Integration von Ursachenerkennung in ein Kontrollzentrum für Softwarelandschaften

Bachelorarbeit

Jens Michaelis

31. März 2015

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
M.Sc. Florian Fittkau

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Automatische Ursachenerkennung ist ein unverzichtbares Werkzeug zur schnellen Behebung von auftretenden Problemen in Softwarelandschaften. Effekte von auftretenden Anomalien propagieren durch die Softwarelandschaft und das manuelle Lokalisieren der Ursache, in einer angemessenen Zeit, stellt eine Herausforderung für Administratoren dar. Im Rahmen eines Projektmoduls wurde der *RanCorr*-Ansatz als Teil eines Kontrollzentrums für komplexe Softwaresysteme in das Analysewerkzeug *ExplorViz* integriert und erweitert. Dieser Ansatz stellt eine vollautomatische Ursachenerkennung vor, um schnell und effizient alle möglichen Ursachen der aufgetretenen Anomalie zu lokalisieren. Zusätzlich wurde ein weiterer Algorithmus als Verfeinerung der bestehenden Algorithmen implementiert und mit diesen verglichen. Die Evaluierung zeigt dabei auf, dass der erweiterte Algorithmus in den betrachteten Szenarien genauere Werte erzeugt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1. Motivation

Automatische Ursachenerkennung ist im Kontext von komplexen Softwaresystemen ein wichtiges Werkzeug, um beim Auftreten einer Anomalie schnell und gezielt reagieren zu können. Anomalien werden nicht immer durch die am stärksten anormale Komponente verursacht. Häufig propagieren Effekte durch das System. Die Zusammenhänge sind bei großen Softwarelandschaften nur schwer in angemessener Zeit nachvollziehbar.
Ausfälle oder Verzögerungen aufgrund der Anomalien sind schnell mit hohen Kosten oder unzufriedenen Endanwendern verbunden. Ungenutzte Serverkapazitäten, welche deaktiviert werden könnten, erzeugen unnötige Kosten. Die Unterstützung des Administratoren durch eine schnelle, präzise und automatisierte Ursachenerkennung im Rahmen eines Kontrollzentrums, ist ein wünschenswerter Beitrag zur Reduzierung von Ausfallzeiten.

1.2. Ziele

Ziel der Arbeit ist es, im Rahmen eines Projektmoduls an der Christian-Albrechts-Universität zu Kiel, die automatische Ursachenerkennung *RanCorr* in ein Kontrollzentrum für Softwarelandschaften zu implementieren und um einen weiteren Algorithmus zu erweitern. Das Kontrollzentrum soll eine Softwarelandschaft automatisch auf Anomalien überwachen und die Ursache für diese finden. Anschließend soll dem Anwender ein vorgefertigter Plan zur Behebung der Anomalie vorgelegt und ausgeführt werden.

1.3. Aufbau

Zuerst werden die verwendeten Technologien und das Abschlussprojekt *Software Engineering* beschrieben. Im Anschluss wird die Ursachenerkennung *RanCorr* erläutert und der erweiterte Algorithmus – der *Refined-Mesh-Algorithmus* – vorgestellt. Im Folgenden wird die Implementierung der Ursachenerkennung in das Kontrollzentrum beschrieben und anschließend die implementierten Algorithmen auf Ihre Zuverlässigkeit und Qualität untersucht. Danach werden verwandte Arbeiten vorgestellt und ein Rückblick über die Erkenntnisse sowie mögliche Erweiterungen gegeben.

Kapitel 2

Grundlagen und Technologien

2.1. ExplorViz

ExplorViz wurde erstmals 2012 von Florian Fittkau in [Fittkau 2012] als Konzept vorgestellt und in [Fittkau u. a. 2013] konkretisiert. Das webbasierte Werkzeug bietet eine Visualisierung der Kommunikation in einer Softwarelandschaft zu Analysezwecken.
Auf Landschaftsebene wird durch eine UML-Darstellung eine Übersicht über das betrachtete System gegeben (siehe Abbbildung 2.1). Dabei geben dunkelgrüne Rahmen eine Knotengruppe, hellgrüne Rahmen einen einzelnen Knoten wieder. Applikationen werden durch blaue Kästen dargestellt und ein Symbol informiert über die Art der Applikation. Die Kommunikation wird durch Linien visualisiert, deren Breite die jeweilige Stärke der Kommunikation angibt.
Auf Applikationsebene wird durch die 3D-Stadt-Metapher ein neuer Ansatz zur Darstellung von Applikationen gewählt, welcher Anwendern ein besseres Verständnis für die Zusammenhänge geben soll. Diese Ansicht ist in Abbildung 2.2 am Beispiel der Live-Demo unter `www.explorviz.net` einzusehen. Die grünen Grundflächen visualisieren Komponenten – im Falle der betrachteten Java-Applikation Pakete. Der Anwender kann einzelne Komponenten öffnen oder schließen, um die jeweiligen Unterkomponenten anzeigen zu lassen. Blaue Kästen stellen Klassen dar, während die Linien die Kommunikation zwischen den Klassen visualisieren.
Durch die Verwendung von *HTML5* in Kombination mit *WebGL* kann *ExplorViz* auf einer Vielzahl an Geräten ohne Installationsaufwand mit handelsüblichen Webbrowsern verwendet werden.

2.2. ΘPADx

Θ*PAD* wurde von Tillmann Bielefeld in [Bielefeld 2012] vorgestellt. Entwickelt als Plugin für *Kieker*, bietet es eine live-Lösung zum Aufspüren von Anomalien in komplexen Softwaresystemen. 2013 wurde Θ*PAD* von Tom Frotscher in [Frotscher 2013] erweitert, um Schwachstellen bei der Erkennung von Langzeitanomalien zu beheben. Die von diesem erweiterten Ansatz – Θ*PADx* genannt – gelieferten Anomaliewerte dienen als Grundlage für die Ursachenerkennung.

2.3. RanCorr

RanCorr wurde in [Marwede u. a. 2009] vorgestellt und präsentiert drei Algorithmen zur automatischen Erkennung von Ursachen für Anomalien in Softwareumgebungen. Dabei wird über gegebene Anomaliewerte und die Architektur des Softwaresystems die wahrscheinlichste Ursache für aufgetretene Anomalien ermittelt. Die Ursachenerkennung erstreckt sich über vier Phasen, welche sequentiell durchgeführt werden. Die drei vorgestellten Algorithmen werden analysiert und die Qualität ihrer Aussagen miteinander verglichen.

2.4. OpenStack

OpenStack ist eine frei verfügbare *Cloud Computing* Plattform, welche 2010 von Rackspace und der NASA veröffentlicht wurde. Das Projekt wird von Firmen wie IBM, Intel oder Red Hat unterstützt. Im Rahmen des Projektmoduls wird *OpenStack* als Grundlage für die Performance- und Qualitätsanalyse sowie für die Demonstration verwendet. Die verwendete Unterkomponente, *OpenStack Nova*, stellt eine Cloudumgebung zur Verteilung von Rechenaufgaben zur Verfügung.

2.5. Hyper-V

Die Berechnungsknoten der *OpenStack Nova* Umgebung werden auf virtuellen Maschinen betrieben. Zur Kommunikation zwischen der realen Hardwareumgebung, dem Hostsystem, und den virtuellen Komponenten wird ein Hypervisor eingesetzt. *Hyper-V* von Microsoft stellt einen solchen Hypervisor zur Verfügung. Es ist als eigenständiges System in Form des kostenlosen *Microsoft Hyper-V Server 2012 R2* oder als Bestandteil der Betriebssysteme Windows (8 Pro und Enterprise) und Microsoft Server (2008 und 2012) verfügbar.

2.6. Super Pi

[*Super Pi*] ist ein von Yasumasa Kanada veröffentlichtes Programm zur Berechnung von 2^i Stellen der Zahl π. Die Ausführung als einzelner Thread belastet einen Kern der CPU für die Dauer der Berechnung. Durch die Möglichkeit die Zahl der zu berechnenden Stellen zu wählen, wird *Super Pi* als Werkzeug zur kontrollierten Überlasterzeugung in der Evaluierung verwendet.

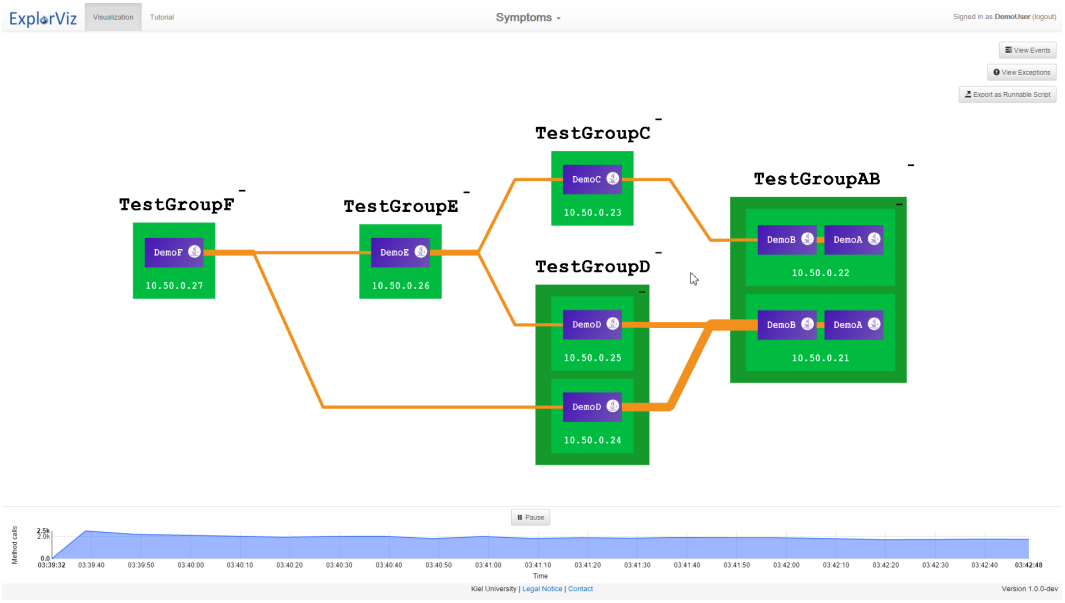


Abbildung 2.1. Darstellung der Landschaft aus der Demoumgebung des Abschlussprojekts in ExplorViz

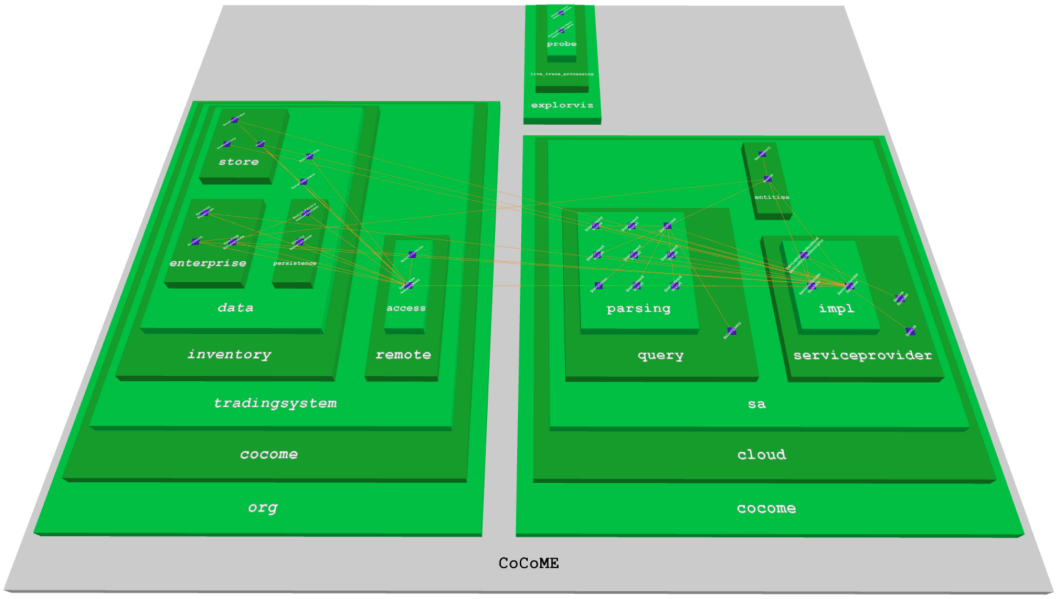


Abbildung 2.2. Beispiel der 3D-Stadt-Metapher aus der Live-Demo unter `www.explorviz.net`

Kapitel 3

Abschlussprojekt - Software Engineering

3.1. Anforderungen

Im Rahmen des Bachelorabschluss/Masterprojekts - Software Engineering für parallele und verteilte Systeme im Wintersemester 2014/15 galt es, das ExplorViz-Projekt um die Funktionalität eines Kontrollzentrums zu erweitern. Das Projekt wurde von acht Studenten bearbeitet.
Ziel des Kontrollzentrums ist es, Anomalien zu erkennen, die Ursachen für diese Anomalien zu lokalisieren, dem Anwender einen Plan zur Behebung der Ursachen vorzuschlagen und diesen im Anschluss durchzuführen.
Dafür wurde das Projekt in vier Phasen aufgeteilt: Symptome, Diagnose, Planung und Umsetzung. Der Anwender soll jederzeit zwischen den ersten drei Phasen wechseln können, um eine visuelle Unterstützung bei der Fehlersuche zu erhalten. Die Anomaliewerte sollen mittels der graphischen Oberfläche präsentiert und das Überschreiten definierter Grenzen über Warnsymbole signalisiert werden. Ebenso soll die Ursachenwahrscheinlichkeit durch eine farbige Markierung kenntlich gemacht werden. Der Anwender soll in der Lage sein, den vorgeschlagenen Plan über einfache Anweisungen in der Oberfläche zu bearbeiten. Außerdem soll die Durchführung des Plans überwacht und bei Fehlern rückgängig gemacht werden.

