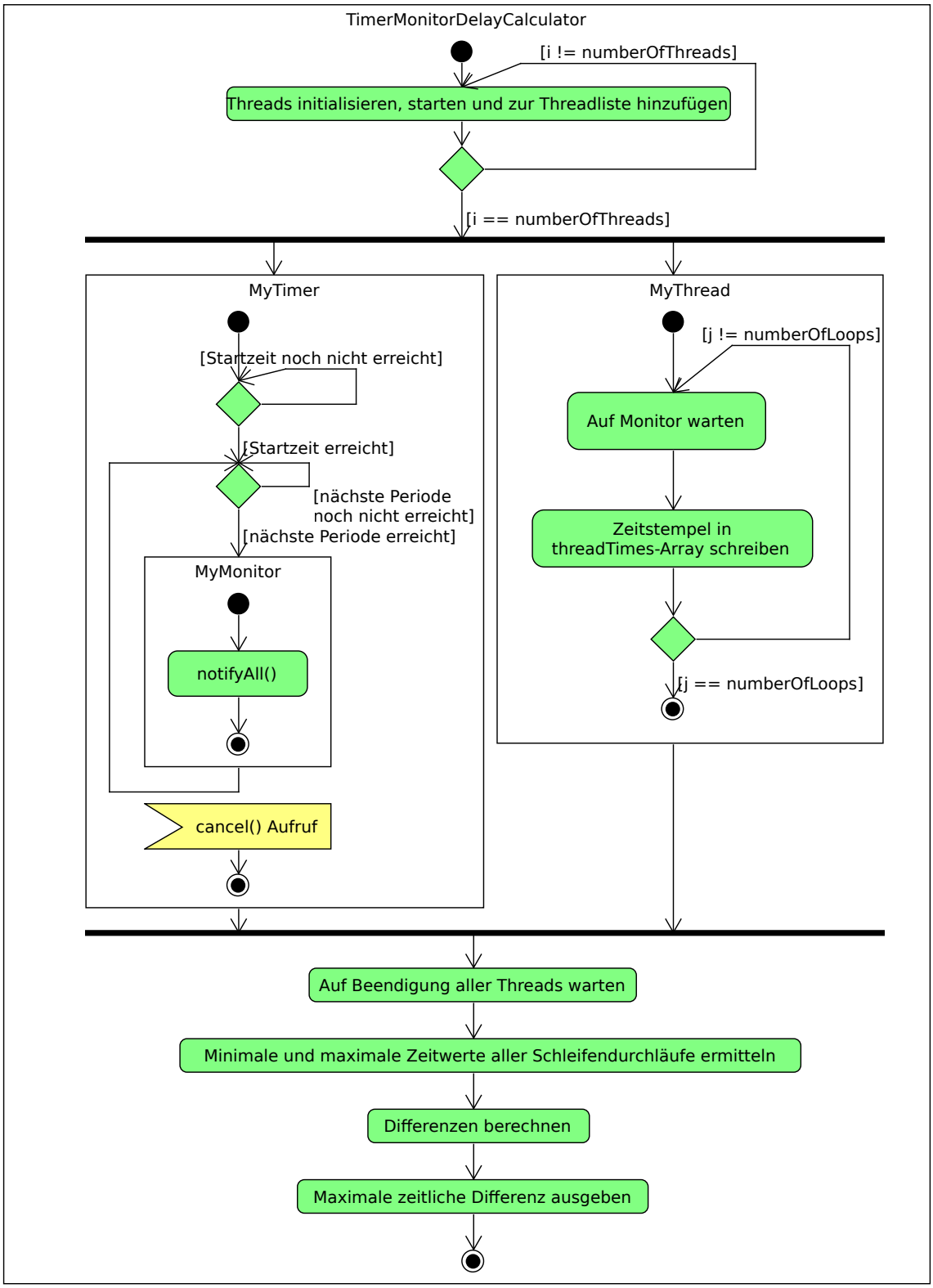

Abbildung 5.1: Aktivitätsdiagramm zum Unterkapitel 5.3

5.4 Esper und Java

Hier läuft die Simulation sehr analog zur vorherigen Implementierung ab. Es werden am Anfang wieder mehrere Variablen deklariert, darunter: die Anzahl der Threads, die Anzahl der Schleifendurchläufe, die Länge des Intervalls und die Startzeit. Anschließend wird ein Timer erstellt mit der Startzeit, dem Intervall und einem Monitor als Argumenten. Jedes Mal wenn der Timer dann feuert, benachrichtigt der Monitor alle Objekte, die auf ihn warten. Es werden auch wieder Threads gestartet, welche die Schleifenanzahl und den Monitor als Argumente haben. Die Threads warten auf den Monitor. Sobald sie von ihm benachrichtigt werden, schreiben sie einen Zeitstempel in ein eindimensionales Array mit der aktuellen Schleifendurchlaufnummer als Index und warten dann auf die nächste Benachrichtigung vom Monitor. Einer der wenigen Unterschiede ist, dass die Threads ihre Zeitstempel dieses Mal nicht in ein Array schreiben, sondern ein Ereignis erstellen und dieses an das CEP-System schicken. Dabei besteht ein Ereignis aus einer Thread-ID, einer Schleifendurchlaufsnummer und einem Integer-Wert. Der Thread führt diese Aktion so oft aus, wie in der Schleifenanzahl angegeben. Als nächstes wird eine simple Anfrage an die Esper Cep-Engine gestellt, welche alle empfangenen Ereignisse auswählt. Außerdem wird ein Listener hinzugefügt, welcher die empfangenen Zeitstempel in ein zweidimensionales Array schreibt, wobei der Zeilenindex gleich der Threadnummer und der Spaltenindex gleich der Schleifennummer ist. Am Ende der Ausführung werden die Zeitstempel des Array verglichen und die maximale Verzögerung zwischen zwei Threads ausgegeben.

5.5 Timer, Monitor und Threads in Echtzeit-Java

Nachdem bei der endgültigen C-Implementierung mit der Benutzung eines Echtzeit-Betriebssystemkerns deutliche Performanzgewinne zu sehen waren, wurde untersucht, ob es nicht auch Echtzeitalternativen zur Java Virtual Machine (JVM) gibt. Eine davon ist die JamaicaVM, welche eine deterministische Garbage Collection, Ahead-Of-Time Kompilierung und harte Echtzeit verspricht. Bei der normalen JVM sorgen die willkürliche Garbage Collection und die Just-In-Time (JIT) Kompilierung dazu, dass Echtzeitgarantien schwer einzuhalten sind. JamaicaVM implementiert die „Realtime specification for Java“ (RTSJ). Es werden außerdem Echtzeit-Threads angeboten, welche strikt priorisiert werden können und welche auch, ähnlich zum Deadline Scheduling, Perioden und Laufzeiten zugewiesen bekommen können. Bei dieser Implementierung läuft die Simulation wie folgt ab. Zuerst wird ein Objekt zum Halten der Daten instanziiert, die Daten die es hält sind unter anderem: die Anzahl der Threads, die Anzahl der Schleifendurchläufe, die Länge des Intervalls und die Startzeit. Anschließend wird ein Echtzeit-Timer erstellt mit der Startzeit, dem Intervall und einem Monitor als Argumenten. Das Echtzeit-Timer beginnt ab der deklarierten Startzeit an, Ereignisse zu feuern und dies wird in Intervallen wiederholt, wobei die Dauer zwischen zwei Ereignissen durch die Intervalllänge festgelegt ist. Jedes Mal wenn der Echtzeit-Timer ein Ereignis feuert, benachrichtigt der Monitor alle

Objekte, die auf ihn warten. Außerdem werden Echtzeit-Threads gestartet, welche die Schleifenanzahl, den Monitor, eine Priorität und eine Periode als Argumente haben. Als Priorität wird die maximal mögliche Priorität für Echtzeit-Threads benutzt. Es wird eine Periode von 1 Mikrosekunde benutzt. Die Threads warten auf den Monitor. Sobald die Threads vom Monitor die Freigabe zum Fortfahren erhalten, wird die jetzige Uhrzeit von ihnen mittels *Clock.getRealtimeClock().getTime()* in das Zeitstempel-Array geschrieben und warten dann auf die nächste Benachrichtigung vom Monitor. Dies wird solange durchgeführt, wie in der Schleifenanzahl angegeben. Nachdem alle Threads beendet sind, werden ihre Arrays zu einem zweidimensionalem Array fusioniert, nach Thread-ID und Schleifendurchlaufnummer als Indizes. Zum Schluss werden die minimalen und maximalen Thread-Zeiten von jedem Schleifendurchlauf aufgenommen und deren Differenz berechnet. Das Maximum dieser Differenzen wird berechnet und dann als die maximale zeitliche Verzögerung zwischen zwei Threads in allen Schleifendurchläufen ausgegeben. Die *RealtimeThread*-Objekte kümmert sich um die Aufnahme der Zeitwerte. Sie besitzt die Anzahl der Schleifendurchläufe, das myMonitor-Objekt und ein Array namens threadTimes für die Zeitwerte als Attribute. In ihrer Run-Methode wird eine for-Schleife aufgerufen, welche so oft durchlaufen wird, wie in der Anzahl der Schleifendurchläufe angegeben. In der for-Schleife wird auf das MyRTMonitor-Objekt gewartet und dann die jetzige Zeit in das Array threadTimes geschrieben mit der Schleifenanzahl als Index.

5.6 Threads mit integrierten Timern in Echtzeit-Java

Es wurde auch noch eine zweite Implementierung in Echtzeit-Java entwickelt, um zu untersuchen, ob sich eine bessere Performanz erreichen lässt, falls nicht alle Threads nur durch einen Timer und Monitor synchronisiert werden, sondern dass alle Threads ihre eigenen Timer besitzen. Die Simulation ist wie folgt aufgebaut. Zuerst werden mehrere Variablen deklariert, unter anderem: die Anzahl der Threads, die Anzahl der Schleifendurchläufe, die Länge des Intervalls und die Startzeit. Im Hauptprogramm wird dieses Mal kein Echtzeit-Timer erstellt. Es werden, wie in der vorherigen Implementierung, Echtzeit-Threads gestartet, welche die Schleifenanzahl, den Monitor, eine Priorität und eine Periode als Argumente haben. Jeder Thread bekommt jeweils genau einen Timer-Objekt zugewiesen. Das Timer-Objekt beinhaltet einen Echzeit-Timer. Jeder Thread wartet auf sein Timer-Objekt. Sobald die Threads von ihren Timer-Objekten die Freigabe zum Fortfahren erhalten, wird die jetzige Uhrzeit von ihnen mittels *Clock.getRealtimeClock().getTime()* in das Zeitstempel-Array geschrieben und warten dann auf die nächste Benachrichtigung vom Monitor. Dies wird solange durchgeführt, wie in der Schleifenanzahl angegeben. Nachdem alle Threads beendet sind, werden ihre Arrays zu einem zweidimensionalem Array fusioniert, nach Thread-ID und Schleifendurchlaufnummer als Indizes. Zum Schluss werden die minimalen und maximalen Thread-Zeiten von jedem Schleifendurchlauf aufgenommen und deren Differenz berechnet. Das Maximum dieser Differenzen wird berechnet und dann als die maximale zeitliche Verzögerung zwischen zwei Threads in allen Schleifendurchläufen ausgegeben.

