Automatisierte Verlässlichkeitsmodelle verteilter Anwendungen

Stefan Naumann

Technische Universität Chemnitz, Professur Betriebssysteme
stefan.naumann@s2012.tu-chemnitz.de

Zusammenfassung. Verteilte Systeme werden immer größer und dynamischer. Knoten und Prozesse können zur Laufzeit ersetzt, hinzugefügt und entfernt werden. Die Verlässlichkeitsmodellierung dieser Systeme ist mit herkömmlichen oder manuellen Methoden sehr aufwändig. Hier wird gezeigt, dass es möglich ist aus aufgezeichneten Kommunikationsdaten Modelle verteilter Systeme zu erstellen und diese auf ihre Verlässlichkeit hin zu untersuchen.
Dabei werden Graphen automatisiert erzeugt, die das verteilte System darstellen - erst mit einer groben Rechnersicht, später mit Prozessen als Knoten, bis hin zu gerichteten Kanten als Abhängigkeiten zwischen Prozessen. Diese Graphen werden einer Brückenerkennung unterzogen, um Single Points of Failure im System zu finden.

1 Einführung

Computersysteme werden vernetzter, Unternehmen, wie bspw. Soundcloud, setzen für ihre Produkte nicht mehr auf einzelne Großrechner, sondern auf viele (schwächere) Rechner, die nur durch Zusammenarbeit und Kommunikation über ein Netzwerk ihr Ziel erreichen können. Dabei bietet ein Rechner einen Dienst für das System an, der wiederum von den anderen Rechnern benutzt wird, bspw. bietet ein Rechner eine Datenbank an, die von anderen Rechnern im Netzwerk als Datenspeicher benutzt werden kann.

Durch diese Aufteilung eines größeren Produkts in einzelne Dienste wird eine Performanzsteigerung erhofft. Zudem sind einzelne Rechner, welche jeweils Teilaufgaben erfüllen, einfacher zu warten als ein großer Monolith, dessen Ausfall oder Wartung zu einem Gesamtausfall des Systems führen würde.

Die Abschätzung der Verlässlichkeit solcher Systeme wird sowohl für Wissenschaftler als auch für Ingenieure schnell komplex. Nicht nur weil mehrere (eventuell heterogene) Knoten/Rechner im System vorhanden sind, sondern auch, weil eine Dynamik dadurch entsteht, dass zur Laufzeit Knoten ersetzt, ausfallen oder neu hinzugefügt werden können.

Dieser Artikel beschäftigt sich damit von solchen verteilten Systemen automatisiert zur Laufzeit Modelle zu erstellen. Die darin enthaltenen Informationen über die Kommunikation im Netzwerk erlaubt es dieses auf Schwachstellen (bspw. Single Point of Failure) zu untersuchen.

2 Verwandte Arbeiten

Verteilte Systeme können durch Generierung von Abhängigkeiten aus den aktuell auf einem Rechner geöffneten TCP-Verbindungen modelliert werden. Dies benötigt eine synchrone Abfrage der Verbindungen auf allen Computern, was schwer umzusetzen ist.

Uhle [Uhl14] hat eine manuelle Modellierung eines größeren verteilten Systems vorgenommen und ist zu dem Schluss gekommen, dass diese Art sehr umständlich und wenig handhabbar ist.

Außerdem zeigte Uhle, dass es möglich ist aus Verbindungsdaten einen Abhängigkeitsgraphen einer laufenden dynamischen Infrastruktur zu erzeugen. Dabei ist kein menschliches Eingreifen vonnöten. Uhle hat als Fallstudie ein verteiltes System mit Linux-Rechnern und einer Bazooka-Deployment-Infrastruktur. Für die Untersucht hat Uhle das Tool *netstat* verwendet um die geöffneten Verbindungen zwischen Prozessen zu beobachten.

Eine weitere Möglichkeit zur Modellierung verteilter Systeme wäre, die eintreffenden Nachrichten eindeutig zu markieren, und aus der Anfrage resultierende weitere Anfragen ebenfalls mit dieser ID zu markieren, wie es Fonseca et al [FPK+07] vorschlagen. Diese Markierung wird durch ein hinzugefügtes Headerfeld im Protokoll realisiert. Es muss von den Rechnern bzw. Prozessen vermerkt werden, wenn eine markierte Nachricht sie erreicht. Damit ist es möglich den Weg, den eine Anfrage gegangen ist im Netzwerk, nachzuvollziehen. Es ist erkennbar, welche Dienste welche anderen Dienste im Netzwerk verwenden, d.h. von welchen Diensten sie abhängen. Allerdings ist es nicht in jedem Protokoll möglich, weitere Headerfelder hinzuzufügen. Deshalb wird auf die Veränderung von Anfragen verzichtet.

Buzacott [Buz83] stellt einen Algorithmus vor, wie ein gewichteter Graph danach ausgewertet werden kann, ob ein Knoten zu einer bestimmten Zeit unerreichbar ist, weil alle Wege zum dem Knoten versagen. Die Kanten seien dabei mit Werten versehen, die angeben, wie wahrscheinlich es ist, dass eine Kante zu einem bestimmten Moment nicht verfügbar ist. Es lässt sich durch den im Artikel gegebenen Algorithmus berechnen, wie wahrscheinlich ein komplettes Abschneiden von Knoten im Graphen ist. Die Ausfallwahrscheinlichkeiten der Kanten könnten mit dem hier dargestellten Verfahren ermittelt werden, allerdings ist das recht unelegant, da die Verwaltung von zwei Zählvariablen auf beiden Seiten einer Kante (ein Sende- und ein Empfangsereigniszähler pro Seite) genügen würde um die Verfügbarkeit der Kanten zu ermitteln. Hier wird versucht Kommunikation zwischen Rechnern (bzw. Prozessen) in einen Kontext zu setzen, anstelle Verfügbarkeiten von Kanten zu ermitteln.