3.2. Anomaly Detection (Symptome)

Ziel der *Anomaly Detection* ist es, die drei Anomaliearten Punkt-, Kontext- und Kollektivanomalie zu erkennen. Das für Kieker entwickelte Plugin Θ*PADx* wurde zu diesem Zweck an *ExplorViz* angepasst.
Die *Anomaly Detection* analysiert die Antwortzeiten vergangener Aufrufe und ermittelt über einen Vorhersagealgorithmus – *Forecaster* genannt – eine Vorhersage der nächsten Antwortzeit. Die Abweichung von dieser Vorhersage bestimmt den Anomaliewert. In Abbildung 3.1 zeigt die grüne Linie den Vorhersagewert und die blaue Linie den Anomaliewert an. Dabei unterscheidet die *Anomaly Detection* zwischen Unterlastanomalien, welche von 0 bis -1 abgebildet werden, und Überlastanomalien, welche von 0 bis 1 angegeben werden.
Bei definierten Schwellwerten wird auf der analysierten Komponente eine Warnungs- oder Fehlermarkierung gesetzt, welche zur Darstellung eines entsprechenden Symbols

in der graphischen Ausgabe führt (siehe Abbildung 3.2a). Es wurden insgesamt drei Vorhersagealgorithmen implementiert:

NaiveForecaster gibt die zuletzt aufgetretene Antwortzeit als Vorhersagewert zurück.

MovingAverageForecaster liefert den Durchschnitt aller betrachteten Antwortzeiten als Vorhersagewert zurück.

WeightedForecaster gewichtet die betrachteten Antwortzeiten nach ihrer Aktualität. Die Art der Gewichtung kann durch den Benutzer zwischen logarithmisch, linear oder exponentiell gewählt werden. Dieser Algorithmus ist als Teil der Bachelorarbeit von Kim Christian Mannstedt implementiert worden [Mannstedt 2015].

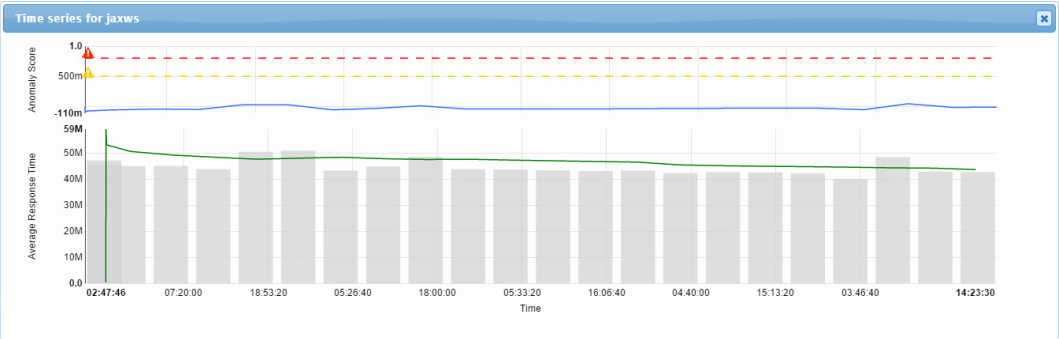
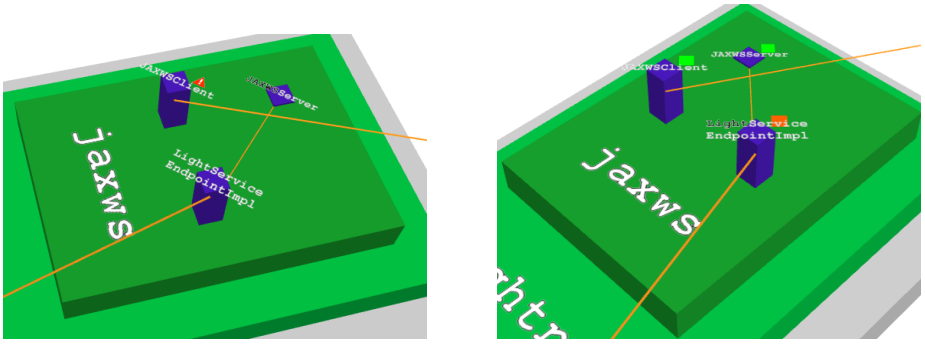


Abbildung 3.1. Darstellung von Anomaliewerten aus der Demoumgebung des Abschlussprojekts

3.3. Root Cause Detection (Diagnose)

Da die Ursachenerkennung als Teil dieser Arbeit integriert wurde, findet eine ausführliche Erklärung in Kapitel 4 auf Seite 11 und eine Beschreibung der Implementierung in Kapitel 5 auf Seite 17 statt.
Ziel der Ursachenerkennung ist es, allen Ursachen für Anomalien den höchsten Wahrscheinlichkeitswert zuzuweisen und dem *Capacity Planning* zur Erstellung eines Maßnahmenplans zu übergeben. Des Weiteren sollen die errechneten Wahrscheinlichkeitswerte dem Anwender über die Benutzeroberfläche visuell dargestellt werden (siehe Abbildung 3.2b). Die Ursachenerkennung wird gestartet sobald die Anomalieerkennung eine Überschreitung der Grenzwerte festgestellt hat.



(a) Warnzeichen der Anomalieerkennung

(b) Farbliche Markierung der Ursachenwahrscheinlichkeit

Abbildung 3.2. Beispiele aus der Demoumgebung des Abschlussprojekts

3.4. Capacity Planning (Planung)

Das *Capacity Planning* erweitert den vorhanden *Capacity-Manager CapMan*. Es werden neue Strategien und die Möglichkeit, Applikationen zu steuern, eingefügt. Sollte die Anomalieerkennung eine Warnung oder einen Fehler auslösen, wartet die Planungsphase die Berechnung der Wahrscheinlichkeitswerte durch die Ursachenerkennung ab und erzeugt auf Grundlage dieser Daten einen Plan (siehe Abbildung 3.3a). Der Plan berücksichtigt dabei das Element mit der höchsten Ursachenwahrscheinlichkeit und alle Elemente, welche in einem definierten Rahmen ähnlich wahrscheinlich sind.
Auf Basis der Anomaliewerte wird ermittelt, ob das jeweilige Element durch eine Unter- oder eine Überlast für eine Anomalie verantwortlich ist. Die Art der Anomalie bestimmt die für das jeweilige Element vorgeschlagene Aktion. Für die Cloud-Knoten steht dabei das Neustarten, Beenden und Verfielfachen eines Knotens zur Verfügung. Applikationen können ebenfalls neu gestartet und beendet werden. Im Rahmen der Bachelorarbeit von Julian Gill wurde zusätzlich das Migrieren einer Applikation implementiert [Gill 2015].
Der Anwender kann einen erzeugten Plan akzeptieren, bearbeiten oder ablehnen. Das Bearbeiten erfolgt dabei über die Benutzeroberfläche von *ExplorViz*. Sobald der Nutzer einen Plan bestätigt, wird dieser an die *Capacity Execution* weitergeleitet.

3.5. Capacity Execution (Umsetzung)

In der Umsetzungsphase wird der von der Planungsphase übergebene Plan in das benötigte Format – eine *ActionList* – übertragen. Die Liste beinhaltet alle *ExecutionActions* – Anweisungen, welche für jedes zu bearbeitende Element durchzuführen sind.

(a) Darstellung eines Plans

(b) Manuelle Anpassung eines Plans

Abbildung 3.3. Beispiele der Planungsphase aus der Demoumbgebung des Abschlussprojekts

Die Bearbeitung der Liste erfolgt nebenläufig. Die erfolgreiche Durchführung einer Aktion wird überwacht und, im Falle einer fehlenden Erfolgsnachricht, nach einer vordefinierten Höchstzeit abgebrochen und erneut gestartet.
Sollte die vordefinierte, maximale Anzahl an Versuchen erreicht werden und die Ausführung einer Aktion im Plan nicht erfolgreich sein, versucht die *Capacity Execution* den Zustand vor der Ausführung des Planes wiederherzustellen. Dazu wird jeder Aktion – bis auf den Neustart – eine Gegenaktion zugeordnet und ausgeführt.
Die Kommunikation mit der *OpenStack Cloud* wurde über den *phyton-nova-client* realisiert. Dieser stellt ein Kommandozeilenwerkzeug für die Cloud zur Verfügung. Erweitert wurde die Umsetzungsphase auch um eine Unterstützung des *LoadBalancer* auf Applikationsebene. Der *LoadBalancer* verteilt die Last zwischen gleichen Applikationen auf unterschiedlichen Knoten. Gleiche Applikationen werden in Skalierungsgruppen zusammengefasst und dem *LoadBalancer* übergeben. Die Skalierungsgruppen wurden in das Landschaftsmodell von *ExplorViz* eingefügt.

Kapitel 4

Ursachenerkennung

4.1. Allgemein

Das Papier [Marwede u. a. 2009] stellt drei Algorithmen zur automatischen Ursachenerkennung vor und analysiert diese bezüglich der Klarheit und Genauigkeit der Aussage. Diese Arbeit erweitert die drei Algorithmen um einen vierten, welcher neue Informationen der Anomalieerkennung einbezieht. Der gebildete Ursachenwahrscheinlichkeitswert – im Folgenden RCR für *RootCauseRating* genannt – stellt die Wahrscheinlichkeit dar, mit der die betrachtete Applikation oder Komponente die Ursache für eine der aufgetretenen Anomalien ist. Auf Klassenebene wird der erzeugte Wert nicht auf einen Wahrscheinlichkeitsraum abgebildet.

4.2. Grundlagen

Im Folgenden werden die mathematischen Grundlagen der Ursachenerkennung beschrieben.

4.2.1. Ungewichtetes arithmetisches Mittel

Das ungewichtete arithmetische Mittel stellt eine Durchschnittsbildung dar und findet sowohl im *Local-* als auch im *Neighbour-Algorithmus* Verwendung. Sei S_i die betrachteten Werte zu Operation i, dann definiert Gleichung 4.1 das Mittel.

$$\overline{S_i} := \frac{1}{|S_i|} \sum_{s \in S_i} s \tag{4.1}$$

4.2.2. Gewichtetes Hölder-Mittel

Das gewichtete Hölder-Mittel – auch das gewichtete Potenzmittel genannt – ist eine spezielle Form des Hölderschen Mittelwertes. Dieser erweitert die arithmetischen Mittelwerte um einen Parameter p, über welchen der Einfluss von extremen Werten auf das errechnete Mittel gesteuert werden kann. Zusätzlich kann jeder Wert gewichtet werden. In der *RanCorr*-Implementierung nach [Marwede u. a. 2009] kommt eine Erweiterung des gewichteten Hölder-Mittels zum Einsatz, welche auch negative Werte als Eingabe zulässt

und in Gleichung 4.2 für die Anomaliewerte S_i abgebildet ist. γ stellt die Erweiterung zur Behandlung negativer Werte dar.

$$\widetilde{S}_{i,p} := \gamma \left(\frac{\sum_{j=1}^{n} w_j \cdot \gamma(s_j, p)}{\sum_{j=1}^{n} w_j}, \frac{1}{p} \right), \tag{4.2}$$
$$\gamma(a, q) := |a|^q \cdot \begin{cases} 1, & a \geqslant 0 \\ -1, & a < 0 \end{cases}$$

Über den Parameter p kann der Einfluss von Extremwerten auf den Mittelwert gesteuert werden. $0 < p < 1$ glättet den Einfluss von Werten außerhalb des Mittelfeldes, während $p > 1$ den Einfluss dieser Werte stärkt.
Als Unterart des Hölder-Mittels wird in den Algorithmen auch das ungewichtete Hölder-Mittel verwendet, bei welchem die Gewichte alle mit 1 belegt sind.

4.3. Architektur

Die Ursachenerkennung durchläuft auf der untersten Ebene, im implementierten Fall der Klassenebene, vier Phasen. Die Korrelationsphase wird dabei im Falle des *Local-Algorithmus* nicht durchgeführt.

Modellbau Die zur Verfügung gestellten Daten werden analysiert und verwendet, um ein Modell der Softwarelandschaft zu generieren. Dazu werden die Kommunikation zwischen aufrufenden und aufgerufenen Klassen, die Anomaliewerte und die Anzahl der Aufrufe, gesammelt und verarbeitet. Es entsteht eine Graphenstruktur, welche die gesamte, aktive Softwarelandschaft abbildet. Klassen ohne Kommunikation werden nicht berücksichtigt, da für diese keine Kommunikationsdaten vorliegen und sie keine Ursache einer Anomalie sein können.

