Humboldt-Universität zu Berlin
Institut für Informatik
Informatik in Bildung und Gesellschaft

Seminararbeit Autonomic Computing - Zusammenfassung: Pinpoint

Oliver Stadie[1]

June 27, 2010

[1] oliver.stadie@googlemail.com

Inhaltsverzeichnis

1 Einführung

Dies ist eine Seminararbeit zum Seminar "Autonomic Computing" im Sommersemester 2010 an der HU Berlin. Aufgabe war es, die Publikation [Chen] zusammenzufassen. Diese behandelt das automatisierte Identifizieren von Fehlern in dynamischen Softwaresystemen, wie z. B. e-commerce Systeme.

1.1 Aufbau & Abgrenzung

Da [Chen] bereits selbst eine recht kurze und präzise formulierte Arbeit ist, scheint ein Zusammenfassen nur möglich indem einige Fakten weggelassen werden und andere hervorgehoben werden. Dieser Abschnitt soll kurz den Aufbau des Papers zusammenfassen und aufzeigen, auf welche Teile des Papers in dieser Zusammenfassung wie stark eingegangen wird.

Das Paper unterteilt sich in folgende 7 Abschnitte:

Introduction: Hier geben die Autoren eine Motivation - wozu sollen wir Fehlersuche automatisieren? Vorgestellt wird das Ziel des Papers und das Problem bisheriger Analyse-Verfahren. In Abschnitt 2.1 wird das Problem und das Ziel behandelt, andere Analyse-Verfahren werden hier ausgeblendet.

The Pinpoint Framework: behandelt die zugrunde liegende Idee und die benötigten SW-Komponenten zur Umsetzung dieser Idee. Diese Idee ist der Hauptfokus der Zusammenfassung, da sich das ganze Paper um diese dreht. Behandelt wird dies in Abschnitt 2.2.

Pinpoint Implementation: behandelt die konkrete technische Umsetzung der Idee. Diese wurde in der Zusammenfassung stark gekürzt, da die Idee auch in jeder anderen technischen Umgebung denkbar ist. Die Zusammenfassung der technische Umsetzung ist in 2.3 zu finden.

Evaluation: wertet die technische Umsetzung durch Fehlerinjektion aus. Definiert die Testumgebung. Führt eigene Metriken ein. Vergleicht Pinpoint mit anderen Analyse-Verfahren. Behandelt den Performance Verlust, durch die zusätzliche Analyse. Die Zusammenfassung kürzt die Bewertung stark, ebenso den Vergleich mit anderen Analyse-Verfahren. Die Metrik-Definitionen werden umgangen, indem intuitive Metriken benutzt werden. Die Evaluation wird in Abschnitt 2.4 behandelt.

Discussion: Die Grenzen von Pinpoint und Ansätze zur Erweiterung sind Inhalt dieses Abschnitts. Außerdem rechtfertigen die Autoren noch einmal wozu Pinpoint entwickelt wurde. Sie geben Literaturhinweise und Hinweise auf zukünftige Arbeiten. Hier wird nichts davon behandelt, außer die Grenzen und Ansätze zur Erweiterung in Abschnitt 2.5.

Conclusions: Hier sagen die Autoren ihre Meinung zu ihren Ergebnissen. In Abschnitt 3 gibt der Autor der Zusammenfassung stattdessen seine Meinung ab.

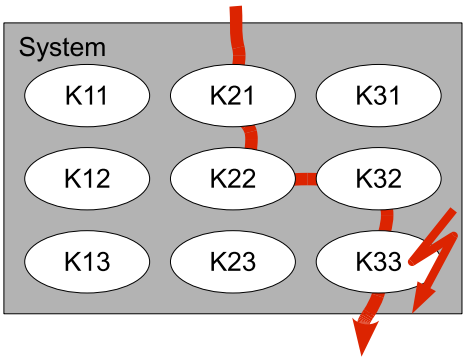

Abbildung 1: Problem beim Ermitteln der Fehlerquelle

Acknowledgements: Diese entfallen hier komplett.