3 Aufzeichnung der Kommunikation

Mithilfe des Wissens über die Kommunikation zwischen den Rechnern im verteilten System kann erkannt werden, welche Rechner welche Dienste nutzen. Hierzu wird davon ausgegangen werden, dass eine Kommunikation zwischen Rechnern das Benutzen eines Dienstes darstellt.

Um diese Kommunikation abzufangen, wurde ein *Sniffer* implementiert. Dieses Programm wird auf jedem Rechner gestartet um, unter Zuhilfenahme der Bibliothek PCAP, empfangene und gesendete Netzwerkpakete aufzuzeichnen und zu einer zentralen Datenbank zu senden. PCAP (Packet Capture Library) ist eine Bibiliothek, die es einem Programm ermöglicht Netzwerknachrichten, die gesendet oder empfangen werden, mitzuhören. Dabei werden folgende Informationen pro Paket aufgezeichnet:

- HostId (eindeutige Nummer des aufzeichnenden Rechners)
- IP-Adresse des Empfängers
- IP-Adresse des Senders
- Zielport
- Quellport
- Zeitstempel (mit Nanosekunden)
- Anwendungsname (wenn verfügbar)

Jedem Rechner wird eine eindeutige HostId zugeordnet. In einer seperaten Datenbankrelation werden folgende Daten gespeichert, um zu wissen, welche Rechner zum Netzwerk gehören und welche Anfragen von außerhalb an das System gestellt werden:

- HostId
- Hostname
- eine Liste von
  - IP-Adresse
  - Devicename (bspw. eth0)

Für die Auswertung der in der Datenbank gesammelten Daten wurde ein *Compute*-Programm implementiert, welches die Daten ausliest und mithilfe der unten dargestellten Verfahren ein Modell des Systems (Graph) erzeugt und dieses verarbeitet. Das *Compute*-Programm kann unabhängig von den *Sniffer*n ausgeführt werden, sodass eine spätere Auswertung der Daten möglich ist, auch wenn die *Sniffer* zu dem Zeitpunkt bereits abgeschaltet worden sind.

4 Modelle

Jedes der dargestellten Modelle ist in sich eine Momentaufnahme und nimmt ein statisches System an. Während der Aufzeichnung der Kommunikation sollten keine Änderungen im System passieren, d.h. Prozesse sollten nicht gestartet oder beendet werden und es sollten keine Rechner hinzugefügt oder entfernt werden. Die erstellten Modelle können drei verschiedene Detailgrade enthalten:

Die erstellten Modelle werden dabei durch Graphen dargestellt, es wurden drei Detailgrade der Modellierung betrachtet.

- Topologie mit Rechnern als Knoten, Kommunikation als ungerichtete Kante (Topologiegraphen)
- Prozesse als Knoten, Kommunikation als ungerichtete Kante (Prozessgraphen)
- Prozesse als Knoten, Dienstnutzung als gerichtete Kante (Abhängigkeitsgraphen)

Dabei werden alle Prozesse bzw. Rechner, die zum untersuchten verteilten System gehören als Knoten modelliert. Ein zusätzlicher Knoten stellt einen virtuellen Nutzer dar, der die Anfragen an das System sendet.

4.1 Topologiegraphen

In den Topologiegraphen wird Kommunikation zwischen Rechnern als ungerichtete Kante eingetragen. Wird ein Paket gefunden, werden die IP-Adressen zu HostIds aufgelöst, d.h. die beteiligten Knoten im Graphen werden identifiziert. Zwischen den beiden Knoten wird eine ungerichtete Kante eingefügt. Es entsteht genau ein Graph bei der Auswertung eines Netzwerktrace eines verteilten Systems.

4.2 Prozessgraphen

Bei der Erstellung der Prozessgraphen besteht die Schwierigkeit darin, die Prozesse als solche zu Erkennen. Ein Prozess wird auf einem Rechner ausgeführt und hat eine bestimmte Portnummer, angenommen diese ändert sich nicht. So ist es möglich Prozesse zu erkennen, d.h. durch den Rechnernamen und den Port eindeutig zu bezeichnen.

Das Verfahren basiert auf der Annahme, dass ein Empfangsereignis zu Sendeereignissen am gleichen Prozess führt. Um abschätzen zu können, welche folgenden gesendeten Nachrichten von dem Prozess gesendet wurden, der eine Nachricht erhalten hat, wird innerhalb eines Zeitintervalls ab Erhalt einer Nachricht nach gesendeten Nachrichten gesucht. Die beliebig konstante Zeitkonstante μ (mit $\mu > 0$) gibt dabei die Länge des Intervalls an.

Zur Erzeugung des Modells wird eine nach Zeitstempel sortierte Liste der Trace-Paketdaten durchsucht. Wird eine Nachricht gefunden, die noch nicht betrachtet wurde, wird eine ungerichtete Kante eingefügt. Am Zielknoten der Kommunikation wird nach weiteren Nachrichten gesucht, die innerhalb eines Zeitintervalls $[t, t + \mu]$ (t ist der zur Nachricht zugeordnete Timestamp, siehe Abschnitt 3) an andere Prozesse versendet wurden.