6 Evaluation

In diesem Kapitel werden die Implementierungen mittels Messergebnissen evaluiert. Die Evaluation wurde auf einem Rechner vom Modell *Lenovo ideapad 100S-14IBR* durchgeführt. Der Rechner besitzt einen *Intel Pentium CPU N3710* Prozessor, welcher auf 1.6 GHz getaktet ist und 4 Kerne sowie eine Wortlänge von 64 bit besitzt. Zusätzlich enthält der Prozessor einen L1 Cache mit 32 KB Größe und einen L2 Cache mit 1 MB Größe. Außerdem verfügt der Rechner über insgesamt 4 GB Arbeitsspeicher, welcher aus zwei Einheiten vom Typ *Samsung SODIMM DDR3 Synchronous* besteht, welche jeweils 2 GB Speicher besitzen und auf 1600 MHz getaktet sind. Das Motherboard ist vom Typ *Aristotle 14*. Die Festplatte des Rechners ist vom Typ *Samsung MZNTY256* und besitzt 238 GiB Speicher. Das installierte Betriebssystem ist Ubuntu 16.04.3 LTS. Die installierten Kernel sind *Linux 4.10.0-37-generic x86_64* und *Linux 4.9.40-rt30 x86_64*.

Die Evaluation ist wie folgt gegliedert: zuerst wird die C-Implementierung besprochen, dann die Java-Implementierung, danach eine Java-Implementierung mit Anschluss an Esper, dann noch zwei Implementierung in Echtzeit-Java und am Schluss werden die Ergebnisse diskutiert.

6.1 Implementierung in C

Diese Implementierung wurde mit dem Befehl *gcc -O3 -lrt -pthread <Eingabedatei> -o <Ausgabedatei>* kompiliert. Das *-lrt*-Flag sagt aus, dass die Ausgabedatei mit der Bibliothek *rt* statisch gelinkt sein sollte. Da dies zur Kompilierzeit geschieht, weiß der Compiler, welche Funktionen benötigt werden und kann eventuell Optimierungen vornehmen. Beim dynamischen Linken der Bibliothek, also das Linken zur Laufzeit, wäre das nicht möglich. Das *-O3*-Flag sagt aus, dass der Maschinencode soweit wie möglich auf Performanz optimiert werden soll und ist die höchste Optimierungsstufe, welche vom GNU C Compiler angeboten wird. Dies spiegelt sich in einer höheren Kompilierzeit wider, was jedoch wegen der geringen Größe des Programms vernachlässigbar ist. Es wurden 50 Tests mit jeweils 10 Schleifendurchläufen durchgeführt. Es wurde mit folgenden Threadanzahlen getestet: 10, 50, 100, 175, 250, 375, 500 und 1000. Jede Threadanzahl wurde mit folgenden Intervallen (in Millisekunden) getestet: 1, 10, 25, 50 und 100.

Die Evaluation dieser Implementierung wurde auf dem Echtzeit-Kernel ausgeführt. Abbildung 6.2 zeigt, wie performant das Programm ist. Die Median-Latenzen lagen bei 0 für alle Anzahlen von Threads. In C lässt sich somit Wertegenerierung mit 1000 Threads

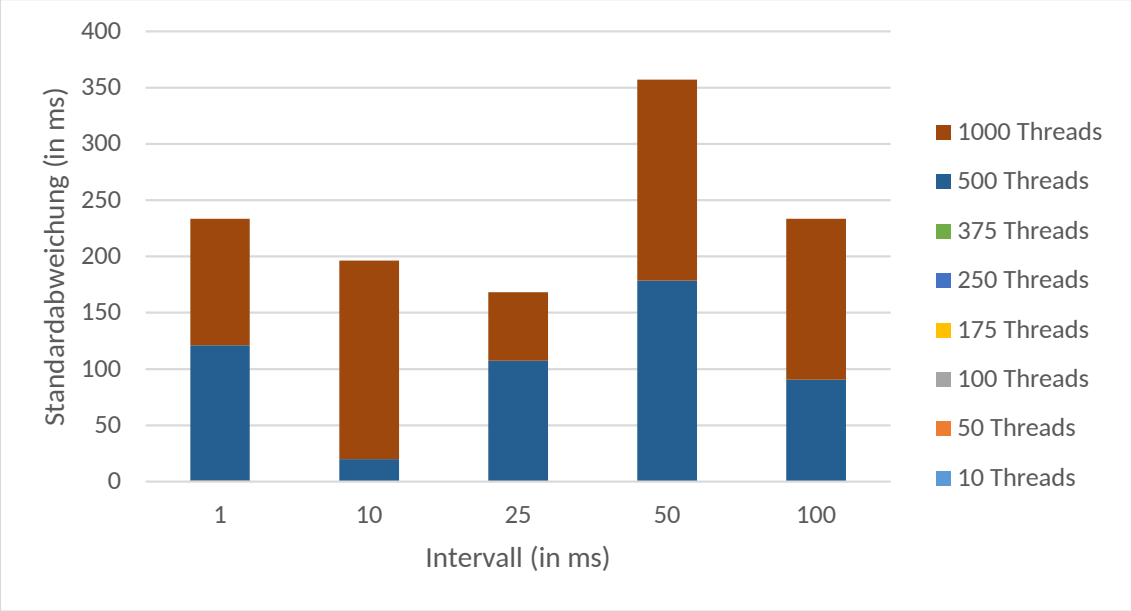

Abbildung 6.1: Kumulative Standardabweichung der Latenzen der C-Implementierung

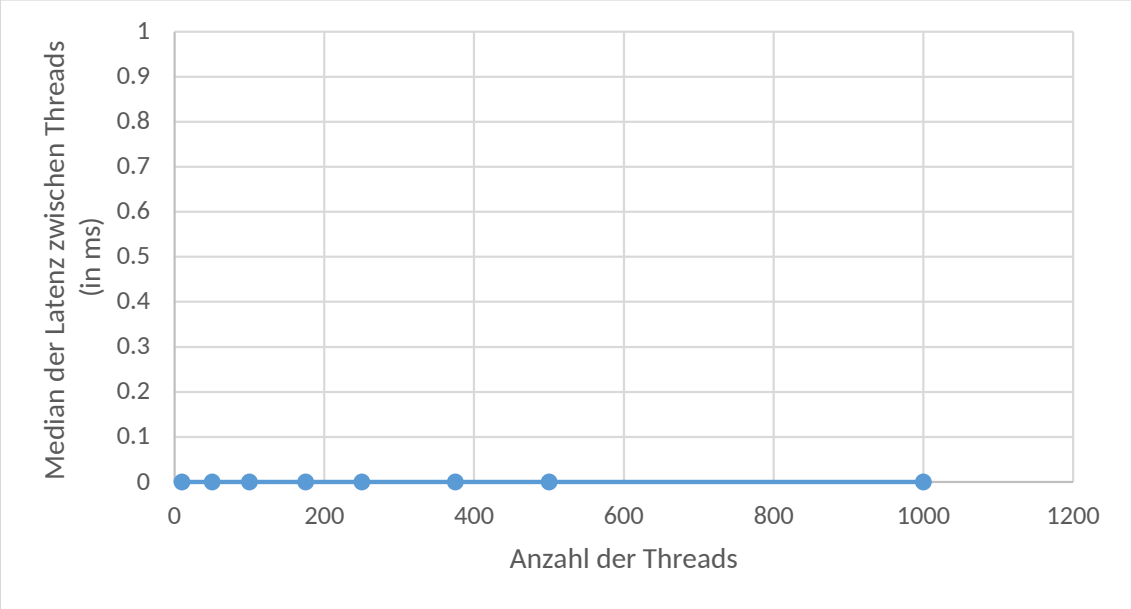

Abbildung 6.2: Median der Latenzen der C-Implementierung bei 1 Millisekunde Intervall

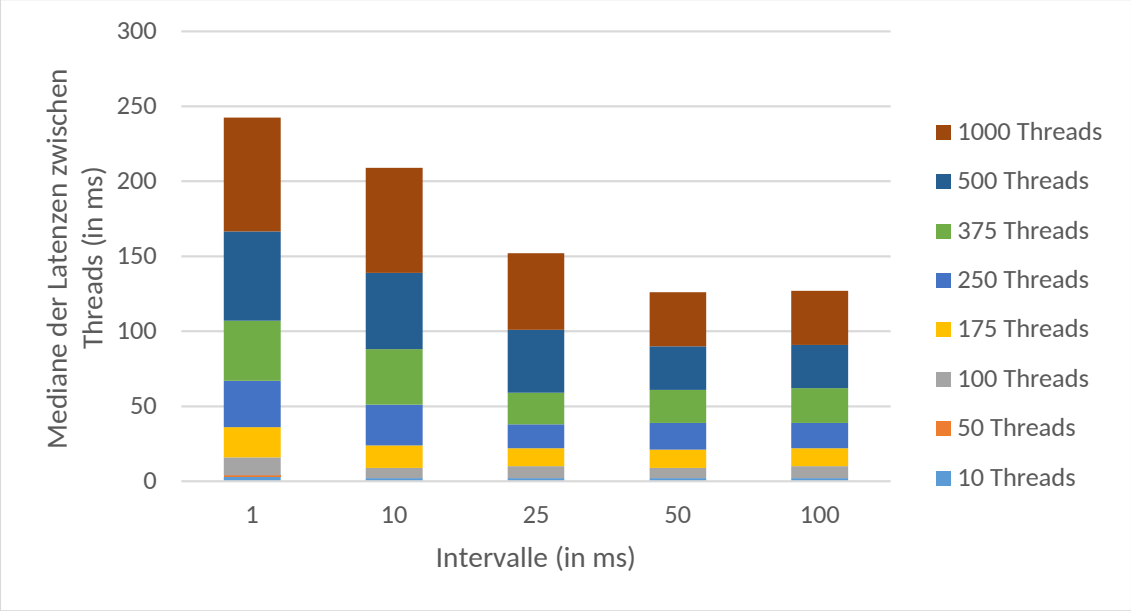

Abbildung 6.3: Kumulative Latenzen der Timer-Monitor-Implementierung in Java

Intervall	1 ms	10 ms	25 ms	50 ms	100 ms
Median	7.95	11.6 ms	5.47 ms	3.41 ms	3.24 ms

Tabelle 6.1: Mediane aller Standardabweichungen der Latenzen der Timer-Monitor-Implementierung in Java

auf die Millisekunde synchron betreiben. Jedoch zeigt Abbildung 6.1, dass es bei 500 und 1000 Threads auch Abweichungen der Latenzen gab, welche im dreistelligen Millisekundenbereich liegen und damit sehr groß sind. Bis einschließlich 375 Threads waren die Standardabweichungen der Latenzen für alle Intervalle 0. Dies lässt vermuten, dass mit dem Vierfachen an Prozessorkernen es möglich ist, bei der Simulation von 1000 Threads eine Median-Latenz von 0 Millisekunden und eine Standardabweichung der Latenzen von 0 Millisekunden möglich ist. Außerdem besitzt die Implementierung noch Optimierungspotenzial. Die derzeitige Implementierung ist Busy-Waiting. Jeder Thread prüft, ob er die Uhrzeit ausgeben soll.