2 Zusammenfassung

2.1 Problem & Ziel

In diesem Unterabschnitt soll die Problematik erläutert werden und das Ziel, welches durch Pinpoint erreicht werden soll.

Das Problem bei der Analyse heutiger dynamischer Systeme ist, dass sie unüberschaubar groß sind. Sie unterliegen ständigem Wachstum und ständigen Änderungen. Jedes Mal wenn eine neue Funktionalität eingeführt wird, ändert sich das System und es kommt häufig vor, dass man nicht weiß, wie genau das neue Teilsystem intern aussieht. Diese Systeme setzen sich aus interagierenden, verteilten Komponenten zusammen und selbst wenn jede Komponente für sich einwandfrei funktioniert, können Fehler aus der Interaktion zwischen verschiedenen Komponenten emergieren. Betrachtet werden hier Systeme, die nacheinander Requests von Clients erhalten und einen Response an diese zurücksenden. Das ist die Funktionsweise der meisten heutigen Internetanwendungen. Das in diesen Systemen Fehler auftreten ist normal und das Paper versucht nicht diese Fehler zu vermeiden, sondern ihre Quellen zu identifizieren.

Ziel ist es also, ein Analyse-Verfahren zu entwickeln, welches die Fehlerquellen findet, ohne das konkrete System kennen oder gar anpassen zu müssen.

Als Beispiel sehe man sich Abbildung 1 an. Ein Client-Request der einen Fehler erzeugt (roter Pfeil) durchläuft das zu untersuchende System. Das System benutzt zur Bearbeitung des Request nacheinander seine Komponenten K21, K22, K32 und K33 und sendet danach einen Fehler-Response an den Client zurück. Selbst wenn wir wissen, dass der Fehler in Komponente K33 auftrat, könnte die Fehlerursache genauso gut in K21, K22 oder K32 liegen. Ziel ist nun also die Fehlerquelle zu identifizieren.

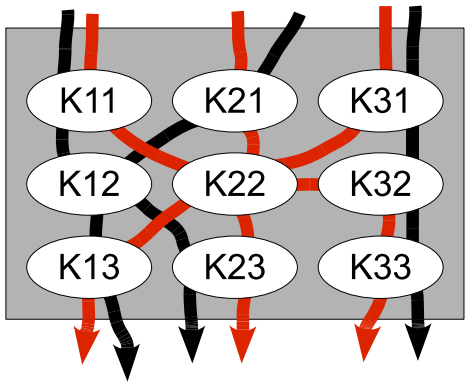

Abbildung 2: Idee: viele Requests um Fehlerquelle zu identifizieren. Rote Pfeile = Requests mit Fehler; Schwarze Pfeile = Requests ohne Fehler

2.2 Idee

Die Idee ist nun, statt nur einem einzelnen Request, viele Requests zu analysieren, siehe als Beispiel Abbildung 2.

Man kann nun sehen, dass einige Komponenten öfter Fehler haben als andere, z. B. hat K22 in 100% der Fälle Fehler (3 von 3), K21 nur in 50% (1 von 2) und K12 in 0% (0 von 2).

Man kann auch das Zusammenspiel mehrerer Komponenten auswerten. So haben z. B. K32 und K33 eine Fehlerquote von 50% (1 von 2) wenn sie zusammenarbeiten.

Mit diesem Vorgehen kann man zwar die Fehlerquellen nicht mit hundertprozentiger Sicherheit identifizieren, dafür kann man jedoch die Sensibilität einstellen, ab der eine Komponente als fehlerhaft angenommen wird (z. B. bei Fehlerquoten über 5%).