Es entsteht ein Graph pro Durchlauf mit konstantem Zeitwert μ.

4.3 Abhängigkeitsgraphen

Nutzt ein Prozess die Dienste eines anderen, wird dies als Abhängigkeit bezeichnet (d.h. der eine Prozess ist vom anderen Prozess abhängig). Die Nutzung von Diensten sei nur über Netzwerkkommunikation möglich, auch dann, wenn beide Prozesse auf der gleichen Maschine ausgeführt werden. Außerdem wird angenommen, dass jede Anfrage beantwortet wird. Jede weitere Anfrage vom Serverprozess ist notwendig um die erhaltene zu beantworten, d.h. es besteht ein kausaler Zusammenhang zwischen der erhaltenen Anfrage und kurz darauf versendeten Anfragen von einem Prozess.

Für die Darstellung dieses Modells wird ein Graph verwendet, wobei die Knoten Prozesse und die gerichteten Kanten eine Abhängigkeit ($A \rightarrow B \Rightarrow A$ ist abhängig von B) symbolisieren. Das Verfahren beginnt bei einer ins System eintreffenden Nachricht (einer Anfrage vom Nutzer) und geht rekursiv Wege durch das System ab. Wird eine Nachricht an einen Prozess erkannt, wird in der Liste von Nachrichten nach weiteren Nachrichten gesucht, die vom aktuellen Prozess aus innerhalb eines Zeitintervalls μ gesendet werden. Für jede Anfrage von außerhalb wird so ein neuer Graph erzeugt.

Abhängigkeitsgraphen ermöglichen das Erkennen von Pfaden im Netzwerk, die eine Anfrage wahrscheinlich genommen hat. Dadurch kann abgeschätzt werden, wie sich Ausfälle von Prozessen oder Verbindungskanten tatsächlich auf genau diese Anfrage auswirken würden.

Der Algorithmus könnte noch verbessert werden, indem nur solange bei einem Knoten nach folgenden Nachrichten gesucht wird, bis eine nächste Anfrage diesen Rechner erreicht. So könnten für größere μ die Anzahl der Kanten, die nicht kausal von der Anfrage abhängig sind, im Graphen verringert werden.

Im Algorithmus findet keine Prüfung statt, ob eine folgende Nachricht auch wirklich vom gleichen Prozess versendet wird. Dazu wäre es notwendig auf dem Rechner weitere Untersuchungen anzustellen und Prozesse genauer zu beobachten, um herauszufinden, ob tatsächlich dieser Prozess die Anfrage versendet hat und auch ob dieser Prozess das auch ohne das Empfangsereignis getan hätte. Das wiederum könnte schnell zu einer sehr großen und sehr unübersichtlichen Datenansammlung führen.

5 Verarbeitung der Modelle

5.1 Brücken und Artikulationspunkte

In [Sch13] wird ein Algorithmus zur Erkennung von Brücken und Artikulationspunkten in zusammenhängenden einfachen Graphen beschrieben, der auf die oben dargestellten Modelle anwendbar ist.

Eine Brücke in einem Graphen ist eine Kante, deren Entfernen den Graphen in wenigstens zwei Teilgraphen zerfallen lässt; ein Artikulationspunkt ist ein Knoten, dessen Entfernen den Graphen in wenigstens zwei Teilgraphen zerfallen lässt. Der minimale Grad $\delta(G)$ gibt an, wie viele Ausgangskanten ein Knoten im Graphen mindestens aufweist.

Im vorgestellten Algorithmus wird eine Tiefensuche über dem ungerichteten Eingabegraphen durchgeführt. Alle Baumkanten werden zu den Blättern hin gerichtet, alle Kanten, die im Graphen aber nicht im Tiefensuchbaum enthalten sind, werden zur Wurzel hin gerichtet eingefügt. Allen Knoten wird eine von der Wurzel zu den Blättern aufsteigende Nummer zugeordnet und werden in dieser Reihenfolge (aufsteigend) besucht. Vom aktuellen Knoten werden nacheinander alle Rückkanten (also zu den Blättern führenden Kanten) verfolgt, und schließlich ein Weg zurück zur Wurzel gesucht. Die besuchten Kanten und Knoten werden markiert.

Die so gefundenen Wege werden Ketten (Chain) C_i genannt. Dieses Verfahren berechnet die Menge aller Ketten $C = C_1 + C_2 + \cdots + C_n$. Es können anhand der Menge der Ketten folgende Eigenschaften für Knoten und Kanten abgelesen werden:

- G sei einfach. G heißt 2-kantenzusammenhängend genau dann wenn C die Kanten von G partitioniert, also restlos in Ketten einteilt.
- G sei 2-kantenzusammenhängend. G heißt 2-knotenzusammenhängend genau dann wenn C_1 der einzige Kreis in C ist.
- Eine Kante ist eine Brücke genau dann wenn sie in keiner Kette von C vorkommt.
- G sei einfach und der minimale Grad im Graphen sei $\delta(G) \geq 2$.[1] Ein Knoten v heißt Artikulationspunkt genau dann wenn v inzident zu einer Brücke ist oder v der erste Knoten eines Kreises in $C \setminus C_1$ ist.