Aggregation Die gesammelten Anomaliewerte einer Klasse werden über eine Mittelwertsmethode auf ein temporäres RCR abgebildet. Da die Anomalieerkennung andere Werte erzeugt als die Algorithmen der Ursachenerkennung nach [Marwede u. a. 2009] voraussetzen, werden diese mit Gleichung 4.3 angepasst, bevor die Aggregation erfolgt. Bei dem erweiterten Algorithmus *Refined-Mesh* ist die Anpassung in dem Aggregationsalgorithmus implementiert.

Korrelation Die temporären RCR werden mit dem generierten Landschaftsmodell aus der Modellbauphase in Beziehung zueinander gesetzt. Die verwendeten Regeln werden in der Beschreibung der einzelnen Algorithmen erläutert. Ziel dieser Phase ist es, Effekte, die aufgrund einer Anomalie durch die Softwarelandschaft propagieren, möglichst

realistisch abzubilden. Dabei werden die temporären RCR angepasst oder unverändert als finaler RCR gespeichert.

Visualisierung Abschließend werden die generierten Werte visuell dargestellt. Dazu werden die erzeugten Werte auf die höheren Ebenen, in diesem Fall der Komponenten- und Applikationsebene, aggregiert, auf einen Wahrscheinlichkeisraum abgebildet und an *ExplorViz* zur Darstellung der Softwarelandschaft übergeben.

$$s_i := |a_i| * 2 - 1 \tag{4.3}$$

4.4. Der Local-Algorithmus

Beim *Local-Algorithmus*, von [Marwede u. a. 2009] mit *Trivial Algorithm* betitelt, wird für jede betrachtete Klasse das ungewichtete arithmetische Mittel über die Anomaliewerte, wie in Gleichung 4.1 beschrieben, gebildet. Es findet keine Korrelationsphase statt.

4.5. Der Neighbour-Algorithmus

Im Falle des *Neighbour-Algorithmus*, welcher in [Marwede u. a. 2009] als *Simple Algorithm* bezeichnet wird, wird der eigene RCR mit denen der direkt benachbarten Klassen in Beziehung zueinander gesetzt.
In der Aggregationsphase erhält jede Klasse ein vorläufiges RCR in Form des arithmetischen Mittels. In der Korrelationsphase wird dann der arithmetische Mittelwert aller direkt aufrufenden Klassen, sowie das maximale RCR aller direkt aufgerufenden Klassen, generiert.
Sollte der Mittelwert aller aufrufenden Klassen größer als der eigene RCR sein, ist davon auszugehen, dass dies eine Konsequenz der Anomalie der betrachteten Klasse ist. Weist zusätzlich keine aufgerufene Klasse einen höheren RCR auf, wird die Anomalie voraussichtlich von der betrachteten Klasse verursacht oder verstärkt. Der RCR der betrachteten Klasse wird erhöht.
Im Gegensatz dazu gilt, dass kein höherer Mittelwert bei aufrufenden Klassen und ein höherer, maximaler RCR bei aufgerufenen Klassen darauf hinweist, dass die betrachtete Klasse nicht als Ursache für die Anomalie in Frage kommt. In diesem Fall wird der eigene RCR reduziert.
Falls keiner der beschriebenen Fälle eintrifft, wird das errechnete vorläufige Ergebnis unverändert abgespeichert. Die Korrelation für Klasse i wird durch Gleichung 4.4 beschrieben.

$$r_i := \begin{cases} \frac{1}{2} \cdot \left(\overline{S}_i + 1\right), & \overline{S}_i^{in} > \overline{S}_i \wedge max_i^{out} \leqslant \overline{S}_i \\ \frac{1}{2} \cdot \left(\overline{S}_i - 1\right), & \overline{S}_i^{in} \leqslant \overline{S}_i \wedge max_i^{out} > \overline{S}_i \\ \overline{S}_i, & else \end{cases} \tag{4.4}$$

$\overline{S}_i$ bezeichnet den RCR der betrachteten Klasse i, welcher mit dem ungewichteten arithmetischen Mittel berechnet wird.

$\overline{S}_i^{in}$ stellt das ungewichtete arithmetische Mittel der vorläufigen RCR aller Klassen dar, welche die betrachtete Klasse direkt aufrufen.

max_i^{out} ist der maximale, vorläufige RCR aller von der betrachteten Klasse direkt aufgerufenen Klassen.

4.6. Der Mesh-Algorithmus

Der Mesh-Algorithmus erweitert die betrachteten Klassen um indirekt aufrufende oder aufgerufene Klassen. Er wird in [Marwede u. a. 2009] als *Advanced Algorithm* bezeichnet. Im Gegensatz zu den beiden vorherigen Algorithmen wird hier in der Aggregationsphase das Hölder-Mittel statt dem arithmetischen Mittel verwendet. Bei aufgerufenen Klassen wird das Maximum rekursiv über alle direkt und indirekt aufgerufenen Klassen gebildet. Im Falle der aufrufenden Klassen werden ebenfalls alle indirekt verbundenen Klassen betrachtet, welche dabei, wie in Gleichung 4.2 beschrieben, aggregiert werden. Gleichung 4.6 gibt dabei an, wie hier die Gewichte ermittelt werden.

$$r_i := \begin{cases} \frac{1}{2} \cdot \left(\widetilde{S}_{i,0.2} + 1\right), & \widetilde{S}_{i,1}^{in} > \widetilde{S}_{i,0.2} \wedge max_i^{out} \leqslant \widetilde{S}_{i,0.2} \\ \frac{1}{2} \cdot \left(\widetilde{S}_{i,0.2} - 1\right), & \widetilde{S}_{i,1}^{in} \leqslant \widetilde{S}_{i,0.2} \wedge max_i^{out} > \widetilde{S}_{i,0.2} \\ \widetilde{S}_{i,0.2}, & else \end{cases} \tag{4.5}$$

$$w := \frac{e}{d^z} \tag{4.6}$$

$\widetilde{S}_{i,0.2}$ stellt den vorläufigen RCR der betrachteten Klasse i dar, errechnet mit dem erweiterten ungewichteten Hölder-Mittel. $p = 0.2$ kann dabei in der Konfiguration angepasst werden.

$\widetilde{S}_{i,1}^{in}$ bezeichnet das erweiterte, gewichtete Hölder-Mittel aller direkt oder indirekt verbundenen, aufrufenden Klassen. Die vorläufigen RCR aller Klassen werden mit dem erweiterten Hölder-Mittel zusammengefasst. Die Gewichte werden dabei mit der in Gleichung 4.6 beschriebenen Formel errechnet, in welcher e das Kantengewicht und d die Distanz zur betrachteten Klasse angibt. Z ist in der Konfiguration definiert und reguliert den Einfluss der Distanz zur betrachteten Klasse auf den Mittelwert.

max_i^{out} bezeichnet den höchsten, vorläufigen RCR aller direkt oder indirekt von der betrachteten Klasse aufgerufenen Klassen.

4.7. Der Refined-Mesh-Algorithmus

Im Gegensatz zu den in [Marwede u. a. 2009] verwendeten Anomaliewerten, ermöglichen die von Θ*PADx* erzeugten Anomaliewerte eine Unterscheidung zwischen Über- und Unterlastanomalien. Die drei beschriebenen Algorithmen führen diese Unterscheidung nicht durch. Als Konsequenz wird eine Überlast- und eine Unterlastanomalie innerhalb des betrachteten Zeitraums bei der RCR-Bestimmung gleichwertig behandelt. Um eine Unterscheidung zu ermöglichen, wurde der *Refined-Mesh-Algorithmus* in der Aggregations- und in der Korrelationsphase angepasst. Im Gegensatz zu den angepassten Anomaliewerten – -1 für keine Anomalie und 1 für eine starke Anomalie – stellt hier -1 eine starke Unterlastanomalie und 1 eine starke Überlastanomalie dar. 0 steht für keine Anomalie. Diese Werte sind für jedes Element i in der Liste $A_i = \{a_1, ..., a_n\}$ hinterlegt.

Aggregationsphase In dieser Phase wird zuerst die Art der Anomalie bestimmt. Es werden die zur Verfügung gestellten Anomaliewerte durch ein gewichtetes Höldersches Mittel, wie in Gleichung 4.7 beschrieben, zusammengefasst. Der Wert w_0 ist in der Konfiguration definiert, der Wert p ist analog zum *Mesh-Algorithmus* in der Konfiguration auf Klassenebene definiert. Ist der gebildete Mittelwert größer oder gleich 0, wird als Anomalieart eine Überlast, beziehungsweise im anderen Fall eine Unterlast zugeordnet. Anschließend wird, wie in Gleichung 4.8 beschrieben, der Mittelwert als temporärer RCR bestimmt. Anomaliewerte der anderen Anomalieart werden mit -1 belegt und Anomaliewerte gleichen Vorzeichens in den Wertebereich übertragen, welcher in den anderen drei Algorithmen verwendet wird. Die Gewichte sind hierbei konstant mit 1 belegt.

$$
\begin{aligned}
t_{i,p} &:= \gamma\left(\frac{\sum_{j=1}^{n} w(a_j)\cdot\gamma(a_j,p)}{\sum_{j=1}^{n} w(a_j)}, \frac{1}{p}\right), \\
\gamma(a,q) &:= |a|^q \cdot \begin{cases} 1, & a \geqslant 0 \\ -1, & a < 0 \end{cases}, \\
wr(a) &:= \begin{cases} w_o, & a >= 0 \\ 1, & a < 0 \end{cases}
\end{aligned}
\tag{4.7}
$$

$$
\begin{aligned}
\widetilde{A}_{i,p} &:= \gamma\left(\frac{\sum_{j=1}^{n}\gamma(\beta(a_j, t_{i,p}), p)}{n}, \frac{1}{p}\right), \\
\gamma(a,q) &:= |a|^q \cdot \begin{cases} 1, & a \geqslant 0 \\ -1, & a < 0 \end{cases}, \\
\beta(a,t) &:= \begin{cases} a \cdot 2 - 1, & a \geqslant 0 \wedge t \geqslant 0 \\ |a| \cdot 2 - 1, & a < 0 \wedge t < 0 \\ -1, & sonst \end{cases}
\end{aligned}
\tag{4.8}
$$

Korrelationsphase Diese Phase wird in Gleichung 4.9 beschrieben. Hier werden bei den ausgehenden Aufrufen – dem Wert max_i^{out} – alle Klassen einer anderen Anomalieart mit dem niedriegsten Wert, -1, belegt und das Sammeln von RCR weiterer, indirekt verbundener Klassen auf diesem Pfad abgebrochen. Analog zum *Mesh-Algorithmus* wird der größte, vorläufige RCR aus der Aggregationsphase aller betrachteten Klassen zurückgegeben.
Bei aufrufenden Klassen – dem Wert $\widetilde{A}_{i,1}^{in}$ – werden Klassen einer anderen Anomalieart ebenfalls mit -1 belegt. Weitere, indirekt verbundene Klassen auf diesem Pfad werden jedoch berücksichtigt (Gleichung 4.10). Das Gewicht ist analog zum *Mesh-Algorithmus*, wie in Gleichung 4.6 beschrieben, gewählt. Zusätzlich gibt es in der Korrelation eine weitere Bedingung – den Buffer. Dieser verhindert eine Erhöhung des RCR bei Werten unter dem in der Konfiguration definierten Wert.

$$
r_i := \begin{cases} \frac{1}{2} \cdot \left(\widetilde{A}_{i,0.2} + 1\right), & \widetilde{A}_{i,1}^{in} > \widetilde{A}_{i,0.2} \wedge max_i^{out} \leqslant \widetilde{A}_{i,0.2} \wedge \widetilde{A}_{i,0.2} > (-1 + 2 \cdot buf) \\ \frac{1}{2} \cdot \left(\widetilde{A}_{i,0.2} - 1\right), & \widetilde{A}_{i,1}^{in} \leqslant \widetilde{A}_{i,0.2} \wedge max_i^{out} > \widetilde{A}_{i,0.2} \\ \widetilde{A}_{i,0.2}, & else \end{cases}
\tag{4.9}
$$

$$
\begin{aligned}
\widetilde{A}_{i,p}^{in} &:= \gamma\left(\frac{\sum_{j=1}^{n} w_j \cdot \gamma(\alpha(a_j, t_{j,p}, t_{i,p}), p)}{\sum_{j=1}^{n} w_j}, \frac{1}{p}\right), \\
\gamma(a,q) &:= |a|^q \cdot \begin{cases} 1, & a \geqslant 0 \\ -1, & a < 0 \end{cases}, \\
\alpha(a, t_j, t_i) &:= \begin{cases} a, & t_j \geqslant 0 \wedge t_i \geqslant 0 \vee t_j < 0 \wedge t_i < 0. \\ -1, & sonst \end{cases}
\end{aligned}
\tag{4.10}
$$

Kapitel 5

Implementierung

5.1. Anpassungen an RanCorr

In [Marwede u. a. 2009] wird in der Korrelationsphase nicht erläutert, wie im Falle von nicht vorhandenen aufrufenden oder aufgerufenen Klassen verfahren wird. In der vorliegenden Implementierung – [*Kieker plug-in for anomaly correlation based on RanCorr*] – wurde in diesem Fall der unveränderte, temporäre RCR ausgegeben. Dieses Vorgehen wurde bei der Implementierung von *RanCorr* in das Kontrollzentrum übernommen.
RanCorr arbeitet mit Operationen als kleinste Ebene. Die von *ExplorViz* gelieferten Daten lassen eine Zuordnung von Kommunikationen nur auf der Klassenebene zu, weshalb diese als kleinste Ebene gewählt wurde.