6.2 Timer, Monitor und Threads in Java

Diese Implementierung wurde als Java Archive (JAR) verpackt und dann in der Konsole ausgeführt. Es wurden 100 Tests mit jeweils 10 Schleifendurchläufen durchgeführt. Es wurde mit folgenden Threadanzahlen getestet: 10, 50, 100, 175, 250, 375, 500 und 1000. Jede Threadanzahl wurde mit folgenden Intervallen (in Millisekunden) getestet: 1, 10,

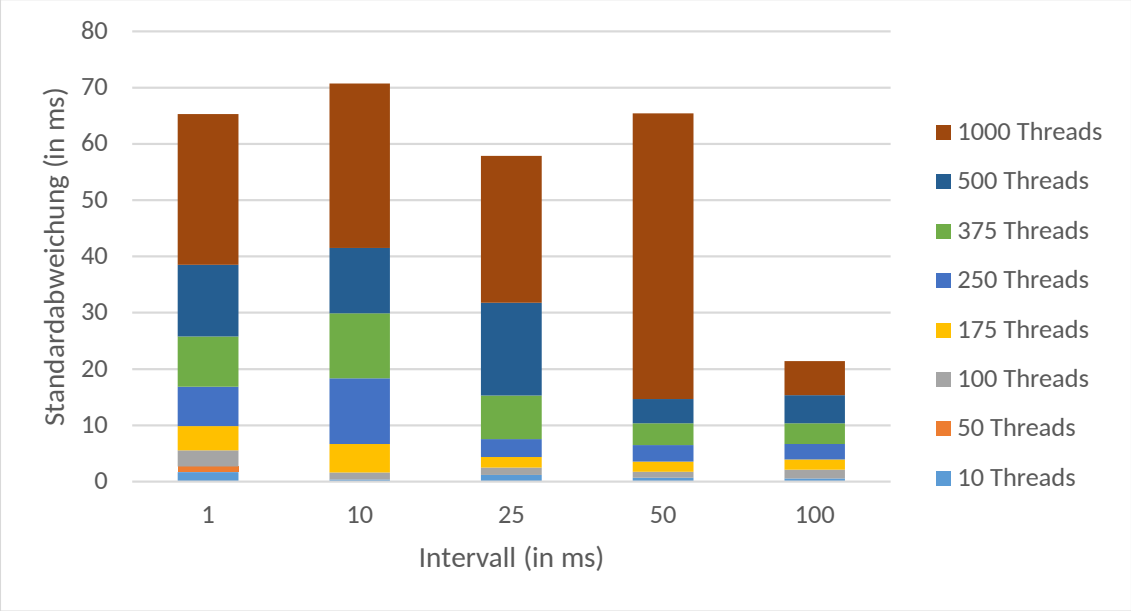

Abbildung 6.4: Kumulative Standardabweichungen der Timer-Monitor-Implementierung in Java

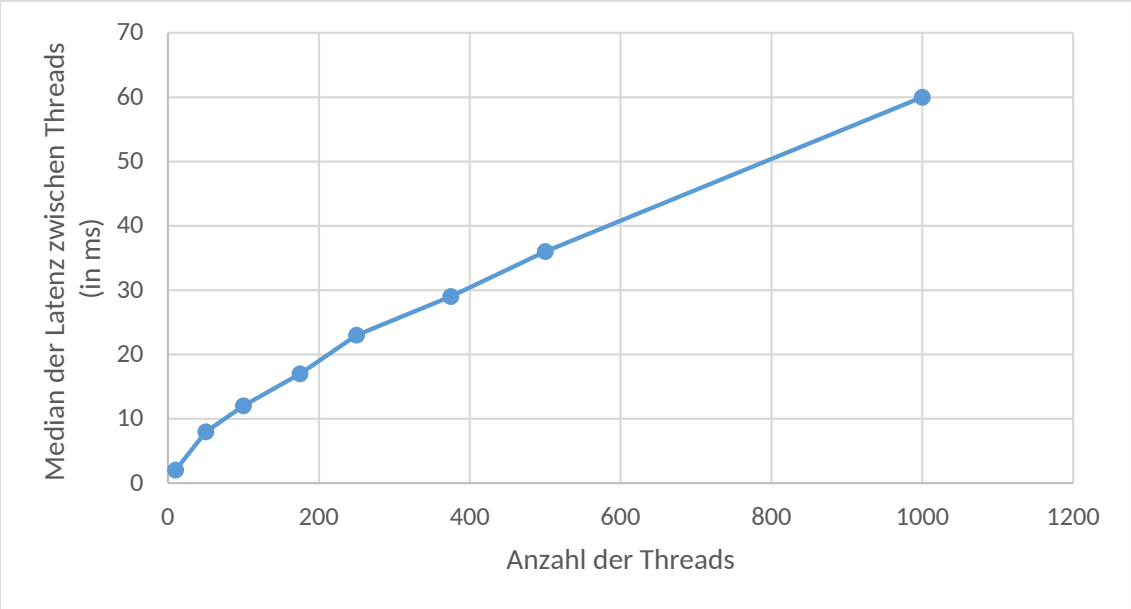

Abbildung 6.5: Medianlatenzen der Timer-Monitor-Implementierung in Java bei Intervall von 100 Millisekunden

Intervall	1 ms	10 ms	25 ms	50 ms	100 ms
Median	2.99 ms	3.18 ms	3.07 ms	2.26 ms	2.35 ms

Tabelle 6.2: Mediane aller Standardabweichungen der Latenzen der Timer-Monitor-Implementierung in Java auf Echtzeit-Betriebssystemkern

25, 50 und 100. Die Evaluation dieser Implementierung wurde auf dem Standard-Kernel ausgeführt.

In Abbildung 6.3 sind die aufsummierten Mediane der Worst-Case-Latenzen zwischen Threads zu sehen. Hier ist das Intervall von 100 Millisekunden am besten, wobei das Intervall von 50 Millisekunden nah dran ist. Weiterhin sind in Abbildung 6.4 die aufsummierten Standardabweichungen der Latenzen zwischen Threads zu sehen. Das Intervall von 100 Millisekunden schneidet dabei am besten ab. Zusätzlich wurde der Median aller Standardabweichungen der verschiedenen Intervalle berechnet. Die Ergebnisse sind in Tabelle 6.1 zu sehen.

Da das Intervall von 100 Millisekunden mit 3.24 Millisekunden den niedrigsten Median aller Standardabweichungen hat, wird es als „bestes“ Intervall zur Wertegenerierung gesehen, da die aufgenommenen Werte am robustesten und somit am aussagekräftigsten sein sollten. Es macht auch Sinn, dass die Performanz bei einem Intervall von 100 Millisekunden besser ist, da die Threads nicht so oft laufen und CPU-Last beanspruchen. Bei den kurzen Intervallen kommt es zu einem ähnlichen Verhalten wie bei *Busy-Waiting*. In Abbildung 6.5 wird das Wachstum der Latenzen für verschiedene Threadzahlen gezeigt. Für 1000 Threads beträgt der Median der Latenzen zwischen Threads 60 Millisekunden und für 500 Threads beträgt er 36 Millisekunden. Die Messungen lassen einen linearen Trend vermuten. Mithilfe von linearer Regression lässt sich die Latenz Y zwischen Threads basierend auf X Threads angeben als: $Y = 0.056 * X + 6.04$. Für ein angeschlossenes CEP-System wie Esper könnte man für 1000 Threads ein Zeitfenster von 92.4 Millisekunden (60 + 10 * 3.24) wählen. Zu der gemessenen Latenz für 1000 Threads wird das zehnfache der Standardabweichung aufaddiert. Diese große Sicherheitsmarge sollte garantieren, dass die erzeugten Werte von Esper in jedem Schleifendurchlauf angenommen werden.

Außerdem wurde dieselbe Implementierung auch noch auf dem Echtzeit-Betriebssystemkern getestet. Es wurden wieder 100 Tests mit jeweils 10 Schleifendurchläufen durchgeführt. Auch wurde wieder mit folgenden Threadanzahlen getestet: 10, 50, 100, 175, 250, 375, 500 und 1000. Die getesteten Intervalle (in Millisekunden) blieben auch gleich: 1, 10, 25, 50 und 100.

In Abbildung 6.6 sind die aufsummierten Mediane der Worst-Case-Latenzen zwischen Threads zu sehen. Hier ist das Intervall von 25 Millisekunden am besten, wobei das Intervall von 50 Millisekunden nah dran ist. Weiterhin sind in Abbildung 6.7 die aufsummierten Standardabweichungen der Latenzen zwischen Threads zu sehen. Das Intervall von 100 Millisekunden schneidet dabei am besten ab. Zusätzlich wurde der Median aller

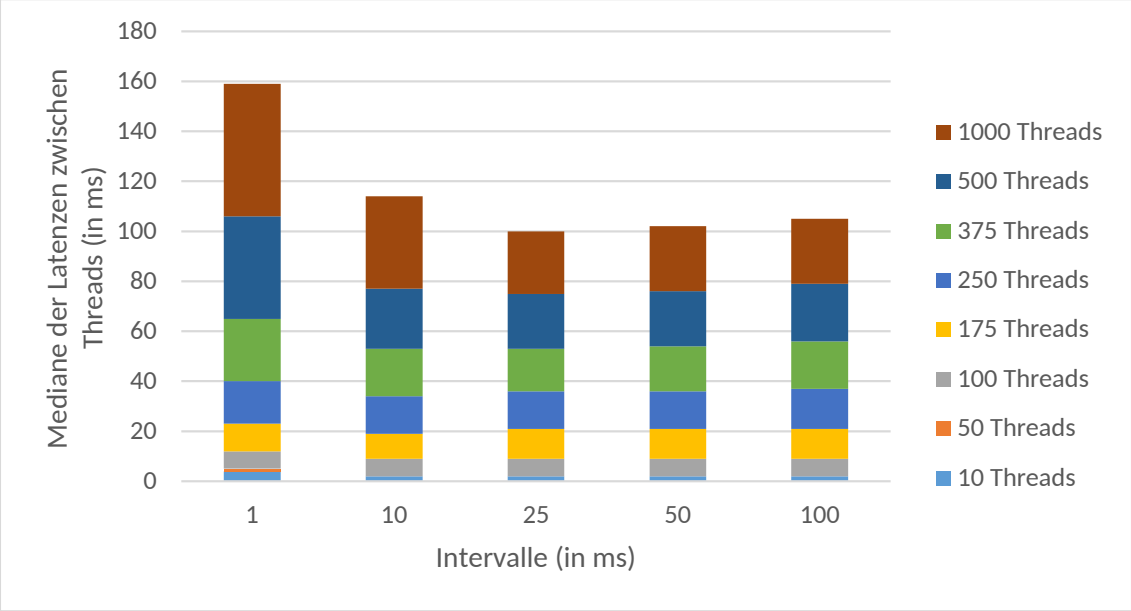

Abbildung 6.6: kumulierte Latenzen der Timer-Monitor-Implementierung in Java auf Echtzeit-Betriebssystemkern