2.2.1 Komponenten von Pinpoint

Abbildung 3 zeigt die Architektur von Pinpoint - dem System zum Lösen des Problems. Die grauen Kästen sind die Komponenten von Pinpoint, die Komponenten A bis C sind das zu analysierende System und die weißen Kästen sind Daten die von Pinpoint erzeugt werden. Pinpoint hat nun 3 Teilaufgaben:

1. Client Request Tracing: Protokollierung welche Komponenten für jeden Request benutzt werden. D. h. Erstellung des Trace Log aus Abbildung 3.

2. Failure Detection: Protokollierung welche Requests fehlerhaft sind und welche nicht. Dabei werden sowohl interne, als auch externe Fehler ausgewertet (siehe Unterabschnitt 2.2.3 für Details). D. h. Erstellung des Fault Log aus Abbildung 3.

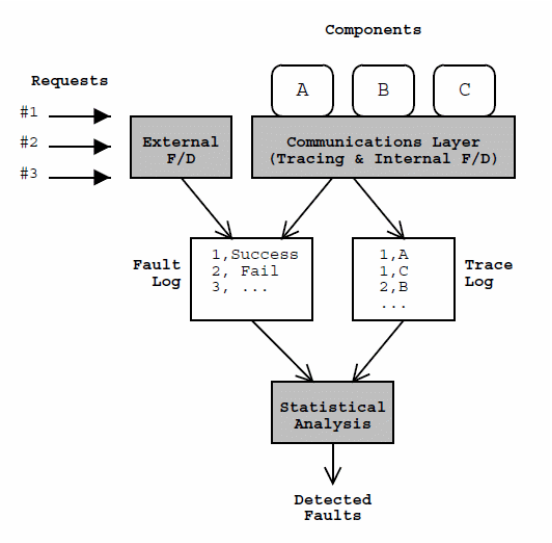

Abbildung 3: Pinpoints Architektur

3. Data Clustering Analysis: Auswertung der protokollierten Daten, um fehlerhafte Einzelkomponenten & fehlerhafte Interaktionen zu ermitteln.

Diese Teil-Aufgaben werden detailliert in den folgenden Abschnitten 2.2.2 bis 2.2.4 betrachtet.

2.2.2 Client Request Tracing

Diese Teilaufgabe beinhaltet das Aufzeichnen aller Komponenten die bei jedem Request benutzt werden. Alternativ ist auch das Aufzeichnen aller Maschinen (grob) oder Dateien (fein) möglich, je nachdem in welcher Granularität man nach Fehlern sucht. Es wird jedem ankommenden Requests eine ID zugewiesen.

Um das Ganze zu realisieren ohne die System-Komponenten zu berühren wird die Middleware instrumentiert[1].

Anschaulich wird das eben Beschriebene noch einmal in Abbildung 4 dargestellt.

2.2.3 Failure Detection

Bei der Failure Detection geht es darum aufzuzeichnen welche Requests Fehler erzeugen und welche nicht. Das Ganze soll vollkommen unabhängig vom Client Request Tracing funktionieren, lediglich die Request-IDs müssen mit denen aus dem Client Request Tracing identisch sein.

Wie ein fehlerfreier Request protokolliert wird, ist in Abbildung 5 zu sehen.

[1] Instrumentierung bedeutet Erweitern des Quellcodes, so dass dieser noch seine eigentlichen Aufgaben erfüllt, zusätzlich jedoch weitere Aufgaben (z. B. die Erstellung von Logs).