5.2. Integration in ExplorViz

In diesem Kapitel wird beschrieben, wie die Ursachenerkennung in *ExplorViz* als Teil des Kontrollzentrums implementiert wurde. Abbildung 5.1 gibt dabei eine Übersicht über die Architektur der Implementierung, wobei graue Klassen einen bereits vorhanden Bestandteil von *ExplorViz* darstellen.

5.2.1. Aufbau

Ziel der Implementierung war es, das Austauschen oder Erweitern der Ursachenerkennung möglichst benutzerfreundlich zu gestalten. Daher wurden die vier Phasen zusammengefasst und durch abstrakte Klassen abgebildet.

AbstractRanCorrAlgorithm fasst die Phasen Modellbau, Aggregation auf Klassenebene und Korrelation zusammen. Als Grundlage dienen die von *ExplorViz* erstellten Kommunikationen. Diese beschreiben, welche Klasse eine andere Klasse aufgerufen hat, wie oft dies geschehen ist und welche Anomaliewerte dabei ermittelt wurden. Diese Daten werden verwendet, um einen Beziehungsgraphen der Klassen untereinander aufzubauen und die Anomaliewerte den jeweiligen Klassen zuzuordnen. Anschließend wird die Aggregation der Anomalien zur Bildung des vorläufigen RCR durchgeführt. Sobald das Modell gebaut und die Daten gesammelt sind, wird für jede Klasse ein Thread zur

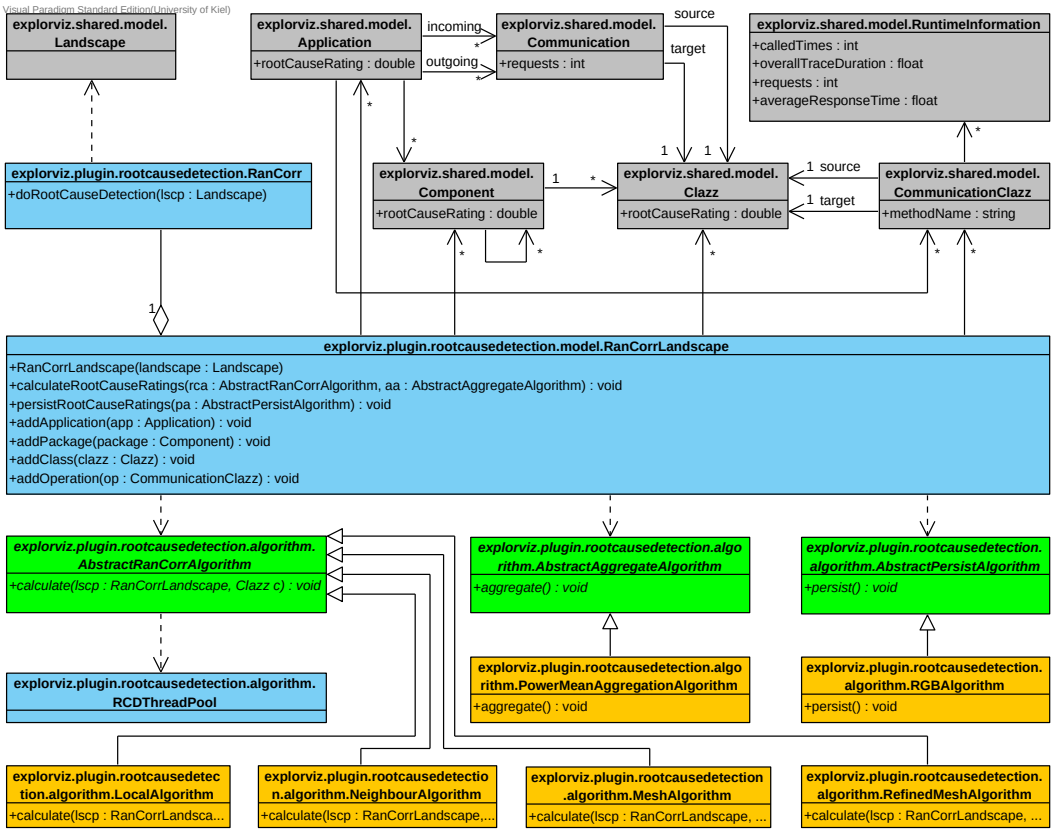


Abbildung 5.1. Klassendiagramm der Ursachenerkennung

Korrelation des RCR in einen ThreadPool hinterlegt, welcher nebenläufig abgearbeitet wird.

AbstractAggregationAlgorithm überträgt Anomaliewerte von der Klassenebene auf die Komponenten- und Applikationsebene. Im Falle des Projekts wird der *PowerMeanAggregationAlgorithm* verwendet, welcher, wie in [Marwede u. a. 2009] beschrieben, die RCR aller Unterkomponenten einer Komponente über das gewichtete Hölder-Mittel zusammenfasst und als RCR der betrachteten Komponente ausgibt. Auf Applikationsebene wird identisch mit allen Komponenten einer Applikation verfahren. Damit übernimmt der Algorithmus den zweiten Teil der Aggregation.

AbstractPersistAlgorithm übernimmt die Visualisierung und Übergabe der Werte an das *Capacity Planning*. Im Falle der Implementierung werden die RCR jedes Elements auf einen RGB-Wert übertragen, welcher dann in der Landschaft gespeichert wird. Dadurch

gibt die Diagnoseansicht von *ExplorViz* eine farbige Markierung beim betrachteten Element wieder.

Die Hilfsfunktionen, zum Beispiel die Mittelwertsfunktionen, wurden in eine externe Klasse ausgelagert. Zusätzlich wurde ein Landschaftsmodell erstellt, welches die von *ExplorViz* zur Verfügung gestellte Softwarelandschaft in ein Format passend zur Ursachenerkennung umwandelt.

5.2.2. Konfiguration

Da betrachtete Softwarelandschaften stark voneinander abweichen können, wurde eine Konfigurationsklasse angelegt. In dieser können die entscheidenden Aspekte der Ursachenerkennung manuell angepasst werden.

ranCorrAlgorithm legt fest, welcher der implementierten Algorithmen zur Erzeugung der RCR verwendet wird.

ranCorrPersistAlgorithm definiert, welcher Algorithmus zur Visualisierung und Übergabe der Werte an die Planungsphase verwendet wird.

ranCorrAggregationAlgorithm steuert, welcher Algorithmus zur Aggregation der Werte von der Klassenebene auf darüberliegende verwendet wird.

RootCauseRatingFailureState ist der Wert, welcher eingetragen wird sobald kein RCR gebildet werden kann. Dies kann an fehlerhaften Verknüpfungen in der Landschaft oder fehlenden Anomaliewerten liegen.

numberOfThreads definiert, wie viele Threads im Threadpool parallel ausgeführt werden.

DistanceIntensityConstant wird im *Mesh-* und *Refined-Mesh-Algorithmus* verwendet und definiert die Stärke des Einflusses von weiter entfernten Klassen auf den Mittelwert, welcher in der Korrelationsphase bei aufrufenden Klassen verwendet wird.

PowerMeanExponentXLevel wird in der Aggregationsphase verwendet. Dieser Wert gibt für jede Ebene das p des Hölder-Mittels und damit den Einfluss von Extremwerten an. Es gibt für die Klassen-, Komponenten- und Applikationsebene einen Eintrag.

RefinedNegativeFactor wird jedem positiven Anomaliewert als Gewicht bei Gleichung 4.7 im *Refined-Mesh-Algorithmus* zugeordnet.

RefinedBuffer steuert, wie weit sehr kleine RCRs bei der Korrelation keinen Einfluss auf das Ergebnis nehmen. Liegt der eigene RCR und der maximale RCR der aufgerufenen Klassen unter oder genau auf diesem Wert, wird keine Korrelation durchgeführt.

Die Konfiguration bietet damit eine Möglichkeit die Implementierung den eigenen Anforderungen anzupassen.

5.2.3. Laufzeiten und Speicherbedarf

Als Grundlage für die Errechnung der Laufzeiten und des Speicherplatzes der Implementierung dient die Menge an Kommunikationen n, welche *ExplorViz* ermittelt. Zu diesen werden die Anomaliewerte durch die Anomalieerkennung erzeugt. Dabei ist die Menge an Anomaliewerten je Kommunikation auf die letzten 15 beschränkt.

Phase	Laufzeit	Speicher
Modellbau	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Aggregation	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Korrelation	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Visualisierung	$\mathcal{O}(n)$	$\mathcal{O}(n)$

5.2.4. Nebenläufigkeit

Aufgrund der erhöhten Laufzeit in der Korrelationsphase auf Klassenebene, wurde entschieden, diese Kalkulation nebenläufig zu gestalten. Bei der Implementierung des *Mesh-* und *Refined-Mesh-Algorithmus* ist es theoretisch notwendig bei der Berechnung jeder Klasse die Werte aller vorhandenen Klassen auszulesen, um die benötigten Zwischenwerte zu berechnen. Die Nebenläufigkeit wurde bereits auf der Ebene des abstrakten Korrelationsalgorithmus realisiert und muss von jeder Implementierung eingehalten werden. Die benötigten, in der Modellbauphase generierten Daten, werden in jeweils einer *ConcurrentHashMap* hinterlegt, um eine hohe Anzahl an parallelen Lesezugriffen auf die Elemente zu gewährleisten.

Kapitel 6

Evaluation

Im Folgenden werden die implementierten und im Laufe des Abschlussprojekts entwickelten Algorithmen bezüglich ihrer Qualität überprüft und miteinander verglichen.

6.1. Ziele

6.1.1. Vergleich der Algorithmen zur Ursachenerkennung

Der im Rahmen des Abschlussprojekts entwickelte, erweiterte *Mesh-Algorithmus – Refined-Mesh* genannt – wird mit den drei bereits vorgestellten Algorithmen verglichen. Es gilt zu prüfen, welcher Algorithmus für welches Anwendungsszenario die aussagekräftigsten Ergebnisse bezüglich der Ursachenwahrscheinlichkeit erzeugt.

6.1.2. Bestimmung der Größe des Zeitfensters

Die in der Aggregationsphase gesammelten Anomaliewerte stammen aus einem definierten Zeitfenster. Es gilt zu bestimmen, welches Zeitfenster für die Ursachenerkennung – auf alle vier Algorithmen bezogen – die qualitativ hochwertigsten Ergebnisse erzeugt.

6.2. Methodik

Um die Algorithmen vergleichen zu können, müssen für alle vier Berechnungen identische Zustände gegeben sein. Es wird jedes Szenario auf einer Cloudumgebung drei mal durchlaufen und die Anomaliewerte werden gesammelt. Der Durchschnitt aus den drei Durchläufen wird in einer nachgebauten Landschaft an die vier Algorithmen übergeben. Um die Qualität vergleichbar zu machen, muss eine Kennzahl definiert werden. Für die Auswertung wird ein Qualitätskriterium, welches in Gleichung 6.1 beschrieben ist, verwendet. Für die Planungsphase ist eine möglichst starke Abgrenzung zwischen den tatsächlichen Ursachen für Anomalien und den erzeugten Nebeneffekten für die Erstellung eines effizienten Maßnahmenplans entscheidend. Daher müssen die Algorithmen eine möglichst große, prozentuale Distanz zwischen den korrekt als Ursache erkannten RCR und dem höchsten, nicht als Ursache definierten, RCR aufweisen. Im Falle mehrerer, gleichzeitig auftretender Ursachen für Anomalien ist es außerdem entscheidend, wie nah

die ermittelten RCR der Ursachen beieinanderliegen. Dieser Wert wird in Gleichung 6.2 beschrieben.

$$quali(\{u_1,...,u_i\},\{v_1,...,v_j\}) := \begin{cases} \frac{\overline{U}-v_1}{\overline{U}}, & u_1 \geqslant v_1 \\ 0, & u_1 < v_1 \vee \overline{U} = 0 \end{cases} \tag{6.1}$$

Es sind alle einer tatsächlichen Ursache zugeordneten, absteigend sortierten RCR in Menge $U = \{u_1,...,u_i\}$ eingetragen. Menge $V = \{v_1,...,v_j\}$ beinhaltet alle RCR, welche Elementen zugeordnet wurden, die nicht Ursache der entstandenen Anomalien sind. Diese Werte sind ebenfalls absteigend sortiert. $\overline{U}$ bezeichnet das ungewichtete arithmetische Mittel der Menge U nach Gleichung 4.1.

$$naehe(\{u_1,...,u_i\} := \begin{cases} \frac{u_1-u_i}{u_1}, & u_1 > 0 \\ 0, & sonst \end{cases} \tag{6.2}$$

Die übermittelten Anomaliewerte stammen aus vier unterschiedlichen Zeitfenstern. $[-10,+4]$ besteht aus zehn Anomaliewerten vor der Warnung durch die Anomalieerkennung, der eigentlichen Anomalie und vier Werten nach der Warnung. $[-5,+4]$ beinhaltet fünf statt zehn Werte vor der Anomalie und $[-2,+2]$ zwei Werte vor und zwei Werte nach der Anomalie sowie die Anomalie selbst. $[0]$ besteht ausschließlich aus dem Anomaliewert zum Zeitpunkt der Warnung.