Im Topologiegraphen stellt eine Brücke und ein Artikulationspunkt eine Single-Point-of-Failure dar. Fällt diese Kante bzw. der Knoten aus, zerfällt der Graph in mehrere Teilgraphen, d.h. im verteilten System ist eine Kommunikation zwischen zwei Teilen des Systems nicht mehr möglich.

Der vorgestellte Algorithmus arbeitet nur auf zusammenhängenden Graphen, allerdings ist es möglich, dass das Netzwerk nicht zusammenhängt. Ein Graph kann also aus wenigstens einer Zusammenhangskomponente bestehen. Der Algorithmus kann auf jeder der Komponenten einzeln ausgeführt werden. Dadurch werden Brücken und Artikulationspunkte unabhängig davon zurückgeliefert, ob der Graph zusammenhängt oder nicht.

6 Versuchsdurchführung

Zur empirischen Untersuchung des Ansatzes und der Auswertungsstrategien wurden die implementierten Anwendungen nach dem initialen Testen mit mehreren virtuellen Maschinen anhand eines Fallbeispiels auf echter Hardware untersucht.

[1] Diese Bedingung wird praktisch aufgelöst, da ein Beachten des Grades eines Knoten nicht implementiert wurde.

6.1 Fallbeispiel

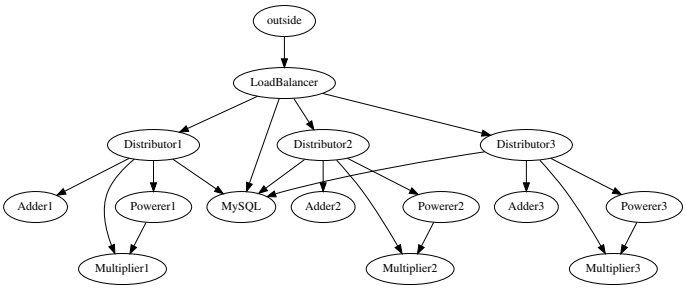


Abb. 1. Aufbau des Testnetzwerks - ein Formelrechner; Knoten symboliseren Prozesse

Abbildung 1 zeigt den logischen Aufbau des Testnetzwerks. Das Netzwerk besteht aus 14 Prozessen und dient als einfacher Formelrechner. Der Knoten „outside" (dt. außerhalb) dient als Verdeutlichung der Nutzerrechner/-prozesse. Dieser Knoten symbolisiert keinen Rechner/Prozess im Netzwerk, sondern stellt in einem Knoten alle anfragenden Rechner/Prozesse dar.

Die zu berechnende Formel besteht aus aneinandergereihten Operationen (Addition, Multiplikation, Potenzrechnung). Die Formel wird als String übermittelt.

Der Loadbalancer überprüft ob die Anfrage bereits gestellt wurde und liefert das gespeicherte Ergebnis aus der MySQL-Datenbank zurück. Existiert kein gespeichertes Ergebnis verteilt er die Anfrage an einen der Distributoren mithilfe des Round-Robin-Prinzips. Eine Fehlertoleranz ist nicht implementiert worden. Der Distributor interpretiert die Formel und verteilt die einzelnen Aufgaben an seine Prozesse, sodass der Adder alle Additionen, der Powerer die Potenzierungen und der Multiplier die Multiplikationen (einschließlich die des Powerers) übernimmt. Das Endergebnis wird vom Distributor in die Datenbank gespeichert. Es gibt drei Distributoren mitsamt der rechnenden Prozesse im System, einen Loadbalancer und eine MySQL-Datenbank.

Die Prozesse unterhalb eines Distributors werden nur von dem einen Distributor verwendet, d.h. Adder1 nur von Distributor1, Adder2 nur von Distributor2, etc.

6.2 Versuchsaufbau

In Abbildung 1 ist die logische Topologie des Testnetzwerks dargestellt. Dabei sind Prozesse zu sehen, die auf Rechner aufgeteilt werden müssen. Die Rechner sind allerdings sternförmig an einen Switch angeschlossen und in einem VLAN zusammengefasst. Der Switch kann alle Nachrichten dieses VLANs an einen beliebigen Port spiegeln. Allein auf diesem PC läuft die *Sniffer*-Anwendung, die die Netzwerkkarte im promiscuous mode betreibt um alle Nachrichten des VLANs mithören zu können. Auf diesem PC wird auch die Datenbank betrieben, in der die Nachrichten gespeichert werden. Ebenfalls dargestellt ist der Knoten „outside", der die Anfragen an das Netzwerk stellt.

Die implementierte *Sniffer*-Anwendung war nicht für einen solchen Einsatz gedacht, daher müssen vorab die Informationen über die Rechner (IP-Adressen, Namen, etc.) manuell gesammelt

und in die Datenbank eingetragen werden. Zudem ist eine Abfrage des Prozessnamens, der eine Nachricht versendet hat, nicht möglich.

Bei der Aufteilung der Prozesse auf Rechner muss außerdem besonderer Wert darauf gelegt werden, dass Nachrichten nicht lokal ausgeliefert werden können, weil diese sonst den Switch und damit den *Sniffer* nicht erreichen. D.h. alle Prozesse, die in Abbildung 1 direkte Nachbarn sind, dürfen nicht auf demselben Rechner ausgeführt werden.