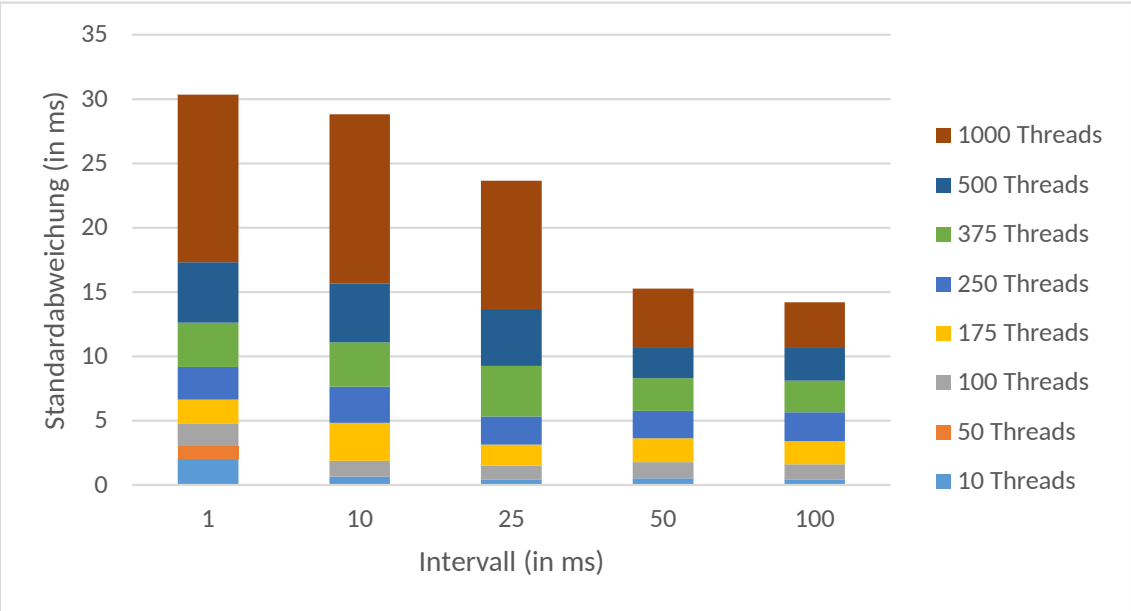

Abbildung 6.7: kumulierte Standardabweichungen der Latenzen der Timer-Monitor-Implementierung in Java auf Echtzeit-Betriebssystemkern

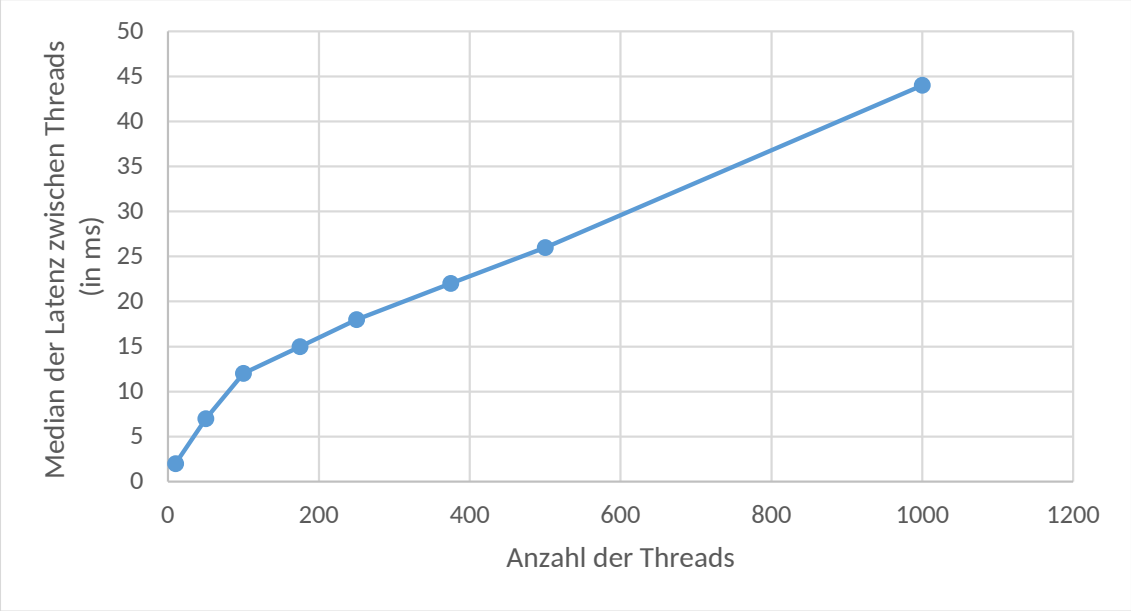

Abbildung 6.8: Medianlatenzen der Timer-Monitor-Implementierung in Java auf Echtzeit-Betriebssystemkern bei 50 Millisekunden Intervall

Standardabweichungen der verschiedenen Intervalle berechnet. Die Ergebnisse sind in Tabelle 6.2 zu sehen.

Da das Intervall von 50 Millisekunden mit 2.26 Millisekunden den niedrigsten Median aller Standardabweichungen hat, wird es als „bestes“ Intervall zur Wertegenerierung gesehen, da die aufgenommenen Werte am robustesten und somit am aussagekräftigsten sein sollten. In Abbildung 6.8 wird das Wachstum der Latenzen für verschiedene Threadzahlen bei einem Intervall von 50 Millisekunden gezeigt. Für 1000 Threads beträgt der Median der Latenzen zwischen Threads 44 Millisekunden und für 500 Threads beträgt er 26 Millisekunden. Die Messungen lassen einen linearen Trend vermuten. Mithilfe von linearer Regression lässt sich die Latenz Y zwischen Threads basierend auf X Threads angeben als: $Y = 0.039 * X + 6.17$. Für ein angeschlossenes CEP-System wie Esper könnte man für 1000 Threads ein Zeitfenster von 66.6 Millisekunden (44 + 10 * 2.26) wählen. Zu der gemessenen Latenz für 1000 Threads wird das Zehnfache der Standardabweichung aufaddiert. Diese große Sicherheitsmarge sollte garantieren, dass die erzeugten Werte von Esper in jedem Schleifendurchlauf angenommen werden.

6.3 Esper und Java

Diese Implementierung wurde als JAR verpackt und dann in der Konsole ausgeführt. Es wurden 10 Tests mit jeweils 10 Schleifendurchläufen durchgeführt. Es wurde mit folgenden Threadanzahlen getestet: 10, 50, 100, 175, 250, 375, 500 und 1000. Jede Threadanzahl wurde mit folgenden Intervallen (in Millisekunden) getestet: 1, 10, 25, 50 und 100. Bei

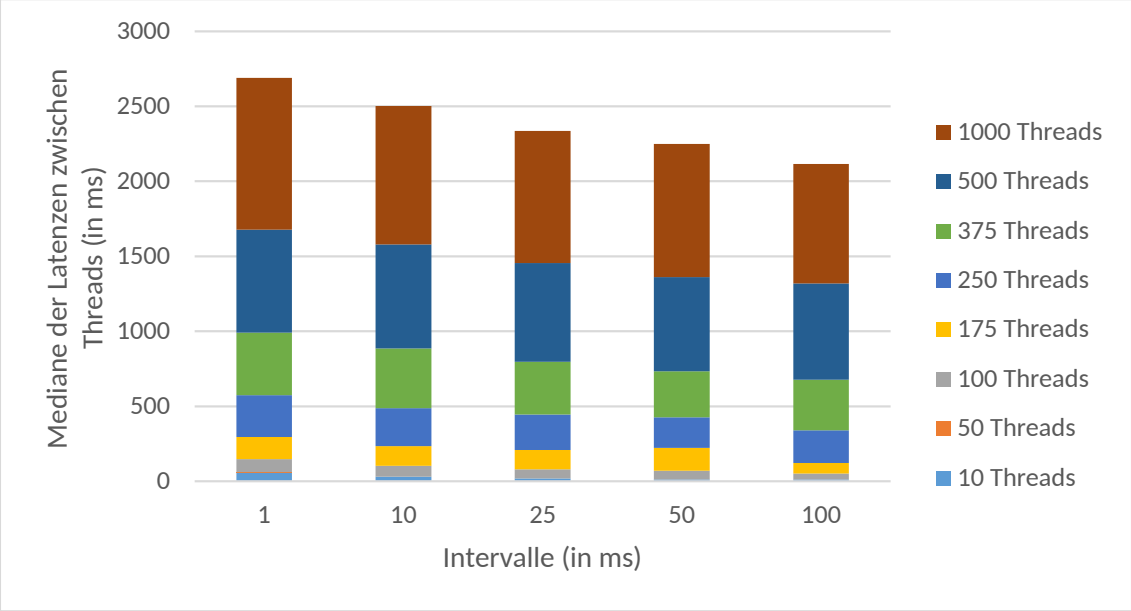

Abbildung 6.9: Kumulative Latenzen der Java-Esper-Implementierung

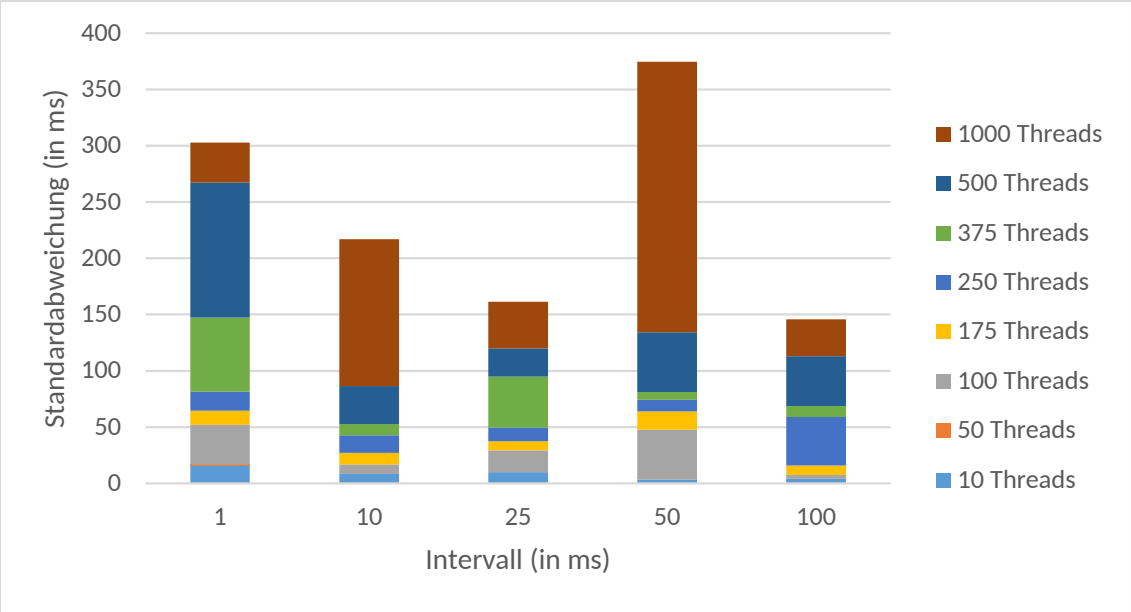

Abbildung 6.10: Kumulative Standardabweichungen der Latenzen der Java-Esper-Implementierung