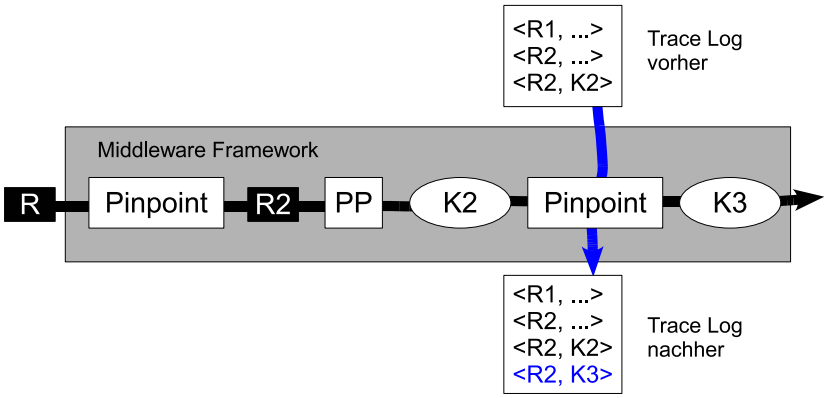

Abbildung 4: Client Request Tracing: Pinpoint weist dem anonymen Request "R" die ID "R3" zu und erweitert vor dem betreten jeder Komponente den Trace Log.

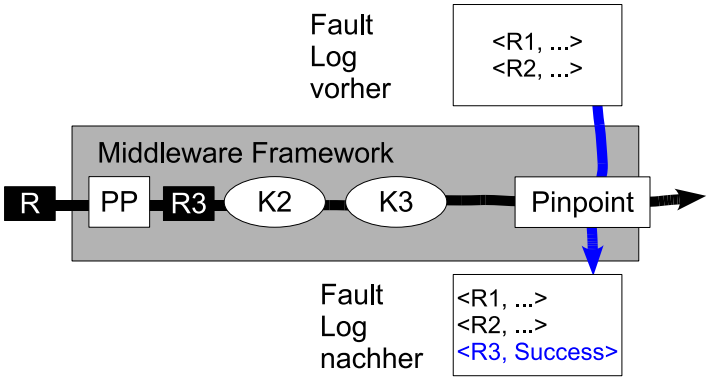

Abbildung 5: Failure Detection: Erfolgreiche Requests werden am Ende der Abarbeitung einfach protokolliert.

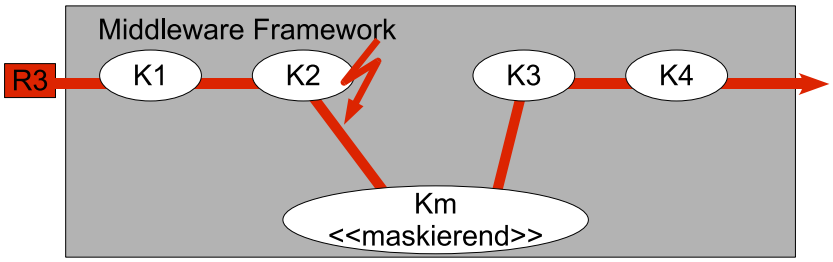

Abbildung 6: Interner Fehler ohne Pinpoint: In K2 entsteht ein Fehler, dieser wird von Km maskiert und der Client erhällt einen scheinbar fehlerfreien Response.

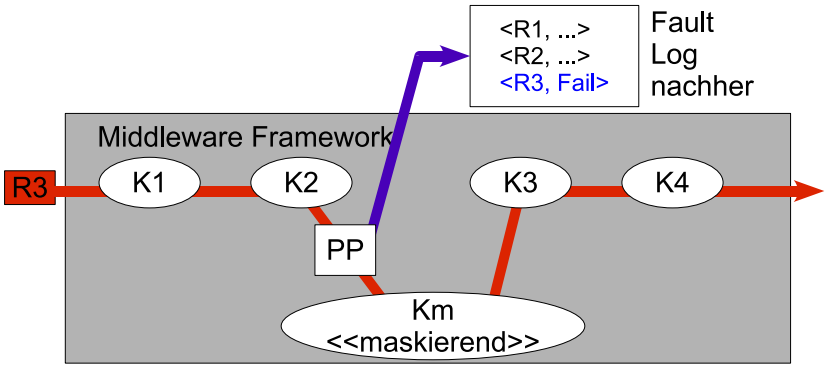

Abbildung 7: Interner Fehler mit Pinpoint: In K2 entsteht ein Fehler, dieser wird von Pinpoint erkannt und protokolliert bevor das System seinen normalen Programmverlauf fortsetzt.