Beim eigentlich angedachten Vorgehen, einen *Sniffer*-Prozess pro Rechner zu starten, wäre es möglich gewesen die Datenbankanwendung mehrfach zu instantiieren, d.h. bspw. pro Rechner einmal, sodass jeder *Sniffer* auf seinen eigenen Datenbankserver zugreift (*localhost*) um einen eventuellen Flaschenhalseffekt an der Datenbank zu verringern. Das ist hier nicht ohne Weiteres möglich, es müsste der *Sniffer* so implementiert werden, dass er nebenläufig nach Round-Robin-Prinzip auf mehreren Datenbankservern jeweils einen Teil der Nachrichten speichert.

6.3 Durchführung

Der Versuch besteht aus neun Rechnern, die das im Fallbeispiel definierte verteilte System ausführen. Es gibt einen zusätzlichen Rechner, der automatisiert Anfragen stellt. Das System verwendet das TCP-Protokoll.

Alle Anwendungen des verteilten Systems werden gestartet, d.h. die Verbindungen zum MySQL-Server bestehen zum Start der Aufzeichnung bereits. Nun wird die Aufzeichnung begonnen und das Skript auf dem „outside"-Rechner gestartet. Dieses stellt im Abstand von 10 Millisekunden vordefinierte Anfragen.

TCP hat die Eigenschaft, dass beim Verbindungsaufbau zu einem Port für die weitere Kommunikation der Port gewechselt wird. Das führt zu Problemen Kommunikation akkurat abzubilden. Daher wird der Versuch zweimal durchgeführt. Dabei werden:

- im 1. Durchlauf alle Nachrichten mit gesetztem TCP-PUSH-Flag aufgezeichnet
- im 2. Durchlauf alle Nachrichten mit ausschließlich gesetztem TCP-SYN-Flag aufgezeichnet.

Im ersten Durchlauf werden also alle Payload-Pakete aufgezeichnet[2], im zweiten nur die Pakete, die eine TCP-Verbindung initialisieren. Es wird davon ausgegangen, dass ein Prozess, der eine Verbindung aufbaut auch später eine Anfrage stellen will.

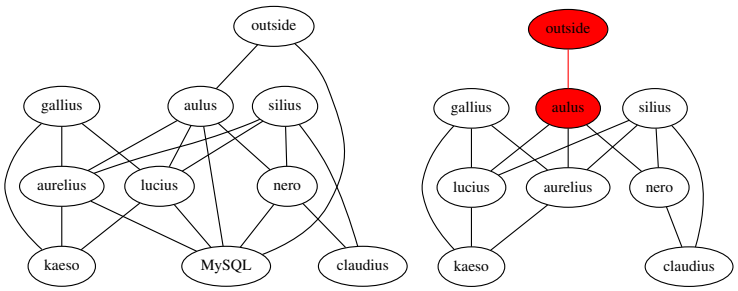
Testfälle Das Testskript sendet automatisch Anfragen an das System mit einem Abstand von 10 Millisekunden. Dabei wird kein Wert darauf gelegt bestimmte Anfragen zu versenden, es geht nur um die Menge. Es werden Formeln der Form $a \hat{} b * c + d$ an den Loadbalancer gesendet. Dieser wählt einen Distributor für die Weiterverarbeitung aus. Bei jeder Anfrage werden vom Distributor alle Rechenprozesse angefragt, auch dann, wenn die Variablen a, b, c oder $d = 0$ sind.

7 Ergebnisse

Es folgt die Auswertung der automatischen Modellierung und der Brücken- und Artikulationspunkterkennung anhand der beiden durchgeführten Versuchsdurchläufe. Rote Kanten sind erkannte Brücken, rot ausgefüllte Knoten stellen Artikulationspunkte dar. Die Graphen wurden mithilfe des GraphViz-Tools visualisiert.

[2] davon ausgehend, dass alle Payload-Nachrichten mit gesetztem PUSH-Flag versendet werden

7.1 Topologiegraph



(a) Topologiegraph erstellt mit PSH-Nachrichten

(b) Topologiegraph erstellt mit SYN-Nachrichten

Abb. 2. Topologiegraphen

Abbildung 2 zeigt beide resultierende Topologiegraphen. Wie erwartet enthält der SYN-Graph den MySQL-Knoten nicht. Im PSH-Graphen ist eine Kante zwischen „outside" und dem MySQL-Knoten enthalten. Das könnte Kommunikation zwischen dem Switch oder einem nicht im System befindlichen Rechner und dem Rechner mit dem MySQL-Server sein. Eine Verbindung zu MySQL wird im Testskript nicht aufgebaut.

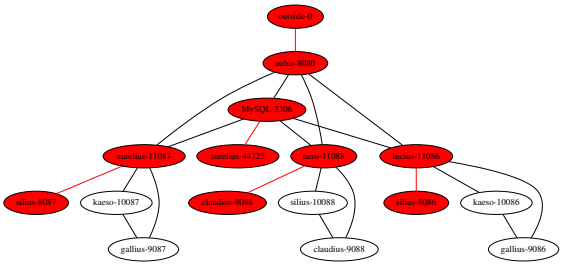
Beide Graphen stellen die verwendeten Kanten zwischen den Rechnern dar. Die Brücken- und Artikulationspunkterkennung bringt keine sinnvollen Ergebnisse hervor. Nur im SYN-Graphen werden überhaupt Brücken erkannt, nämlich die zwischen „outside" und dem Loadbalancer. Durch die dichte Vermaschung kann der Algorithmus keine Brücken bzw. Artikulationspunkte im Netzwerk erkennen.