6.3. Versuchsaufbau

Die verwendete Programmstruktur besteht aus fünf Berechnungsknoten – *ComputeNode* genannt – der *OpenStack Nova* Umgebung. *OpenStack Nova* stellt die Technologien zur Verfügung, um Berechnungsknoten zu steuern und in einer Cloudumgebung zusammenzuführen. Als Hypervisor wird in diesem Versuch *Hyper-V* als Bestandteil von [*Microsoft Hyper-V Server 2012 R2*] verwendet.
Um die Cloudumgebung zu realisieren, werden zwei Rechner in einem geschlossenen Netzwerk mit dem *Hyper-V* Server betrieben. Jeder dieser Server ist mit einen Intel Core i7-3770k mit je vier Prozessorkernen mit *Hyperthreading* – zwei virtuellen Kernen pro Prozessorkern – und 16GB Arbeitsspeicher ausgestattet. Somit stehen pro Server 8 virtuelle Prozessorkerne zur Verfügung. Die Steuerung erfolgt über einen dritten Computer und dem *Hyper-V* Manager.
Auf dem ersten Server wird der *OpenStack* Kontrollknoten [*DevStack*] eingerichtet. Dieser stellt die Schnittstellen zur Steuerung der Cloud und Kommunikation zwischen den Knoten zur Verfügung. Dafür wird der Kontrollknoten auf sechs der acht virtuellen Prozessorkerne mit 12 GB Arbeitsspeicher betrieben. Die Berechnungsknoten N1 bis N5 werden ebenfalls auf den Servern betrieben. Jedem dieser Knoten wird ein virtueller Prozessorkern und 3 GB Arbeitsspeicher zugewiesen. Auf allen Knoten läuft ein Abbild von [*Ubuntu Server*] als grundlegendes Betriebssystem.

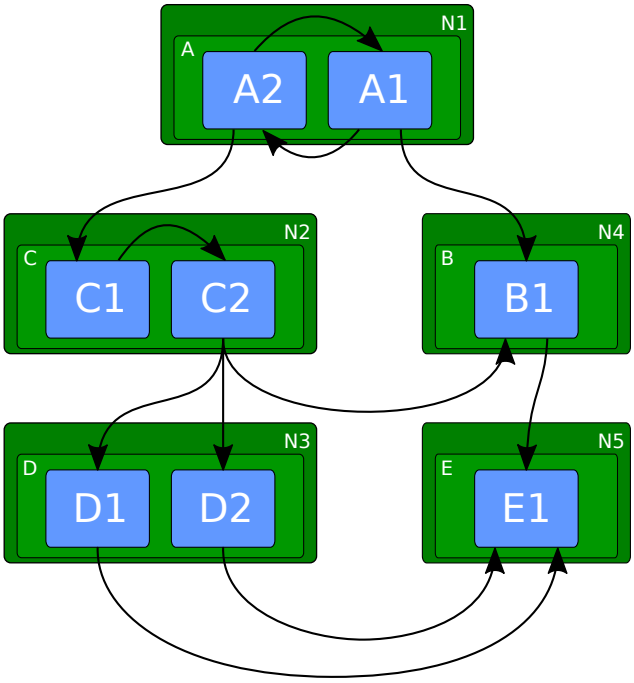
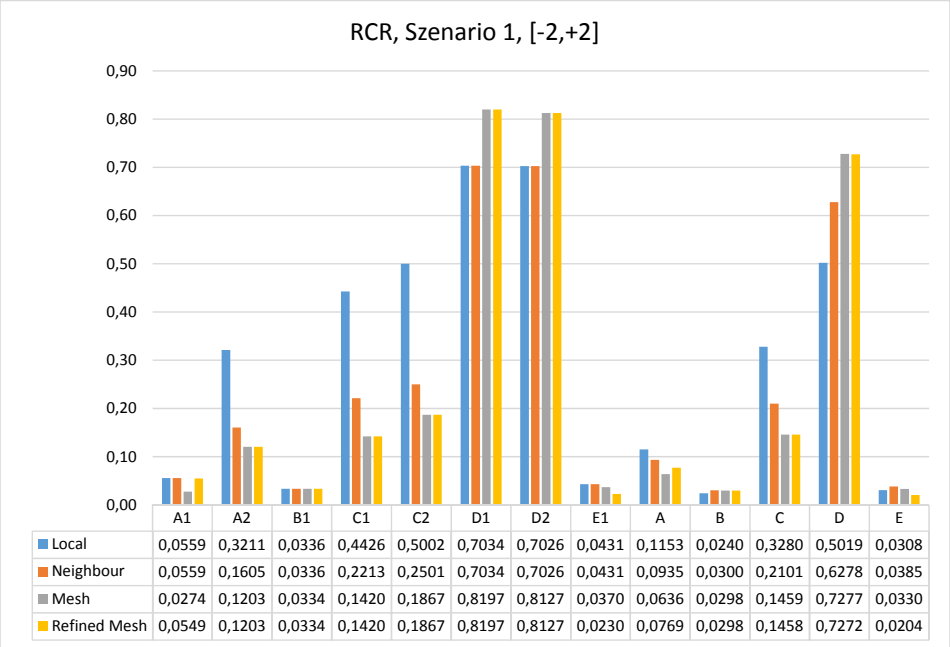


Abbildung 6.1. Die untersuchte Softwarelandschaft

Auf den Berechnungsknoten wird die Softwarelandschaft, wie in Abbildung 6.1 beschrieben, aufgebaut. Jedem Knoten wird dabei eine Applikation A bis E zugeordnet. Es handelt sich um eine Java-Umgebung, welche per Remote Procedure Call eine Anfrage an die jeweils nächste Klasse schickt. Diese Klasse führt eine Berechnung von Pi mit der Monte-Carlo-Simulation durch (beschrieben in [Andrieu u. a. 2001]). Dabei werden zufällige Punkte in einem Einheitsquadrat gewählt und überprüft, ob diese innerhalb des Einheitskreises liegen. Die Berechnung wird bei einem virtuellen Prozessorkern und 50.000 Punkten im Durchschnitt in 80ms durchgeführt. Anschließend wird, falls notwendig, die nächste Klasse aufgerufen und auf die Antwort dieser gewartet, bevor eine Antwort an die aufrufende Klasse erfolgt.
Die Monte-Carlo-Simulation bietet den Vorteil stark CPU-gebunden zu sein, wodurch Anomaliewerte durch Auslastung der CPU mit [*Super Pi*] erzeugt werden können.



	A1	A2	B1	C1	C2	D1	D2	E1	A	B	C	D	E
Local	0,0559	0,3211	0,0336	0,4426	0,5002	0,7034	0,7026	0,0431	0,1153	0,0240	0,3280	0,5019	0,0308
Neighbour	0,0559	0,1605	0,0336	0,2213	0,2501	0,7034	0,7026	0,0431	0,0935	0,0300	0,2101	0,6278	0,0385
Mesh	0,0274	0,1203	0,0334	0,1420	0,1867	0,8197	0,8127	0,0370	0,0636	0,0298	0,1459	0,7277	0,0330
Refined Mesh	0,0549	0,1203	0,0334	0,1420	0,1867	0,8197	0,8127	0,0230	0,0769	0,0298	0,1458	0,7272	0,0204

Abbildung 6.2. Ermittelte RCR für Szenario 1 im Zeifenster $[-2, +2]$

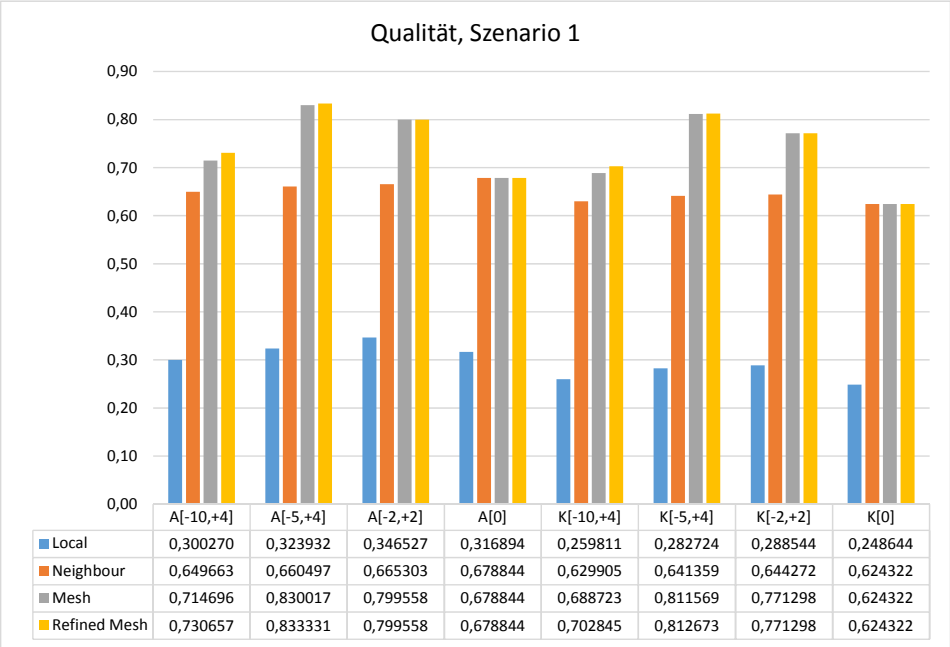
6.4. Szenarien

6.4.1. Überlastszenario

Das erste Szenario stellt ein klassisches Überlastszenario dar. Knoten D wird durch eine parallel ausgeführte Pi-Berechnung CPU-seitig für 50 Sekunden belastet. Die dadurch erzeugte Anomalie beeinflusst die von D abhängigen Knoten und Applikationen.

6.4.2. Gemischtes Über- und Unterlastszenario

Im zweiten Szenario kommt es parallel zu einer Über- und einer Unterlast.
Knoten B wurde vor dem Auftreten der eigentlichen Anomalie durch einen parallel laufenden Prozess mittelstark belastet. *Super Pi* benötigt für die Kalkulation von 16.384 Stellen im Schnitt 170ms auf einem Berechnungsknoten und wird zuerst alle 300ms, nach 20 Sekunden alle 200ms aufgerufen. Damit wird eine Überlastanomalie auf Knoten B erzeugt, welche noch unter den Grenzwerten der Anomalieerkennung liegt. Nach 40 Sekunden wird die Berechnung eingestellt, wodurch eine starke Unterlastanomalie entsteht. Diese überschreitet die Grenzwerte der Anomalieerkennung, was zu einem Anomaliefehler führt.



	A[-10,+4]	A[-5,+4]	A[-2,+2]	A[0]	K[-10,+4]	K[-5,+4]	K[-2,+2]	K[0]
Local	0,300270	0,323932	0,346527	0,316894	0,259811	0,282724	0,288544	0,248644
Neighbour	0,649663	0,660497	0,665303	0,678844	0,629905	0,641359	0,644272	0,624322
Mesh	0,714696	0,830017	0,799558	0,678844	0,688723	0,811569	0,771298	0,624322
Refined Mesh	0,730657	0,833331	0,799558	0,678844	0,702845	0,812673	0,771298	0,624322

Abbildung 6.3. Qualität in Szenario 1 nach Gleichung 6.1

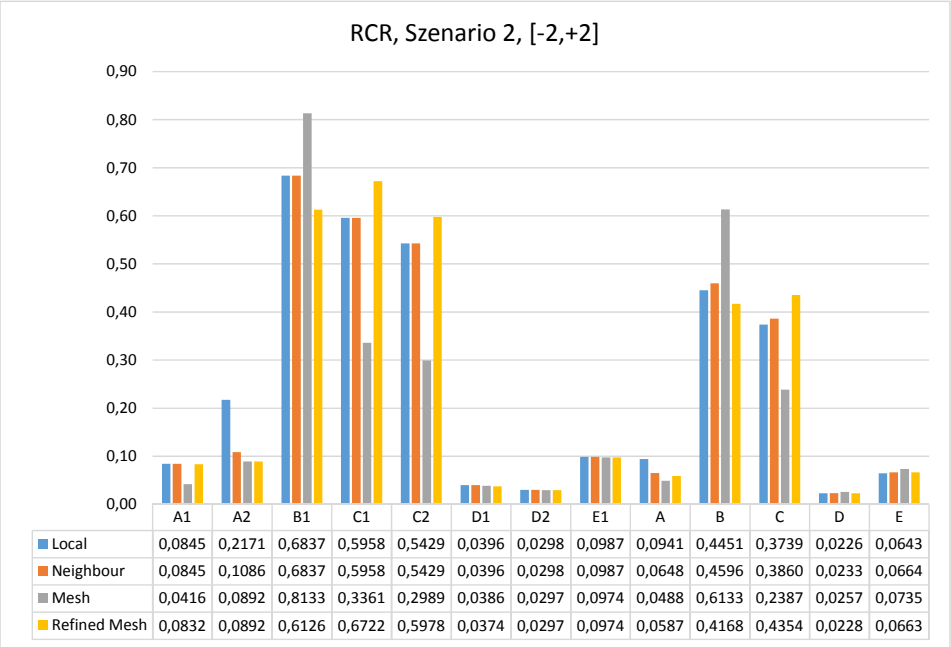
Auf Knoten C wird gleichzeitig eine Überlastanomalie verursacht, indem alle 200*ms* eine Berechnung von 16.384 Stellen von Pi durchgeführt wird.
Dieses Szenario soll darstellen, wie unterschiedliche Anomalien die Ursachenerkennung beeinflussen. Parallel laufende Hintergrundprozesse, wie die Indizierung einer Datenbank oder hohe Anfragen von externen Quellen auf Applikationen, die von *ExplorViz* nicht erfasst werden, können zu Unterlastanomalien führen, sobald diese beendet sind. Das gleichzeitige Auftreten von Überlastanomalien kann in komplexen Softwarelandschaften nicht ausgeschlossen werden.