Intervall	1 ms	10 ms	25 ms	50 ms	100 ms
Median	35.63 ms	12.87 ms	22.2 ms	30.25 ms	21.38 ms

Tabelle 6.3: Mediane der Standardabweichungen der Latenzen der Java-Esper-Implementierung

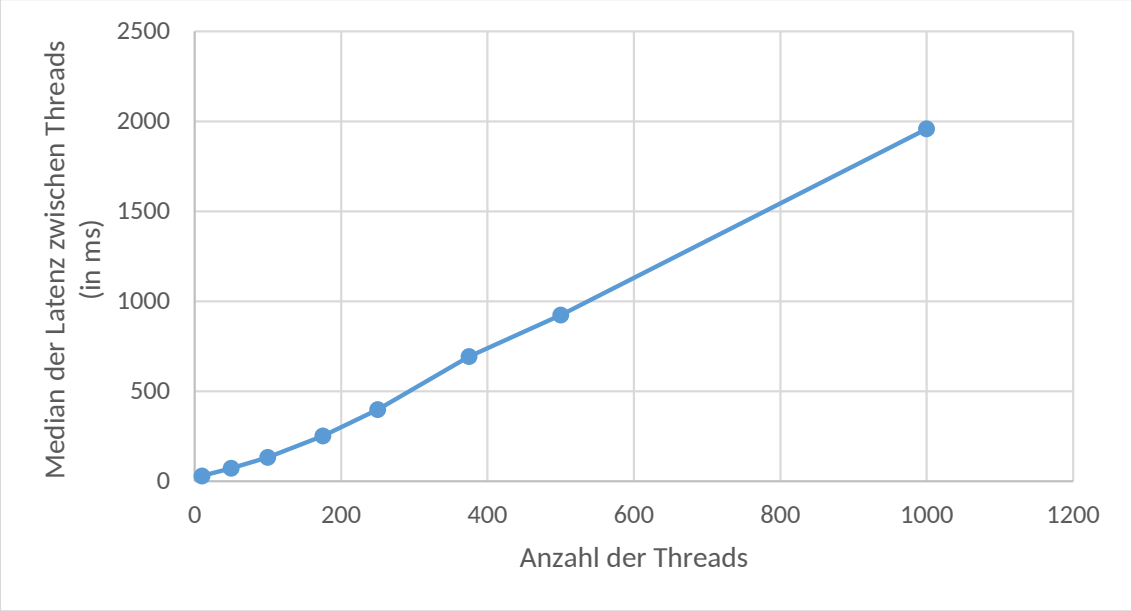

Abbildung 6.11: Medianlatenzen der Java-Esper-Implementierung bei Intervall von 10 Millisekunden

dieser Implementierung waren die Laufzeiten so hoch, dass mehr Testdurchläufe zeitlich nicht realisierbar gewesen wären. Die Evaluation dieser Implementierung wurde auf dem Standard-Kernel ausgeführt.

In Abbildung 6.9 sind die aufsummierten Mediane der Worst-Case-Latenzen zwischen Threads zu sehen. Das Intervall von 100 Millisekunden schneidet dabei am besten ab. Weiterhin sind in Abbildung 6.10 die aufsummierten Standardabweichungen der Latenzen zwischen Threads zu sehen. Auch hier ist das Intervall von 100 Millisekunden am besten. Zusätzlich wurde der Median aller Standardabweichungen der verschiedenen Intervalle genommen. Die Ergebnisse sind in Tabelle 6.3 zu sehen.

Da das Intervall von 10 Millisekunden mit 12.87 Millisekunden den niedrigsten Median aller Standardabweichungen hat, wird es als „bestes“ Intervall zur Wertegenerierung gesehen, da die aufgenommenen Werte am aussagekräftigsten sein sollten. In Abbildung 6.11 wird das Wachstum der Latenzen für verschiedene Threadzahlen gezeigt. Für 1000 Threads beträgt der Median der Latenzen zwischen Threads 1958 Millisekunden. Die Messungen lassen einen linearen Trend vermuten. Mithilfe von linearer Regression lässt sich die Latenz Y zwischen Threads basierend auf X Threads angeben als: $Y = 1.99 * X - 53.03$.

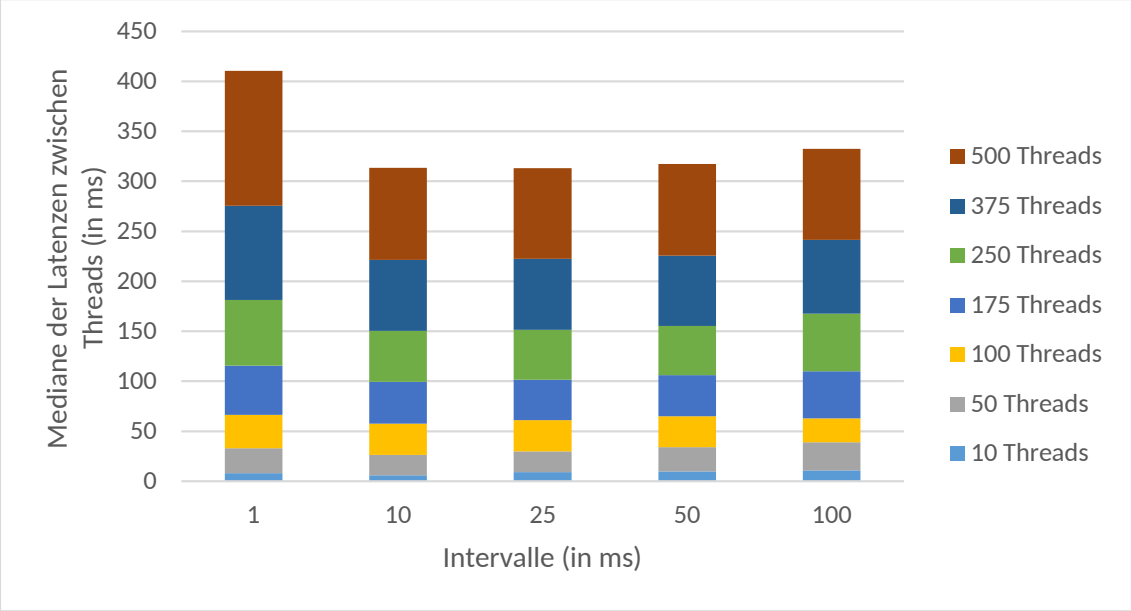

Abbildung 6.12: Kumulative Latenzen der Timer-Monitor-Implementierung in Echtzeit-Java

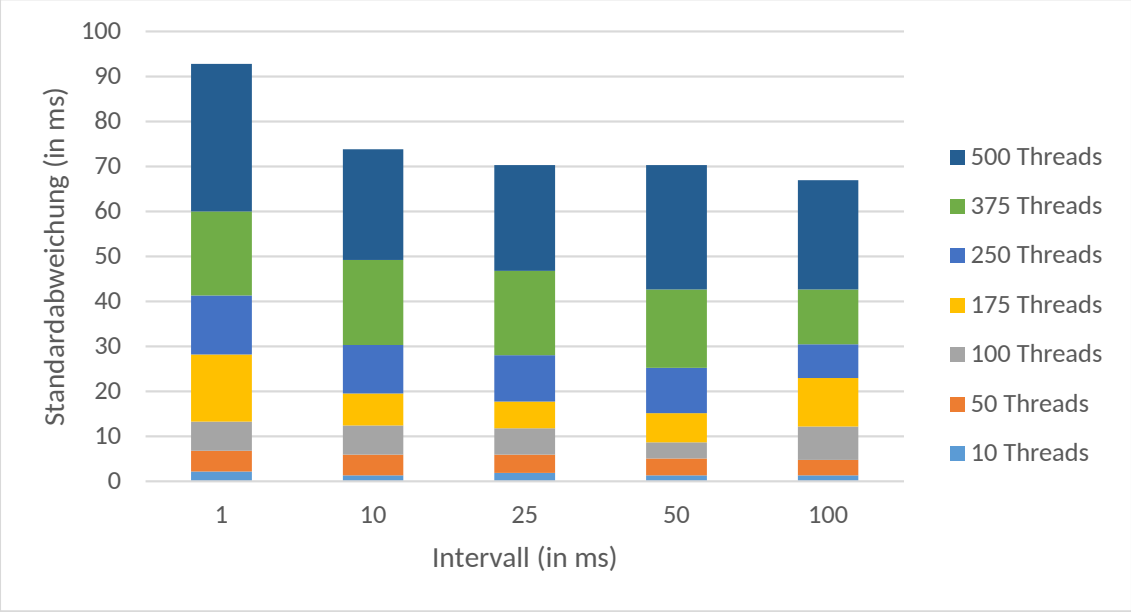

Abbildung 6.13: Kumulative Standardabweichungen der Latenzen der Timer-Monitor-Implementierung in Echtzeit-Java

Intervall	1 ms	10 ms	25 ms	50 ms	100 ms
Median	13.14	7.14 ms	5.98 ms	6.46 ms	7.51 ms

Tabelle 6.4: Mediane aller Standardabweichungen der Latenzen der Timer-Monitor-Implementierung in Echtzeit-Java