Im Fehlerfall wird zwischen zwei Fehlerarten unterschieden:

Interne Failures werden maskiert bevor sie den Client erreichen. Diese Maskierung ist in Abbildung 6 skizziert. Diese Fehler werden im Communications Layer von Pinpoint (vlg. Abbildung 3) durch Instrumentierung der Middleware ermittelt. Wie Pinpoint interne Fehler protokolliert ist in Abbildung 7 noch einmal veranschaulicht.

Externe Failures sind z.B. Maschinen-Crashs oder veraltete Links. Abbildung 8 zeigt den Verlauf externer Fehler grafisch. Diese Fehler werden in der Pinpoint-Komponente “External F/D” ermittelt (vlg. Abbildung 3). Wie Pinpoint externe Fehler protokolliert ist in Abbildung 9 noch einmal veranschaulicht. Dabei ist zu beachten, dass “External F/D” den Requests selbst eine ID zuweisen kann, sollten diese noch keine erhalten haben.

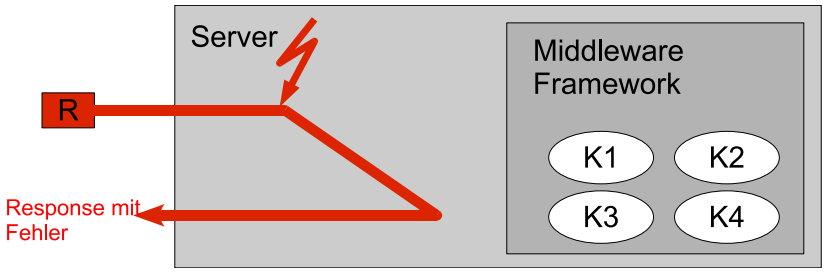

Abbildung 8: Externer Fehler ohne Pinpoint: Ein Fehler sorgt dafür, dass keine der Komponenten überhaupt betreten wird. Der Client erhält einen Fehler-Response.

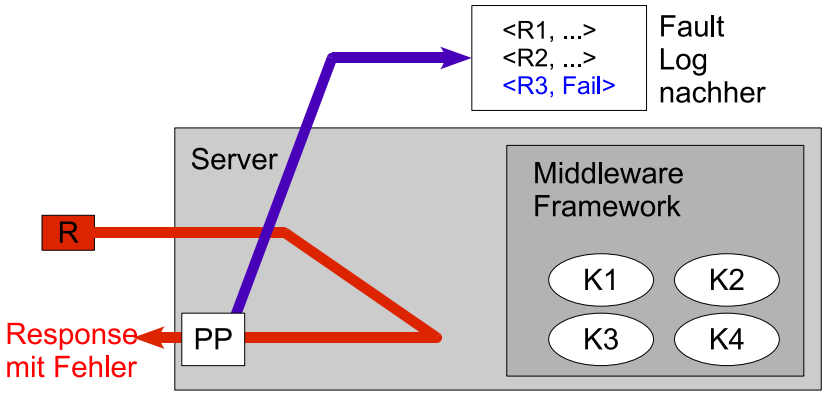

Abbildung 9: Externe Fehler mit Pinpoint: Bevor das System einen Fehler-Response zum Client sendet, wird eine ID (falls nötig) für den Request erzeugt und der Fehler im Fault Log protokolliert.

Client Request ID	Failure	Component A	Component B	Component C
1	0	1	0	0
2	1	1	1	0
3	1	0	1	0
4	0	0	0	1

Tabelle 1: Data Analysis: Vereinigung beider Logs, um daraus Fehlerquellen zu ermitteln

2.2.4 Data Analysis

Diese Teilaufgabe besteht darin, die Trace Logs und Failure Logs zusammenzuführen und auszuwerten. Die vereinigten Logs könnten beispielsweise wie in Tabelle 1 aussehen.