6.4.3. Überlastszenario mit falscher Korrelation

Als drittes Szenario wird ein weiteres Überlastszenario durchgeführt. Hier wird der Knoten C – ähnlich dem Knoten D in Szenario 1 – und der Knoten D gering belastet.

6.5. Ergebnisse

Abbildung 6.2, 6.4 und 6.7 stellen die ermittelten RCR im Zeitfenster $[-2, +2]$ dar. Die ersten acht Spalten geben die Klassenebene, die letzten fünf die Applikationsebene wieder. Das



	A1	A2	B1	C1	C2	D1	D2	E1	A	B	C	D	E
Local	0,0845	0,2171	0,6837	0,5958	0,5429	0,0396	0,0298	0,0987	0,0941	0,4451	0,3739	0,0226	0,0643
Neighbour	0,0845	0,1086	0,6837	0,5958	0,5429	0,0396	0,0298	0,0987	0,0648	0,4596	0,3860	0,0233	0,0664
Mesh	0,0416	0,0892	0,8133	0,3361	0,2989	0,0386	0,0297	0,0974	0,0488	0,6133	0,2387	0,0257	0,0735
Refined Mesh	0,0832	0,0892	0,6126	0,6722	0,5978	0,0374	0,0297	0,0974	0,0587	0,4168	0,4354	0,0228	0,0663

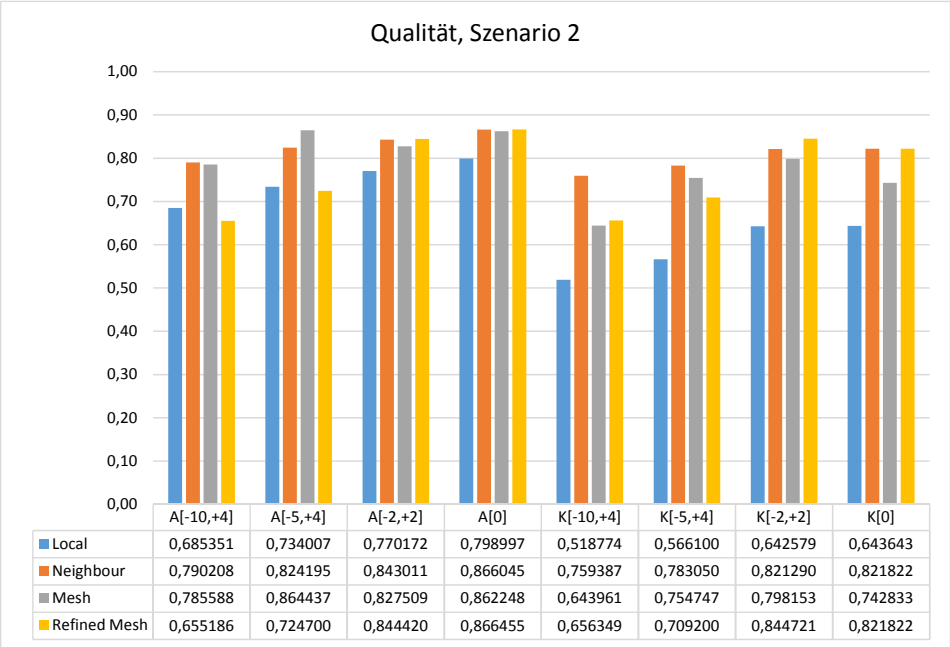
Abbildung 6.4. Ermittelte RCR für Szenario 2 im Zeifenster $[-2, +2]$

Zeitfenster wurde gewählt, um die Unterschiede in der Aggregation von Anomaliewerten aufzuzeigen.
Abbildung 6.3, 6.5 und 6.8 stellen die errechneten Qualitätsmerkmale nach Gleichung 6.1 dar. Bei den Qualitätsdiagrammen bilden die ersten vier Spalten die Applikationsebene und die anderen vier die Klassenebene ab. Abbildung 6.6 zeigt die Nähe zwischen korrekten Anomalien im Fall des zweiten Szenarios nach Gleichung 6.2. Abbildung 6.9 bildet den ungewichteten arithmetischen Mittelwert der Qualitätskennzahlen aus allen drei Szenarien ab.

6.6. Diskussion

6.6.1. Szenario 1

Abbildung 6.3 zeigt auf, dass der *Refined-Mesh-Algorithmus* auf der Applikationsebene in allen Zeitfenstern mit durchschnittlich $0,761$ die qualitativ hochwertigsten Werte erzeugt, wobei der *Mesh-Algorithmus* mit $0,756$, mit weniger als 1% Differenz, nur knapp unter den Werten des *Refined-Mesh-Algorithmus* liegt.



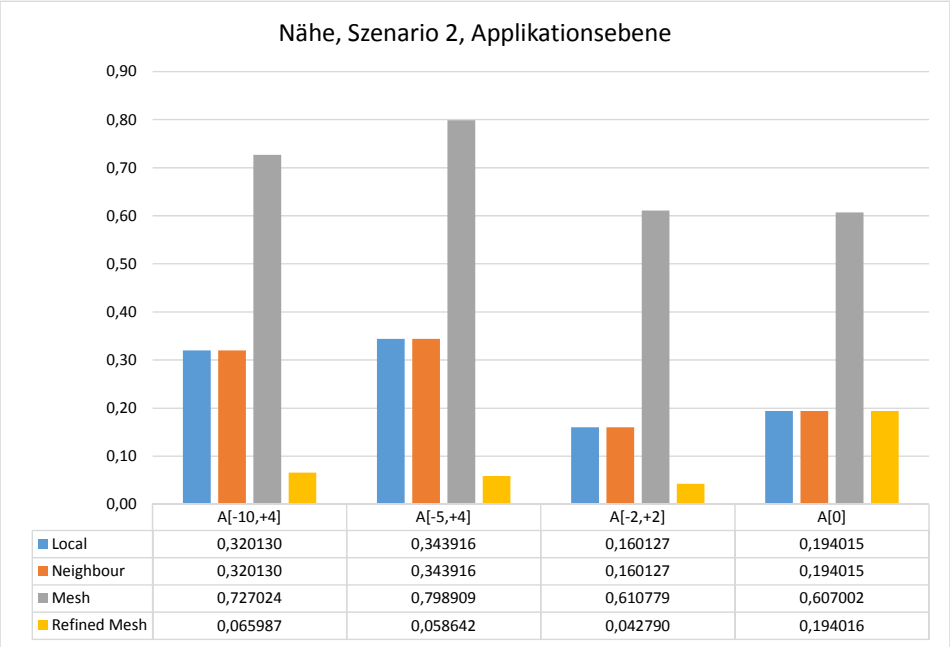
	A[-10,+4]	A[-5,+4]	A[-2,+2]	A[0]	K[-10,+4]	K[-5,+4]	K[-2,+2]	K[0]
Local	0,685351	0,734007	0,770172	0,798997	0,518774	0,566100	0,642579	0,643643
Neighbour	0,790208	0,824195	0,843011	0,866045	0,759387	0,783050	0,821290	0,821822
Mesh	0,785588	0,864437	0,827509	0,862248	0,643961	0,754747	0,798153	0,742833
Refined Mesh	0,655186	0,724700	0,844420	0,866455	0,656349	0,709200	0,844721	0,821822

Abbildung 6.5. Qualität in Szenario 2 nach Gleichung 6.1

Mit 0.664 weist der *Neighbour-Algorithmus* einen um 13% schlechteren Wert als die zuvor betrachteten Algorithmen auf. Diese Differenz wird durch die unterschiedlichen Aggregationsmethoden in der Aggregationsphase erklärt.
Der *Local-Algorithmus* erzeugt mit $0,322$ Werte, die um $48\% - 58\%$ niedriger als die Werte der vorher betrachteten Algorithmen sind. Diese Differenz wird durch die fehlende Korrelationsphase begründet. Während die ersten drei Algorithmen die RCR von A2, C1 und C2 reduzieren, um den Einfluss der Anomalie in D auszugleichen, fehlt dieser Schritt in der Ausführung des *Local-Algorithmus*. Als Konsequenz sind die erzeugten RCR dieser Klassen höher, was die Qualität der Aussage verringert (siehe Abbildung 6.2).

6.6.2. Szenario 2

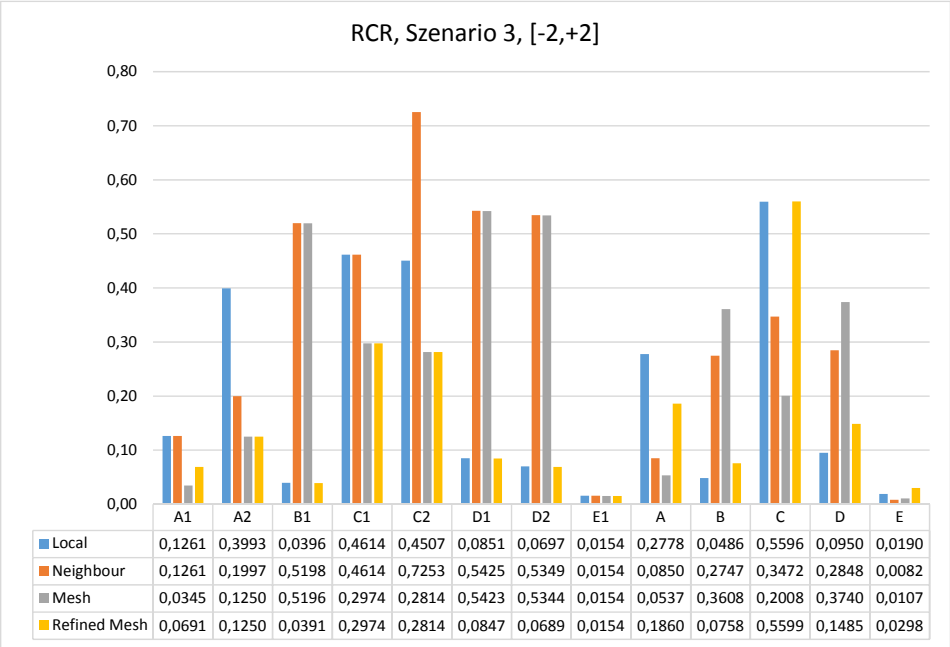
Im zweiten Szenario ist aufgrund mehrerer, gleichzeitig auftretender Anomalien, neben der Qualität der Aussage, auch die Nähe der erzeugten RCR aller Ursachen wichtig. Um den Anwender effizient zu unterstützen, muss der verwendete Algorithmus sowohl die Unterlast- als auch die Überlastanomalie korrekt erkennen und ihre Ursachen in einen dichten Wertebereich zusammenlegen.