7.2 Prozessgraphen

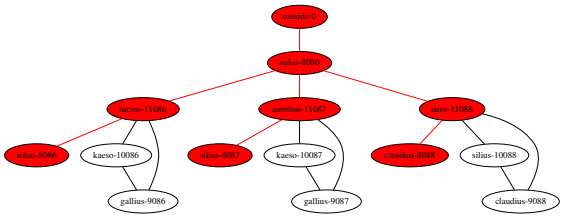
Abbildung 3 zeigt Prozessgraphen mit unterschiedlichen Zeitkonstanten μ. Das Verfahren wurde mit den Zeitkonstanten 1 ms, 5 ms, 10 ms und 50 ms, 500 ms, 5000 ms und 50000 ms durchgeführt. Die besten Ergebnisse lieferte der Threshold von 10 ms, was exakt der Zeit entspricht, die das Testskript zwischen zwei Anfragen wartet. In der PSH-Variante mit 10 ms Threshold erscheint eine Kante vom „MySQL" Knoten zum Rechner „aurelius" auf Port 44725. Das weißt darauf hin, dass hier eine Anfrage eventuell recht lang gedauert hat, somit dessen Sendeereignis unabhängig vom früheren Empfangsereignis gesehen wurde.

Graphen mit zu kleinem Threshold μ enthalten sehr viele Knoten und Kanten. Es zeigt sich, dass der Algorithmus Empfangs- und Sendeereignis nicht zusammenfassen kann, weil diese weiter als μ voneinander entfernt liegen. Hohe Thresholds führen zu falsch erkannten Kanten. Hier wird das nächste Sendeereignis noch zum früheren Empfangsereignis zugeordnet, obwohl eigentlich kein kausaler Zusammenhang besteht.

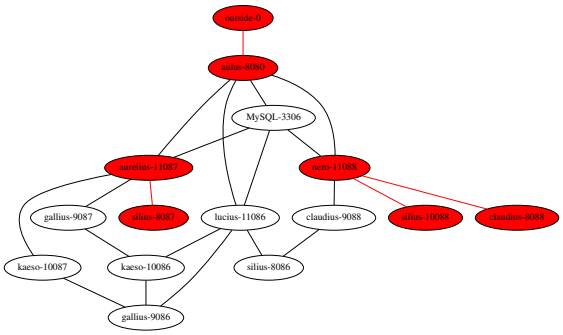
In realen Systemen mit variablen Verarbeitungszeiten oder wenn mehrere Anfragen nebenläufig im System bearbeitet werden ist es eventuell sehr schwer den Threshold passend zu wählen.



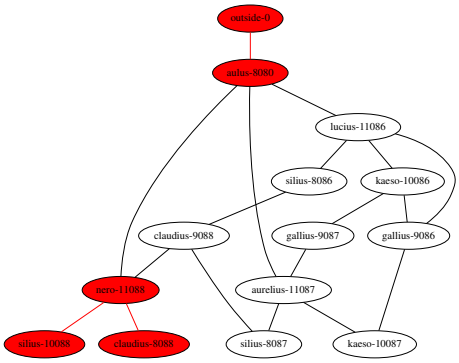
(a) Prozessgraph PSH-Nachrichten, $\mu = 10000$ Mikrosekunden (10 ms)



(b) Prozessgraph SYN-Nachrichten, $\mu = 10000$ Mikrosekunden (10 ms)



(c) Prozessgraph PSH-Nachrichten, $\mu = 50000$ Mikrosekunden (50 ms)



(d) Prozessgraph SYN-Nachrichten, $\mu = 50000$ Mikrosekunden (50 ms)

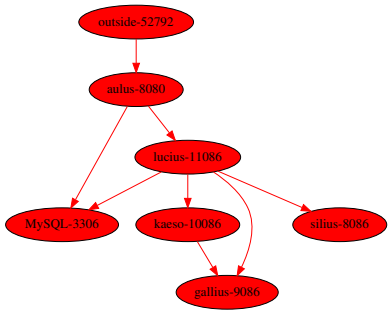
Abb. 3. Prozessgraphen mit unterschiedlichen Thresholds μ

7.3 Abhängigkeitsgraphen

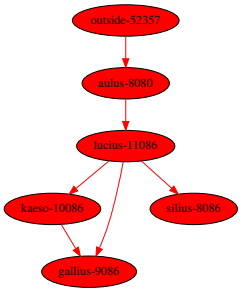
Mit jeder Anfrage an das System entsteht ein Graph, weshalb hier nur einige ausgewählte gezeigt werden können. Da die Kanten gerichtet sind, ist eine sinnvolle Brückenerkennung nicht mehr möglich, d.h. alle Kanten werden als Brücken und alle Knoten als Artikulationspunkte erkannt. Es gibt keine Abhängigkeiten im Netzwerk zu höheren Prozessen (d.h. keine Kreise). Das Verfahren erkennt auch keine Redundanzen.

1000 Mikrosekunden Es ist beim Threshold von 1 ms zu erkennen, dass der PSH-Graph eine Kante von „aulus" zu „MySQL" beinhaltet. Es kann also angenommen werden, dass in 1 ms der Loadbalancer eine Anfrage an den MySQL-Datenbankserver gestellt hat, aber im selben Zeitintervall keine Antwort bekommen hat bzw. keine weitere Anfrage an einen Distributor stellen konnte. Dieser Threshold ist zu klein um sinnvolle Aussagen über das System anstellen zu können.