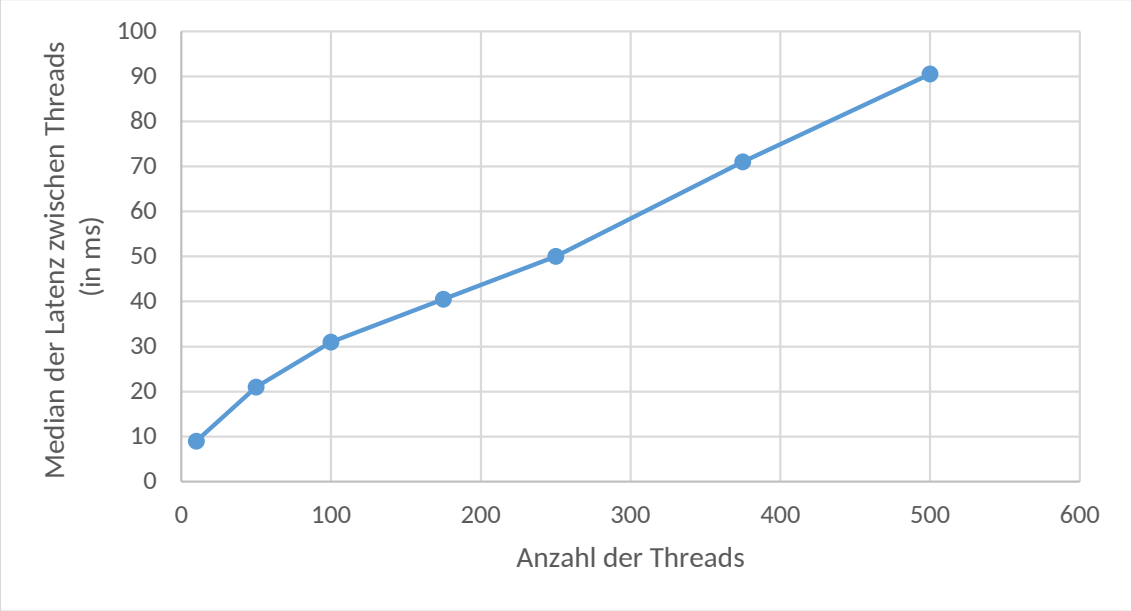

Abbildung 6.14: Medianlatenzen der Timer-Monitor-Implementierung in Echtzeit-Java bei Intervall von 25 Millisekunden

6.4 Timer, Monitor und Threads in Echtzeit-Java

Die Timer-Monitor-Implementierung in Echtzeit-Java wurde mit dem *Jamaica Builder* zu einer ausführbaren Datei kompiliert. Es wurden 100 Tests mit jeweils 10 Schleifendurchläufen durchgeführt. Es wurde mit folgenden Threadanzahlen getestet: 10, 50, 100, 175, 250, 375, 500 und 1000. Jede Threadanzahl wurde mit folgenden Intervallen (in Millisekunden) getestet: 1, 10, 25, 50 und 100. Die Evaluation dieser Implementierung wurde auf dem Echtzeit-Kernel ausgeführt. Es war nicht möglich, mit 1000 Threads zu testen, da der Jamaica Builder dies in der Standardkonfiguration nicht zulässt. In Abbildung 6.12 sind die aufsummierten Mediane der Worst-Case-Latenzen zwischen Threads zu sehen. Das Intervall von 25 Millisekunden hat dabei die niedrigste kumulative Latenz, dicht gefolgt vom Intervall mit 10 Millisekunden. Weiterhin sind in Abbildung 6.13 die aufsummierten Standardabweichungen der Latenzen zwischen Threads zu sehen. Das Intervall von 100 Millisekunden schneidet hier am besten ab. Zusätzlich wurde der Median aller Standardabweichungen der verschiedenen Intervalle genommen. Die Ergebnisse sind in Tabelle 6.4 zu sehen.

Da das Intervall von 25 Millisekunden mit 5.98 Millisekunden den niedrigsten Median aller Standardabweichungen hat, wird es als „bestes“ Intervall zur Wertegenerierung gesehen, da die aufgenommenen Werte am aussagekräftigsten sein sollten. In Abbildung 6.14 wird das Wachstum der Latenzen für verschiedene Threadzahlen gezeigt. Für 500 Threads beträgt der Median der Latenzen zwischen Threads 90.5 Millisekunden. Die Messungen lassen einen linearen Trend vermuten. Mithilfe von linearer Regression lässt sich die Latenz Y zwischen Threads basierend auf X Threads angeben als: $Y = 0.15 * X + 11.64$.

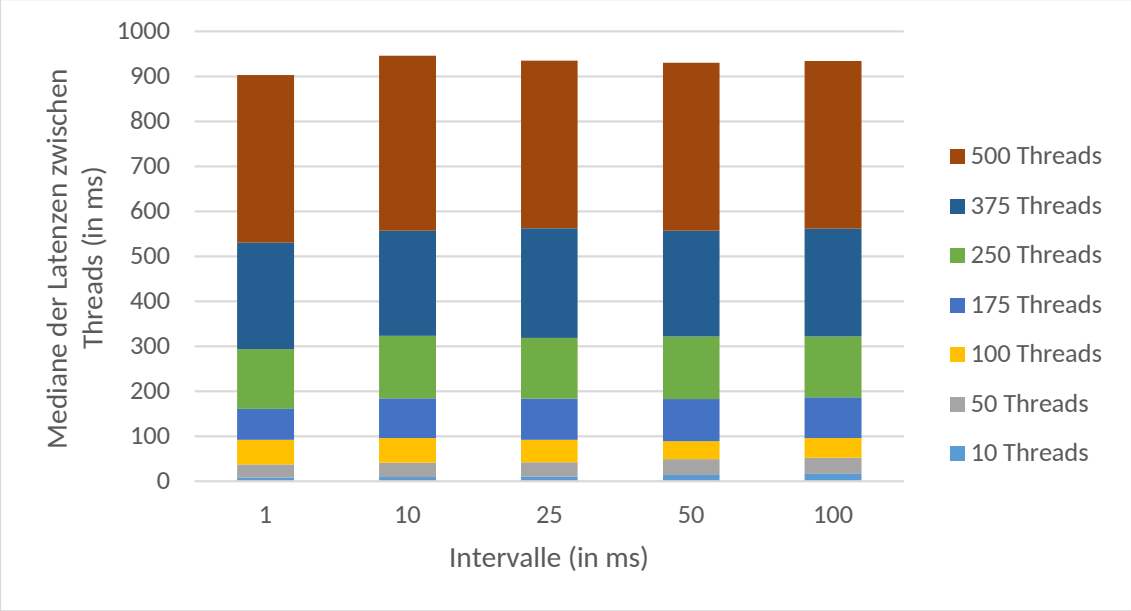

Abbildung 6.15: Kumulative Latenzen der Threads mit integrierten Timern in Echtzeit-Java

Intervall	1 ms	10 ms	25 ms	50 ms	100 ms
Median	25.2 ms	36.89 ms	40.64 ms	47.95 ms	26.73 ms

Tabelle 6.5: Mediane aller Standardabweichungen der Threads mit integrierten Timern in Echtzeit-Java für die verschiedenen Intervalle

Unter Benutzung dieser Formel liegt die geschätzte Latenz für 1000 Threads bei 162 Millisekunden.

6.5 Threads mit integrierten Timern in Echtzeit-Java

Die Implementierung der Threads mit integrierten Timern in Realtime Java wurde mit dem *Jamaica Builder* zu einer ausführbaren Datei kompiliert. Es wurden 100 Tests mit jeweils 10 Schleifendurchläufen durchgeführt. Es wurde mit folgenden Threadanzahlen getestet: 10, 50, 100, 175, 250, 375, 500 und 1000. Jede Threadanzahl wurde mit folgenden Intervallen (in Millisekunden) getestet: 1, 10, 25, 50 und 100. Die Evaluation dieser Implementierung wurde auf dem Echtzeit-Kernel ausgeführt. Wie im vorherigen Kapitel erwähnt, war es nicht möglich, mit 1000 Threads zu testen. In Abbildung 6.15 sind die aufsummierten Mediane der Latenzen zwischen Threads zu sehen. Das Intervall von 1 Millisekunde hat dabei die niedrigste kumulative Latenz. Weiterhin sind in Abbildung 6.16 die aufsummierten Standardabweichungen der Latenzen zwischen Threads zu sehen. Das Intervall von 100 Millisekunden schneidet hier am besten ab, wobei das Intervall von

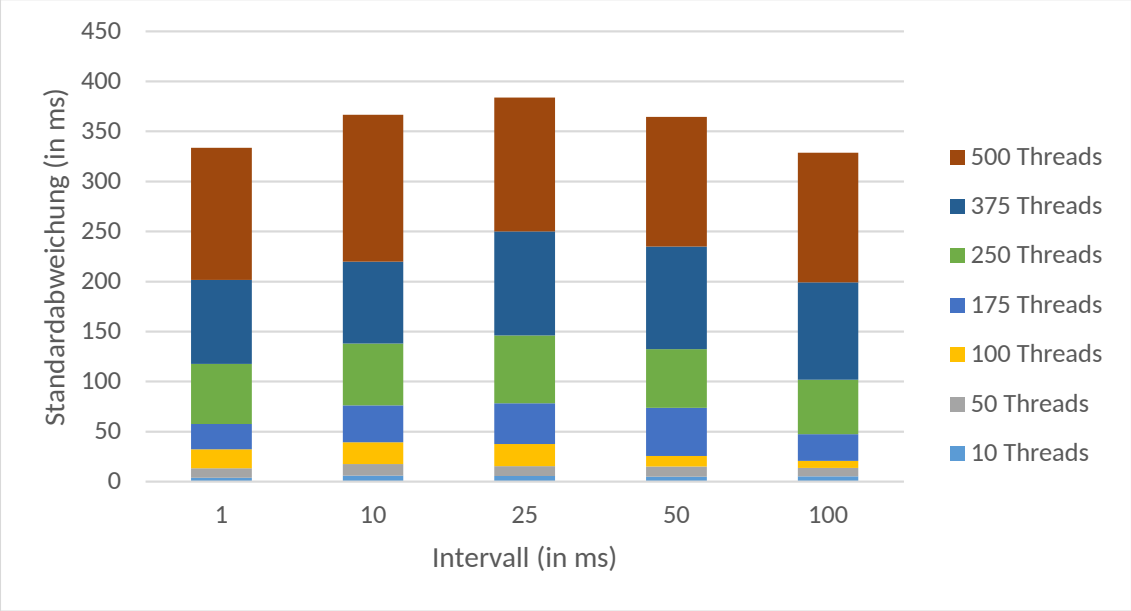

Abbildung 6.16: Kumulative Standardabweichungen der Latenzen der Threads mit integrierten Timern in Echtzeit-Java