	A[-10,+4]	A[-5,+4]	A[-2,+2]	A[0]
Local	0,320130	0,343916	0,160127	0,194015
Neighbour	0,320130	0,343916	0,160127	0,194015
Mesh	0,727024	0,798909	0,610779	0,607002
Refined Mesh	0,065987	0,058642	0,042790	0,194016

Abbildung 6.6. Nähe in Szenario 2 nach Gleichung 6.2

Anormal sind in diesem Szenario der Knoten B mit einer Unterlastanomalie und der Knoten C mit einer starken Überlast. Eine Betrachtung der erzeugten RCR in Abbildung 6.4 zeigt, dass der *Mesh-Algorithmus* eine starke Abweichung von 60% zwischen dem RCR von B mit $0,61$ und C mit $0,24$ aufweist. Dies führt in der Planungsphase dazu, dass der erzeugte Plan von Klasse B als Ursache für die Anomalien ausgeht und hier prüft, ob die Unterlast behoben werden kann. Da B kein Teil einer Skalierungsgruppe ist und somit nicht weiter reduziert werden kann, wird keine Maßnahme vorgeschlagen. Der niedrige RCR von C sorgt außerdem für eine schwache Einfärbung der betroffenen Elemente und erschwert somit die manuelle Fehlerdiagnose.
Ursache für diese Abweichung ist die fehlende Unterscheidung zwischen Anomaliearten im *Mesh-Algorithmus*. Da B1 einen höheren Anomaliewert als C1 und C2 aufweist, wird in der Korrelationsphase der Wert von C1 und C2 reduziert.
Beim *Neighbour-Algorithmus* erfolgt diese Reduzierung nicht, da im Falle von C1 nur der Wert von A2 und C2 in der Korrelationsphase verwendet wird. Im Falle von C2 verhindert der höhere RCR von C1 eine Reduzierung in der Korrelationsphase.
Der *Refined-Mesh-Algorithmus* erkennt die unterschiedlichen Anomaliearten von C1, beziehungsweise C2, im Vergleich zu B1. Daher wird nicht der höhere Wert von B1 sondern der Wert von E1 als maximaler Ausgangswert für die Korrelation von C2 und der Wert von



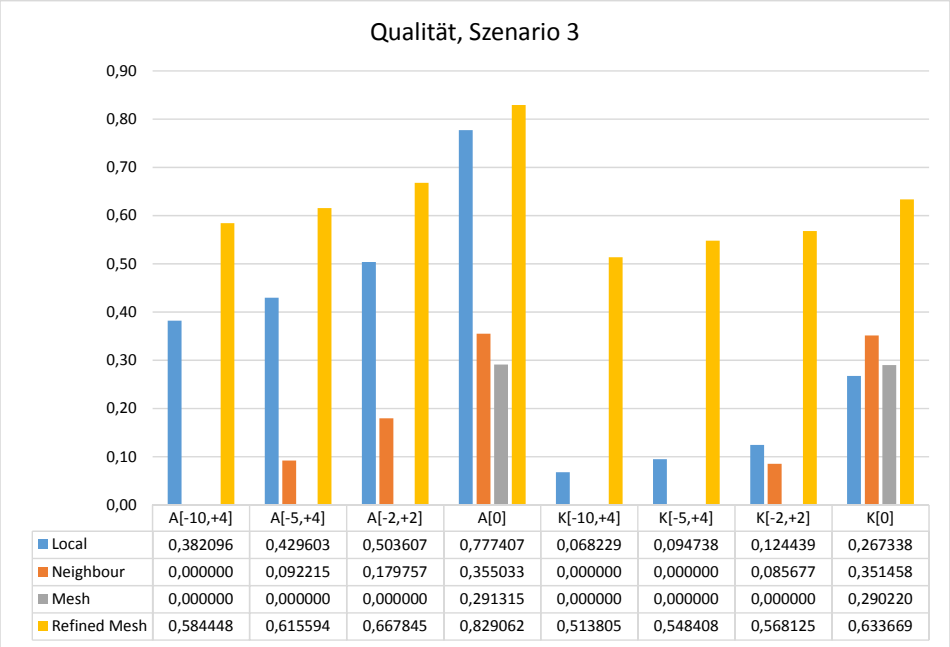
	A1	A2	B1	C1	C2	D1	D2	E1	A	B	C	D	E
Local	0,1261	0,3993	0,0396	0,4614	0,4507	0,0851	0,0697	0,0154	0,2778	0,0486	0,5596	0,0950	0,0190
Neighbour	0,1261	0,1997	0,5198	0,4614	0,7253	0,5425	0,5349	0,0154	0,0850	0,2747	0,3472	0,2848	0,0082
Mesh	0,0345	0,1250	0,5196	0,2974	0,2814	0,5423	0,5344	0,0154	0,0537	0,3608	0,2008	0,3740	0,0107
Refined Mesh	0,0691	0,1250	0,0391	0,2974	0,2814	0,0847	0,0689	0,0154	0,1860	0,0758	0,5599	0,1485	0,0298

Abbildung 6.7. Ermittelte RCR für Szenario 3 im Zeifenster $[-2, +2]$

C2 für die Korrelation von C1 herangezogen, was zu keiner Reduzierung der betrachteten Werte führt.
Beim *Local-Algorithmus* spielen diese Zusammenhänge durch die fehlende Korrelationsphase keine Rolle.
Bei der Analyse der Nähe von korrekt erkannten Ursachen in Abbildung 6.6 zeigt sich ein klarer Vorsprung für den *Refined-Mesh-Algorithmus* im Falle der ersten drei Zeitfenster und ein nahezu identischer Wert im Vergleich zum *Neighbour-Algorithmus*, sowie dem *Local-Algorithmus* im letzten Zeitfenster. Für alle betrachteten Zeitfenster ergibt sich ein Durchschnittswert von $0,09$ für den *Refined-Mesh-*, $0,26$ für den *Local-* sowie den *Neighbour-* und $0,69$ für den *Mesh-Algorithmus*, wobei ein niedrigerer Wert für einen engen, korrekten Ergebnisraum steht.

6.6.3. Szenario 3

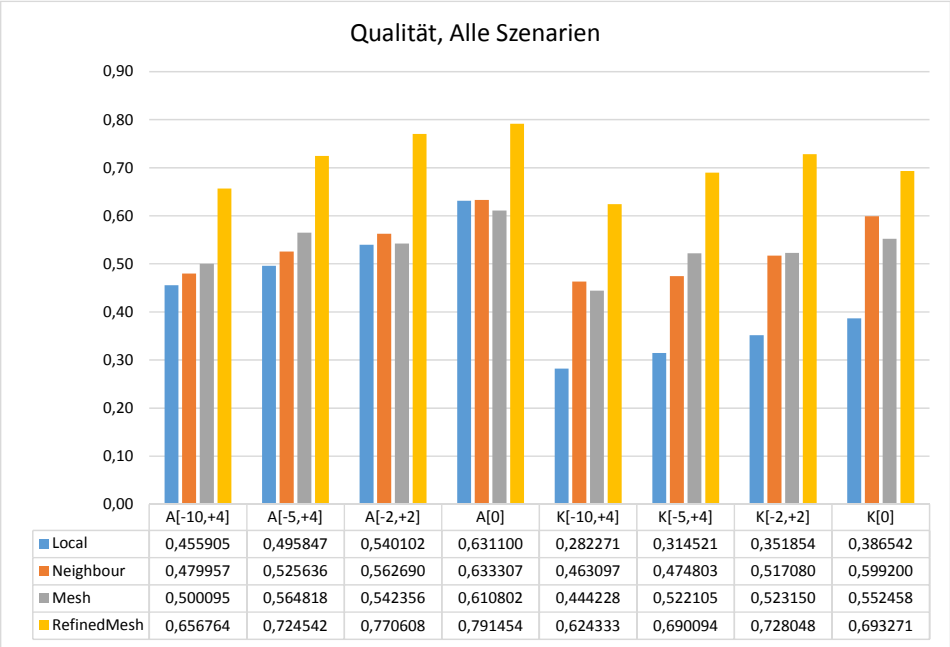
Im dritten Szenario kommt es in mehreren Fällen zu Fehldiagnosen bezüglich der Ursache der Anomalie (in Abbildung 6.8 durch den Wert 0 dargestellt). Der *Neighbour-Algorithmus* ordnet in einem Zeitfenster auf Applikationsebene und zwei Zeitfenstern auf Klassenebene dem falschen Element den höchsten RCR zu, wie es in Abbildung 6.7 zu sehen ist. Der



	A[-10,+4]	A[-5,+4]	A[-2,+2]	A[0]	K[-10,+4]	K[-5,+4]	K[-2,+2]	K[0]
Local	0,382096	0,429603	0,503607	0,777407	0,068229	0,094738	0,124439	0,267338
Neighbour	0,000000	0,092215	0,179757	0,355033	0,000000	0,000000	0,085677	0,351458
Mesh	0,000000	0,000000	0,000000	0,291315	0,000000	0,000000	0,000000	0,290220
Refined Mesh	0,584448	0,615594	0,667845	0,829062	0,513805	0,548408	0,568125	0,633669

Abbildung 6.8. Qualität in Szenario 3 nach Gleichung 6.1

Refined-Mesh-Algorithmus ordnet in jeweils drei von vier Fällen dem falschen Element die höchste Ursachenwahrscheinlichkeit zu. Der *Local-Algorithmus* und der *Refined-Mesh-Algorithmus* erkennen die korrekte Ursache auf Applikation C, beziehungsweise Klasse C1 und C2.
Der Grund für dieses Verhalten liegt im Umgang mit niedrigen Anomaliewerten in der Korrelationsphase. Die Klassen D1, D2 und B1 werden von C2 aufgerufen und rufen die Klasse E1 auf. Der durch die Anomalie stark erhöhte Wert von C2 erfüllt die erste Bedingung zur Erhöhung der vorläufigen RCR von D1, D2 und B1 in der Korrelation. Der niedrigere Wert von E1 erfüllt die zweite Bedingung, was trotz niedriger Anomaliewerte und geringen Unterschieden zwischen dem RCR von E1 und der betrachteten Klassen zu einer Erhöhung der vorläufigen Werte von D1, D2 und B1 führt: Bereits ein vorläufiger RCR von 0.0001 bei D1, D2 oder B1 – im Vergleich zu 0.00001 bei E1 – würde in der Korrelationsphase zu einem RCR von 0.5 führen.
Aus diesem Grund bietet der *Refined-Mesh-Algorithmus* die Möglichkeit sehr kleine, vorläufige RCR in der Korrelationsphase nicht zu erhöhen. Die Grenze kann in der Konfiguration definiert werden.
Durch die fehlende Korrelationsphase lässt sich das Verhalten des *Neighbour-* und *Mesh-Algorithmus* beim *Local-Algorithmus* nicht beobachten.



	A[-10,+4]	A[-5,+4]	A[-2,+2]	A[0]	K[-10,+4]	K[-5,+4]	K[-2,+2]	K[0]
Local	0,455905	0,495847	0,540102	0,631100	0,282271	0,314521	0,351854	0,386542
Neighbour	0,479957	0,525636	0,562690	0,633307	0,463097	0,474803	0,517080	0,599200
Mesh	0,500095	0,564818	0,542356	0,610802	0,444228	0,522105	0,523150	0,552458
RefinedMesh	0,656764	0,724542	0,770608	0,791454	0,624333	0,690094	0,728048	0,693271

Abbildung 6.9. Durchschnitt der Qualität über alle Szenarien nach Gleichung 6.1

In diesem Szenario werden die Werte von D durch eine minimale Überlast geringfügig erhöht. Natürliche Schwankungen in den Laufzeiten haben jedoch in der Demoumgebung des Testsystems bereits zu einem solchen Verhalten geführt.

6.6.4. Zeitfenster

Abbildung 6.9 zeigt auf, dass auf der Applikationsebene das Zeitfenster [0] die aussagekräftigsten Ergebnisse für alle Algorithmen liefert.

6.6.5. Qualität über alle Szenarien

Wie in Abbildung 6.9 zu sehen ist, liegt der *Refined-Mesh-Algorithmus* im Durchschnitt über allen drei Szenarien und in jedem Zeitfenster qualitativ vor den anderen Algorithmen. Im entscheidenden Zeitfenster von [0] ist die Qualität des *Refined-Mesh-Algorithmus* in jedem Szenario, sowohl auf Klassen- als auch auf Applikationsebene, über oder gleichauf mit den Qualitätswerten der übrigen Algorithmen. Des Weiteren liegen die RCR in Szenario 2 bei dieser Implementierung der Ursachenerkennung am dichtesten beieinander.

Aufgrund der falschen Korrelation beim *Mesh-Algorithmus* im zweiten Szenario, liefert der *Neighbour-Algorithmus* im direkten Vergleich minimal höhere Werte.
Ebenso profitiert der *Local-Algorithmus* hier von den Ergebnissen des zweiten Szenarios. Dieser zeigt jedoch bei der Überlastsituation in Szenario 1 klare Schwächen im Vergleich zu den übrigen Algorithmen, wie es bereits in Kapitel 6.6.1 dargestellt wurde.

6.7. Einschränkung der Validität

Die dargestellten Szenarien decken nur einen kleinen Teil der möglichen Verläufe in einem komplexen Softwaresystem ab. Szenario 1 stellt dabei einen Standardfall dar, Szenario 2 und 3 decken Sonderfälle ab, wie sie im Laufe des Abschlussprojekts beobachtet werden konnten.
Aufgrund der kleinen Softwarelandschaft und der Abbildung der drei Szenarien in einer einzigen Umgebung können die Ergebnisse bei anderen Landschaften abweichen.
Durch die Übertragung der Werte in eine nachgebildete Landschaft ging die genaue Zahl der Aufrufe auf Klassenebene für jeden Zeitpunkt verloren. Die verwendeten Werte sind Durchschnittswerte.
Der qualitative Unterschied zwischen dem *Mesh-Algorithmus* und *Refined-Mesh-Algorithmus* wäre bei der Betrachtung von Szenarien nach ihrer relativen Wahrscheinlichkeit geringer, da das einzelne Auftreten einer Anomalie wahrscheinlicher ist als das gleichzeitige Auftreten mehrerer.

Kapitel 7

Verwandte Arbeiten

7.1. X-ray

In [Attariyan u. a. 2012] wird von der University of Michigan und Google, Inc. ein Endanwenderansatz zur Ursachenerkennung vorgestellt. Ziel des Werkzeuges ist es, Anwendern ohne Zugriff auf den Quellcode einer Anwendung potentielle Ursachen für anormale Laufzeiten aufzuzeigen.
Dabei konzentriert sich *X-ray* auf Konfigurationsmöglichkeiten, welche im Rahmen eines Endandwenders oder Systemadministratoren liegen und von diesen im Voraus spezifiziert werden müssen.
X-ray zeichnet während der Ausführung der Anwendung sämtliche unterstützen Metriken auf (Netzwerk- und Datenträgeraktivität, CPU-Auslastung, Anfragenverzögerungen) und stellt diese für eine spätere Analyse bereit.
Über dynamische Informationsflussanalyse wird ermittelt, welcher Konfigurationswert an welchem Punkt im Programm eingelesen wird und welchen Einfluss dieser auf die aktuelle Operation ausübt.
Es wird für jede Aktion ein Durchschnittswert zur betrachteten Metrik gebildet und mit der Anzahl der Aufrufe multipliziert. Dadurch wird der Konfigurationswert mit dem größten Einfluss auf die betrachtete Metrik bestimmt.
Diese Art der Gewichtung könnte als Erweiterung der im *Mesh-* und *Refined-Mesh-Algorithmus* verwendeten Gewichtung genutzt werden. Im Bereich der aufgerufenen Klassen findet bei diesen Algorithmen keine Gewichtung statt. Bei aufrufenden Klassen wird ausschließlich die Anzahl der Aufrufe gezählt.

7.2. PRCA

2013 wurde von Christoph Heger in [Heger u. a. 2013] *PRCA* vorgestellt. Der Ansatz soll Programmierer während der Entwicklung bei der Identifikation von Performanceproblemen helfen und die Ursachen für diese aufspüren.
Der Ansatz verwendet Unit-Tests zur Performancebestimmung und einen Änderungsgraphen des Codes zur Identifizierung der Ursache. Diese beiden Werkzeuge werden entsprechend vorausgesetzt und müssen von den Entwicklern angelegt und gepflegt werden.

PRCA durchläuft, sobald es angestoßen wird, die Unit-Tests und vergleicht die Performancewerte mit den vorherigen Durchläufen. Hier findet eine Anomalieerkennung statt. Kommt es zu einer Verschlechterung der Werte, lädt *PRCA* die durchgeführten Änderungen aus dem Änderungsgraphen der Versionsverwaltung. Die betroffenen Tests werden erneut ausgeführt, für alle betroffenen Methoden Performancedaten erzeugt und ein Aufrufgraph aufgebaut.
Aus den so gewonnenen Daten wird in einer Korrelationsphase geprüft, welche Methoden für die Performanceregression in Frage kommen. Anschließend wird bei diesen Methoden geprüft, ob eine Änderung vorgenommen wurde. Auf diese Weise wird die Anzahl an möglichen Methoden als Ursache deutlich eingeschränkt und der Entwickler kann bei der Fehlersuche gezielt vorgehen.
Die verwendeten Methoden der Ursachenerkennung ähneln dem Ansatz von [Marwede u. a. 2009]. Es findet sowohl eine Aggregation der Werte, als auch eine Korrelation über den Aufrufgraph statt.

7.3. MonitorRank

In Zusammenarbeit mit LinkedIn wurde 2013 von Myunghwan Kim *MonitorRank* vorgestellt [Kim u. a. 2013]. Der Ansatz konzentriert sich auf breit gefächerte Softwaresysteme – im Falle von LinkedIn mit 400 Diensten.
MonitorRank wertet die Daten tausender Sensoren aus und spürt die Ursachen für Anomalien auf. Bei Softwaresystemen dieser Größe ist nach [Kim u. a. 2013] die Korrelation von Anomalien nicht aussagekräftig genug, da das System ständiger Änderungen unterliegt und sich der Callgraph bei einer Anomalie stark vom normalen Verhalten unterscheiden kann. Der Callgraph alleine reicht nicht aus, um komplexe Zusammenhänge zu erfassen. So können zwei Sensoren von einer CPU abhängig sein, ohne im Callgraph direkt oder indirekt verbunden zu sein. Auch die Werte zusammenhängender Sensoren können sich nach [Kim u. a. 2013] stark unterscheiden: So erzeugt beispielsweise ein Anwender mit 1000 Verbindungen im Netzwerk einen anderen Aufruf und damit eine andere Abhängigkeit als ein Anwender mit 10.
Daher nutzt *MonitorRank* drei Komponenten, um ein RCR zu bestimmen. Zuerst werden die Informationen aller Sensoren gesammelt und aggregiert. Anschließend wird aus den gewonnenen Daten ein Callgraph erzeugt und die gesammelten Werte in sogenannte *Cluster* gebündelt. Diese *Cluster* fassen Sensoren zusammen, die durch interne Faktoren, wie ähnlichem Anomalieverhalten oder externen Faktoren, wie räumliche Anordnung, in Bezug zueinander stehen.
Sobald eine Anomalie auftritt, wird die letzte Phase, die eigentliche Ursachenerkennung, gestartet. Diese Phase durchläuft von einer Anomalie ausgehend alle aufgerufenen Sensoren und sucht diejenigen, welche ein ähnliches Anomalieverhalten aufweisen. Dabei werden Muster in den gesammelten Metriken gesucht und verglichen. Jeder abhängige Sensor erhält auf diese Weise einen Ähnlichkeitsfaktor.

Anschließend läuft ein Algorithmus den Graph, beginnend mit der Anomalie, ab. Die Wahrscheinlichkeit mit der an einer Gabelung ein Sensor gewählt wird, wird durch den Ähnlichkeitsfaktor bestimmt. Die Klasse mit den meisten Besuchen durch den Algorithmus wird als wahrscheinlichste Ursache für die Anomalie angesehen.
Diese Methode gleicht dem personalisierten *PageRank-Algorithmus* mit Teleportation [Jeh und Widom 2003]. Der gewählte Algorithmus ähnelt dem Verhalten eines Mitarbeiters ohne Wissen über das System, welcher auf der Suche nach der Ursache für die Anomalie ist.
Der beschriebene Ansatz beinhaltet einige potentielle Verbesserungsmöglichkeiten für *RanCorr*. Die Bündelung von Anomalien nach Ähnlichkeiten grenzt den betrachteten Werteraum stark ein. Große Softwaresysteme sind somit leichter zu überprüfen. Die Einbeziehung von externen Faktoren spielt dabei eine wichtige Rolle, die bisher von *RanCorr* nicht berücksichtigt wird.

Kapitel 8

Fazit und Ausblick

8.1. Fazit

Die automatische Ursachenerkennung stellt einen wichtigen Teil des Kontrollzentrums für Softwarelandschaften dar. Die ermittelten Werte unterstützen die Planerstellung und geben dem Anwender wichtige Hinweise zur Lokalisierung von Fehlern und manuellen Korrektur an der Landschaft. Die Arbeit hat die vier Phasen der Ursachenerkennung vorgestellt und erläuterte die unterschiedlichen Vorgehensweisen der einzelnen, implementierten Algorithmen. In einer Fallbetrachtung wurden drei mögliche Szenarien in einem Softwaresystemen durchgespielt und analysiert.
Wie die Evaluation zeigt, stellt der *Refined-Mesh-Algorithmus* eine sinnvolle Erweiterung zu den in [Marwede u. a. 2009] vorgestellten Algorithmen dar. Die Ursachenerkennung hat in 93% der Fälle die höchste Wahrscheinlichkeit den Ursachen der Anomalien zugewiesen. Im Falle des *Refined-Mesh-Algorithmus* war die Zuweisung in jedem Fall erfolgreich. Die implementierten Algorithmen stellen somit ein gutes Werkzeug zur Lokalisierung von Ursachen für Anomalien zur Verfügung.

8.2. Ausblick

Die Algorithmen arbeiten jeweils nur mit den Anomaliewerten einer einzelnen Metrik (im Falle des Kontrollzentrums der Antwortzeiten). Während es durch einen Tausch seitens der Anomalieerkennung möglich ist, andere Metriken zu unterstützen, wird der Aspekt von mehreren verfügbaren Metriken nicht betrachtet. Eine Unterstützung mehrerer verschiedener Quellen durch die Ursachenerkennung ist mit der Implementierung insofern möglich, dass für jede der Quellarten eine unabhängige Ursachenwahrscheinlichkeit ermittelt werden kann. Der Einfluss der verschiedenen Metriken aufeinander, wie zum Beispiel der CPU-Belastung auf die Antwortzeiten, könnte jedoch auch in der Korrelationsphase berücksichtigt und die Algorithmen entsprechend erweitert werden.
Die Möglichkeiten, die durch unterschiedliche Anomaliearten in der Implementierung des *Refined-Mesh-Algorithmus* gegeben sind, wurden im Rahmen der Arbeit bei der Aggregationsphase auf Klassenebene und der Korrelation berücksichtigt. Es fand keine Betrachtung der Aggregationsphase für übergeordnete Ebenen statt – der Komponenten- und Applikationsebene. Hier gibt es Potential für weitere Verbesserungen zur klaren Abgrenzung der Wahrscheinlichkeitswerte.

Abschließend ist anzumerken, dass die betrachtete Testumgebung deutlich kleiner ist als komplexe Softwaresysteme. Die Beurteilung der Qualität unter realen Bedingungen und der Einfluss auf die gewählte Visualisierung, gilt es weiter zu untersuchen.

Anhang

Dateien auf Datenträger

*explorviz** Der Quellcode des kompletten Kontrollzentrums, inklusive kompilierter Dateien.

*RanCorr** Der Quellcode der im Rahmen der Projektarbeit implementierten Ursachenerkennung (nicht ausführbar).

Daten.xlsx und Daten.pdf Die in den Testszenarien gesammelten Daten.

presentation.pdf Die Präsentation der Ergebnisse der Arbeit.

thesis.pdf Die Arbeit in Druckform.

Literaturverzeichnis

[Andrieu u. a. 2001] C. Andrieu, N. de Freitas, A. Doucet und M. I. Jordan. An introduction to MCMC for machine learning (Sep. 2001). (Siehe Seite 23)

[Attariyan u. a. 2012] M. Attariyan, M. Chow und J. Flinn. X-ray: automatic root-cause diagnosis of performance anomalies in production software. In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. USENIX Association, 2012, Seiten 307–320. (Siehe Seite 33)

[Bielefeld 2012] T. C. Bielefeld. Online Performance Anomaly Detection for Large-Scale Software Systems. Diplomarbeit. Christian-Albrechts-Universität zu Kiel, 2012. (Siehe Seite 3)

[*DevStack*] DevStack. `https://git.openstack.org/cgit/openstack-dev/devstack/`, Zugriff am 01.03.2015. (Siehe Seite 22)

[Fittkau 2012] F. Fittkau. Online Trace Visualization for System and Program Comprehension in Large Software Landscapes. Vortrag. 2012. (Siehe Seite 3)

[Fittkau u. a. 2013] F. Fittkau, J. Waller, C. Wulf und W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE Computer Society, Sep. 2013. (Siehe Seite 3)

[Frotscher 2013] T. Frotscher. Architecture-Based Multivariate Anomaly Detection for Software Systems. Masterarbeit. Christian-Albrechts-Universität zu Kiel, 2013. (Siehe Seite 3)

[Gill 2015] J. Gill. Integration von Kapazitätsmanagement in ein Kontrollzentrum für Softwarelandschaften. Bachelorarbeit. Christian-Albrechts-Universität zu Kiel, 2015. (Siehe Seite 9)

[Heger u. a. 2013] C. Heger, J. Happe und R. Farahbod. Automated root cause isolation of performance regressions during software development. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, Seiten 27–38. (Siehe Seite 33)

[Jeh und Widom 2003] G. Jeh und J. Widom. Scaling personalized web search. In: *Proceedings of the 12th International Conference on World Wide Web*. ACM, 2003, Seiten 271–279. (Siehe Seite 35)

[*Super Pi*] Y. Kanada. Super Pi. `ftp://pi.super-computing.org/Linux/super_pi.tar.gz`, Zugriff am 02.03.2015. (Siehe Seiten 4 und 23)

[Kim u. a. 2013] M. Kim, R. Sumbaly und S. Shah. Root cause detection in a service-oriented architecture. In: *Proceedings of the ACM SIGMETRICS/International conference on measurement and modeling of computer systems*. ACM, Juni 2013, Seiten 93–104. (Siehe Seite 34)

[Mannstedt 2015] K. C. Mannstedt. Integration von Anomalieerkennung in ein Kontrollzentrum für Softwarelandschaften. Bachelorarbeit. Christian-Albrechts-Universität zu Kiel, 2015. (Siehe Seite 8)

[Marwede u. a. 2009] N. Marwede, M. Rohr, A. van Hoorn und W. Hasselbring. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In: *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*. IEEE, März 2009, Seiten 47–57. (Siehe Seiten 4, 11–15, 17, 18, 34 und 37)

[*Microsoft Hyper-V Server 2012 R2*] Microsoft Hyper-V Server 2012 R2. `http://www.microsoft.com/en-us/evalcenter/evaluate-hyper-v-server-2012-r2`, Anmeldung erforderlich, Zugriff am 01.03.2015. (Siehe Seite 22)

[*Kieker plug-in for anomaly correlation based on RanCorr*] D. Olp und Y. Noller. Kieker plug-in for anomaly correlation based on RanCorr. Version 0.2. (Siehe Seite 17)

[*Ubuntu Server*] Ubuntu Server. 14.04.01 LTS (Trusty Tahr), `http://cloud-images.ubuntu.com/releases/14.04/release-20150209.1/`, Version vom 10.02.2015, Zugriff am 02.03.2015. (Siehe Seite 22)