Um zu berechnen wie oft Komponenten (oder Kombinationen) an Fehlern beteiligt sind wird ein Data Clustering Algorithmus benutzt. Dieser ist im Paper nicht näher erklärt. Er liefert jedoch ähnliche Ergebnisse wie die Fehlerquoten aus Abschnitt 2.2, weswegen diese Vereinfachung hier genügen soll.

2.3 Technische Umsetzung

Dieses Thema fällt im original Paper deutlich länger aus, da die Idee in Abschnitt 2.2 jedoch genauso gut auf jeder anderen technischen Plattform denkbar ist, wird hier nur sehr kurz darauf eingegangen, wie diese Idee technisch realisiert wurde.

Zur Umsetzung wurde ein Prototyp des Communication Layers (vgl. Abb. 3) auf J2EE Middleware implementiert. Dieser realisiert das bereits erklärte Client Request Tracing und die interne Fault Detection. Des Weiteren bietet er die Möglichkeit Fehler in System-Komponenten zu injizieren, um Pinpoint damit zu verifizieren. Die Verifikation wird in Abschnitt 2.4 zusammengefasst.

Für die externe Failure Detection wurde ein sogenannter “Network Sniffer” erstellt. Dieser prüft unabhängig vom Framework die ausgehenden TCP- und HTTP-Responses nach Fehlercodes.

Der “Analyzer” benutzt nicht näher erläuterte Data Clustering Algorithmen.

2.4 Evaluation

Zur Bewertung der geleisteten Arbeit wurde Pinpoint auf den e-commerce service “J2EE PetStore demonstration application” angewendet.

Der Communication Layer von Pinpoint hat dabei zu Testzwecken 4 Typen von Fehlern in die Komponenten des PetStore injiziert. Diese Fehlerinjektion musste natürlich getrennt von der Fault Detection ablaufen, um diese zu verifizieren. Gleichzeitig hat ein Client Browser Emulator generierte Requests an den PetStore geschickt.

Insgesamt wurden 133 Tests für Einzelkomponenten-Fehler und Interaktions-Fehler durchgeführt. Abbildung 10 zeigt die wichtigsten Ergebnisse. Anzustre-

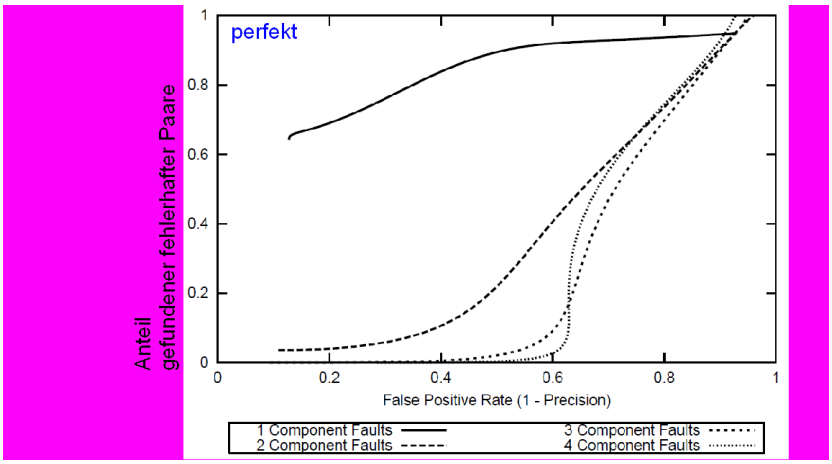

Abbildung 10: Auswertung von Pinpoint: Weit links sind alle von Pinpoint gemeldeten Fehler auch wirklich Fehler, weit rechts sind dagegen viele Fehlalarme dabei. Weit oben wird ein großer Anteil der wirklichen Fehler auch gefunden, weit unten werden dagegen viele von Pinpoint Fehler übersehen.

ben ist hier eine Kurve, die möglichst weit in die obere linke Ecke des Diagramms reicht. In dieser Ecke würden einerseits alle Fehler gefunden werden (y-Achse), und anderseits keine Fehlalarme, also falsche Fehlermeldungen, ausgelöst werden (x-Achse). Pinpoint findet, je nach eingestellter Sensibilität, um die 65 - 95% aller Fehler, jedoch sind 10 - 90% der gefundenen Fehler Falschmeldungen. Im Diagramm sieht man auch, dass Pinpoint bei Kombinationen mehrerer Komponenten deutlich schlechter abschneidet, als bei der Fehlersuche auf Einzelkomponenten.

Des Weiteren vergleichen die Autoren Pinpoint mit zwei älteren und einfacheren Analyse-Verfahren: dedection und dependency checking. Diese werden im original Paper gut beschrieben und auch der Vergleich ist sehr ausführlich. Hier sei nur gesagt, dass Pinpoint bei allen Vergleichen besser abschneidet, oder nur minimal schlechter ist, als die beiden anderen Verfahren.

Integriert man Pinpoint in ein laufendes System, so geht natürlich wertvolle Rechenleistung für die Analyse durch Pinpoint verloren. Diese Rechenleistung sind jedoch gerade mal 8,4%, da die Data Analysis (siehe Abschnitt 2.2.4) offline vollzogen werden kann. Die Logfiles verbrauchen pro Request 2,5kB Speicher oder in komprimierter Form sogar nur 100 Byte.

2.5 Einschränkungen & Ansätze

Pinpoint ist nicht in der Lage alle Fehler zu finden. Welche Fehler Pinpoint nicht finden kann, und wie man Pinpoint erweitern müsste, um sie doch finden zu können, soll hier kurz zusammengefasst werden.

Komponenten die immer zusammenarbeiten sind für Pinpoint nicht unter-

scheidbar. Gefundene fehlerhafte Komponentengruppen sind daher immer eine Obermenge der tatsächlich fehlerhaften Komponenten. Die Autoren schlagen synthetische Requests vor, welche die Komponenten in anderen Kombinationen ansteuern, als möglichen Lösungsansatz vor.

Pinpoint analysiert jeden Request einzeln. Die Gründe für Fehler die in einer Folge von Requests auftreten (Beispiel: Login ist nicht möglich, wenn Registrierung schon fehlschlug), können so nicht erkannt werden. Ein möglicher Ansatz wäre es, zusätzlich zu den genutzten Komponenten auch den Speicher (z. B. die Datenbank) zu protokollieren. So könnte man mittels gemeinsam genutzter Daten Zusammenhänge zwischen verschiedenen Requests herstellen.

Auch "krankhafte Eingaben" (z.B. fehlerhafte Cookies / defekter Browser) sind nicht von Systemfehlern zu unterscheiden. Ein Ansatz wäre hier, zusätzlich zu den benutzten Komponenten, die Requests selbst aufzuzeichnen.

Sogenannte „fail-stutter" faults (d. h. Performance-Verlust durch maskierte Fehler) findet Pinpoint auch nicht. Hierfür müssten Timing Informationen genutzt werden.

3 Bewertung

Die automatisierte Fehlersuche ist ein Teil des autonomic computing, den Pinpoint zu verwirklichen versucht. Dabei liefert Pinpoint, aus der Sicht des Autoren der Zusammenfassung, eher mittelmäßige Ergebnisse, denn um 90% der realen Fehler mit Pinpoint zu finden, muss man in Kauf nehmen, dass man in 90% der Fälle nach Fehlern sucht, wo keine sind. Stellt man sich jedoch vor, dass auch die Fehlerbehebung automatisiert abläuft, so scheinen 90% false positives akzeptabel. Außerdem ist das vorgestellte Pinpoint nur ein erster Schritt, für den bereits einige sinnvolle Erweiterungen vorgeschlagen wurden und auch schon in Arbeit sind. Also auch wenn diese Technologie noch nicht ausgereift ist, so hat sie doch deutliches Potential für Verbesserungen.

Literatur

[Chen] Chen, Mike et al.: *Pinpoint: Problem Determination in Large, Dynamic Internet Services*