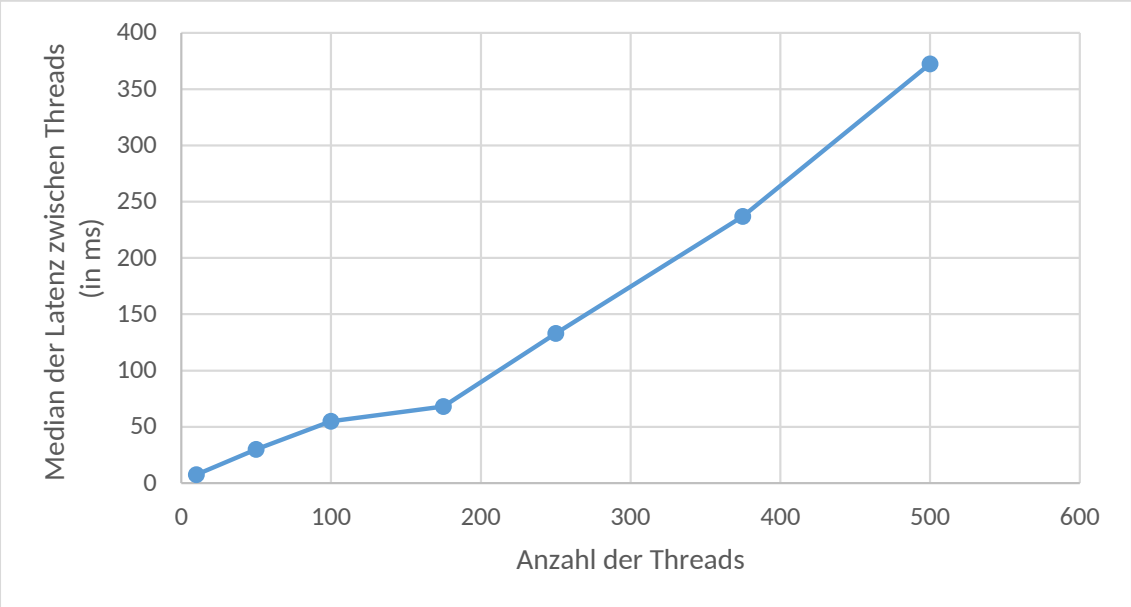

Abbildung 6.17: Medianlatenzen der Threads mit integrierten Timern in Echtzeit-Java bei Intervall von 1 Millisekunden

1 Millisekunde ziemlich nah dran ist. Zusätzlich wurde der Median aller Standardabweichungen der verschiedenen Intervalle genommen. Die Ergebnisse sind in Tabelle 6.5 zu sehen.

Da das Intervall von 1 Millisekunde mit 25.2 Millisekunden den niedrigsten Median aller Standardabweichungen hat, wird es als „bestes“ Intervall zur Wertegenerierung gesehen, da die aufgenommenen Werte am aussagekräftigsten sein sollten. In Abbildung 6.17 wird das Wachstum der Latenzen für verschiedenen Threadzahlen gezeigt. Für 500 Threads beträgt der Median der Latenzen zwischen Threads 372.5 Millisekunden. Der Graph lässt einen linearen Trend erkennen. Mithilfe von linearer Regression lässt sich die Latenz Y zwischen Threads basierend auf X Threads angeben als: $Y = 0.72 * X - 22.14$. Für 1000 Threads liegt die geschätzte Latenz bei 697.86 Millisekunden.

6.6 Diskussion

Bei fast allen Implementierungen ist eine Korrelation zwischen aufsteigendem Intervall und absteigender kumulativer Latenz zu beobachten. Zusätzlich zeigen fast alle Implementierungen einen Ausreißer bei den Standardabweichungen der Latenzen beim Intervall mit 50 Millisekunden Länge ab einer Threadanzahl von 500 oder 1000. Es ist nicht klar, warum es dieses lokale Maximum bei 50 Millisekunden gibt. Bei der C-Implementierung und den Threads mit integrierten Timern in Echtzeit-Java, zwei Implementierungen bei denen die Threads sich selbst synchronisieren, ist bei 1 Millisekunde der Median der Standardabweichungen der Latenzen am niedrigsten. Bei den Timer-Monitor-Implementierungen verbessert sich der Median der Standardabweichungen der Latenzen erst bei höheren Intervallen. Hier gab es keine Gemeinsamkeiten. Außerdem deuten fast alle Implementierungen auf einen linearen Zusammenhang zwischen Threadanzahl und Latenz zwischen Threads. Einzige Ausnahme ist die C-Implementierung, die sehr performant ist. Es ist jedoch zu vermuten, dass sich das C-Programm ab einer bestimmten Anzahl von Threads analog verhält.

Verglichen zur Timer-Monitor-Implementierung auf dem Echtzeit-Betriebssystemkern gibt es bei der Implementierung auf dem normalen Linux-Betriebssystemkern ein Plus von 36 Prozent bei der besten Median-Latenz für 1000 Threads und ein Plus von 43 Prozent bei der besten Median-Standardabweichung der Latenzen. Dies macht Sinn, da auf dem Echtzeit-Betriebssystemkern hoch priorisierte Ausführungen gar nicht oder nicht so oft von unwichtigen Prozessen unterbrochen werden.

Im Vergleich der Timer-Monitor-Implementierung auf dem normalen Betriebssystemkern und der Java-Esper-Implementierung gibt es bei dieser ein Plus von 3163 Prozent bei der besten Median-Latenz für 1000 Threads. Der kleinste Median der Standardabweichung der Latenzen ist im Vergleich zur reinen Java-Implementierung um 297 Prozent gestiegen. Dieser Anstieg ist nicht überraschend wenn man bedenkt, dass nach der Werteerzeugung noch die Verarbeitung in Esper ablaufen muss.

Verglichen zur Timer-Monitor-Implementierung in Java auf dem Echtzeit-Betriebssystemkern gibt es bei der Timer-Monitor-Implementierung in Echtzeit-Java ein Plus von 248 Prozent bei der besten Median-Latenz bei 500 Threads und einem Plus von 164 Prozent bei der besten Median-Standardabweichung der Latenzen. Dies ist überraschend, da die Echtzeitversion von Java strengere Garantien geben müsste und die Kompilierung auch noch Performanzgewinne liefern sollte. Es kann jedoch daran liegen, dass nur die Standardkonfiguration der aicas-Werkzeuge benutzt wurde und auch kein Profil der Ausführung erstellt wurde, welches die Performanz noch weiter optimieren soll. Trotzdem kann es sich lohnen, diese Echtzeit-Implementierung zu benutzen, da durch die *JamaicaVM* mehr Echtzeit-Garantien geboten werden und somit eine verlässlichere, auf einem höheren Niveau reproduzierbare, Ausführung versprochen wird.

Verglichen zu der Timer-Monitor-Implementierung in Echtzeit-Java gibt es bei den Threads mit integrierten Timern beim besten Median der Latenz ein Plus von 311 Prozent und beim besten Median der Standardabweichung der Latenzen ein Plus von 321 Prozent. Anhand von diesen Werten lässt sich zeigen, dass die Threads mit integrierten Timern wesentlich ineffizienter sind. Dies ist nicht überraschend, da die Threads sich um ihre Zeitwerte kümmern und die gemeinsame Ressource *Timer* von diesen mehreren Threads angefragt wird und es somit zu größeren Verzögerungen kommt.

Den Vergleichen nach, scheint die Timer-Monitor-Implementierung in „normalem“ Java auf einem Echtzeit-Betriebssystemkern die performanteste zu sein und sich am besten für die Wertegenerierung für CEP-Systeme zu eignen.

7 Zusammenfassung und Ausblick

Immer mehr Haushaltsgeräte und Maschinen werden mit dem Internet verbunden. Diesen Trend nennt man das Internet der Dinge (Englisch: Internet Of Things, kurz: IoT). Ein zentrales Problem dabei ist die Simulation von IoT-Umgebungen. Diese ist sehr teuer, da man für jede Simulation Hardware bereitstellen muss. Daher ist eine Software-basierte Simulation erstrebenswert. Ein weiteres Problem im Internet der Dinge ist die automatische Überwachung von IoT-Umgebungen und das Reagieren auf Situationen. Dies geschieht meist mithilfe von Complex-Event-Processing-Systemen (kurz: CEP-Systeme). Diese bekommen einen Strom an Ereignissen, lesen diesen ein und treffen, basierend auf den Ereignisdaten, Entscheidungen. Für CEP-Systeme gibt es bislang noch keine Benchmarks. Dieses Problem soll durch andere Arbeiten gelöst werden. In dieser Arbeit wurde deswegen auch versucht, eine Datengenerierung für Benchmarks zu entwickeln. Zusätzlich wurde erforscht, wie viele Threads synchron Daten erzeugen können und wie groß die maximale zeitliche Abweichung bei asynchronen Threads ist.

Das Simulationswerkzeug besteht aus einem React Frontend und einem Spring Boot Backend. Es erlaubt die Angabe von: Datentyp, Startwert, Änderungsrate, Ausreißerwahrscheinlichkeit, Intervalllänge und der maximalen positivn und negativen Abweichung. Es können beliebig viele Sensorsimuationen erstellt werden mit unterschiedlichen Parametern. Bei der Simulation werden Werte über MQTT ausgegeben und an Topics mit dem Sensornamen verschickt. Im Frontend wird eine Liste der Sensorsimulationen und eine Konsole mit den derzeit generierten Werten angezeigt. Das Backend verarbeitet die HTTP-Anfragen des Frontend und verwaltet die Sensorsimulationen und deren Wertegenerierung, sowie die Erzeugung der MQTT-Nachrichten.

Es wurden verschiedene Implementierungen vorgestellt. Die erste Implementierung ist eine Abwandlung des Simulationswerkzeuges, wobei die Threads sich zusammen durch Busy Waiting synchronisieren. Als nächstes wurde ein analoges Programm in C vorgestellt, welches über das Java Native Interface aufgerufen wird. Bei dem C-Programm wurden auch unterschiedliche Scheduling-Verfahren ausprobiert, unter anderem Deadline-Scheduling, welches die besten Resultate liefert. Beim Deadline-Scheduling wird jedem Thread pro Periode eine bestimmte, maximale Laufzeit zugewiesen. Der Vergleich der Zeiten lief über das Vergleichen von Dateien mit den Zeitwerten. Das Arbeiten mit Dateien hat jedoch einen großen Zusatzaufwand, weshalb als nächstes untersucht wurde, ob sich mit Kommunikation über TCP eine bessere Performanz erreichen lässt. Nach diesen Tests hatte es sich gezeigt, dass es mit Java auf der gegebenen Hardware nicht möglich ist, 1000 Threads mit einem Minimalintervall von 1 Millisekunde zu synchronisieren. Deswegen wurde eine weitere reine C-Implementierung vorgenommen, bei der

die Zeitwerte in Arrays geschrieben und am Ende miteinander verglichen werden. Für die Java-Implementierungen wurde die Zielsetzung aufgelockert und es wurde stattdessen untersucht, wie groß die maximale zeitliche Abweichung zwischen zwei Threads ist. Diese Umstellung war möglich, da Esper es erlaubt, dass man ein Zeitfenster definiert, in dem Werte berücksichtigt werden. Als nächstes wurde eine Implementierung in Java vorgestellt, welche einen Monitor mit Timer beinhaltet, welcher Threads synchronisiert. Außerdem wurde mit *JamaicaVM*, einer virtuellen Maschine welche die Echtzeitspezifikation von Java implementiert, experimentiert. Hier war es möglich, Echtzeit-Threads und Ahead-Of-Time-Kompilierung zu benutzen.