5000 Mikrosekunden Mit 5 ms Threshold zeigt sich eine starke Varianz in den Graphen. Einige Graphen zeigen alle Kanten, die von der Anfrage genommen wurden. Andere stellen nur die Verwendung des Powerers und des Multipliers, wieder Andere den Weg bis zum Distributor. Fünf Millisekunden reichen somit nicht aus um dieses System akkurat zu erfassen.

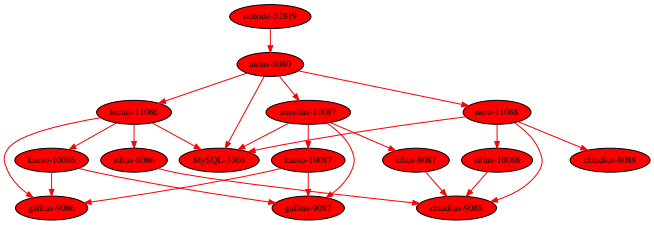


(a) Abhängigkeitsgraph mit PSH-Nachrichten

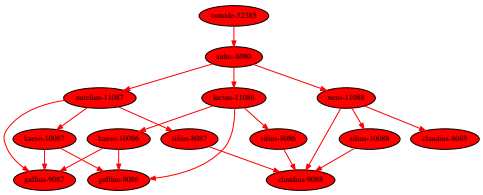


(b) Abhängigkeitsgraph mit SYN-Nachrichten

Abb. 4. Abhängigkeitsgraphen mit Threshold $\mu = 10000$ Mikrosekunden (10 ms)



(a) Abhängigkeitsgraph mit PSH-Nachrichten



(b) Abhängigkeitsgraph mit SYN-Nachrichten

Abb. 5. Abhängigkeitsgraphen mit Threshold $\mu = 50000$ Mikrosekunden (50 ms)

Der PSH-Graph stellt den Weg einer Anfrage dar, allerdings fehlt bei den meisten Graphen die Kante vom Power-Prozess (Port 1008X) zum Multiplier-Prozess (Port 908X). Im Algorithmus wird bei jedem neuen Knoten nach Nachrichten gesucht, die 5000 Mikrosekunden nach Erhalt einer Nachricht versendet worden. Daher erschließt sich nicht ganz, warum gerade diese Kante fehlt. Der Distributor wartet bis eine Anfrage beantwortet wurde, und sendet erst dann eine neue, sodass es durchaus passieren kann, dass nicht alle drei Anfragen in dieses Zeitintervall fallen, allerdings passiert das beim Power-Prozess nicht.

10000 Mikrosekunden Abbildung 4 zeigt mit einem Threshold von 10000 Mikrosekunden (10 ms) erstellte Abhängigkeitsgraphen. Sowohl die PSH- als auch die SYN-Graphen sind sehr akkurate Darstellungen des Systems. In der PSH-Variante gibt es einige wenige Ausnahmen, die die Einfüge-Operation an den MySQL-Server vom Distributor nicht verzeichnet haben. Die Abhängigkeiten zwischen den Prozessen sind dargestellt, auch vom Loadbalancer und Distributor vom MySQL-Server (in der PSH-Variante).

Prozessen wird hier genügend Zeit eingeräumt Anfragen zu bearbeiten und weitere zu stellen. Außerdem fällt ins Gewicht, dass Anfragen nie nebenläufig im System sind. In einem echten System kann diese Annahme nicht getroffen werden.

50000 Mikrosekunden Abbildung 5 zeigt zwei mit $\mu = 50$ ms erstellte Abhängigkeitsgraphen. Nachrichten von späteren Anfragen werden mit verarbeitet, sodass ein vollständigeres Bild vom System entsteht. Allerdings sind in beiden Graphen Kanten enthalten, die keine kausale Abhängigkeit zwischen Prozessen darstellen, sondern fehlerhafterweise früheren Empfangsereignissen zugeordnet wurden.

7.4 Fehlerbetrachtung

Beim Versuchsaufbau sind Fehler nicht komplett auszuschließen. Zum Beispiel wäre es möglich, dass einzelne Nachrichten aufgrund eines vollen PCAP-Buffers nicht aufgezeichnet wurden. Unter der Annahme, dass dass gleiche oder ähnliche Anfragen mehrmals an das System gesendet werden (eventuell auch von unterschiedlichen Nutzern), dann ist die Wahrscheinlichkeit, dass jede Nachricht wenigstens einmal aufgezeichnet wurde, hinreichend hoch. Selbst wenn so nicht jede Nachricht gespeichert wird, kann dennoch ein vollständiges Bild des verteilten Systems erzeugt werden.

Eine weitere Fehlerquelle könnte im VLAN-Mirroring liegen. Will Prozess / Rechner A mit B kommunizieren, so sendet er die Nachricht über die Netzwerkkarte (NIC, Network Interface Card). Die Nachricht wird über den Switch an B weitergeleitet, mit aktiviertem Mirroring auch an den Capture PC. Es ist nicht auszuschließen, dass der Switch die Weiterleitung an den Capture PC zeitversetzt vornimmt, d.h. erst die Nachricht an B ausliefert, dann (eventuell sogar nachdem etwas Zeit vergangen ist), die Nachricht an den Capture PC weiterleitet. Eine Rekonstruktion des eigentlichen Zeitpunktes, wann die Nachricht versendet wurde bzw. wann sie ankommt, ist nicht möglich. So können sich Probleme bei den Algorithmen der Prozesssicht und der Abhängigkeitsgraphen ergeben, da diese Algorithmen auf Zeit basieren um einen Zusammenhang von einem Sendeereignis zu einem Empfangsereignis zu erkennen.

Somit können weder Empfangs- noch Sendeereignisse genau terminiert werden, beide erhalten für eine Nachricht den gleichen Zeitstempel, wenn sie den Capture PC erreicht. Eine Trennung der beiden Ereignisse ist nicht mehr möglich. Würden auf allen Computern die Nachrichten aufgezeichnet, müsste eine sehr genaue Uhrensynchronisation durchgeführt werden. Eine perfekte Synchronisation ist nicht möglich, weshalb auch hier Fehler entstehen würden.

8 Ausblick

In zukünftiger Forschung könnten weitere Modelle oder Ansätze ausgewertet werden, bspw. könnte die Verfügbarkeit von Kanten unter der Zuhilfenahme von historischen Daten untersucht werden. Dabei könnte es genügen zu zählen wie viele Pakete gesendet worden sind, und wie viele beim Empfänger tatsächlich angekommen sind.

Mit Multigraphen, bei denen jeder Knoten einen Rechner (oder Prozess) darstellt, und jede Kante eine Kommunikation zwischen zwei Knoten über einen bestimmten Port darstellt, könnte es möglich sein abzuschätzen, wie viele Ausgangskanten ein Rechner hat, d.h. von wie vielen anderen Rechnern dieser benötigt wird. Rechner mit vielen Kanten sind offensichtlich im System sehr wichtig, und sollten durch StandBy-Rechner oder andere Fehlertoleranzmaßnahmen gesichert sein.

Anhand der Länge von Paketen kann untersucht werden, auf welchen Wegen durch das System wie viele Daten versendet werden. Die Wege mit besonders hoher Last sind für das System offenbar sehr wichtig, und sollten daher besonders verstärkt werden.

Anhand der Informationen, wann eine Anfrage eingetroffen ist, und wann eine Antwort gesendet wurde, d.h. an den gleichen Port des Anfragenden, kann es möglich werden Bearbeitungszeiten im System abzuschätzen und die langsamsten Knoten zu identifizieren. Diese Knoten könnten dann durch stärkere Hardware ersetzt werden, oder die Software kann restrukturiert werden und ggf. auf mehrere Prozesse aufgeteilt werden.

Die Auswertung der in Abschnitt 7.3 aufgeführten Graphen mit dem Brückenerkennungsverfahren ist zwar formal korrekt[3], aber für die Einschätzung der Verlässlichkeit des Systems wenig hilfreich. Es müssten geeignetere Verfahren gefunden werden um diese Modelle auswerten zu können. Denkbar wäre die Auswertung der Abhängigkeiten zu Fehlerbäumen.

Ein weiteres Forschungsfeld könnte die Erkennung von Prozessen über IP und Port sein. Hier wurde versucht über einen zeitlichen Schwellwert zu erkennen, ob eine gesendete Nachricht mit einer gerade empfangenen Nachricht zusammenhängt. Es könnte untersucht werden, ob es genauere Methoden dafür gibt; eventuell muss dabei auch auf Betriebssystem-Datenstrukturen zurückgegriffen werden.

9 Fazit

In diesem Artikel wurde gezeigt, wie ein verteiltes System durch die Auswertung von aufgezeichneten Kommunikationsinformationen automatisiert modelliert und auf Single Points of Failure untersucht werden kann. Es wurden drei verschiedene Detailgrade beleuchtet: von einer recht ungenauen Sicht (Topologiegraph) über eine genauere (Prozessgraph), bis später Richtung an Abhängigkeiten zwischen Prozessen angetragen werden konnten (Abhängigkeitsgraphen).

Die Topologiegraphen können ebenfalls genutzt werden um zu erkennen, welche Rechner miteinander kommunizieren. Für die Analyse der Verlässlichkeit eines verteilten Systems genügt das aber noch nicht. Die Prozesssicht führt einige Fehler in das Modell ein, dafür kann damit eine genauere Sicht auf das verteilte System erlangt werden. Die Abhängigkeitsgraph-Methode führt, mit dem richtigen Schwellwert dazu, dass Wege betrachtet werden können, die eine Anfrage genommen hat, ohne dabei die Nachrichten verändern zu müssen.

Es wurde mit der Brücken- und Artikulationspunkterkennung ein Verfahren dargestellt, mit dem die Auswirkungen von Ausfällen analysiert werden können. Damit ist es möglich auch dynamische Systeme zur Laufzeit zu evaluieren und zu erkennen, welche Rechner oder Prozesse im verteilten System besonders kritisch sein könnten. Das Verfahren ist für die Abhängigkeitsgraphen nicht sinnvoll einsetzbar.

Literatur

[Buz83] John A Buzacott. A recursive algorithm for directed-graph reliability. *Networks*, 13(2):241–246, 1983.

[FPK+07] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[Sch13] Jens M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters*, 113(7):241 – 244, 2013.

[Uhl14] Johan Uhle. On dependability modeling in a deployed microservice architecture. Master's thesis, Universität Potsdam, June 2014.

[3] unter Lockerung des Artikulationspunktbegriffs