Zum Abschluss, wurden die verschiedenen Implementierungen evaluiert. Die C-Implementierung wurde als Maßstab genommen, da sie die untere Schranke für die Performanz ist. Hier war es möglich, 1000 Threads synchron laufen zu lassen. Bei den Java-Implementierungen war die Timer-Monitor-Implementierung auf dem Echtzeit-Betriebssystemkern die beste mit einer Medianlatenz von 44 Millisekunden bei 1000 Threads. Verglichen dazu hat die Timer-Monitor-Implementierung in Echtzeit-Java ein Plus von 248 Prozent bei der besten Median-Latenz bei 500 Threads und einem Plus von 164 Prozent bei der besten Median-Standardabweichung der Latenzen. Dies ist überraschend, da die Echtzeitversion von Java strengere Garantien geben müsste und die Ahead-Of-Time-Kompilierung auch noch Performanzgewinne liefern sollte.

Basierend auf dieser Arbeit ist es möglich, Sensoren zu simulieren und die simulierten Sensoren an Monitoring-Tools oder CEP-Systeme anzuschließen. Außerdem wird geschätzt, dass bei der Wertegenerierung in Java für CEP-Systeme die maximale Verzögerung für 1000 Threads bei 44 Millisekunden liegt. Mit dem Zehnfachen des Median der Standardabweichung addiert, sollte es für Benchmarks ausreichend sein, das Zeitfenster in Esper auf 66.6 Millisekunden festzulegen.

Ausblick

Die Simulation ist noch sehr simpel. In Zukunft könnten noch komplexere Simulationsmodelle benutzt werden. Zum Beispiel, zur Simulation von Temperatursensoren gibt es bessere Modelle, die auf Thermodynamik beruhen und realistischere Simulationen ermöglichen. Das gilt analog für Geräusch- und Lichtsensoren. Man könnte spezialisierte Unterklassen der Sensorsimulation erstellen, die besser angepasst sind an die Art der Sensoren, die man simulieren möchte.

Bei der Wertegenerierung für CEP-Systeme gibt es noch folgende Herausforderungen für zukünftige Arbeiten. Das Werkzeug muss in die Benchmarking-Systeme integriert werden. Zukünftig sollen verschiedene CEP-Systeme unterstütz werden. Weiterhin müssen Adapter implementiert werden und es muss sichergestellt werden, dass es keine Zeitverluste bei den verschiedenen Verarbeitungsschritten gibt.

Literaturverzeichnis

[15] *Trends in the cost of computing*. https://aiimpacts.org/trends-in-the-cost-of-computing/. 2015 (zitiert auf S. 13).

[17a] https://nest.com/. 2017 (zitiert auf S. 13).

[17b] *Bosch Feuermelder*. https://www.bosch-smarthome.com/de/de/produkte/smart-system-solutions/rauchmelder?WT.mc_id=iot_hub. 2017 (zitiert auf S. 13).

[17c] *Bosch IoT Suite*. https://www.bosch-si.com/iot-platform/bosch-iot-suite/homepage-bosch-iot-suite.html. 2017 (zitiert auf S. 13).

[17d] *Nest Cam IQ Outdoor*. https://nest.com/cameras/nest-cam-iq-outdoor/overview/. 2017 (zitiert auf S. 13).

[BW01] A. Burns, A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001 (zitiert auf S. 16).

[CCM+01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana et al. *Web services description language (WSDL) 1.1*. 2001 (zitiert auf S. 19).

[CKE+15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. „Apache flink: Stream and batch processing in a single engine“. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (zitiert auf S. 22).

[DG08] J. Dean, S. Ghemawat. „MapReduce: simplified data processing on large clusters“. In: *Communications of the ACM* 51.1 (2008), S. 107–113 (zitiert auf S. 18).

[DM02] E. D. Dolan, J. J. Moré. „Benchmarking optimization software with performance profiles“. In: *Mathematical programming* 91.2 (2002), S. 201–213 (zitiert auf S. 16).

[Fac17a] Facebook. *React License*. https://github.com/facebook/react/blob/master/LICENSE. Okt. 2017 (zitiert auf S. 24).

[Fac17b] Facebook. *React Open Source*. https://github.com/facebook/react. Okt. 2017 (zitiert auf S. 24).

[Gar13] N. Garg. *Apache Kafka*. Packt Publishing Ltd, 2013 (zitiert auf S. 21).

[GBB17] Z. Gao, C. Bird, E. T. Barr. „To type or not to type: quantifying detectable bugs in JavaScript“. In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press. 2017, S. 758–769 (zitiert auf S. 25).

[HBS+16] P. Hirmer, U. Breitenbücher, A. C. F. da Silva, K. Képes, B. Mitschang, M. Wieland. „Automating the Provisioning and Configuration of Devices in the Internet of Things“. Englisch. In: *Complex Systems Informatics and Modeling Quarterly* 9 (Dez. 2016), S. 28–43. ISSN: 2255 - 9922. DOI: 10.7250/csimq.2016-9.02. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2016-23&engl=0 (zitiert auf S. 13).

[HLC+14] S. N. Han, G. M. Lee, N. Crespi, N. Van Luong, K. Heo, M. Brut, P. Gatellier. „DPWSim: A simulation toolkit for IoT applications using devices profile for web services“. In: *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE. 2014, S. 544–547 (zitiert auf S. 19).

[HWBM16] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. „Dynamic Ontology-based Sensor Binding“. Englisch. In: *Advances in Databases and Information Systems. 20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31, 2016, Proceedings*. Bd. 9809. Information Systems and Applications, incl. Internet/Web, and HCI. Prague, Czech Republic: Springer International Publishing, Aug. 2016, S. 323–337. ISBN: 978-3-319-44039-2. DOI: 10.1007/978-3-319-44039-2. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-25&engl=0 (zitiert auf S. 13).

[icc17] icclab. *Iot Simulator*. http://icclab.github.io/iot-simulator/. Okt. 2017 (zitiert auf S. 19).

[IH11] T. Issariyakul, E. Hossain. *Introduction to network simulator NS2*. Springer Science & Business Media, 2011 (zitiert auf S. 19, 20).

[IS15] M. H. Iqbal, T. R. Soomro. „Big data analysis: Apache storm perspective“. In: *International journal of computer trends and technology* (2015), S. 9–14 (zitiert auf S. 21).

[JHD+04] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack et al. „The spring framework–reference documentation“. In: *Interface* 21 (2004) (zitiert auf S. 27).

[JMM11] C. Janiesch, M. Matzner, O. Müller. „A blueprint for event-driven business activity management“. In: *International Conference on Business Process Management*. Springer. 2011, S. 17–28 (zitiert auf S. 15).

[LLWC03] P. Levis, N. Lee, M. Welsh, D. Culler. „TOSSIM: Accurate and scalable simulation of entire TinyOS applications“. In: *Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM. 2003, S. 126–137 (zitiert auf S. 19).

[Loc10] D. Locke. „Mq telemetry transport (mqtt) v3. 1 protocol specification“. In: *IBM developerWorks Technical Library* (2010) (zitiert auf S. 15).

[MPD+02] G. Mein, S. Pal, G. Dhondu, T. K. Anand, A. Stojanovic, M. Al-Ghosein, P. M. Oeuvray. *Simple object access protocol*. US Patent 6,457,066. Sep. 2002 (zitiert auf S. 19).

[ODE+06] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, T. Voigt. „Cross-level sensor network simulation with cooja“. In: *Local computer networks, proceedings 2006 31st IEEE conference on*. IEEE. 2006, S. 641–648 (zitiert auf S. 19, 21).

[Rob10] D. Robins. „Complex event processing“. In: *Second International Workshop on Education Technology and Computer Science. Wuhan*. Citeseer. 2010, S. 1–10 (zitiert auf S. 15).

[SBH+17] A. C. F. da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke. „Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments“. Englisch. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER)*. Hrsg. von D. Ferguson, V. M. Muñoz, J. Cardoso, M. Helfert, C. Pahl. Bd. 1. ScitePress. SciTePress Digital Library, Juni 2017, S. 358–367. ISBN: 978-989-758-243-1. DOI: 10.5220/0006243303580367. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2017-28&engl=0 (zitiert auf S. 13).

[SHB+17] A. C. F. da Silva, P. Hirmer, U. Breitenbücher, O. Kopp, B. Mitschang. „Customization and provisioning of complex event processing using TOSCA“. Englisch. In: *Computer Science - Research and Development* (Sep. 2017), S. 1–11. ISSN: 1865-2042. DOI: 10.1007/s00450-017-0386-z. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2017-10&engl=0 (zitiert auf S. 13).

[SHK+06] A. Sobeih, J. C. Hou, L.-C. Kung, N. Li, H. Zhang, W.-P. Chen, H.-Y. Tyan, H. Lim. „J-Sim: a simulation and emulation environment for wireless sensor networks“. In: *IEEE Wireless Communications* 13.4 (2006), S. 104–119 (zitiert auf S. 19, 20).

[SHWM16] A. C. F. da Silva, P. Hirmer, M. Wieland, B. Mitschang. „SitRS XT – Towards Near Real Time Situation Recognition“. Englisch. In: *Journal of Information and Data Management* 7.1 (Apr. 2016), S. 4–17. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2016-14&engl=0 (zitiert auf S. 13).

[Sim17] Simplesoft. *SimpleIoTSimulator*. http://www.smplsft.com/SimpleIoTSimulator.html. Okt. 2017 (zitiert auf S. 18).

[Sma+05] J. F. Smart et al. „An introduction to Maven 2“. In: *JavaWorld Magazine. Available at: http://www. javaworld. com/javaworld/jw-12-2005/jw-1205-maven. html* (2005) (zitiert auf S. 27).

[SS15] A. G. Shoro, T. R. Soomro. „Big data analysis: Apache spark perspective“. In: *Global Journal of Computer Science and Technology* 15.1 (2015) (zitiert auf S. 22).

[Tel17] Telit. *Telit Iot Portal*. https://www.telit.com/products/iot-platforms/telit-iot-portal/. Online; accessed 05-October-2017. 2017 (zitiert auf S. 17).

[VH08] A. Varga, R. Hornig. „An overview of the OMNeT++ simulation environment“. In: *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST (Institute for Computer Sciences, Social-Informatics und Telecommunications Engineering). 2008, S. 60 (zitiert auf S. 19).

[VW12] A. Videla, J. J. Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012 (zitiert auf S. 19).

[XYWV12] F. Xia, L. T. Yang, L. Wang, A. Vinel. „Internet of things“. In: *International Journal of Communication Systems* 25.9 (2012), S. 1101 (zitiert auf S. 15).

[ZGS+17] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, R. Ranjan. „Iotsim: A simulator for analysing iot applications“. In: *Journal of Systems Architecture* 72 (2017), S. 93–107 (zitiert auf S. 17).

Alle URLs wurden zuletzt am 06. 10. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift