Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit

Planung von verlässlichen Workflowausführungen

Patric Höhn

Studiengang: Softwaretechnik

Prüfer/in: Prof. Dr. K. Rothermel

Betreuer/in: Dipl.-Inf. David Richard Schäfer

Beginn am: 29. September 2016

Beendet am: 31. März 2017

CR-Nummer: C.2.4,C.4

Kurzfassung

Workflows beschreiben den zeitlichen Ablauf eines komplexen Prozesses. In dieser Arbeit beschäftigen wir uns mit deren Ausführung auf fehleranfälligen Systemen. Wir entwickeln Strategien um diese Ausfühung möglichst effizient und optimal zu planen. Die Planung und Ausführung auf verteilten Systemen stehen dabei im Fokus. Unsere Strategien führen die Planung unter anderem verteilt aus oder teilen einen Workflow in mehrere Teile auf, um die Planungszeit zu verkürzen. Wir testen die Strategien in einem simulierten Netzwerk und stellen die Ergebnisse in einer Evaluation detailiert dar. Es zeigt, sich dass sich durch unsere Strategien unter bestimmten Bedingungen die Dauer für die Ausführungsplanung drastisch verkürzen lässt und die Zeit für eine erfolgreiche Workflowausführung dabei verhältnismäßig gering ansteigt, sodass die Gesamtdauer von Planung und Ausführung deutlich verkürzt werden kann.

Inhaltsverzeichnis

1 Einleitung **11**

2 System-Modell **13**
2.1 Netzwerk-Modell 13
2.2 Workflow-Modell 14
2.3 Ausführungs-Modell 15

3 Problemstellung **17**
3.1 eine Gruppe von Aktivitäten ausführen 17
3.2 Synchronisation nach einer Gruppenausführung 18
3.3 zu erwartende Kosten einer Gruppenausführung 19
3.4 zu erwartende Kosten einer Workflowausführung 20
3.5 Ausführungsplan mit minimalen zu erwartenden Kosten 20
3.6 Beispiel 20

4 Strategien zum Erstellen von Ausführungsplänen **23**
4.1 Optimaler Ausführungsplan 23
4.2 Bestes Replikat 25
4.3 Line Of Sight 28
4.4 Beste Gruppen 35

5 Evaluation **41**
5.1 Implementierung des Netzwerkmodells 41
5.2 Implementierung der Strategien 43
5.3 Metriken und Konfiguration 50
5.4 Testumgebung 51
5.5 Ergebnisse 51
5.6 Fazit 59

6 Verwandte Arbeiten **61**

7 Zusammenfassung und Ausblick **63**

Literaturverzeichnis **65**

Abbildungsverzeichnis

2.1 Netzwerk mit drei Execution Engines und drei Services 14
2.2 Workflow mit Aktivitäten, Gruppen und Services 15

3.1 Workflow der Länge 5 in einem Netzwerk mit zwei Execution Engines und zwei Services 21

4.1 Workflow-Graph zum Finden des günstigsten Pfads 24
4.2 Beispiel eines Planungsschritts 31
4.3 Vergleich der Anzahl zu berechnender Gruppenkosten pro Execution Engine in Abhängikeit der gewählten SightLength und der Anzahl der Planungsschritte. Als Referenz dient dieselbe Zahl bei Verwendung von *Bestes Replikat* 33

5.1 Berechnungszeit der Synchronisationszuverlässigkeit einer Execution Engine 52
5.2 Zeit der Ausführungsplanung 53
5.3 Durchschnittliche Anzahl der Gruppen im Ausführungsplan 55
5.4 Overhead 57

Tabellenverzeichnis

4.1 Ablauf der Strategie - Optimaler Ausführungsplan 23
4.2 Ablauf der Strategie - Bestes Replikat 26
4.3 Übersicht eines Planungsschritts 30
4.4 Ablauf der Strategie - Beste Gruppen 37

5.1 Parameter für die Berechnung der Gruppenkosten 50

1 Einleitung

Die Leistungsfähigkeit von mobilen Endgeräten nimmt stetig zu. Im Geiste der Internet of Things ermöglichen diese mittlerweile *Location-Based Services* und andere personalisierte Services und deren Verfügbarkeit nimmt ständig zu [SWT+16][STR16]. Um diese Services auch in Kombination sinnvoll zu nutzen wird das Ausführen von Workflows notwendig. Workflows sind der Standard für automatisierte Geschäftsprozesse und finden in vielen Bereichen Verwendung [SBTR14] [KLL09].

Workflows beschreiben den zeitlichen Ablauf eines komplexen Prozesses. Ein Workflow besteht aus einer Menge von Aktivitäten, welche einzelne Arbeitsschritte darstellen und in einer vorgegebenen Reihenfolge auszuführen sind [SSB+14]. Die resultierenden Abhängigkeiten jeder Aktivität werden dabei durch eine definierte Menge von Vorgängern und eine definierte Menge von Nachfolgern beschrieben. Eine Aktivität läuft entweder als isolierter Prozess ab oder beinhaltet das Aufrufen eines externen Services [SSB+14]. Gibt der Service bei Aufruf einen Wert zurück, kann dieser Wert an die als nächstes auszuführende Aktivität weitergegeben werden.

Enthält ein Workflow externe Serviceaufrufe können diese in Abhängigkeit des zugrundeliegenden Netwerks in erheblichem Maße zur gesamten Ausführungsdauer beitragen. Mobile Umgebungen sind hochdynamisch. Latenzen, Ausfallraten und Bandbreiten können aufgrund der drahtlosen Verbindung und Bewegung der Endgeräte stark schwanken und sind grundsätzlich schlechter als in kabelgebundenen Netzwerken.

Besonders für längere Workflows ist das problematisch. Die Ausführung auf einem mobilen Endgerät schlägt wesentlich häufiger fehl und muss dann komplett neu gestartet werden. Die durchschnittliche Ausführungsdauer wird somit deutlich erhöht.

Diesen Problemen kann man mit Replikation begegnen [SSB+14]. Der Workflow wird auf mehreren Geräten verteilt ausgeführt. Zwischenergebnisse werden dabei unter den Geräten ausgetauscht, um bei Ausfällen nicht komplett von vorn beginnen zu müssen. Die Verlässlichkeit der Ausführung kann optimiert werden, indem Aktivitäten mit Serviceaufrufen auf solchen Geräten ausgeführt werden, welche eine bessere Verbindung zum Service-Server aufweisen, zum Beispiel da sie geographisch nah an diesen befinden.

Um vor der Ausführung eines Workflows festzulegen, welche Aktivitäten auf welchem Gerät im Netzwerk ausgeführt werden sollen und nach welchem Ausführungsschritt das Zwischenergebnis des Workflows synchronisiert werden soll, kann man Ausführungspläne verwenden [SWT+16]. Ein Ausführungsplan fasst Aktivitäten zu Gruppen zusammen und weißt jeder

Gruppe ein Endgerät zur Ausführung zu. Die Endgeräte, welche den Workflow ausführen, werden auch als Execution Engines bezeichnet. Wurde eine Gruppe vollständig ausgeführt synchronisiert die Execution Engine das Ergebnis mit den anderen Geräten der Workflowausführung [SWT+16].

In dieser Arbeit schlagen wir mehrere Strategien zum Berechnen von Ausführungsplänen vor. Anschließend vergleichen wir diese einerseits anhand der durchschnittlichen Ausführungsdauer der Ausführungspläne und andererseits anhand der Zeit, die es im Durchschnitt benötigt mit dem jeweiligen Algorithmus einen Ausführungsplan zu berechnen. Wir betrachten dabei mehrere unterschiedliche Replikationsgrade.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – System-Modell: Wir gehen zunächst auf die Details von einem in dieser Arbeit betrachteten Netzwerks, eines Workflows und der Form eines Ausführungsplans ein.

Kapitel 3 – Problemstellung: Die Ausführung von Aktivitäten auf Execution Engines läuft mit einer gewissen Zuverlässigkeit ab und verursacht Kosten, die zu Gesamtkosten eines Ausführungsplans führen. Wie diese Zuverlässigkeiten und Kosten berechnet werden, wird in diesem Kapitel besprochen.

Kapitel 4 – Strategien zum Erstellen von Ausführungsplänen: Ausführungspläne mit geringen Kosten zu finden ist nicht trivial. Wir stellen in diesem Kapitel mehrere Strategien vor optimale und nicht optimale aber möglichst gute Ausführungspläne unter jeweils möglichst geringem Aufwand zu finden.

Kapitel 5 – Evaluation: Die in Kapitel 4 vorgestellten Strategien wurden durch Ausführung getestet. In diesem Kapitel stellen wir die Messergebnisse dieser Tests vor.

Kapitel 6 – Verwandte Arbeiten: In diesem Kapitel werden Arbeiten mit ähnlichen Themen betrachtet.

Kapitel 7 – Zusammenfassung und Ausblick Wir fassen die Arbeit hier zusammen und weisen auf offene Fragen und ein mögliches weiteres Vorgehen hin.

2 System-Modell

Im Folgenden stellen wir unser System-Modell, wie es in [Sch] beschrieben ist, vor. Das System-Modell enthält ein Netzwerk-Modell für das Netzwerk in welchem die Ausführungsplanung und die folgende Workflowausführung stattfinden soll, ein Workflow-Modell, welches auszuführenden Workflow mit seinen Aktivitäten und dessen Reihenfolge bestimmt und ein Ausführungs-Modell, welches die Form und Inhalt des Ausführungsplans beschreibt.

2.1 Netzwerk-Modell

Ein in dieser Arbeit betrachtetes Netzwerk besteht aus Execution Engines und Services sowie unidirektionalen Links. Zwischen jedem paar von Execution Engines existieren für beide Richtungen jeweils eine Verbindung. Zusätzlich enthält jede Execution Engine zu jedem Service im Netzwerk wiederum zwei Verbindungen, für jede Richtung eine. Ein Beispiel für ein solches Netzwerk ist in Abbildung 2.1 für drei Execution Engines e_1, e_2 und e_3 mit den Services s_1, s_2, s_3 dargestellt. Zusätzlich besitzt jede Netzwerkkomponente, also Execution Engines, Services und Verbindungen, einen Zuverlässigkeitswert für deren Verwendung.

Im Netzwerk-Modell beschreiben wir das Netzwerk als gerichteten Graphen $N = (V, L, r)$. V ist die Menge aller Knoten im Netzwerk, L die Menge aller unidirektionalen Verbindungen (gerichtete Kanten) zwischen den Knoten und r eine Funktion für die Zuverlässigkeit der Knoten und Kanten.

Die Menge der Knoten $V = (E, S)$ besteht aus den Execution Engines E, welche die den Workflow ausführenden Endgeräte darstellen und den Services S, welche von den Execution Engines aufgerufen werden können.

Während einer Workflowausführung hält eine Execution Engine einen Zustand mit Informationen über deren Fortschritt. Dieser Zustand wird unter allen Execution Engines im Netzwerk in bestimmten Situationen synchronisiert, um bei einem Ausfall einer Execution Engine auf den restlichen Execution Engines einen gültigen Zwischenstand zum Fortsetzen der Ausführung zu besitzen. Fällt eine Execution Engine während der Ausführung eines Workflows aus, wird eine andere Execution Engine die Ausführung auf Basis des zuletzt bekannten gültigen Zustand fortsetzen. Wenn die zuvor ausgefallene Execution Engine wieder betriebsbereit ist, kann ihr Zustand ein anderer sein, als der der restlichen Execution Engines. Als Resultat existieren im Netzwerk inkonsistente Zustände. Ein Zustand wird deshalb nur dann als gültig erachtet,

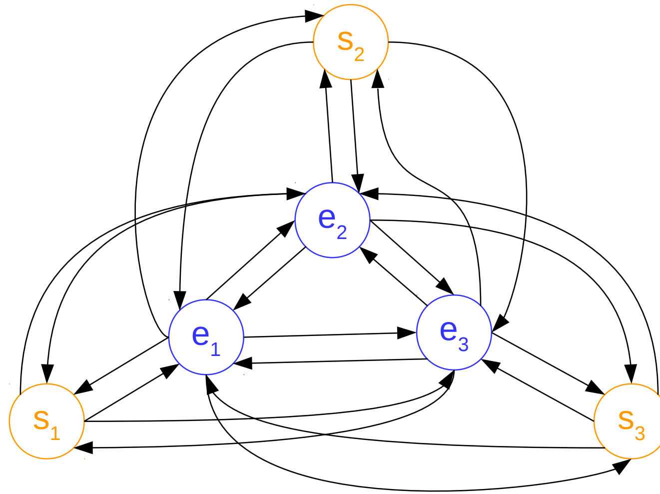


Abbildung 2.1: Netzwerk mit drei Execution Engines und drei Services

wenn er auf der absoluten Mehrheit der Knoten konsistent ist. Infolgedessen können in einem Netzwerk mit $2f + 1$ Execution Engines maximal f fehlerhafte Zustände toleriert werden.

Die Menge der Verbindungen $L : V \times V$ beschreibt alle unidirektionalen Verbindungen zwischen Knoten. $l_{v_1,v_2} \in L$ ist die Verbindung mit dem Startpunkt v_1 und dem Endpunkt v_2, wobei $v_1, v_2 \in V$. Wir gehen in dieser Arbeit von einem vollständig verbundenen Execution Engines aus und davon, dass jede Execution Engine mit jedem Service verbunden ist. Das heißt für jedes beliebige paar von Execution Engines $e_i, e_j \in E$ mit $i \neq j$ existiert eine Verbindung $l_{e_i,e_j} \in L$ und eine Verbindung $l_{e_j,e_i} \in L$. Zusätzlich besitzt jede beliebige Execution Engine $e_i \in E$ Verbindungen mit allen Services $s_k \in S$ der Form $l_{e_i,s_k} \wedge l_{s_k,e_i} \in L$.

Die Funktion $r : V \cup L- > [0;1]$ für die Zuverlässigkeit bietet eine Abbildung von einer Netzwerkkomponente (Knoten oder Link) in V oder L auf einen Wert zwischen 0 und 1. Eine Zuverlässigkeit von 1 bedeutet, dass die Komponente nie ausfällt und eine Zuverlässigkeit von 0, dass die Komponente immer ausgefallen ist.

2.2 Workflow-Modell

Wir beschreiben einen Workflow durch den gerichteten Graphen $W = (A, D)$. A ist die Menge der Aktivitäten, die im Workflow ausgeführt werden und $D : A \times A$ die Menge der gerichteten Kanten zwischen den Aktivitäten. Die Kante d_{a_1,a_2} geht von a_1 aus und in a_2 ein und beschreibt, dass die Aktivität a_2 nur ausgeführt werden kann, wenn die Aktivität a_1 bereits ausgeführt wurde.

Die Menge aller Kanten verbindet alle Aktivitäten und gibt jeweils einen Start- und einen Endpunkt des Workflows vor. Startpunkt ist die einzige Aktivität ohne eingehende Kante und Endpunkt die einzige Aktivität ohne ausgehende Kante. Jede andere Aktivität besitzt sowohl eingehende als auch ausgehende Kanten. In dieser Arbeit betrachten wir nur Workflows in welchen alle Aktivitäten maximal eine eingehende Kante und eine ausgehende Kante besitzen in anderen Worten sind die Aktivitäten des Workflows streng sequentiell abzuarbeiten.

Abbildung 2.2 zeigt einen Workflow mit den Aktivitäten a_o bis a_9. Die Ausführung muss stets mit a_0 beginnen und endet wenn a_9 ausgeführt wurde. Dazwischen kann eine Aktivität immer dann ausgeführt werden, wenn die vorherige Aktivität bereits ausgeführt wurde. Ist beispielsweise a_0 ausgeführt kann mit a_1 und nur mit a_1 weitergemacht werden.

2.3 Ausführungs-Modell

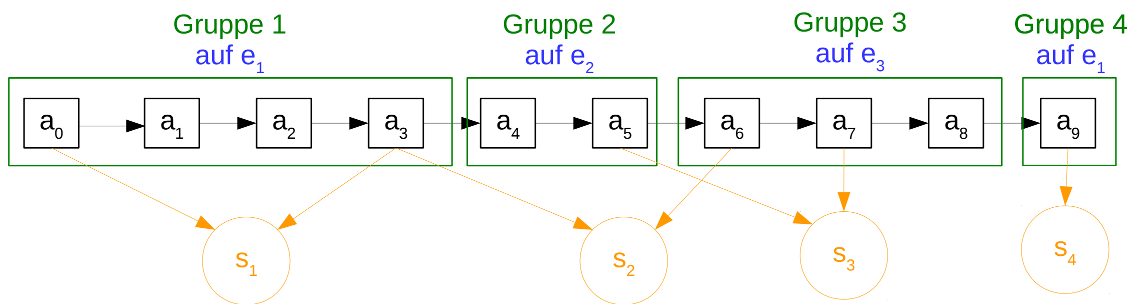


Abbildung 2.2: Workflow mit Aktivitäten, Gruppen und Services

Das Ausführungsmodell dient der Beschreibung, wie der Workflow im Hinblick auf verfügbare Execution Engines, Services und Aktivitäten ausgeführt wird und wird im Ausführungsplan verwendet.

Für jede Aktivität $a_i \in A$, die einen Serviceaufruf beinhaltet, gibt es eine Menge von kompatiblen Services $S_{a_i} \in S$. Diese Menge beinhaltet einen oder mehrere kompatible Services. Wir definieren die Funktion $s_comp : A \to \wp(S)$, welche für eine Aktivität die Potenzmenge der kompatiblen Services zurück gibt.

Für jede Aktivität wird festgelegt auf welcher Execution Engine diese ausgeführt werden soll. Hierbei können aufeinanderfolgende Aktivitäten zu einer Gruppe zusammengefasst werden, die dann als Ganzes auf einer einzigen Execution Engine ausgeführt wird. Die Gruppen dürfen dabei nicht überlappen, sodass bei einer fehlerfreien Ausführung des Workflows jede Aktivität nur einmal ausgeführt wird. G ist die Menge der Gruppen für die Ausführung des Workflows und es gilt: $\forall g \in G : a \in g \wedge a \in g' \Rightarrow g = g'$. Gleichzeitig muss jede Aktivität des Workflows

in einer Gruppe $g \in G$ vorhanden sein. Die Reihenfolge in der die Gruppen ausgeführt werden folgt der Reihenfolge der Aktivitäten im Workflow.

Wir definieren ferner eine Gruppe als eine nicht leere Menge von aufeinanderfolgenden Aktivitäten. Die Länge einer Gruppe entspricht der Menge der in ihr enthaltenen Aktivitäten.

Hat eine Execution Engine eine Gruppe vollständig ausgeführt synchronisiert sie ihren Zustand als Zwischenergebnis mit den anderen Execution Engines. Die Synchronisation ist erfolgreich, sobald eine Mehrheit der Execution Engines diesen Zustand kennt. Erst nach erfolgreicher Synchronisation kann die nächste Gruppe ausgeführt werden. Für die Dauer einer Synchronisation nehmen wir die Zeit t_sync an. Diese kann durch Messungen bestimmt oder schlicht abgeschätzt werden. t_sync wird benötigt, um die Kosten eines Ausführungsplans zu bestimmen, was im nächsten Kapitel genau beschrieben wird.

Jede Aktivität hat eine definierte Ausführungszeit in Millisekunden. Hierfür definieren wir die Funktion $t : A \rightarrow \mathbb{N}$ welche für eine Aktivität eine Zeit in Millisekunden zurück gibt.

Die Ausführung einer Aktivität kann in Zusammenhang mit einem Service-Aufruf reale geldwerte Kosten verursachen. So könnte dieser in etwa den Kauf einer Aktie zu einem bestimmten Preis bewirken. Fällt eine Execution Engine nach der Ausführung einer solchen Aktivität aus oder kann aufgrund eines Netzwerkproblems nach einer Gruppenausführung nicht vor Ablauf eines Timeouts synchronisieren, so wird eine andere Execution Engine die Gruppe erneut ausführen. In diesem Fall gehört die erste Ausführung der Aktivität nicht mehr zur gültigen Ausführung des Workflows und muss kompensiert werden. Für die Kompensierungskosten einer Aktivität definieren wir die Funktion $c : A \rightarrow \mathbb{N}$, welche die Kompensierungskosten in Cent zurück gibt.

Abbildung 2.2 zeigt als Beispiel für den Workflow mit den Aktivitäten a_0 bis a_9 einen Ausführungsplan mit den Gruppen 1 bis 4. Gleichzeitig ist für jede Aktivität mit einem Serviceaufruf die Menge der kompatiblen Services dargestellt. a_0 benötigt beispielsweise zwingend s_1 wohingegen der Serviceaufruf von a_3 sowohl mit s_1 als auch mit s_2 durchgeführt werden kann. Für jede Aktivität kann prinzipiell eine beliebige Menge von Services kompatibel sein.

Gruppe 1 des Ausführungsplans beinhaltet die Aktivitäten a_0 bis a_3 und es ist festgelegt, dass sie und damit alle enthaltenen Aktivitäten auf der Execution Engine e_1 ausgeführt werden soll. Für die restlichen Gruppen und Aktivitäten sind weitere Verhältnisse dieser Art dargestellt.

3 Problemstellung

Ein Ausführungsplan wird danach bewertet, welche Kosten, sowohl in Zeit als auch in Geld gemessen, bei einer entsprechenden Workflow-Ausführung zu erwarten sind. In diesem Kapitel erläutern wir die Berechnung dieser Kosten. Dazu betrachten wir zunächst die Ausführung einzelner Aktivitäten sowie Gruppen und wie die Zuverlässigkeit dieser jeweils festgestellt werden können. Anschließend stellen wir dar, wie unter Einbeziehung dieser Zuverlässigkeiten die zu erwartenden Kosten einer Workflowausführung berechnet werden können.

3.1 eine Gruppe von Aktivitäten ausführen

Die Zuverlässigkeit einer Aktivität $a \in A$ ohne Serviceaufruf entspricht genau der Zuverlässigkeit der Execution Engine auf welcher diese ausgeführt wird, also $r_{execActWithoutService}(e) = r(e)$. Bei Aktivitäten mit Serviceaufruf müssen die Zuverlässigkeiten des von der Execution Engine $e \in E$ gewählten Service $s \in S$ und die beiden Verbindungen zwischen Execution Engine und Service mit in Betracht gezogen werden (3.1).

$$r_{execAct}(e, a, s, N) = r(e) \cdot r(l_e, s) \cdot r(s) \cdot r(l_s, e) \tag{3.1}$$

Gibt es mehrere kompatible Services im Netzwerk, wählt die Execution Engine den Service welcher sich mit der höchsten Zuverlässigkeit aufrufen lässt. So erhält man die maximale Zuverlässigkeit der Ausführung einer Aktivität auf einer Execution Engine (3.2).

$$r_{maxEcecAct}(e, a, N) = max_{s \in s_{comp}(a)} r_{execAct}(e, a, s, N) \tag{3.2}$$

Als nächstes soll die Zuverlässigkeit der Ausführung einer Gruppe g auf der Execution Engine e berechnet werden. Hierfür bilden wir für die in der Gruppe enthaltenen Aktivitäten das Produkt aus den zuvor berechneten $r_{maxExecAct}(e, a, N)$ in folgender Weise.

$$r_{group}(e, g, N) = \prod_{a \in g} r_{maxExecAct}(e, a, N) \tag{3.3}$$

3.2 Synchronisation nach einer Gruppenausführung

Am Ende einer Gruppenausführung muss der sich daraus ergebende Zwischenstand der Workflow-Ausführung, also der aktuelle Zustand auf der entsprechenden Execution Engine, synchronisiert werden. Dies ist dann erfolgreich, wenn der Mehrheit der Execution Engines dieser Zwischenstand bekannt wurde. Bei $2f + 1$ Execution Engines muss eine Execution Engine somit von mindestens f anderen Execution Engines nach dem Senden des Zwischenstands eine Bestätigung erhalten. Damit die Synchronisation mit einer Execution Engine funktionieren kann, müssen die Verbindungen von und zu dieser, sowie die Execution Engine selbst betriebsbereit sein.

Dazu betrachten wir alle möglichen Zustände des Netzwerks in Bezug auf Betriebsbereitschaft der einzelnen Komponenten. In jedem Zustand kann eine Komponente entweder betriebsbereit oder ausgefallen sein. Somit gibt es insgesamt $2^{V \cup L}$ Zustände. Für jeden Zustand ϕ gilt für eine Komponente x entsprechend $\phi(x) = 1$ oder $\phi(x) = 0$. Die Menge $\Phi\phi_1, \phi_2, ..., \phi_{2^{V \cup L}}$ enthält alle möglichen Zustände.

Jeder Netzwerkzustand $\phi \in \Phi$ tritt mit einer gewissen Wahrscheinlichkeit entsprechend der Zuverlässigkeiten der Netzwerkkomponenten auf. Eine Komponente $x \in V \cup L$ ist betriebsbereit mit der Wahrscheinlichkeit $r(x)$ und ausgefallen mit der Wahrscheinlichkeit $1 - r(x)$. Die Wahrscheinlichkeit für einen bestimmten Netzwerkzustands ergibt sich wie folgt.

$$p(\phi, N) = \prod_{x \in V \cup L} [1 - r(x) + \phi(x)(2 \cdot r(x) - 1)] \tag{3.4}$$

In der Gleichung (3.4) werden die Wahrscheinlichkeiten der jeweiligen Zustände (betriebsbereit oder ausgefallen) der einzelnen Komponenten multipliziert. Für eine betriebsbereite Komponente x mit $\phi(x) = 1$ wird in den eckigen Klammern der erste Teil $1 - r(x)$ aufgehoben und es bleibt $r(x)$. Für eine ausgefallene Komponente x mit $\phi(x) = 0$ wird der zweite Teil mit 0 multipliziert und es bleibt nur der erste Teil $1 - r(x)$ übrig.

Zu entscheiden ist, ob in einem vorliegenden Netzwerkzustand eine erfolgreiche Synchronisation für eine ausführende Execution Engine $e_exec \in E$ möglich ist. Mit einer weiteren Execution Engine $e_{other} \in E, e_{exec} \neq e_{other}$ kann dann synchronisiert werden, wenn $\phi(e_{other}) = 1$, $\phi(l_{e_{exec}, e_{other}}) = 1$ und $\phi(l_{e_{other}, e_{exec}}) = 1$. Wenn also alle zur Kommunikation erforderlichen Komponenten betriebsbereit sind.

Wir definieren die Funktion welche zurück gibt, ob eine Execution Engine $e \in E$ im Netzwerkzustand ϕ erfolgreich synchronisieren kann wie folgt:

$$\theta(\phi, N, e) = \begin{cases} 1, & \text{falls e mit} \geq f \text{anderen Execution Engines synchronisieren kann} \\ 0, & \text{ansonsten} \end{cases} \tag{3.5}$$

Die Wahrscheinlichkeit, ob e erfolgreich synchronisieren kann, lässt sich nun wie folgt berechnen:

$$r_{sync}(N, e) = \sum_{\phi \in \Phi} \theta(\phi, N, e) \cdot p(\phi, N) \tag{3.6}$$

In r_{sync} addieren wir die Wahrscheinlichkeiten aller Netzwerkzustände in welchen eine erfolgreiche Synchronisation stattfinden kann.

Wir fassen zusammen, die Wahrscheinlichkeit, dass eine Gruppe erfolgreich ausgeführt und anschließend synchronisiert wird, lässt sich wie folgt berechnen.

$$r_{execsync}(e, g, N) = r_{group}(e, g) \cdot r_{sync}(N, e) \tag{3.7}$$

3.3 zu erwartende Kosten einer Gruppenausführung

Um die zu erwartenden Kosten einer Gruppenausführung zu ermitteln, werden die Effekte bei einer erfolgreichen Ausführung und die Effekte bei einem Fehlschlagen der Gruppenausführung jeweils entsprechend ihrer Wahrscheinlichkeiten wie folgt addiert:

$$\begin{aligned} f_{group}(e, g, N) = {} & r_{execsync}(e, g, N) \cdot \frac{t_{sync}}{t_{user}} \\ & + (1 - r_{execsync}(e, g, N)) \cdot (\frac{t_{elec}}{t_{user}} + \frac{\sum_{a \in g} t(a)}{t_{user}} + \frac{\sum_{a \in g} c(a)}{c_{user}} + \frac{t_{sync}}{t_{user}}) \end{aligned} \tag{3.8}$$

Die erfolgreiche Ausführung und Synchronisation ist mit der Wahrscheinlichkeit $r_{execsync}(e, g, N)$ gewichtet, das Fehlschlagen mit $1 - r_{execsync}(e, g, N)$. Eine erfolgreiche Ausführung kostet lediglich die Zeit der Synchronisation t_{sync} welche durch den Gewichtungsfaktor für Zeit t_{user} geteilt wird. Schlägt die Gruppenausführung fehl, so gehen wir vom schlimmsten Fall aus, wo die gesamte Gruppe ausgeführt, aber nicht synchronisiert wurde. In diesem Fall muss zunächst ein neue Execution Engine abgestimmt werden, was die Zeit t_{elec} in Anspruch nimmt. Für die Dauer von $\sum_{a \in g} t(a)$ werden alle Aktivitäten erneut ausgeführt. Daneben müssen alle Aktivitäten der fehlgeschlagenen Ausführung kompensiert werden. Die Kosten dafür erhalten wir durch $\sum_{a \in g} c(a)$. Gewichtet werden diese geldwerten Kosten mit dem Faktor c_{user}. Die Gewichtungsfaktoren t_{user} und c_{user} machen Zeit und Geld vergleichbar und werden vom Benutzer bestimmt. $t_{user} = 70$ und $c_{user} = 100$ würden zum Beispiel bedeuten, dass eine 70ms längere Ausführungszeit kosten von 1€ verursachen.

Für die Ausführung einer Gruppe wählen wir die Execution Engine, welche dabei die geringsten Kosten verursacht.

$$f_{minGroup}(g, N) = min_{e \in E} f_{group}(g, N) \tag{3.9}$$

3.4 zu erwartende Kosten einer Workflowausführung

Sind die Gruppen für den Ausführungsplan eines Workflows gewählt, erhalten wir die Gesamtkosten der Worklowausführung in dem wir die Kosten aller Gruppen addieren.

$$f_{cost}(G, N) = \sum_{g \in G} f_{minGroup}(g, N) \tag{3.10}$$

3.5 Ausführungsplan mit minimalen zu erwartenden Kosten

Für einen Workflow der Länge n beträgt die Anzahl der möglichen Gruppen mit minimalen Kosten $\frac{n^2+n}{2}$. Wir zeigen dies durch die Anzahl der Gruppen pro Gruppenlänge. Für die Gruppenlänge n gibt es genau eine Gruppe, welche von der ersten bis zur letzten Aktivität reicht. Für die Gruppenlänge $n-1$ gibt es zwei Gruppen, die Gruppe, die bei der ersten Aktivität beginnt und die Gruppe, die bei der zweiten Aktivität beginnt. Für $n-2$ sind es entsprechend 3 Gruppen und so weiter. Für die Gruppenlänge 1 gibt es dann n Gruppen, nämlich je eine für jede einzelne Aktivität. Daraus können wir schließen, dass die Anzahl der Gruppen in einem Workflow mit n Aktivitäten genau $1 + 2 + 3 + .. + n$ beträgt. Anhand der *Gaußschen Summenformel* können wir daraus $\frac{n^2+n}{2}$ schließen. Durch die große Menge der möglichen Gruppen ist auch die Menge der möglichen Ausführungsplänen entsprechend hoch.

Ziel ist es für den Ausführungsplan eine Abfolge von Gruppen zu finden, die möglichst wenig Kosten verursacht. Die Menge von möglichen Ausführungsplänen für einen Workflow W definieren wir als $\Omega(W)$. Für einen gegebenen Workflow W und ein Netzwerk N suchen wir den Ausführungsplan mit den minimalen zu erwartenden Kosten.

$$min_{G \in \Omega(W)} f_{cost}(G, N) \tag{3.11}$$

3.6 Beispiel

Abbildung 3.1 zeigt ein Netzwerk mit zwei Execution Engines, zwei Services und den Verbindungen zwischen diesen mit den jeweiligen Zuverlässigkeiten. In diesem Netzwerk soll der Workflow mit den Aktivitäten a_0 bis a_4 ausgeführt werden. a_1 benötigt den Service s_1 und a_4 benötigt den Service s_2. Für eine Ausführung der Aktivität a_1 bietet sich vermutlich die Execution Engine e_1 an, da diese im Vergleich zu e_2, bei gleicher Zuverlässigkeit der Execution Engine selbst, zum Service s_1 eine stabilere Verbindung hat. Was bedeutet, dass $r_{maxEcecAct}(e_1, a_1, N) > r_{maxEcecAct}(e_2, a_1, N)$. Im Fall für a_4 gilt hingegen $r_{maxEcecAct}(e_1, a_4, N) < r_{maxEcecAct}(e_2, a_4, N)$.

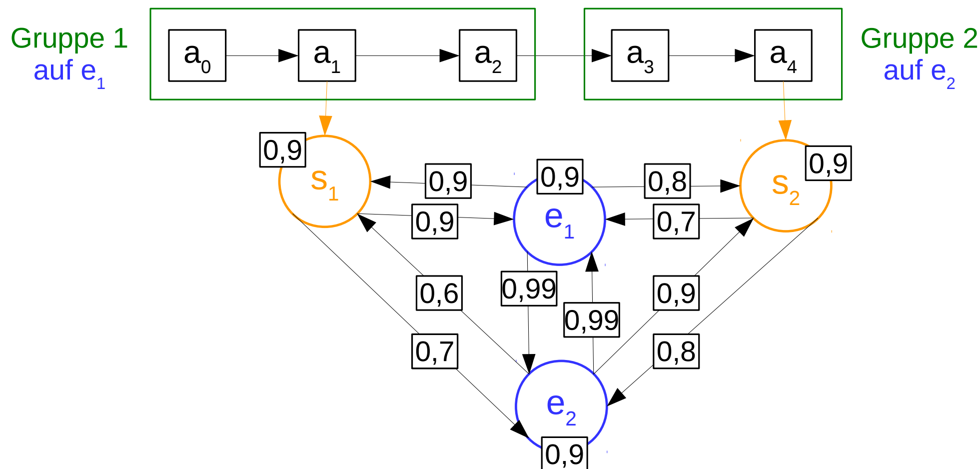


Abbildung 3.1: Workflow der Länge 5 in einem Netzwerk mit zwei Execution Engines und zwei Services

Um die Zuverlässigkeit für die gegebene Gruppe 1 zu bestimmen, muss das Produkt der Ausführungszuverlässigkeiten von a_0, a_1 und a_2 auf e_1 bestimmt werden.

Für die Berechnung der Synchronisationszuverlässigkeit einer Execution Engine müssen in diesem Netzwerk bei vier relevanten Komponenten, nämlich die beiden Execution Engines und den beiden Verbindungen zwischen diesen, jeweils 16 Netzwerkzustände betrachtet werden, in welchen jede Komponente jeweils betriebsbereit oder ausgefallen ist. Wir berechnen die Synchronisationszuverlässigkeit für e_1. Bei der Betrachtung aller Netzwerkzustände kann festgestellt werden, dass nur in einem dieser erfolgreich synchronisiert werden kann. Um ihren Zustand mit der Mehrheit der Execution Engines synchronisieren zu können, muss e_1 mit e_2 kommunizieren können, da die einzige Mehrheit in diesem Netzwerk aus e_1 und e_2 besteht. Hierfür müssen alle Komponenten e_1, e_2, l_{e_1,e_2} und l_{e_2,e_1} betriebsbereit sein. Dies ist der einzige Netzwerkzustand in welchem e_1 erfolgreich synchronisieren kann. Die Eintrittswahrscheinlichkeit dieses Netzwerkzustands berechnet sich durch $r(e_1 \cdot e_2 \cdot r(l_{e_1,e_2}) \cdot l_{e_2,e_1}$ ergibt im Beispiel also $0,9 \cdot 0,9 \cdot 0,99 \cdot 0,99 = 0,793881$

Im Beispiel haben e_1 und e_2 dieselbe Synchronisationszuverlässigkeit. Wäre diese stark unterschiedlich könnte es natürlich vorteilhafter sein beide Gruppen auf der Execution Engine mit der besseren Synchronisationszuverlässigkeit auszuführen, auch wenn die Zuverlässigkeit des Serviceaufrufs dafür schlechter ist.

Im Beispiel nicht gezeigt sind für jede Aktivität die Dauer der Ausführung $t(a)$ sowie die Synchronisationszeit t_{sync}. Je nach Verhältnis dieser muss eine Synchronisation für verhältnismäßig hohe $t(a)$s zusätzlich häufiger oder für verhältnismäßig hohe t_{sync} weniger häufig stattfinden, um einen optimalen Ausführungsplan zu erhalten. Hierdurch können sich die optimalen Gruppen also auch wiederum ändern.

4 Strategien zum Erstellen von Ausführungsplänen

Im folgenden stellen wir unterschiedliche Strategien zum Erstellen von Ausführungsplänen vor. Die erste Strategie *Optimaler Ausführungsplan* bestimmt anhand globalen Wissens auf einer einzelnen Execution Engine den optimalen Ausführungsplan. Die folgenden Strategien *Bestes Replikat* und *Line Of Sight* stellen Wege dar, um die Ausführungsplanung zu beschleunigen. Diese laufen verteilt ab und funktionieren ohne globales Wissen. Der Nachteil dieser Strategien ist jedoch, dass ein optimaler Ausführungsplan nicht mehr garantiert ist. Zum Schluss beschreiben wir mit Beste Gruppen noch eine Strategie die es erlaubt den optimalen Ausführungsplan verteilt zu berechnen.

4.1 Optimaler Ausführungsplan

Die Strategie *Optimaler Ausführungsplan*, welche den optimalen Ausführungsplan finden soll, sieht vor, die Planung auf einer einzelnen Execution Engine auszuführen. Die Wahl dieser Execution Engine kann zufällig sein, allerdings muss auf dieser globales Wissen vorhanden sein. Globales Wissen heißt hier, dass die planende Execution Engine den gesamten Workflow, als auch das gesamte Netzwerk kennt. Wie der Ausführungsplan berechnet wird, wird im folgenden beschrieben. Eine Zusammenfassung des Ablaufs auf einer Execution Engine bietet die Tabelle 4.1.

1.	Berechne die Kosten $f_{minGroup}(g, N)$ aller möglichen Gruppen $g \in \Omega(W)$.
2.	Füge für jede Gruppe g die entsprechende Kante mit dem Gewicht $f_{minGroup}(g, N)$ in den Workflow-Graphen W ein.
3.	Finde im Graphen W den Pfad mit den geringsten Kosten von a_0 bis a_i, $i = \|A\| - 1$.
4.	Verwende die Kanten des günstigsten Pfads für den Ausführungsplan.
5.	Sende den Ausführungsplan an alle Execution Engines.
6.	Die Execution Engines führen den Workflow anhand des Ausführungsplans aus.

Tabelle 4.1: Ablauf der Strategie Optimaler Ausführungsplan

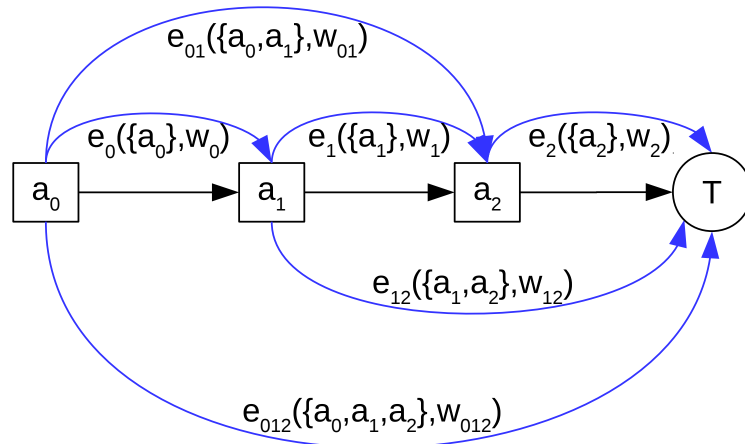


Abbildung 4.1: Workflow-Graph zum Finden des günstigsten Pfads

4.1.1 Ausführungsplan anhand des kürzesten Pfads

Entsprechend der möglichen Gruppen in einem Workflow gibt es eine bestimmte Menge möglicher Ausführungspläne. Für einen Workflow der Länge n gibt es 2^{n-1} mögliche Ausführungspläne. All diese Ausführungspläne zu vergleichen, das heißt für jeden Ausführungsplan die Kosten zu bestimmen, ist entsprechend teuer.

Wir fügen zum Workflowgraphen W die gewichteten Kanten $G = A \times A \vee A \times T$ hinzu. Jedes $g \in G$ entspricht einer möglichen Gruppe in W. G enthält alle möglichen Gruppen für W.

Die Kosten der Gruppen in G bestimmen jeweils das Gewicht der Kante. Die Knoten A stellen jeweils die als nächstes auszuführende Aktivität dar. Die Kanten verbinden die Aktivität, welche in der Gruppe zuerst ausgeführt wird, mit der Aktivität, die nach erfolgreicher Gruppenausführung als nächstes auszuführen ist. Das Ziel der Kante ist also nicht mehr Teil der Gruppe. Die Gruppe endet somit im Vorgänger des Ziels. Eine Kante von a_1 nach a_2 entspricht also einer Gruppe, die nur a_1 enthält. Eine Kante von a_3 nach a_7 entspricht entsprechend einer Gruppe mit den Aktivitäten a_3 a_4, a_5 und a_6.

Den zusätzlichen Knoten T führen wir ein, um signalisieren zu können, dass keine weiteren Aktivitäten auszuführen sind, also der Workflow komplett ausgeführt wurde. Dies ist nötig, um eine Kante darstellen zu können, deren Gruppe die letzte Aktivität beinhaltet. Die letzte Kante eines Ausführungsplans endet also immer in T.

Um alle möglichen Gruppen in G zu bestimmen fügen wir dort für jede Aktivität $a_i \in A$ jeweils eine Kante zu jeder späteren Aktivität $a_j \in A$ ein. Wir weisen jeder Kante e ein Gewicht w zu, welches wir mit $f_{minGroup}(g, N)$ (3.9) für die entsprechende Gruppe berechnen. Abbilding 4.1 zeigt für einen Workflow mit drei Aktivitäten alle solche Kanten.

Um den optimalen Ausführungsplan zu finden, suchen wir nun im Graph W den Pfad mit dem geringsten Gesamtgewicht von der ersten Aktivität bis T durch den Graphen. Die Kanten, die in diesem Pfad enthalten sind, geben die Gruppen für den optimalen Ausführungsplan vor. Den gewünschten Pfad kann mit einem beliebigen Shortest-Path-Algorithmus gefunden werden.

Ist der Ausführungsplan berechnet muss er im Netzwerk an alle anderen Execution Engines vor Ausführung des Workflows verteilt werden. In unserem Fall, wo alle Execution Engines mit einander direkt verbunden sind, kann dies mit $|E| - 1$ direkten Nachrichten an alle anderen Execution Engines gelöst werden.

4.1.2 Bewertung

Mit einem optimalen Ausführungsplan ist eine Workflowausführung mit den geringst möglichen Kosten in Bezug auf unser Systemmodell garantiert. Die Berechnung eines optimalen Ausführungsplans bringt aber auch gewisse Nachteile mit sich. Da für Netzwerk und Workflow globales Wissen benötigt wird, also beides in Gänze an einer Stelle verfügbar sein muss, ist eine verteilte Berechnung nicht möglich.

Die Ausführungsplanung auf einem einzelnen Knoten stellt einen Single-Point-Of-Failure dar. Fällt der planende Knoten aus, muss die Ausführungsplanung auf einem anderen Knoten erneut gestartet werden. Möchte man das verhindern, muss man mehrere Knoten mit globalem Wissen ausstatten. Dann kann die Ausführunsplanung, der selbe deterministische Prozess, auf mehreren Knoten ausgeführt werden. Hierbei wird die Planungszeit allerdings nicht verkürzt.

Daneben ist die Berechnung sehr aufwendig. Für alle $\frac{n^2+n}{2}$ möglichen Gruppen müssen die Kosten für jede Execution Engine berechnet werden. Entsprechend muss die Synchronisationszuverlässigkeit zuvor für jede Execution Engine bestimmt werden, wobei jeweils 2^V mögliche Netzwerkzustände ausgewertet werden müssen. Es stellt sich die Frage, ob die Zeit und die geldwerten Kosten, die bei einer Ausführung mit optimalem Ausführungsplan eingespart werden, den zusätzlichen Zeitaufwand bei der Planung Wert sind.

Aus diesen Gründen betrachten wir im folgenden weitere alternative Strategien. Diese sollen es ermöglichen, die Berechnungen entweder mithilfe von Heuristiken zu vereinfachen und dennoch akzeptable Ergebnisse zu erzielen oder diese auf mehreren Execution Engines verteilt zu berechnen.

4.2 Bestes Replikat

Bei dieser Strategie wird ein Ausführungsplan erstellt, der vorsieht den gesamten Workflow nur auf einer einzigen Execution Engine auszuführen. Das Verfahren wird verteilt auf allen Execution Engines ausgeführt.

1.	Ermittle den optimalen lokalen Ausführungsplan durch Berechnung aller Gruppenkosten und Verwendung eines *Shortest Path* Algorithmus.
2.	Bestimme das beste Replikat anhand der Kosten des jeweils berechneten Ausführungsplans
3.	Sende den besten Ausführungsplan an alle anderen Execution Engines.
4.	Das beste Replikat führt den Workflow anhand des eigenen Ausführungsplans aus.

Tabelle 4.2: Ablauf der Strategie Bestes Replikat

Jede Execution Engine benötigt nur die Informationen über ihre direkten Nachbarn und den Verbindungen zu und von diesen. Wissen über die Verbindungen zwischen Nachbarn oder zwischen Nachbarn und Services muss hier in einer Execution Engine nicht bekannt sein. Die verteilte Ausführung dieser Strategie spielt sich in mehreren Schritten ab. Vor der Ausführung des Workflows muss der Ausführungsplan bei allen Execution Engines bekannt sein.

4.2.1 Ablauf

Tabelle 4.2 zeigt den groben Ablauf in 4 Schritten.

4.2.2 Lokaler Ausführungsplan

Jede Execution Engine e berechnet zunächst lokal den optimalen Ausführungsplan unter der Einschränkung, dass jede Gruppe nur auf e ausgeführt werden kann. Das eingeschränkte Wissen auf jeder Execution Engine erlaubt es immer noch die eigene Synchronisationszuverlässigkeit zu berechnen und für jede Aktivität mit Serviceaufruf den idealen Service zu finden. Die Synchronisationszuverlässigkeiten anderer Execution Engines werden nicht benötigt, da keine Gruppe im lokal erstellten Ausführungsplan auf einer anderen Execution Engine ausgeführt werden muss. Somit können die Zuverlässigkeit für die lokale Ausführung jeder Aktivität und die Kosten jeder Gruppe berechnet werden. Sind alle möglichen Gruppen berechnet wird auch hier, wie bei der Strategie für den optimalen Ausführungsplan, der Workflow Graph um Kanten für jede Gruppe erweitert und ein *Shortest Path* Algorithmus verwendet, um den kürzesten Pfad und damit die lokal optimalen Gruppen zu finden. Die Kosten dieser lokal optimalen Gruppen werden addiert, um die Gesamtkosten des lokalen Ausführungsplans zu erhalten.

4.2.3 Wahl des besten Replikats

Der nächste Schritt stellt für alle Execution Engines fest, welche Execution Engine den besten Ausführungsplan hat und verteilt diesen dann unter allen Execution Engines. Dafür müssen die jeweiligen Gesamtkosten der Ausführungspläne aller Teilnehmer verglichen und das Minimum

festgestellt werden. Wir besprechen im Folgenden mehrere Möglichkeiten dieses Problem zu lösen.

vollverbundene Execution Engines Eine einfache Möglichkeit ist es, allen Execution Engines die Kosten der Ausführungspläne aller anderer Execution Engines bekannt zu machen. In unserem Netzwerk, wo jeder Execution Engine mit jeder anderen Execution Engine verbunden ist, schickt jede Execution Engine eine Nachricht mit den Kosten des eigenen Ausführungsplans an alle anderen Execution Engines. Dafür müssen in einem Netzwerk mit n Execution Engines $n^2 - n$ Nachrichten verschickt werden. Alle Execution Engines können daraufhin die Execution Engine mit dem besten Ausführungsplan bestimmen. Die Execution Engine mit dem besten Ausführungsplan sendet diesen dann wiederum direkt an alle anderen Execution Engines. Hier werden also noch einmal $n - 1$ Nachrichten versendet. Sobald der Erhalt des besten Ausführungsplans von allen anderen Execution Engines bestätigt wurde, kann die entsprechende Execution Engine mit der Ausführung des Workflows beginnen.

andere Netzwerke Soll die Anzahl der zu verschickenden Nachrichten reduziert werden oder existiert nicht zwischen allen Execution Engines eine Verbindung, können andere Verfahren verwendet werden.

In Anlehnung an die zuvor beschriebene Methode kann in einem nicht voll verbundenen Netzwerk ein *Echo*-Algorithmus wie in [Cha82] beschrieben verwendet werden, um eine Nachricht an alle anderen Execution Engines im Netzwerk zu schicken. Auf diese Weise können die Kosten aller Pläne wieder allen Execution Engines bekannt gemacht werden und anschließend der beste Plan im Netzwerk verteilt werden.

Es lässt sich vermeiden die Kosten aller Pläne auf allen Execution Engines zu verteilen. Die Execution Engine im Netzwerk zu finden, die den Ausführungsplan mit den geringsten Kosten hat, ist ein *Leader Election* Problem für welches es unterschiedliche Lösungen gibt.

Zunächst wird ein Overlay-Netzwerk der Execution Engines in der Form eines Baums benötigt. Durch die Verwendung eines *Echo*-Algorithmus kann ein Spannbaum im Netzwerk erzeugt werden.

Wir beschreiben im folgenden unsere Anwendung des *Leader Election* Verfahrens in Bäumen, welches in [Tel00] beschrieben ist. Die *Explosion Phase* kann entfallen, da nach Erhalt der Nachricht mit dem Echo Algorithmus klar ist, dass die *Leader Election* beginnen soll. Ferner wissen die Blätter im Baum, dass sie Blätter sind, da sie zwei *Explorer*-Nachrichten des *Echo*-Algorithmus erhalten haben.

Die Blätter im Baum beginnen dann mit der *Contraction Phase* indem sie die Kosten ihres Ausführungsplans anhand des Baumes weiterleiten. Jeder andere Knoten im Baum wartet darauf von allen Nachbarn bis auf einen Nachrichten mit Kosten zu erhalten und sendet dabei nur die Nachricht mit den niedrigsten Kosten an den Knoten weiter von welchem keine Nachricht empfangen wurde. Zwei Knoten im Baum erhalten schließlich Nachrichten von all

ihren Nachbarn und kennen damit die Kosten des besten Ausführungsplans. Damit endet die *Contraction Phase*.

Die beiden Knoten, die die Kosten des besten Ausführungsplans kennen, senden jeweils eine Nachricht per *Echo*-Algorithmus an alle andere Knoten. Sobald ein Knoten erfährt, dass er den besten Ausführungsplan hat, sendet er diesen wiederum per *Echo*-Algorithmus an alle anderen Knoten. Sobald der Erhalt des Ausführungsplans bestätigt ist, kann die Execution Engine mit der Ausführung des Workflows beginnen.

In dem vorgestellten Verfahren werden keine Netzwerkfehler behandelt. [MWV00] erläutert Verfahren für *Leader Election* in einem mobilen Ad-Hoc Netzwerk in welchem Verbindungsfehler und Ausfälle von Knoten möglich sind.

4.2.4 Bewertung

Der Vorteil der Strategie *Bester Knoten* ist, dass eine planende Execution Engine nicht das gesamte Netzwerk sondern nur die Zuverlässigkeiten seiner direkten Nachbarn und der Verbindungen zu diesen kennen muss. Die Execution Engine muss nur die eigene Synchronisationszuverlässigkeit und die Gruppenkosten für die lokale Ausführung berechnen, was Zeit bei der Planung spart. Der Faktor, um welchen die Ausführungsplanung im Vergleich zum optimalen Ausführungsplan verkürzt wird, entspricht dabei dem Replikationsgrad. Besonders für hohe Replikationsgrade wird daher die Planungszeit deutlich verkürzt.

Der Nachteil der Strategie ist, dass der resultierende Ausführungsplan nicht in allen Fällen optimal sein kann. Eine Execution Engine die für die Serviceaufrufe der einen Gruppe die beste Zuverlässigkeit besitzt kann für die Serviceaufrufe einer anderen Gruppe deutlich schlechter als andere Execution Engines sein. In einem solchen Fall kann es günstiger sein nicht die selbe Execution Engine für beide Gruppen zu verwenden. Dies ist bei dieser Strategie allerdings nicht möglich. Daneben müssen immer noch die Kosten aller möglichen Gruppen des Workflows berechnet werden.

4.3 Line Of Sight

Line Of Sight ist eine weitere Strategie, die den Ausführungsplan verteilt berechnet. Der Grundgedanke ist hierbei jedoch den Ausführungsplan nicht komplett sondern Stück für Stück zu berechnen. Es wird dabei jeweils nur ein bestimmter Teil des Workflows mit einer definierten Länge, der *Line Of Sight*, betrachtet.

Die Beobachtung, dass sehr lange Gruppen in optimalen Ausführungsplänen für bestimme Kombinationen von Workflow und Netzwerk sehr selten vorkommen, legt den Verdacht nahe, dass es nicht unbedingt immer nötig ist, den gesamten Workflow zu betrachten, um einen

Großteil der Gruppen eines optimalen Ausführungsplan zu finden und damit insgesamt einen guten Ausführungsplan zu erhalten.

Werden mehrere Ausführungspläne jeweils nur für einen Teil des Workflows bestimmt, kann mit Ausführung des Workflows auch bereits begonnen werden, bevor der Ausführungsplan für den gesamten Workflow zur Verfügung steht. Die Ausführungsplanung wird bei *Line Of Sight* nur für den betrachteten Teil des Workflows vorgenommen und bestimmt jeweils nur die nächste auszuführende Gruppe. Während der Ausführung einer Gruppe wird dann parallel die Ausführungsplanung fortgeführt und die nächste auszuführende Gruppe bestimmt. Die Bestimmung der jeweils als nächstes auszuführenden Gruppe und der Execution Engine auf welcher diese Gruppe ausgeführt wird stellt einen Planungsschritt dar.

Dies ermöglicht zusätzlich eine adaptive Ausführungsplanung, welche auf sich ändernde Umstände in einem dynamischen Netzwerk reagieren kann. Für die Berechnung der Gruppenkosten werden in jedem Planungsschritt aktuelle Zuverlässigkeitswerte verwendet, die bei der sehr zeitnahen Ausführung der Gruppe noch wesentlich aktueller sind, als bei einer vollständigen und damit länger andauernden Ausführungsplanung. Ändern sich für eine Execution Engine die Zuverlässigkeitswerte von für die Synchronisationszuverlässigkeit relevanten Komponenten um einen gewissen Schwellenwert, kann außerdem eine erneute Berechnung der Synchronisationszuverlässigkeit ausgelöst werden. Dieser Schwellwert ist sinnvoll, da für höhere Replikationsgrade die Berechnungsdauer der Synchronisationszuverlässigkeit im Vergleich zur restlichen Ausführungsplanung signifikant sein kann.

Ein Planungsschritt sollte nie länger dauern, als das Ausführen der Gruppe, die im letzten Planungsschritt festgelegt wurde. Dauert ein Planungsschritt länger blockiert dies die Ausführung der nächsten Gruppe, bis der Planungsschritt beendet ist. Daher muss gegebenenfalls eine Mindestanzahl an Aktivitäten festgelegt werden, die von einer Execution Engine pro Planungsschritt ausgeführt werden müssen. Dann müsste eine Execution Engine gegebenenfalls mehr als eine Gruppe pro Planungsschritt ausführen. Alternativ könnte eine Mindestgruppenlänge festgelegt werden, wobei dies die Kosten der Workflowausführung noch weiter erhöhen kann. Im Extremfall, dass ein Planungsschritt länger dauert als das Ausführen aller im Ausführungsplan der *Line Of Sight* vorhandenen Gruppen, lässt sich das Blockieren der Workflow-Ausführung nicht verhindern. Im Folgenden vernachlässigen wir dieses Problem und nehmen die Möglichkeit einer zeitweisen Blockierung der Workflow-Ausführung in Kauf.

4.3.1 Planungsschritte

Workflowabschnitt Wir definieren eine Anzahl von aufeinander folgenden Aktionen des Workflows, die jeweils zusammen betrachtet werden sollen, wir nennen diese Anzahl $SightLength$. Außerdem definieren wir einen Referenzpunkt im Workflow, der definiert, welches die erste der aktuell betrachteten Aktionen ist. Der Referenzpunkt wird zunächst auf die erste Aktivität im Workflow gesetzt. Sodass zu Beginn die ersten $SightLength$ Aktionen betrachtet werden.

1.	Setze den Referenzpunkt auf die nächste auszuführende Aktivität.
2.	Betrachte ab dem Referenzpunkt die nächsten $SightLength$ Aktivitäten.
3.	Bestimme lokal die durchschnittliche Zuverlässigkeit für die Ausführung der betrachteten Aktivitäten
4.	Ermittle die Execution Engine mit dem besten Wert für 3.
5.	- Als Gewinner in 4. finde den optimalen lokalen Ausführungsplan für die betrachteten Aktivitäten - als Verlierer warte auf Synchronisation oder Timeout und mache dann bei 1. weiter.
6.	Verteile die erste Gruppe des Ausführungsplans unter den Execution Engines und warte auf Bestätigung der Mehrheit der Execution Engines
7.	Beginne mit der Ausführung der Gruppe

Tabelle 4.3: Übersicht eines Planungsschritts der Strategie Line Of Sight

Bestimmung der nächsten Execution Engine Jede Execution Engine e berechnet dann den Durchschnitt der Zuverlässigkeiten $r_{execAct}(e, a, s, N)$ der betrachteten Aktionen für eine lokale Ausführung. Um festzustellen welche Execution Engine die beste durchschnittliche Zuverlässigkeit für die betrachteten Aktionen hat, können die *Leader Election* Verfahren, wie für *Bestes Replikat* besprochen, verwendet werden.

Finden der nächsten Gruppe Diese Execution Engine berechnet dann den optimalen lokalen Ausführungsplan für die betrachteten Aktionen. Dabei werden wieder die Kosten aller möglichen Gruppen berechnet und mit einem *Shortest Path* Algorithmus der beste Ausführungsplan bestimmt. Erst an dieser Stelle wird die Synchronisationszuverlässigkeit benötigt.

Ausführung einer Gruppe und Fortsetzen der Planung Die Execution Engine schickt dann nur die erste Gruppe des Ausführungsplans an alle anderen Execution Engines. Anschließend wird diese erste Gruppe auf der Execution Engine direkt ausgeführt. Während der Ausführung der Gruppe führen alle Execution Engines die Ausführungsplanung weiter. Dafür wird der Referenzpunkt auf die nächste auszuführende Aktion nach dieser Gruppe gesetzt. Dann beginnt der Algorithmus unter den restlichen Execution Engines wieder mit der Bestimmung der Execution Engine für die nächste Gruppe, welche dann wiederum diese nächste Gruppe bestimmt und ausführt.

4.3.2 Beispiel

Abbildung 4.2 zeigt ein Beispiel für einen Planungsschritt. Die Gruppe deren Ausführung zuletzt gestartet wurde enthält a_{11} als letzte Aktivität. Daher befinden sich der Referenzpunkt zunächst

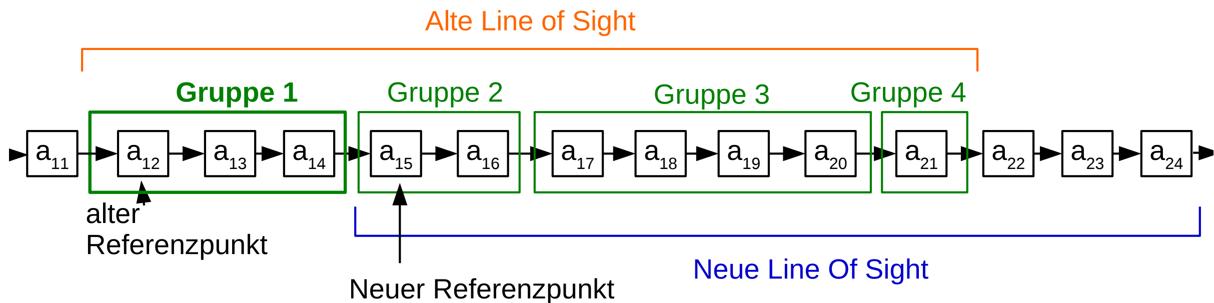


Abbildung 4.2: Beispiel eines Planungsschritts

auf Aktivität a_{12}. Die *Line Of Sight* startet somit bei a_{12} und erstreckt sich bei SightLength 10 bis zu a_{21}.

Für die Aktivitäten a_{12} bis a_{21} berechnen alle Execution Engines ihre jeweilige durchschnittliche Ausführungszuverlässigkeit. Die Execution Engine mit der besten Ausführungszuverlässigkeit wird für die Ausführung der nächsten Gruppe bestimmt und ermittelt diese, indem sie den Ausführungsplan für die aktuelle *Line Of Sight* berechnet.

Als Ergebnis erhalten wir die Gruppen 1 bis 4. Gruppe 1 wird im Netzwerk bekannt gemacht und anschließend ausgeführt, die restlichen Gruppen werden verworfen. Direkt nach Bekanntgabe von Gruppe 1 beginnt der nächste Planungsschritt. Der Referenzpunkt wird nun auf die nächste Aktivität nach Gruppe 1 gesetzt und so eine neue *Line Of Sight* festgelegt. Diese reicht nun von a_15 bis a_24.

Wie zu sehen ist überlappen die *Lines Of Sight* im Beispiel. Dies trifft in den meisten Fällen zu, es sei denn der Ausführungsplan eines Planungsschritts besteht nur aus einer einzigen Gruppe, welche dann die gesamte *Line Of Sight* abdecken würde.

4.3.3 Bewertung

Anzahl zu berechnender Gruppenkosten

Die Komplexität der Ausführungsplanung hängt für die beiden vorherigen Verfahren stark von der Länge des Workflows ab. Die Länge des Workflows diktiert wie viele mögliche Ausführungspläne es gibt und somit die Anzahl der möglichen Gruppen. Umso mehr mögliche Gruppen es gibt, umso länger dauert jeweils die Berechnung deren Kosten und die Suche nach dem kürzesten Pfad im entsprechenden Workflow-Graphen. Bei *Line Of Sight* kann diese Anzahl im Vergleich zu *Bestes Replikat* noch einmal deutlich verringert werden.

Die *Line Of Sight* Strategie betrachtet im Gegensatz zu den vorherigen Strategien nicht den gesamten Workflow sondern nur einen Teil mit der vom Benutzer bestimmten Länge $SightLength$. Einerseits kann dadurch die Ausführungsplanung beschleunigt werden, andererseits ist aber nicht mehr garantiert, dass die für den Ausführungsplan einer Execution Engine gewählten Gruppen optimal sind. Umso kürzer die SightLength gewählt wird umso stärker wirken sich beide Effekte aus.

Wird $SightLength = |A|$ gewählt, dann wird für jeden Planungsschritt der gesamte Workflow betrachtet. Wir gehen davon aus, dass keine Informationen über Gruppenkosten zwischen den Planungsschritten weitergegeben werden, sodass in jedem Planungsschritt die Kosten aller jeweils möglichen Gruppen berechnet werden müssen. In Fall $SightLength = |A|$ ist die Anzahl der zu berechnenden Gruppen in jeder Execution Engine im Vergleich zu *Bestes Replikat* um die Anzahl der Planungsschritte höher. Die Anzahl der Planungsschritte in *Line Of Sight* entspricht der Anzahl der Gruppen im resultierenden Ausführungsplan.

Im schlimmsten Fall ist die Anzahl der Gruppen im Ausführungsplan gleich der Anzahl der Aktivitäten im Workflow, nämlich dann wenn in jedem Planungsschritt die ermittelte Gruppenlänge 1 ist. Im besten Fall gilt $Planungsschritte = \frac{Workflow-Länge}{SightLength}$, was der Fall wäre, wenn stets alle betrachteten Aktivitäten zu einer einzigen Gruppe zusammengesetzt würden. Die Anzahl der Gruppen $G \in \Omega(W)$, für welche bei der Planung die Kosten berechnet werden müssen, bestimmen wir dann folgendermaßen.

$$|G| = Planungsschritte \cdot \frac{SightLength^2 + SightLength}{2}, Planungsschritte <= |A| \tag{4.1}$$

Je nach Länge der zur Ausführung ermittelten Gruppen im Vergleich zur Länge der *Line Of Sight* kann es von einem zum nächsten Planungsschritt große Überlappungen bei den Gruppen, deren Kosten berechnet werden müssen, geben. Falls die durchschnittliche Länge der zur Ausführung bestimmten Gruppen wesentlich kleiner als die Länge der *Line Of Sight* ist, gibt es hier deutliches Optimierungspotential. So könnten die Kosten für die Gruppen für den nächsten Planungsschritt gespeichert werden, sodass bereits berechnete Gruppenkosten nicht neu berechnet werden müssen. Wir gehen hier jedoch davon aus, dass diese Optimierung nicht implementiert wird.

Wahl der SightLength

Die Abbildung 4.3 zeigt die Auswirkung der Wahl der $SightLength$. Umso höher die $SightLength$ gewählt wird, umso genauer ist die Bestimmung der optimalen nächsten Gruppe für eine Execution Engine. Umso niedriger diese gewählt wird umso weniger Berechnungen müssen durchgeführt werden und die Zeit für Ausführungsplanung verkürzt sich für jeden Schritt. Die Anzahl der Schritte hängt dabei von den Längen der Gruppen ab, die in jedem

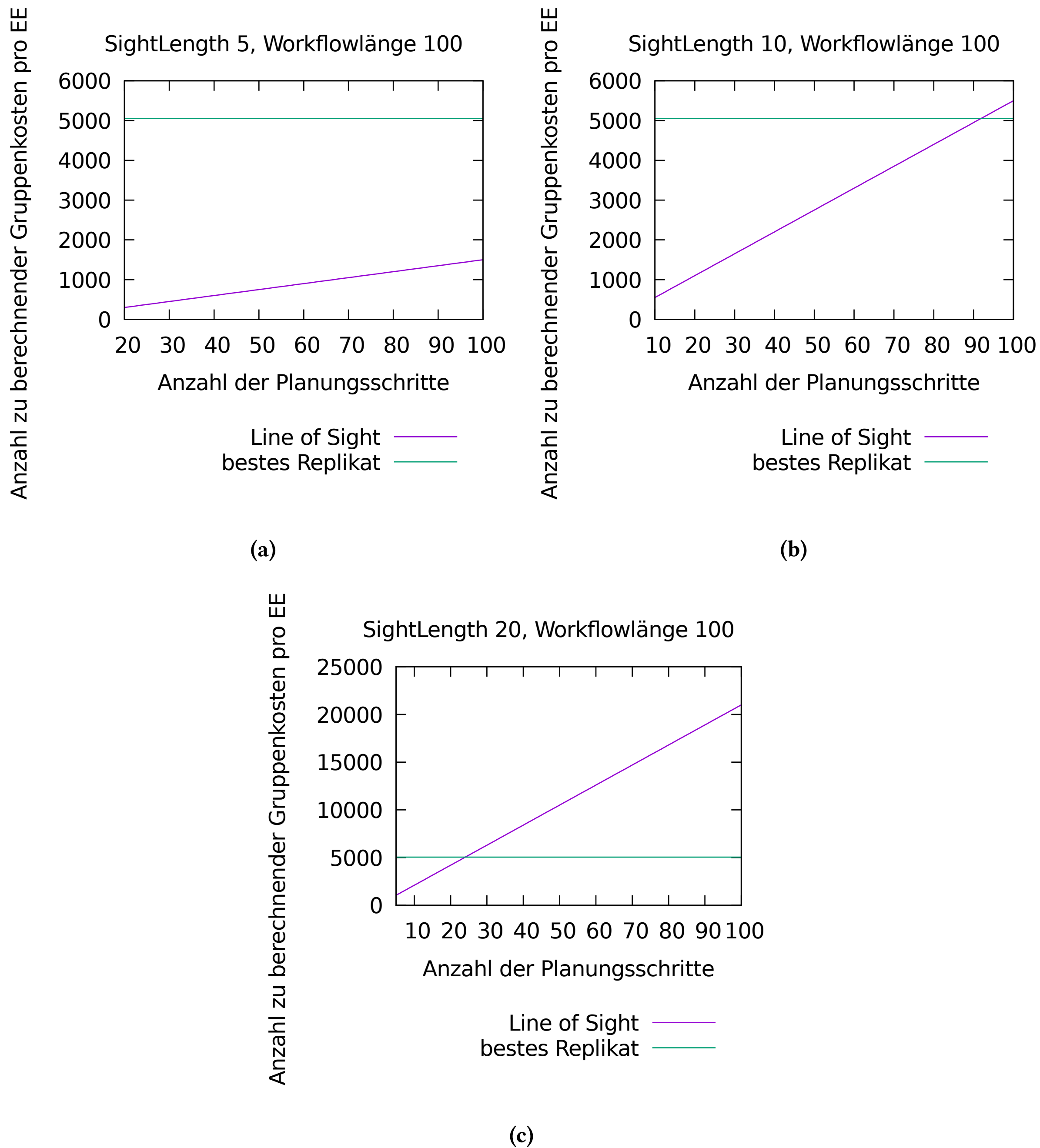


Abbildung 4.3: Vergleich der Anzahl zu berechnender Gruppenkosten pro Execution Engine in Abhängikeit der gewählten SightLength und der Anzahl der Planungsschritte. Als Referenz dient dieselbe Zahl bei Verwendung von *Bestes Replikat*

Schritt bestimmt werden. Die Wahl der SightLength stellt einen Kompromiss zwischen Qualität des Ausführungsplans und der Dauer der Ausführungsplanung dar.

Wie wir in Abbildung 4.3 sehen können führt die SightLength 5 bei einer Workflow-Länge von 100 immer zu einer geringeren Anzahl an Berechnungen von Gruppenkosten, während das bei einer SightLength von 20 und gleicher Workflow-Länge bereits bei nur weniger als 25 Planungsschritten der Fall ist. SightLength 10 führt in den meisten Fällen außer bei mehr als 91 Planungsschritten zu weniger Berechnungen pro Execution Engine als bei der Strategie *Bestes Replikat*. Je nach Netzwerk und Workflow fällt die durchschnittliche Länge von optimale Gruppen und damit die Anzahl der Planungsschritte unterschiedlich aus. Sind typische Gruppenlängen in einem praktischen Anwendungsfall bekannt, kann der Benutzer die SightLength passend wählen.

Adaptive Ausführungsplanung

Die Zuverlässigkeiten der Netzwerkkomponenten können sich in einem dynamischen Netzwerk schnell ändern. Die Ausführungsplanung bei den Strategien *Optimaler Ausführungsplan* und *Bestes Replikat* sind langwierig und werden immer als ganzen ausgeführt, bevor mit der Workflowausführung begonnen werden kann. Bei der Ausführung der letzten Gruppe könnte sich der Zustand des Netzwerks bereits stark verändert haben.

Bei *Line Of Sight* werden die Zuverlässigkeitswerte für die Berechnung der Gruppenkosten stets während der Ausführung der vorherigen Gruppe erfasst und sind somit insgesamt wesentlich aktueller. Die Synchronisationszuverlässigkeit wird dabei nur neu berechnet, wenn die Änderung dieser Werte einen bestimmten Schwellenwert überschreitet. Dadurch kann man verhindern, dass die Berechnungsdauer der Synchronisationszuverlässigkeit nicht den Gewinn an Ausführungszeit der nachfolgenden Gruppen überschreitet.

4.3.4 Abwandlungen der Strategie im Bezug auf die Bestimmung der ausführenden Execution Engines

Die Execution Engine, welche die nächste Gruppe ausführen soll, wird wie bereits beschrieben anhand des Durchschnitts der Zuverlässigkeiten für die Ausführung der Aktivitäten in der *Line Of Sight* ermittelt. Die Wahl dieses Vergleichskriteriums folgt einer groben Vermutung, dass die so bestimmte Execution Engine mit hoher Wahrscheinlichkeit eine Gruppe mit nahezu optimalen Kosten ermittelt.

Zunächst ist hervorzuheben, dass bei dieser Methode zur Bestimmung der nächsten Execution Engine die Synchronisationszuverlässigkeit nicht in Betracht gezogen wird. Diese kann je nach Dauer von t_{sync} in erheblichem Maße zu den Kosten einer Gruppe beitragen.

Daneben ist nicht sichergestellt, dass die im Planungsschritt ermittelte Gruppe tatsächlich auf der Execution Engine ausgeführt wird, welche dies mit der höchsten Zuverlässigkeit tun kann,

da sich der Vergleich der Durchschnittszuverlässigkeiten auf alle Aktivitäten in der *Line Of Sight* bezieht und nicht nur auf diese in der hinterher bestimmten Gruppe. Wir schlagen an dieser Stelle Alternativen für die Bestimmung der jeweiligen Execution Engine vor.

Zunächst könnte jede Execution Engine direkt einen Ausführungsplan für die aktuelle *Line Of Sight* berechnen. Dieser Schritt muss ohnehin vor Ausführung der nächsten Gruppe von mindestens einer Execution Engine ausgeführt werden und wird an dieser Stelle nur vorgezogen. Sind alle Execution Engines gleich schnell beim bestimmen von Ausführungsplänen, dann führt dies auch zu keiner längeren Planung.

Anschließend können entweder die Kosten des gesamten Ausführungsplans verglichen werden oder für die erste Gruppe ein Durchschnittswert der Kosten pro Aktivität oder pro Ausführungszeit der Aktivitäten ermittelt werden, um dann diesen zu vergleichen. Die erste Variante teilt das Problem mit der bestehenden Methode, dass die Kosten sich auf den ganzen Ausführungsplan beziehen und die erste Gruppe von diesem Durchschnitt stark abweichen könnte, dafür ist die Synchronisationszuverlässigkeit hier mit einbezogen. Die zweite Variante vergleicht die tatsächlich produzierten Gruppen und macht diese trotz möglicher unterschiedlicher Gruppenlängen durch die Metrik der Durchschnittskosten vergleichbar.

Beide Varianten fügen keine neuen komplexen Berechnungen hinzu und sollten die Dauer des Planungsschritts daher auch nicht signifikant verlängern.

Als dritte Variante, wäre es möglich die Werte aller ermittelten Gruppen samt ihrer Kosten und zugehöriger Execution Engine im Netzwerk zu verteilen. Dies würde vor der Ermittlung des kürzesten Pfades passieren. Für jede Execution Engine wären dies, für eine Länge der *Line Of Sight* von z, $\frac{z^2+z}{2}$ Gruppen, was den Kommunikationsaufwand unter den Execution Engines deutlich erhöhen würde. Anschließend würde jede Execution Engine all diese Gruppen in den Workflow-Graphen einfügen und den kürzesten Pfad berechnen. Als Resultat bekäme man für die betrachtete *Line Of Sight* einen optimalen Ausführungsplan. Die Execution Engine der ersten Gruppe in diesem Ausführungsplan kann diese dann ausführen. Die folgenden Gruppen werden verworfen und es beginnt der nächste Planungsschritt.

4.4 Beste Gruppen

"Beste Gruppen" stellt eine Möglichkeit für eine verteilte Berechnung des optimalen Ausführungsplans dar. Im Gegensatz zum zentralisierten Ansatz wird kein globales Wissen benötigt. Die Anforderungen für eine Execution Engine bestehen hier, wie bei *Bestes Replikat*, darin alle direkten Nachbarn und die Verbindungen von und zu diesen zu kennen. Das Ergebnis der Strategie ist dennoch der optimale Ausführungsplan und damit identisch zu dem der ersten vorgestellten Strategie.

Zusätzlich liefert die Strategie suboptimale Ausführungspläne als Zwischenergebnisse, was es ermöglicht die Ausführungsplanung vorzeitig zu beenden, falls ein zufriedenstellendes

Ergebnis früh erreicht ist oder die Planung zu lange dauert. Hierfür könnte ein Zielwert für die Kosten des Ausführungsplans definiert werden. Findet eine Execution Engine einen Ausführungsplan welcher diesen Zielwert erreicht oder unterschreitet, kann dieser direkt allen Execution Engines mitgeteilt werden. Die Ausführungsplanung ist dann beendet und mit der Workflow-Ausführung kann begonnen werden. Dadurch kann eine geforderte Qualität des Ausführungsplans garantiert und gleichzeitig die nötige Planungszeit minimiert werden. Auf die Möglichkeit Zwischenergebnisse als Ausführungspläne zu verwenden, gehen wir im folgenden allerdings nicht weiter ein, sondern betrachten den Weg zum optimalen Plan im Detail. Das Verfahren versucht außerdem einen möglichst geringen Kommunikationsaufwand zwischen den Execution Engines bei der Planung zu erreichen. Wir beschreiben zunächst den groben Ablauf der Strategie.

4.4.1 Grober Ablauf

Tabelle 4.4 zeigt eine Zusammenfassung des Ablaufs der Strategie. Jede Execution Engine berechnet für jede mögliche Gruppe jeweils die Kosten für eine lokale Ausführung (Schritt 1). All diese Gruppen werden in der Execution Engine zur Menge der potentiell optimalen Gruppen hinzugefügt. Als optimale Gruppen definieren wir die Gruppen, welche im optimalen Ausführungsplan enthalten sind. In der Menge der potentiell optimalen Gruppen befinden sich alle Gruppen, bei welchen die Execution Engine nicht ausschließen kann, dass diese im optimalen Ausführungsplan enthalten sind.

Danach wird für jede Gruppenlänge, die Gruppe mit den geringsten Kosten ermittelt und samt Kosten und zugehöriger Execution Engine mit allen anderen Execution Engines geteilt (Schritte 2 & 3). Alle empfangenen Gruppen werden gespeichert und die Werte für die geringsten Kosten einer Gruppenlänge werden anhand der empfangenen Gruppen aktualisiert. Außerdem bildet die Execution Engine für die Gruppen für welche die Kosten auf einer anderen Execution Engine bekannt geworden sind jeweils $f_{minGroup}(g, N)$. So werden die global geringsten Kosten einer Gruppenlänge bestimmt (Schritt 4).

Daraufhin wird für den bisherigen Wissenstand der Execution Engine der vorläufig optimale Ausführungsplan per *Shortest Path* Algorithmus berechnet und dessen Kosten als Referenzkosten gespeichert (Schritt 5). Dieser Referenzplan und dessen Kosten werden mit den anderen Execution Engines geteilt, woraufhin diese jeweils ihre Referenzkosten aktualisieren (Schritte 6 & 7).

Als nächstes wird durch zwei, später genauer erläuterte, Verfahren versucht die Anzahl der Kanten im Workflow-Graphen, durch welche der kürzeste Pfad und damit der Ausführungsplan berechnet wird, stark zu reduzieren (Schritte 8 & 9). Dafür werden Gruppen entfernt, die auf keinen Fall Teil des optimalen Ausführungsplan sein können. So wird Dauer der Berechnung des kürzesten Pfads und somit des optimalen Ausführungsplans stark verkürzt. Hierfür werden Gruppen unterschiedlicher Länge untereinander verglichen und insbesondere das Wissen um die Mindestkosten einer Gruppenlänge genutzt.

1.	Jede Execution Engine e bestimmt alle möglichen Gruppen $g \in \Omega(W)$. und berechnet jeweils die Kosten $f_{group}(e, g, N)$.
2.	Bestimme für jede Gruppenlänge die Gruppe mit den geringsten Kosten.
3.	Teile die Gruppen aus 3. mit allen anderen Execution Engines.
4.	Bestimme die geringsten Kosten pro Gruppenlänge, anhand der globalen Werte.
5.	Ermittle den nach bisherigem Wissensstand optimalen Ausführungsplan.
6.	Teile den Ausführungsplan aus 5. mit allen anderen Execution Engines.
7.	Bestimme den besten der empfangenen Ausführungspläne und verwende dessen Kosten als Referenz.
8.	Entferne schlechte Gruppen aus der Menge der potentiell optimalen Gruppen.
9.	Bestimme alle unmöglichen Gruppenlängenkombinationen.
10.	Vertile die potentiell optimalen Gruppen, sodass jede Execution Engine für alle verbleibenden Gruppen $fminGroup(g, N)$ berechnen kann. Wiederhole dabei bei Empfang neuer Gruppen vor dem Senden jeweils die Schritte 8 und 9.
11.	Berechne den optimalen Ausführungsplan.

Tabelle 4.4: Ablauf der Strategie Beste Gruppen

In der letzten Phase werden die verbleibenden Gruppen so zwischen den Execution Engines geteilt, dass jede schließlich für alle Gruppen die minimalen Kosten $f_{minGroup}(g, N)$ berechnen kann (Schritt 10). Daraufhin wird auf jeder Execution Engine ein *Shortest Path* Algorithmus ausgeführt, um den optimalen Ausführungsplan zu bestimmen (Schritt 11).

Um den optimalen Ausführungsplan in jeder Execution Engine zu erhalten, sind lediglich die Schritte 1, 10 und 11 nötig. Die Schritte 2 bis 9 werden angewendet, um den Kommunikationsaufwand zu verringern und die Berechnung des optimalen Ausführungsplan per *Shortest Path* Algorithmus zu beschleunigen, indem die Anzahl der Kanten reduziert wird.

4.4.2 Schlechte Gruppen

Als eine schlechte Gruppe bezeichnen wir eine Gruppe g_i, deren Kosten höher ist als die Summe der Kosten einer Menge kleinerer nicht überlappender Gruppen, welche zusammen die selben Aktivitäten wie g_i, enthalten. Als Beispiel betrachten wir die Aktivitäten a_1 und a_2 mit den Gruppenkosten $f_{group}(e, a_1, N) = 15$, $f_{group}(e, a_2, N) = 25$ und $f_{group}(e, a_1, a_2, N) = 50$. e ist dabei die Execution Engine im Netzwerk N, welche Schritt 8 des Ablaufs ausführen soll. Die Kosten der Gruppe mit a_1 und a_2 betragen 50, während die Summe der Kosten der Gruppen, welche jeweils nur a_1 beziehungsweise a_2 enthalten nur 40 beträgt. In einem solchen Fall wird die Gruppe $\{a_1, a_2\}$ nie im optimalen Ausführungsplan enthalten sein, da die Alternative die

Aktivitäten stattdessen einzeln auszuführen immer möglich ist. In diesem Fall wird die Gruppe $\{a_1, a_2\}$ aus der Menge der potentiell optimalen Gruppen entfernt.

Aus dem umgekehrten Fall können wir nie auf eine Gruppe schließen, die nicht im Ausführungsplan enthalten sein kann. Sollte die Gruppe mit a_1 und a_2 geringere Kosten verursachen als die Einzelausführung, so können wir die Einzelgruppen dennoch nicht verwerfen. Da stattdessen zum Beispiel die Gruppen $\{a_0, a_1\}$ und $\{a_2\}$ im optimalen Ausführungsplan enthalten sein könnten, weil unter anderem $f_{group}(e, a_0, a_1, N) + f_{group}(e, a_2, N) < f_{group}(e, a_0, N) + f_{group}(e, a_1, a_2, N)$ weiterhin möglich ist, also im gleichen Fall die Gruppen $\{a_0, a_1\}$ und $\{a_2\}$ zusammen geringere Kosten haben können, als $\{a_0\}$ und $\{a_1, a_2\}$.

Für die Ausführung dieses Schritts prüft die Execution Engines für alle Gruppen, welche in der Menge der potentiell optimalen Gruppen enthalten sind und mehr als eine Aktivität beinhalten, ob diese schlechte Gruppen sind. Die Execution Engine beginnt für einen Workflow mit mehr als einer Aktivität mit der Gruppenlänge 2 und bearbeitet Gruppenlängen in aufsteigender Reihenfolge. Für jede Gruppe der aktuellen Gruppenlänge wird unter Betrachtung der Kosten aller in der Menge der potentiell optimalen Gruppen enthaltenen Teilgruppen, geprüft, ob es sich um eine schlechte Gruppe handelt. Schlechte Gruppen werden direkt aus der Menge der potentiell optimalen Gruppen entfernt. Durch die Bearbeitung in aufsteigender Reihenfolge wird die Menge der Teilgruppen für höhere Gruppenlängen beim Finden von schlechten Gruppen reduziert. Somit ist dies effizienter als die Gruppenlängen in absteigender Reihenfolge zu betrachten.

4.4.3 Unmögliche Gruppenlängenkombinationen

Wir definieren eine Gruppenlängenkombination als hypothetischen Ausführungsplan, dessen Gruppen jeweils die Mindestkosten der jeweiligen Gruppenlängen zugewiesen bekommen. Die Kosten einer Gruppenlängenkombination stellen eine Untergrenze der Kosten des entsprechenden Ausführungsplans dar.

Als Beispiel dient die Gruppenlängenkombination mit den Gruppen a_0, a_1, a_2, a_3, a_4. Die geringsten bekannten Kosten für eine Gruppe der Länge 1 betragen 20 und für die Länge 2 50. Die Gesamtkosten der Gruppenlängenkombination betragen 120. Diese Kosten werden mit dem Referenzwert aus Schritt 7 verglichen. Ist Letzterer geringer so wird die Gruppenlängenkombination als ungültig definiert. Wurden alle Gruppenlängenkombinationen, welche eine bestimmte Gruppe enthalten, als ungültig definiert, kann die entsprechende Gruppe aus der Menge der potentiell optimalen Gruppen entfernt werden.

Für die Ausführung dieses Schritts werden alle möglichen Gruppenlängenkombinationen auf Unmöglichkeit geprüft. Zunächst befinden sich alle Gruppenlängenkombinationen in der Menge der möglichen Gruppenkombinationen. Unmögliche Gruppenlängenkombinationen werden von der Menge der möglichen Gruppenkombinationen entfernt. Anschließend wir für jede Gruppe, welche sich in der Menge der potentiell optimalen Gruppen befindet, geprüft, ob

diese noch in einer der möglichen Gruppenlängenkombinationen enthalten ist. Ist dies nicht der Fall, kann die Gruppe aus der Menge der potentiell optimalen Gruppen entfernt werden.

4.4.4 Verteilen der potentiell optimalen Gruppen

Wir bezeichnen die Menge der potentiell optimalen Gruppen auf einer Execution Engine e_i als $G_{pot}(E_{included})$ mit $e_i \in E_{included}$. $E_{included}$ enthält dabei e_i selbst und alle Execution Engines welche in zuvor von e_i bereits empfangenen $G_{pot}(E_{included})$ in den jeweiligen $E_{included}$ enthalten waren.

Eine Execution Engine e_i sendet zunächst allen ihren Nachbarn ihre Menge $G_{pot}(E_{included})$ zusammen mit $E_{included}$. Hierbei sind außerdem jeder Gruppe ihre Ausführungskosten und die entsprechende Execution Engine e_{exec} angefügt.

Dadurch erhält eine Execution Engine e_j neue Kosten für die Ausführung von Gruppen auf anderen Execution Engines und berechnet, um die empfangene Menge $G_{pot}E_{included,i})$ mit der eigenen Menge $G_{pot}(E_{included,j})$ zusammenzuführen, zunächst für jede der Gruppen in beiden Mengen die Kosten $f_{tempMinGroup}(g, N) = min_{e \in E_{included}} f_{group}(g, N)$ mit $E_{included} = E_{included,i} + E_{included,j}$.

Anschließend werden die Schritte 8 und 9 für die neue Menge $G_{pot}(E_{included})$ noch einmal ausgeführt, um anhand der neuen Daten, diese Menge noch einmal zu verringern.

Anhand der von ihr empfangenen und gesendeten Nachrichten kennt eine Execution Engine einen Teil der $E_{included}$ für jeden ihrer Nachbarn. Enthält das eigene $E_{included}$ eine Execution Engine welche in der $E_{included}$ eines Nachbarn nicht enthalten ist, wird diesem die neue Menge $G_{pot}(E_{included})$ zusammen mit $E_{included}$ gesendet.

Nach einer begrenzten Anzahl von Nachrichten enthält $E_{included}$ auf jeder Execution Engine alle Execution Engines im Netzwerk. Sobald dies auf einer Execution Engine jeweils der Fall ist, kann mit $G_{pot}(E_{included})$ und einem *Shortest Path* Algorithmus der optimale Ausführungsplan berechnet werden. Entsprechend der Verteilung der Gruppen $G_{pot}(E)$ ist dieser korrekt und somit auf allen Execution Engines gleich.

Stellt eine Execution Engine e_{prime} fest, dass die erste Gruppe laut Ausführungsplan auf e_{prime} ausgeführt werden soll, beginnt diese direkt mit der Ausführung der Gruppe. Alle anderen Execution Engines warten hingegen darauf, dass e_{prime} nach der Ausführung der ersten Gruppe synchronisiert.

Der Umstand, dass die Zeit zu der die Ausführungsplanung auf den Execution Engines jeweils beendet ist, je nach Netzwerk stark divergieren kann, muss bei der Wahl eines Timeouts für die Synchronisierung der ersten Gruppe beachtet werden.

4.4.5 Bewertung

Die Strategie Beste Gruppen bietet eine verteilte Alternative zur Berechnung des optimalen Ausführungsplans, wo kein globales Wissen an einer Stelle verfügbar sein muss. Die Planungszeit lässt sich dadurch vermutlich deutlich verkürzen, wobei das Finden von schlechten Gruppen und unmöglichen Gruppenlängenkombinationen einen Mehraufwand darstellt. Dieser Mehraufwand fällt für höhere Replikationsgrade allerdings weniger ins Gewicht.

Daneben lassen sich wie zuvor erläutert, die Schritte 2 bis 9 vermeiden und dennoch der optimale Ausführungsplan erhalten. Ist das Netzwerk im Vergleich zu den verfügbaren Rechenkapazitäten sehr leistungsfähig, lässt sich dann die Planungszeit im Vergleich zu *Optimaler Ausführungsplan*nahezu um den Faktor des Replikationsgrads verringern.

5 Evaluation

Um die Auswirkungen der besprochenen Strategien in einer realen Umgebung abschätzen zu können wurden diese implementiert und ausgeführt.

Die Strategien *Optimaler Ausführungsplan*, *Bestes Replikat* und *Line Of Sight* wurden durch Durchführung ihrer Ausführungsplanung und anschließender Ausführung des Workflows anhand der resultierenden Ausführungspläne getestet. Eine Ausführungsplanung und die folgende Workflowausführung anhand des erstellten Ausführungsplans bezeichnen wir als einen Testdurchlauf. Gleichermaßen wurde als Benchmark ein Algorithmus welcher zufällige Ausführungspläne ausgibt getestet. Für jede Strategie und das Benchmark wurden jeweils die Planungsdauer, die Ausführungsdauer des Workflows und die Anzahl der Gruppen im Ausführungsplan gemessen.

Alle Tests wurden lokal auf einem einzelnen Rechner ausgeführt. Die Ausführungsplanung wurde jeweils zentralisiert implementiert. Für die Ausführung der Workflows anhand der generierten Ausführungspläne wurde das Netzwerk anhand zufällig gewählter Zuverlässigkeiten der Netzwerkkomponenten simuliert. Im Folgenden erläutern wir zunächst die Implementierung des simulierten Netzwerks sowie der jeweiligen der Strategien und der zufälligen Ausführungsplanung.

Anschließend stellen wir die Ergebnisse der durchgeführten Messungen vor und vergleichen dabei für jede verwendete Metrik die Werte der unterschiedlichen Strategien und Replikationsgrade.

5.1 Implementierung des Netzwerkmodells

Die Implementierung des Netzwerkmodells beinhaltet einerseits die Generierung eines zufälligen Netzwerks von Exectuion Engines und Services, als auch die Simulation von Ausfällen der einzelnen Komponenten anhand ihrer Zuverlässigkeit. Für Letzteres wurden nur die Fälle betrachtet, die eine Auswirkung auf die Workflowausführung haben. Diese sind das Fehlschlagen eines Serviceaufrufs, das Ausfallen einer Execution Engine während der Ausführung einer Aktivität ohne Serviceaufruf und der Ausfall einer Verbindung zwischen den Execution Engines in dem Moment, wo eine Nachricht zwischen diesen gesendet werden soll.

5.1.1 Erstellen eines zufälligen Netzwerks

Für jeden Testdurchlauf, also dem Erstellen eines Ausführungsplans und der anschließenden Workflow-Ausführung anhand dieses Ausführungsplans, wird ein zufälliges Netzwerk N wie folgt generiert.

In Abhängigkeit des Replikationsgrads werden entsprechend viele Execution Engines angelegt. Jede Execution Engine e erhält als Zuverlässigkeit $r(e)$ einen zufälligen Wert zwischen $0,9$ und 1. Zwischen jedem Paar von Execution Engines e_i und e_j werden jeweils die Verbindungen l_{e_i,e_j} und l_{e_j,e_i} angelegt. Für $r_{l_{e_i,e_j}}$ und $r_{l_{e_j,e_i}}$ werden ebenfalls zufällige Werte zwischen $0,9$ und 1 festgelegt.

Unabhängig vom Replikationsgrad werden außerdem jeweils 5 Services angelegt. Jeder Service s erhält als Zuverlässigkeit $r(s)$ einen zufälligen Wert zwischen $0,9$ und 1. Für jede Execution Engine e werden jeweils zwei Verbindungen $l_{e,s}$ und $l_{s,e}$ zu jedem der fünf Services $s \in S$ angelegt. Jede dieser Verbindungen l erhält als Zuverlässigkeit $r(l)$ wiederum einen zufälligen Wert zwischen $0,9$ und 1. Jedem Service wird zudem eine Menge von Aktivitäten zugewiesen. Diese Mengen können sich überschneiden und nicht jeder Aktivität muss ein Service zugewiesen werden.

5.1.2 Fehlschlagen eines Serviceaufruf

Algorithmus 5.1 Fehlschlagen eines Serviceaufruf

```
1: wait for executionTime(a)
2: for all Services of a_k AS s do
3:     executionReliability[s] = r(e) * r(s) * r(l_e, s) * r(l_s, e);
4: end for
5: if Rand[0..1] > max(executionReliability) then
6:     go unrepsonsive for 1500ms
7: end if
```

Der Algorithmus 5.1 beschreibt die Implementierung, um ein Fehlschlagen eines Serviceaufrufs zu simulieren. Zunächst wartet der Algorithmus die Ausführungszeit der Aktivität ab, um einen Ausfall, falls er eintritt, zum Ende der Ausführung zu simulieren. Die Zuverlässigkeit eines Serviceaufrufs wird für einen Service als $executionReliability[s]$ berechnet. Unter den kompatiblen Services wird angenommen, dass die Execution Engine e den Service für einen Aufruf wählt, für welchen diese Zuverlässigkeit am höchsten ist. Ist die zufällig gewählte Zahl zwischen 0 und 1 größer als $max(executionReliability)$ wird ein Fehlschlagen des Serviceaufrufs simuliert und die Execution Engine bleibt unerreichbar für 1500ms.

5.1.3 Ausfall einer Execution Engine beim Ausführen einer Aktivität ohne Serviceaufruf

Algorithmus 5.2 Fehlschlagen einer lokalen Ausführung einer Aktivität a auf der Execution Engine e

1: wait for executionTime(a)
2: executionReliability = $r(e)$;
3: **if** Rand[0..1] > executionReliability **then**
4: go unrepsonsive for 1500ms
5: **end if**

Der Algorithmus 5.2 beschreibt die Implementierung, um ein Ausfall einer Execution Engine bei der Ausführung einer Aktivität ohne Serviceaufruf zu simulieren. Die Schritte sind, bis auf die Berechnung der Zuverlässigkeit, die selben wie beim Ausführen mit Serviceaufruf.

5.1.4 Ausfall einer Verbindung zwischen Execution Engines

Algorithmus 5.3 Fehlschlagen der Kommunikation zwischen Execution Engines e_i und e_j anhand der Nachricht m implementiert beim Sendevorgang in e_i

1: message m is in message queue
2: linkReliability = $r(l_{e_i,e_j})$;
3: **if** Rand[0..1] < linkReliability **then**
4: send message m
5: **end if**
6: remove message m from message Queue

Für den Ausfall einer Verbindung zwischen zwei Execution Engines wird, wie im Algorithmus 5.3 beschrieben, eine Nachricht aus der Queue des Senders entfernt, ohne die Nachricht zu versenden. Um festzustellen, ob eine Verbindung zum Sendezeitpunkt ausgefallen ist oder nicht, wird die Verbindungszuverlässigkeit $linkReliability$ berechnet und mit einer Zufallszahl verglichen.

5.2 Implementierung der Strategien

5.2.1 Synchronisationszuverlässigkeit

Algorithmus 5.4 zeigt die Implementierung der Synchronisationszuverlässigkeit, wie sie in den Strategien *Optimaler Ausführungsplan*, *Bestes Replikat* und *Line Of Sight* verwendet wird.

Algorithmus 5.4 Synchronisationszuverlässigkeit

```
1:  procedure SyncReliability(N,e_master)                         // r_sync(N, e)
2:      syncReliability := 0;
3:      for all ReplicaStates do // |ReplicaStates| = 2^ReplicationDegree
4:          if (master itself is alive) then
5:              count := 1;
6:              stateProbability := r(e_master);
7:              for all other Replicas AS e_other do // Replicas that are not master
8:                  commReliability := r(e_other) * r(l_{e_master,e_other}) * r(l_{e_other,e_master});
9:                  if (replica is alive in this ReplicaState) then
10:                     stateProbability := stateProbability * commReliability;
11:                     count := count + 1;
12:                 else
13:                     stateProbability := stateProbability * (1 - commReliability);
14:                 end if
15:             end for
16:             if (count > |E|/2) then
17:                 syncReliability := syncReliability + stateProbability;
18:             end if
19:         end if
20:     end for
21:     return syncReliability
22: end procedure
```

Ziel ist es alle Netzwerkzustände zu finden, in welchen mit mehr als der Hälfte der Replikate synchronisiert werden kann und deren jeweiligen Eintrittswahrscheinlichkeiten zu addieren. Die Summe gibt dann die Synchronisationswahrscheinlichkeit an.

Damit eine Execution Engine e_{master} mit einer anderen Execution Engine $e_{replica}$ synchronisieren kann, müssen sowohl diese beiden Execution Engines, als auch die beiden Verbindungen zwischen diesen betriebsbereit sein. Um die Anzahl der zu betrachtenden Zustände zu verringern, betrachten wir nur die Fälle, wo für eine Execution Engine e_{master} eine andere Execution Engine e_{other} erreichbar ist. Hierfür müssen zusätzlich zu e_{master} auch e_{other} und die beiden Verbindungen zwischen diesen betriebsbereit sind.

Ein $ReplicaState$ gibt für jede Execution Engine an, ob diese mit $r(e_{master})$ synchronisieren kann. Die Wahrscheinlichkeit hierfür beträgt $r_{partialSync}(N, e_{master}, e_{other}) = r(e_{master}) \cdot r(e_{other}) \cdot r(l_{e_{master},e_{other}}) \cdot r(l_{e_{other},e_{master}})$. Ein $ReplicaState$ kann als Bitstring repräsentiert werden. Ein Bitstring der Form 10110 würde für fünf Replikate angeben, dass Replikat 1,3 und 4 erreichbar sind und für 2 und 5 mindestens eine Verbindung, oder die Execution Engine selbst ausgefallen ist. Eines der Bits repräsentiert e_{master} selbst, wobei *erreichbar* hier bedeutet, dass diese nicht ausgefallen ist. Die Wahrscheinlichkeit hierfür beträgt entsprechend $r(e_{master})$.

Für einen Replikationsgrad $ReplicationDegree$ gibt es entsprechend $2^{ReplicationDegree}$ mögliche zu betrachtende Zustände. In jedem Zustand wird zunächst geprüft, ob e_{master} selbst erreichbar ist. Ist dies nicht der Fall, muss dieser Zustand nicht weiter betrachtet werden. Ist e_{master} erreichbar, wird zusätzlich für alle anderen Replikate geprüft, ob diese erreichbar sind. Dabei wird die Eintrittswahrscheinlichkeit $stateProbability$ für diesen Zustand berechnet. Dazu wird das Produkt der Wahrscheinlichkeiten der einzelnen Replikatszustände $commReliability$ oder $1 - commReliability$, falls dieses Replikat nicht erreichbar ist, gebildet. Sind mehr als $|E|/2$ Replikate, inklusive e_{master} erreichbar, wird die Eintrittswahrscheinlichkeit dieses Zustands zur Summe, welche die Synchronisationszuverlässigkeit bestimmt, hinzu addiert.

5.2.2 Optimaler Ausführungsplan

Die Implementierung der Strategie *Optimaler Ausführungsplan* ist in Algorithmus 5.5 dargestellt. Zunächst wird für jede Execution Engine die Synchronisationszuverlässigkeit, wie in Algorithmus 5.4 beschrieben, berechnet. Anschließend werden alle möglichen Gruppen des Workflows betrachtet. Hierfür werden für jede Aktivität $a_i \in A$ alle Gruppen erfasst, welche mit a_i beginnen und mit derselben oder einer der nachfolgenden Aktivitäten, also $a_j \in A$ mit $j >= i$, enden. Dann wird für jede dieser Gruppen, jeweils die beste Execution Engine und dafür wiederum der beste Service bestimmt, um die Zuverlässigkeit $f_{minGroup}(g, N)$ zu berechnen.

Alle Gruppen werden dann mit ihren errechnenten Kosten $f_{minGroup}(g, N)$ als Gewicht und der bestimmten Execution Engine als zusätzliches Attribut jeweils als Kanten in den Workflow-Graphen eingefügt. Im Workflow-Graphen wird dann mit dem Dijkstra Algorithmus der kürzeste Pfad von der ersten bis zur letzten Aktivität des Workflows bestimmt. Anschließend wird die entsprechende Gruppe für jede Kante des resultierenden Pfades in einen Ausführungsplan eingefügt, welcher dann der optimale Ausführungsplan ist.

5.2.3 Bestes Replikat

Die Implementierung der Strategie *Bestes Replikat* für die Ausführung des gesamten Workflows auf einer einzigen Execution Engine ist in Algorithmus 5.6 dargestellt. Der Inhalt der äußersten Schleife stellt jeweils die Ausführungsplanung, wie sie auf der entsprechenden Execution Engine e auszuführen wäre, dar. Im Gegensatz zur Implementierung von *Optimaler Ausführungsplan* kann $f_{minGroup}(g, N)$ hier nicht berechnet werden, sondern nur $f_{group}(e, g, N)$, da jeweils nur eine Execution Engine betrachtet wird. Die ermittelten Gruppen und ihre Gruppenkosten, werden jeweils als Kante und entsprechendes Kantengewicht in W eingefügt. Mit dem *Dijkstra*-Algorithmus wird danach wieder der kürzeste Pfad von der ersten bis zur letzten Aktivität in W bestimmt und der entsprechende Ausführungsplan der Execution Engine e wird als p_e gespeichert. Die einzelnen, nicht optimalen Ausführungspläne der Execution Engines

Algorithmus 5.5 Optimaler Ausführungsplan

1: **procedure** Optimal Planning(W,N, $S_{comp}(a)$)
2: create empty Execution Plan p
3: **for all** E AS e **do**
4: calculate $r_{sync}(N, e)$
5: **end for**
6: **for all** Activities $a_i \in W$ AS $startOfGroup$ **do**
7: **for all** Activities $a_j \in W, j >= i$ AS $endOfGroup$ **do**
8: add $G_{i,j}$ starting at a_i and ending in a_j as edge from a_i to a_j to W;
9: **for all** E AS e **do**
10: **for all** Activities from $startOfGroup$ until $endOfGroup$ AS a_k **do**
11: **for all** Services of a_k AS s **do**
12: calculate $r_{execAct}(e, a_k, s, N)$
13: **end for**
14: **if** Services of a = {} **then**
15: calculate $r_{execActWithoutService}(e, a_k, N)$
16: **else**
17: calculate $r_{maxExecAct}(e, a_k, N)$
18: **end if**
19: **end for**
20: calculate $r_{group}(e, g, N)$;
21: calculate $r_{exec\&sync}(e, g, N)$;
22: calculate $f_{group}(e, g, N)$;
23: **end for**
24: calculate $f_{minGroup}(g, N)$;
25: add $f_{minGroup}(g, N)$ as an edge weight to $G_{i,j}$;
26: add e of $min_{e \in E} f_{group}(g, N)$ as additional attribute to $G_{i,j}$;
27: **end for**
28: **end for**
29: execute Dijkstra Algorithm on W for th shortest path from a_{first} to a_{last} over the edges $G_{i,j}$;
30: **for all** edges G in shortest path **do**
31: add G to p;
32: **end for**
33: **return** p;
34: **end procedure**

Algorithmus 5.6 Bestes Replikat

```
1: procedure BEST REPLICA(W,N, S_comp(a))
2:   for all E AS e do
3:     create empty Execution Plan p_e;
4:     calculate r_sync(N, e);
5:     for all Activities a_i ∈ W AS startOfGroup do
6:       for all Following Activities a_j ∈ W, j > i AS endOfGroup do
7:         add G_{i,j} starting at a_i and ending in a_j as edge from a_i to a_j to W;
8:         for all Activities from startOfGroup until endOfGroup AS a_k do
9:           for all Services of a_k AS s do
10:            calculate r_execAct(e, a_k, s, N)
11:          end for
12:          if Services of a = {} then
13:            calculate r_execActWithoutService(e, a_k, N)
14:          else
15:            calculate r_maxExecAct(e, a_k, N)
16:          end if
17:        end for
18:        calculate r_group(e, g, N);
19:        calculate r_exec&sync(e, g, N);
20:        calculate f_group(e, g, N);
21:        add f_group(e, g, N) as an edge weight to G_{i,j};
22:      end for
23:    end for
24:    execute Dijkstra Algorithm on W for the shortest path from a_first to a_last over the
   edges G_{i,j};
25:    for all edges G in shortest path do
26:      add G to p_e;
27:    end for
28:    remove all edges G_{i,j} from W;
29:  end for
30:  calculate p_minCost as the execution plan with minimal cost of all p_e, e ∈ E;
31:  return p_minCost;
32: end procedure
```

werden am Ende verglichen und das Minimum ihrer Kosten bestimmt. Der entsprechende Ausführungsplan mit den minimalen Kosten ist das Ergebnis der Ausführungsplanung.

5.2.4 Line Of Sight

Die Implementierung von *Line Of Sight* ist in Algorithmus 5.7 beschrieben. Die jeweilige *Line Of Sight* wird durch den $referencePoint$ und die übergebene $SightLength$ definiert. Den Aktivitäten A im Workflow W sind entsprechend ihrer Reihenfolge die Nummern 0 bis $|A| - 1$ zugewiesen. Anhand dieser Nummern lassen sich die Aktivitäten identifizieren. Der $referencePoint$ wird zunächst auf 0 gesetzt und zeigt somit auf die erste Aktivität im Workflow. Solange $referencePoint < |A|$ gilt, müssen weitere Gruppen generiert werden, um einen vollständigen Ausführungsplan zu erhalten. Ein Durchlauf der $while$-Schleife stellt hier einen Planungsschritt von *Line Of Sight* dar. Sind ab dem $referencePoint$ weniger Aktivitäten für das Einfügen in eine weitere Gruppe übrig, als für eine *Line Of Sight* vorgesehen, muss die $SightLength$ für die nächste *Line Of Sight* jeweils mit $SightLength = |A| - referencePoint$ angepasst werden.

Die Implementierung eines Planungsschritts beinhaltet als nächstes zwei for-Schleifen, wobei die erste das nächste den Workflow ausführende Replikat e_{next} bestimmt. In der zweiten for-Schleife wird dann für dieses Replikat und die aktuelle *Line Of Sight* von allen hierbei möglichen Gruppen die Kosten $f_{group}(e_{next}, g, N)$ bestimmt und die entsprechenden Kanten in W einfügt. Anschließend wird für die aktuelle *Line Of Sight* mithilfe des *Dijkstra*-Algorithmus der kürzeste Pfad von der ersten bis zur letzten Aktivität in W bestimmt. Die erste Kante dieses Pfads bestimmt die nächste auszuführende Gruppe. Diese Gruppe wird zum Ausführungsplan hinzugefügt.

Der $referencePoint$ wird dann auf die nächste Aktivität nach der bestimmten Gruppe gesetzt und die eingefügten Kanten $G_{i,j}$ werden aus W entfernt, um den nächsten Planungsschritt ausführen zu können. Zeigt der $referencePoint$ nun auf eine nicht existierende Aktivität a_z mit $z = |A|$ und somit $referencePoint = |A|$ wurden alle Aktivitäten in den Gruppen im Ausführungsplan eingefügt. Dann ist der Ausführungsplan vollständig und die Ausführungsplanung beendet. In unserer Implementierung wird erst an dieser Stelle mit der Workflowausführung begonnen.

5.2.5 Benchmark: Zufälliger Ausführungsplan

Der Algorithmus für den zufälligen Ausführungsplan bestimmt eine zufällige Anzahl von Gruppen für einen Workflow. Die Aktivitäten werden dann gleichmäßig auf alle Gruppen verteilt, wobei für einen Rest r der Division $|A|/$**Anzahl der Gruppen** die ersten r Gruppen jeweils eine Aktivität mehr beinhalten, als die restlichen Gruppen.

Algorithmus 5.7 Line Of Sight

```
1: procedure LineOfSightPlanning(W,N, S_comp(a), SightLength)
2:     create empty Execution Plan p
3:     for all E AS e do calculate r_sync(N, e);
4:     end for
5:     referencePoint = 0;
6:     while referencePoint < |A| do
7:         if |A| − referencePoint < SightLength then
8:             SightLength = |A| − referencePoint;
9:         end if
10:        for all E AS e do                          // find replica to execute next group
11:            for all Activities a_i ∈ LineOfSight AS startOfGroup do
12:                for all Services of a_k AS s do calculate r_execAct(e,a_k, s, N)
13:                end for
14:                if Services of a = {} then r_temp := r_execActWithoutService(e, a_k, N);
15:                elser_temp := r_maxExecAct(e, a_k, N);
16:                end if
17:                r_average[e] := r_average[e] + r_temp;
18:                count = count +1;
19:            end for
20:            r_average[e] = r_average / count;
21:        end for
22:        e_next = e of max(r_average);         // e_next is the replica that executes the next group
23:        for all Activities a_i ∈ LineOfSight AS startOfGroup do // get all group costs
24:            for all Following Activities a_j ∈ LineOfSight, j > i AS endOfGroup do
25:                add G_{i,j} starting at a_i and ending in a_j as edge from a_i to a_j to W;
26:                for all Activities from startOfGroup until endOfGroup AS a_k do
27:                    for all Services of a_k AS s do calculate r_execAct(e_next, a_k, s, N)
28:                    end for
29:                    if Services of a = {} then calculate r_execActWithoutService(e_next, a_k, N)
30:                    elsecalculate r_maxExecAct(e_next, a_k, N)
31:                    end if
32:                end for
33:                calc r_group(e_next, g, N) then r_exec&sync(e_next, g, N) then f_group(e_next, g, N);
34:                add f_group(e_next, g, N) as an edge weight to G_{i,j};
35:            end for
36:        end for
37:        Find shortest path from a_referencePoint to a_{referencePoint+SightLength−1} in W;
38:        add first group g_first(L, e) from the shortest path p_LoS(L, e) to p;
39:        referencePoint = last Activity in g_first(L, e) + 1;
40:        remove all G_{i,j} from W;
41:    end while
42:    return p;
43: end procedure
```

Algorithmus 5.8 Zufälliger Ausführungsplan

```
1: procedure RandomPlanning(W,N)
2:     numberOfGroups := Random[0,|A|];
3:     groupSize := |A| / numberOfGroups;
4:     leftOver := |A| mod numberOfGroups;
5:     lastWorkflowNode = 0; //first Activity in Workflow
6:     for i=0; i<numberOfGroups; i++ do
7:         actualGroupSize := groupSize;
8:         if leftover > 0 then
9:             actualGroupSize := actualGroupSize + 1;
10:            leftover := leftover -1;
11:        end if
12:        Create new empty group g
13:        for j=0; j<actualGroupSite; j++ do
14:            add a_{j+lastWorkFlowNode} to g
15:        end for
16:        lastWorkflowNode := lastWorkflowNode + actualGroupSize;
17:        master := Random element in E;
18:        add Group g to Execution Plan;
19:    end for
20:    return Execution Plan;
21: end procedure
```

t_{user}	1
c_{user}	1000000
t_{sync}	30
t_{elec}	800

Tabelle 5.1: Parameter für die Berechnung der Gruppenkosten

5.3 Metriken und Konfiguration

Bei der Anwendung der Strategien messen wir pro Testdurchlauf jeweils die Zeit für das Erstellen des Ausführungsplans, die Zeit für die Workflowausführung und die Anzahl der Gruppen im Ausführungsplan. Wir bestimmen den Overhead der Ausführungszeit indem wir die minimal mögliche Ausführungszeit von der tatsächlichen Zeit der Workflowausführung abziehen. Die minimal mögliche Ausführungszeit ergibt sich aus der Summe der Ausführungszeiten der einzelnen Aktivitäten im Workflow.

Die Parameter für die Kostenberechnung $f_{group}(e, g, N)$ sind in Tabelle 5.1 aufgelistet.

Der für die Tests verwendete Workflow besitzt 100 Aktivitäten, welche sequentiell ausgeführt werden müssen. Der Workflow bleibt für alle Testduchläufe unverändert, das heißt die Aktivitäten besitzen immer die selbe Ausführungsdauer und Kompensationskosten und die Menge der jeweils kompatiblen Services ändert sich nicht. Das Netzwerk für welches der Ausführungsplan berechnet wird und auf welchem der Workflow anschließend ausgeführt wird ändert sich dagegen für jeden Testdurchlauf entsprechend der Erläuterungen in 5.1.1.

5.4 Testumgebung

Alle Tests wurden auf einem Notebook mit Intel i7-4710HQ Prozessor 4 Kernen mit 2.6Ghz pro Kern, 8 virtuellen CPUs und 16GB RAM ausgeführt. Als Betriebssystem wurde Ubuntu 16.04 verwendet. Die generierten .jar-Dateien wurden mit dem Oracle Java(TM) SE Runtime Environment (build 1.8.0_121-b13) ausgeführt.

5.5 Ergebnisse

Für jede Strategie wurden pro Replikationsgrad 250 Ausführungsplanungen und die entsprechenden Workflowausführungen durchgeführt. Nach jedem Testdurchlauf wurden die Ergebnisse für jede Metrik gespeichert. Nach beenden aller Tests wurde für jede Metrik das arithmetische Mittel der Messungsergebnisse gebildet.

5.5.1 Planungszeit

Wir betrachten im folgenden die gemessenen Zeiten für die Bestimmung eines Ausführungsplans in den unterschiedlichen Strategien. Da jede der Strategien für einen nicht zufälligen Ausführungsplan die Synchronisationszuverlässigkeit der Execution Engines berechnet und diese Berechnung eine hohe Komplexität aufweist, betrachten wir diese zunächst gesondert, bevor wir auf die einzelnen Strategien eingehen.

Messung der Berechnungszeit für die Synchronisationszuverlässigkeit

Um den Effekt der Berechnungszeit der Synchronisationszuverlässigkeit auf die gesamte Planungszeit bestimmen zu können, haben wir diese zusätzlich zu den Strategien gesondert gemessen. In der Abbildung 5.1 zeigen wir die Durchschnittswerte aus jeweils 100 Messungen pro Replikat und pro Replikationsgrad. Die Ergebnisse stellen jeweils die Berechnungszeit für die Synchronisationszuverlässigkeit einer einzelnen Execution Engine dar.

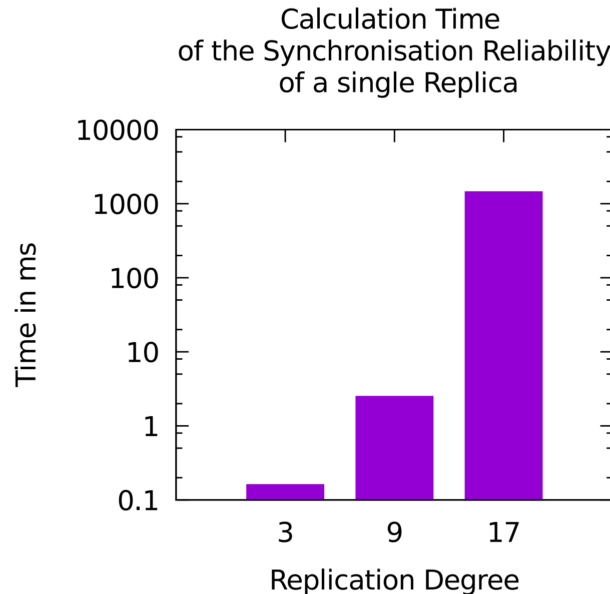


Abbildung 5.1: Berechnungszeit der Synchronisationszuverlässigkeit einer Execution Engine

Die Anzahl der bei der Berechnung jeweils zu betrachtenden Netzwerkzustände beträgt $2^{Replikationsgrad}$. Dies drückt sich, wie in der Abbildung zu erkennen ist, entsprechend in der Rechenzeit deutlich aus. Für den Replikationsgrad 3 beträgt die Rechenzeit lediglich 0,16ms, für den Replikationsgrad 9 sind es bereits 2,48ms und für den Replikationsgrad 17 gar 1441,2ms. Wie in den folgenden Messungsergebnissen deutlich wird, stellt die Rechenzeit der Synchronisationszuverlässigkeit für den Replikationsgrad 17 bei allen nicht zufälligen Strategien entsprechend einen signifikanten Anteil der gesamten Planungszeit dar.

Messungen der gesamten Planungszeit

In der Abbildung 5.2 sehen wir jeweils die insgesamt benötigte Zeit für das Erstellen eines Ausführungsplans für jede Strategie und jeden Replikationsgrad. Bei der Strategie *Bestes Replikat* gibt die Zeitangabe an, wie lange benötigt wurde, um einen Ausführungsplan p_e, wie in 5.6 dargestellt, zu berechnen. Dieser wird jeweils für jede Execution Engine berechnet. Für die beiden anderen nicht zufälligen Strategien ist jeweils die Zeit für die Berechnung des gesamten Ausführungsplans dargestellt. Um die großen Unterschiede der Planungszeit besser sichtbar zu machen wurde eine logarithmische Skala gewählt.

Bei *Line Of Sight* besteht die Möglichkeit mit der Ausführung bereits zu beginnen, sobald nur die erste Gruppe des Ausführungsplans bestimmt wurde. Gehen wir von einer uniformen Verteilung der Berechnungsdauer für eine Gruppe aus, können wir die durchschnittliche nötige Wartezeit bis die erste Gruppe bekannt ist bestimmen. Dafür teilen wir die Zeit für die gesamte

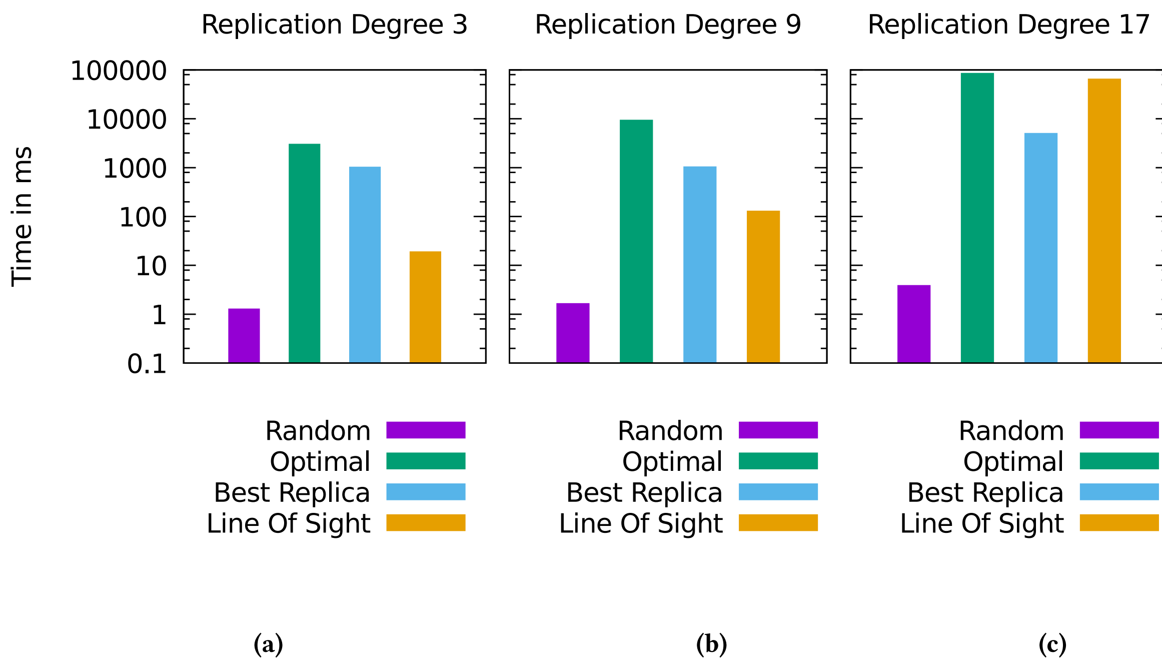


Abbildung 5.2: Zeit der Ausführungsplanung

Ausführungsplanung durch die durchschnittliche Anzahl der Gruppen. Anschließend muss hier noch die Zeit für die Berechnung der Synchronisationszuverlässigkeit für alle Replikate bis auf eines abgezogen werden. Dann kennen wir einerseits die tatsächliche durchschnittliche Verzögerung, bis mit der Ausführung des Workflows begonnen wird. Diese Möglichkeit ist in den Messungsergebnissen nicht dargestellt.

Replikationsgrad 3 Für den Replikationsgrad 3 betrug die durchschnittliche Planungsdauer für den zufälligen Ausführungsplan 1ms, für den optimalen Ausführungsplan 2986ms, für *Bestes Replikat* 1010ms und für *Line Of Sight* 19ms. Die verkürzte Planungszeit für *Bestes Replikat* im Vergleich zum optimalen Ausführungsplan entspricht sehr genau den Erwartungen. Für *Bestes Replikat* muss hier nur ein Drittel der Gruppenkosten ermittelt werden, was die Planungsdauer entsprechend verkürzt.

Line Of Sight zeigt im Vergleich zu den beiden anderen Strategien eine immens verkürzte Planungsdauer, was sich durch die jeweils wesentlich kürzere betrachtete Workflowlänge erklären lässt. Dadurch werden nur für eine wesentlich geringere Anzahl von Gruppen die Ausführungskosten berechnet, was zudem durch weniger entsprechende Kanten im Workflowgraphen das Finden des kürzesten Pfades beschleunigt.

Replikationsgrad 9 Für den Replikationsgrad 9 betrug die durchschnittliche Planungsdauer für den zufälligen Ausführungsplan 2ms, für den optimalen Ausführungsplan 9259ms, für *Bestes Replikat* 1027ms und für *Line Of Sight* 128ms. Im Vergleich zu Replikationsgrad 3 hat sich die Planungsdauer für den optimalen Ausführungsplan in etwa verdreifacht, was einem linearen Anstieg in Abhängigkeit des Replikationsgrads entspricht.

Die Planungszeit von *Bestes Replikat* hat sich kaum verändert. Es war zu erwarten, dass diese von der Anzahl der Replikate relativ unabhängig ist. Die geringe Erhöhung des Werts, lässt sich teilweise durch die für diesen Replikationsgrad länger dauernde Berechnung der Synchronisationszuverlässigkeit erklären. Diese beträgt wie zuvor erläutert hierbei 2,48ms.

Bei *Line Of Sight* hat sich die Planungsdauer mehr als versechsfacht. Durch die zentralisierte Implementierung muss hier die durchschnittliche Ausführungszuverlässigkeit in der *Line Of Sight* nacheinander für jede Execution Engine berechnet werden, statt dies nur für eine Execution Engine zu tun. Genauso verhält es sich mit der Synchronisationszuverlässigkeit, deren Berechnung bei Replikationsgrad 9 für alle Execution Engines bereits 22ms beträgt. Durch diese beiden Umstände lässt sich die Erhöhung der Planungszeit für den erhöhten Replikationsgrad erklären. Bei einer verteilten Ausführung der Planung sollte die Planungsdauer dagegen bis auf die Berechnungsdauer der Synchronisationszuverlässigkeit und einen Kommunikations-Overhead ebenfalls unabhängig vom Replikationsgrad sein.

Replikationsgrad 17 Für den Replikationsgrad 9 betrug die durchschnittliche Planungsdauer für den zufälligen Ausführungsplan 4ms, für den optimalen Ausführungsplan 84079ms, für *Bestes Replikat* 4943ms und für *Line Of Sight* 64074ms. Die Dauer zur Berechnung aller Synchronisationszuverlässigkeiten betrug im isolierten Test ungefähr 24500ms. Für eine Execution Engine waren es entsprechend 1441ms. Es ist nicht auszuschließen, dass die Berechnung der Synchronisationszuverlässigkeiten als Teil der gesamten Ausführungsplanung noch länger dauerte, da hier die Threads aller anderen Execution Engines zusätzlich Rechenleistung in Anspruch nehmen, wohingegen für die isolierte Messung im Vergleich keine weiteren Java-Threads ausgeführt wurden.

Die Planungszeit des optimalen Ausführungsplans wird sowohl durch die höhere Anzahl an Replikaten und damit höherer Anzahl an Vergleichen bei der Bestimmung der Gruppenkosten als auch durch die immens längere Berechnungsdauer der Synchronisationszuverlässigkeit deutlich erhöht. Bei der Strategie *Bestes Replikat* kann allein die Berechnungsdauer der Synchronisationszuverlässigkeit für den Anstieg der Planungsdauer verantwortlich gemacht werden, da alle anderen durchgeführten Berechnungen unabhängig vom Replikationsgrad sind (siehe Algorithmus 5.6). Bei *Line Of Sight* führen die selben Ursachen wie bereits für Replikationsgrad 9 beschrieben zum weiteren Anstieg der Dauer der Ausführungsplanung.

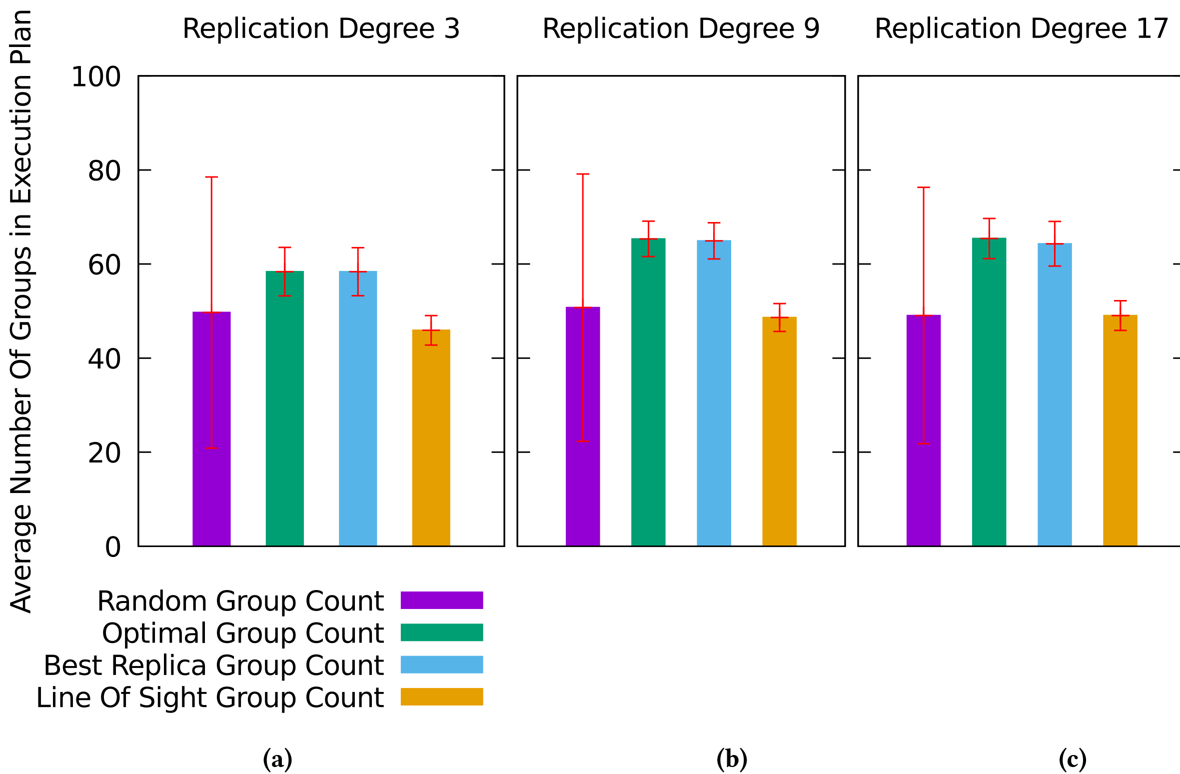


Abbildung 5.3: Durchschnittliche Anzahl der Gruppen im Ausführungsplan

5.5.2 Anzahl der Gruppen im Ausführungsplan

Für jeden erzeugten Ausführungsplan wurden die Anzahl der enthaltenen Gruppen erfasst. In Abbildung 5.3 sind die Durchschnittszahlen der Gruppenanzahl für jede Strategie und jeden Replikationsgrad dargestellt.

Replikationsgrad 3 Die Anzahl der Gruppen des zufälligen Ausführungsplan betragen hier durchschnittlich 50 wobei die Standardabweichung davon mit 29 sehr hoch ist, was für den zufälligen Ausführungsplan zu erwarten war. Die durchschnittliche Anzahl der Gruppen im Optimalen Ausführungsplan beträgt hier 58 mit einer geringeren Standardabweichung von 5,1. Für *Bestes Replikat* sind es 58 bei einer Standardabweichung von 5,1, also leicht geringer als beim optimalen Ausführungsplan. Da hier nur eine Execution Engine den Workflow ausführt, fällt der Grund eine Aktivität, welche sonst in der selben Gruppe als die vorherige Aktivität ausgeführt werden würde, in einer anderen Gruppe auszuführen, um eine Execution Engine mit einer besseren Zuverlässigkeit für einen Serviceaufruf zu erhalten, weg. Die durchschnittliche Anzahl der Gruppen bei *Line Of Sight* ist mit 45,9 bei einer Standardabweichung von 3,1 noch einmal deutlich niedriger als bei den beiden zuvor betrachteten Strategien.

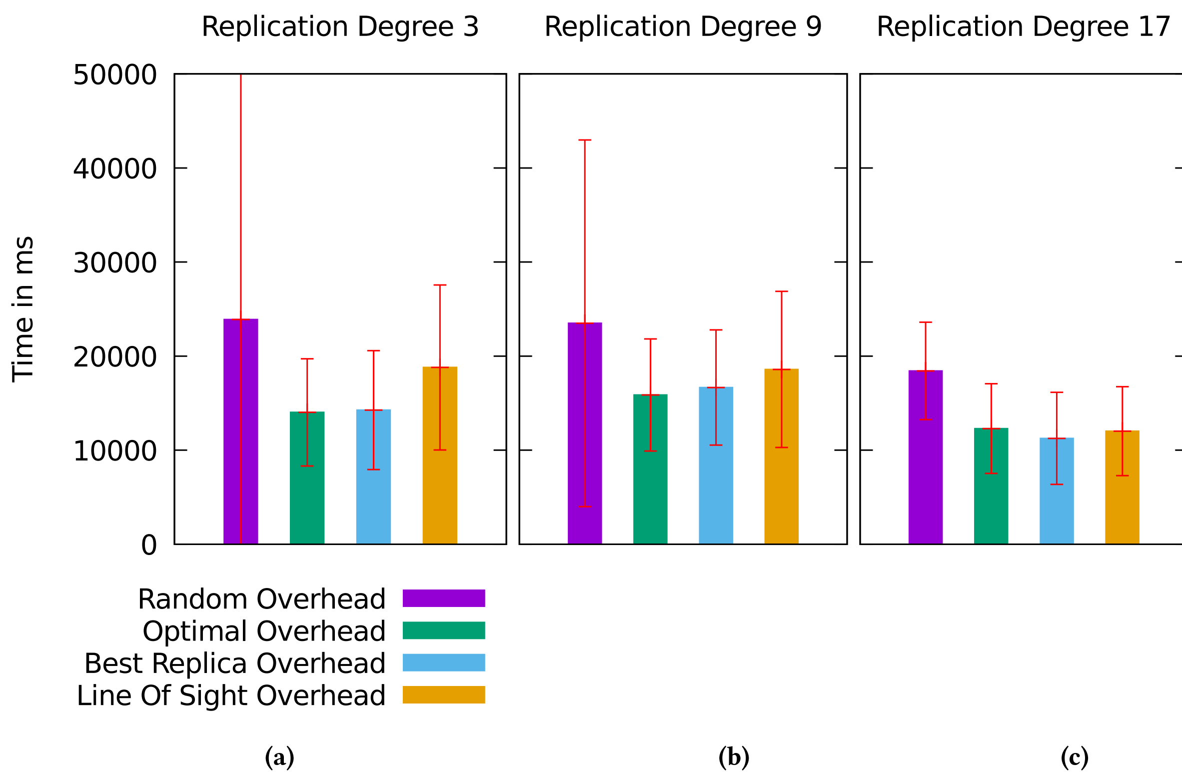
Die beschränkte Sicht bei *Line Of Sight* und damit auch beschränkte Gruppengröße lässt zunächst vermuten, dass die durchschnittliche Gruppengröße hier geringer wäre, als bei den beiden anderen nicht zufälligen Strategien und somit im Schnitt mehr Gruppen im Ausführungsplan enthalten sein müssten. Andererseits wird hier in jedem Planungsschritt für das Finden der nächsten Gruppe, wie auch in *Bestes Replikat* nur eine Execution Engine betrachtet, was es verhindert zu Erkennen, wenn die Gruppe abgekürzt werden sollte, um auf eine andere Execution Engine zu wechseln.

Replikationsgrad 9 Hier beträgt die durchschnittliche Anzahl der Gruppen für den zufälligen Plan 51 mit einer Standardabweichung von 28,4, was in etwa dem Wert von Replikationsgrad 3 entspricht und zu erwarten war. Für den optimalen Ausführungsplan sind es 65,3 Gruppen, bei einer Standardabweichung von 3,8. Die höhere Anzahl von Gruppen lässt sich hier damit erklären, dass bei einer höheren Anzahl von Replikaten die Wahrscheinlichkeit höher ist, dass die für eine Ausführung einer Aktivität beste Execution Engine eine andere ist, als die für die vorherige Aktivität, sodass häufiger eine neue Gruppe begonnen wird. Für *Bestes Replikat* ist die durchschnittliche Gruppengröße mit 64,9 und einer Standardabweichung von 3,8 ebenfalls deutlich größer als im Replikationsgrad 3.

Die durchschnittliche Gruppengröße bei *Line Of Sight* bleibt relativ stabil bei 48,6 mit einer Standardabweichung von 3.

Replikationsgrad 17 Die durchschnittliche Gruppenzahl des zufälligen Ausführungsplans bleibt nach wie vor nahezu unverändert bei 49 mit einer Standardabweichung von 27. Für den optimalen Ausführungsplan ist die Gruppenzahl mit 65 und 4,3 nicht weiter gestiegen. Ebenso verhält es sich bei Best Replica mit 64,3 Gruppen bei einer Standardabweichung von 4,7. Die durchschnittliche Gruppengröße von *Line Of Sight* ist auch bei Replikationsgrad 17 praktisch unverändert bei 49,0 mit einer Standardabweichung von 3,2.

Vergleich Im Allgemeinen lohnt es sich dann eher kleinere Gruppen auszuführen, wenn die Synchronisationszeit im Vergleich zur Ausführungszeit einer Aktivität relativ niedrig ist und die Synchronisationszuverlässigkeit auch relativ hoch ist. Dann ist die Synchronisationszeit als Teil der gesamten Ausführungszeit auch bei kleinen Gruppen nicht signifikant. Hier lässt sich auch ein häufiger Wechsel der Execution Engine, um jeweils die beste Zuverlässigkeit für die Ausführung der Aktivitäten und der Serviceaufrufe zu erhalten, verschmerzen. Ferner vermuten wir, dass die zwischen 0,9 und 1 relativ gering gewählten Zuverlässigkeiten für unser Netzwerk zu einer häufigen Synchronisation und damit kurzen Gruppen beigetragen haben.



(a) (b) (c)

Abbildung 5.4: Overhead

5.5.3 Overhead bei der Ausführungszeit

Replikationsgrad 3 Der Overhead für den zufälligen Ausführungsplan liegt bei 23890ms und zeigt eine extrem hohe Standardabweichung von 32881ms. Beim optimalen Ausführungsplan ist der Overhead mit 14023ms wie zu erwarten deutlich niedriger und die ebenfalls deutlich geringere Standardabweichung von 5696ms deutet auf eine wesentlich zuverlässigere Qualität des Ausführungsplans hin. Der Overhead für Best Replica liegt mit 14257 und einer Standardabweichung von 6319ms nur knapp höher als beim optimalen Ausführungsplans und zeigt auch eine deutliche verbesserte Zuverlässigkeit als der zufällige Plan.

Davon deutlich abheben tut sich der Overhead bei *Line Of Sight* mit 18793ms. Auch die Standardabweichung ist hier mit 8766ms deutlich höher. Die deutlich eingeschränkte Sicht auf nur 10% des gesamten Workflows verschlechtert hier entsprechend den Ausführungsplan und verlängert damit die Ausführungszeit. Dennoch ist der Overhead deutlich geringer als bei einem zufälligen Ausführungsplan und auch die Standardabweichung bescheinigt eine deutlich zuverlässigere Ausführungsplanung.

Replikationsgrad 9 Hier liegt der Overhead für den zufälligen Plan ohne Große Veränderung bei 23492ms, die Standardabweichung allerdings auf 19484ms gesunken. Der Overhead für den optimalen Plan liegt bei 15867ms mit einer Standardabweichung von 5953ms. Diese liegt also nur geringfügig höher als für 3 Replikate. Bei *Bestes Replikat* erhalten wir einen durchschnittlichen Overhead von 16663ms bei einer Standardabweichung von 6122ms. Für die Strategie *Line Of Sight* beträgt der Overhead hier 18582ms bei einer Standardabweichung von 8293ms.

Replikationsgrad 17 Der Overhead für den zufälligen Ausführungsplan liegt hier bei 18423ms mit einer Standardabweichung von 5174ms. Für den optimalen Ausführungsplan liegt dieser Wert bei 12297ms mit einer Standardabweichung von 4768ms. Bei *Bestes Replikat* sind es 11254ms bei einer Standardabweichung von 4892ms. Hier ist der Overhead zum ersten Mal geringer als bei *Optimaler Ausführungsplan*. Für *Line Of Sight* beträgt der Overhead hier 12028ms bei einer Standardabweichung von 4728ms.

Der geringeren Overhead-Wert für *Bestes Replikat* im Vergleich zu *Optimaler Ausführungsplan* in Replikationsgrad 17 wurde vermutlich dadurch verursacht, dass der Workflow auf einem einzigen Rechner mit nur 4 Kernen ausgeführt wurde. Wir erklären den vermuteten Hintergrund im Folgenden. Für jede Execution Engine werden mehrere Threads gestartet. Daher können für höhere Replikationsgrade deutlich mehr Threads anfallen, als von der Testumgebung gleichzeitig ausgeführt werden können. Bei einem Wechsel der ausführenden Execution Engine, muss der entsprechende Thread, möglicherweise, falls dieser bis dahin inaktiv war, zuerst auf einem der Prozessorkerne aktiviert werden, was zusätzliche Zeit kostet, die bei gleichbleibender Execution Engine nicht anfällt. Bei einer Testumgebung mit einer dem Replikationsgrad entsprechenden Anzahl an Rechnern existiert dieses Problem natürlich nicht, sodass dort die Messergebnisse von den hier gesammelten möglicherweise abweichen können.

Daneben wurde die Zeit, die im allgemeinen benötigt wird, um die ausführende Execution Engine nach einer Gruppe zu wechseln, in der Kostenberechnung des Ausführungsplans nicht beachtet, da diese zunächst als vernachlässigbar abgeschätzt wurde.

Vergleich Bei *Line Of Sight* ist es im Hinblick auf die durchschnittliche Gruppenlänge im Vergleich zur Länge der *Line Of Sight* wahrscheinlich, dass aufeinanderfolgende Gruppen häufig auf der selben Execution Engine ausgeführt werden und daher die Anzahl der Wechsel der Execution Engine bei der Workflowausführung wesentlich geringer ist als bei *Optimaler Ausführungsplan*. Die ausführende Execution Engine wird bei *Line Of Sight* durch den Durchschnitt der Zuverlässigkeiten für zehn aufeinanderfolgende Aktivitäten bestimmt. Bei einer Gruppenlänge von 2, werden für die Bestimmung der nächsten Execution Engine acht der zehn betrachteten Aktivitäten wiederverwendet, sodass es wahrscheinlich ist, dass die gleiche Execution Engine wieder die höchste Zuverlässigkeit erzielt. Somit tritt für *Line Of Sight* in geringerem Maße der selbe Effekt, wie bei *Bestes Replikat* auf.

Es fällt zudem auf, dass die Werte der nicht zufälligen Strategien zunächst für den Replikationsgrad 9 erhöht sind, dann aber für den Replikationsgrad 17 allesamt deutlich niedriger ausfallen. Zunächst ist festzustellen, dass die jeweils beste verfügbare Zuverlässigkeit für die Ausführung von Aktivitäten und die Synchronisationszuverlässigkeit bei einer größeren Menge von Replikaten durchschnittlich höher ist, was insgesamt geringere Kosten der Gruppen und somit der Ausführungspläne ermöglicht und damit den Overhead senken kann. Dies wäre die Erklärung für die allesamt deutlich niedrigeren Overheadwerte in Replikationsgrad 17. Für Replikationsgrad 9 wirkt scheinbar allerdings noch ein anderer Effekt, der für eine allgemeine Erhöhung der Overheadwerte sorgt.

5.5.4 Zusammenfassung

Der gering erhöhte Overhead bei der Ausführungszeit von *Bestes Replikat* in den Replikationsgraden 3 und 9 im Vergleich zu *Optimaler Ausführungsplan* wird von der deutlich verringerten Planungszeit durch eine verteilte Ausführungsplanung übertroffen, sodass der Einsatz von *Bestes Replikat* in diesen Fällen gegenüber *Optimaler Ausführungsplan* vorzuziehen ist.

Bis Replikationsgrad 9 lohnt sich die deutlich geringere Planungszeit von *Line Of Sight* auch bei dem etwas erhöhten Overhead der Ausführungszeit. Danach wird die Zeit der Ausführungsplanung hauptsächlich durch die Berechnung der Synchronisationszuverlässigkeit bestimmt, sodass die Planungszeit für *Line Of Sight* hier nicht einmal mehr wesentlich geringer ist, als bei *Optimaler Ausführungsplan*.

Die Berechnung der Synchronisationszuverlässigkeit verursacht bei höheren Replikationsgraden einen erheblichen Teil der Zeit der Ausführungsplanung. Hier lohnt es sich nach Heuristiken zu suchen, um dessen Berechnung zu vermeiden oder bei mehreren im selben Netzwerk auszuführenden Workflows Werte wiederzuverwenden statt neu zu berechnen.

5.6 Fazit

Wir befürchten, dass die Ausführung der Tests auf einem einzelnen Rechner die Ergebnisse leider in derart verfälscht, dass ein Wechsel der Execution Engine in einigen Fällen eine signifikante Zeit in Anspruch nimmt, wie sie sonst in einem realen Netzwerk nicht auftreten würde. Dennoch haben wir gezeigt, dass sich die Planungsdauer im Vergleich zur Berechnung eines optimalen Ausführungsplan auf einer einzelnen Execution Engine erheblich verkürzen lässt. Einerseits bringt eine verteilte Ausführungsplane eine deutliche Zeiteinsparung, als die Zerlegung des Workflows in mehrere Abschnitte. Dabei wurde festgestellt, dass sich die Verlängerung der Workflowausführung im Verhältnis oft als geringer erweist, als die Zeiteinsparung bei der Planung.

6 Verwandte Arbeiten

In [SBTR14] werden deklarative Workflows statt imperativen Workflows, welche in dieser Arbeit betrachtet werden, behandelt. Der Unterschied ist, dass es bei deklarativen Workflows für die Ausführung der Aktivitäten keine fest vorgeschriebene Reihenfolge gibt. Stattdessen ist eine Menge von Regeln festgelegt, welche die Ausführungsreihenfolge in gewissem Maße einschränkt. Um mögliche Gruppen und resultierende erlaubte Ausführungsreihenfolgen für diese zu finden, müssen diese jeweils auf Einhaltung der Regeln geprüft werden.

Dadurch ist die Menge an möglichen Gruppen erhöht und auch eine parallele Ausführung mehrerer Gruppen auf unterschiedlichen Execution Engines wird möglich.

In der zitierten Arbeit werden unterschiedliche Ausführungspläne gleichzeitig auf mehreren Execution Engines ausgeführt.

Wobei bei der Erstellung dieser Ausführungspläne die jeweilige Zuverlässigkeit der Serviceaufrufe nicht betrachtet wird.

7 Zusammenfassung und Ausblick

In dieser Arbeit beschäftigen wir uns mit mehreren Strategien für eine effiziente Planung von Workflowausführungen. Wir haben unser Systemmodell und die darin zu lösenden Probleme vorgestellt. Anschließend haben Lösungsansätze in Form von Strategien zum Erstellen von Ausführungsplänen besprochen.

Wir haben dabei Trade-Offs zwischen Planungszeit und Qualität des resultierenden Ausführungsplans in Betracht gezogen und festgestellt, dass sich die Planungszeit im Vergleich zu einer zentralisierten optimalen Ausführungsplanung signifikant verkürzen lässt ohne dabei die durchschnittliche Zeit der Workflowausführung zu sehr zu verlängern.

Durch Verteilung der Ausführungsplanung und Betrachtung des Workflows in mehreren Teilen lies sich die Planungszeit deutlich reduzieren. Es hat sich daneben gezeigt, dass besonders ein erhöhter Replikationsgrad die Planungszeit enorm verlängern kann und die Berechnung der Synchronisationszuverlässigkeiten dabei einen großen Teil dieser Zeit ausmachen. Alle unserer Strategien konnten andererseits eine Erhöhung des Replikationsgrad nutzen, um die durchschnittliche Ausführungszeit zu verringern, indem jeweils zuverlässigeren Execution Engines Gruppen zur Ausführung zugeteilt wurden.

Ausblick

Für zukünftige Arbeiten erhoffen wir uns die bereits lokal getesteten Strategien in einem realen Netzwerk mit realen Zuverlässigkeiten zu testen, um realitätsnähere Ergebnisse zu erhalten und nicht mit den Limitationen eines einzelnen Rechners kämpfen zu müssen. Hierfür ist unter anderem auch die verteilte Implementierung der Ausführungsplanung von *Bestes Replikat* und *Line Of Sight* nötig. Bei einer verteilten Ausführungsplanung wäre dann der Kommunikations-Overhead während der Planung interessant zu sehen.

Die Strategie *Line Of Sight* mit unterschiedlichen Längen der Line of Sight zu testen könnte zusätzlich interessante Erkenntnisse ermöglichen.

Eine verteilte Berechnung des optimalen Ausführungsplans, wie in der Strategie *Beste Gruppen* vorgestellt, wurde in dieser Arbeit nicht getestet. Hier wäre es interessant zu sehen, ob die verkürzte Planungszeit in den Bereich von *Bestes Replikat* gelangt und in welchen Netzwerken eine Verringerung der zu verteilenden Gruppen sinnvoll ist und in welchen nicht.

Außerdem wäre es interessant Möglichkeiten für eine effiziente verteilte Ausführung eines *Shortest Path* Algorithmus zu betrachten, um zum Beispiel ein Verteilen der Gruppen wie in *Beste Gruppen* vorgeschlagen zu vermeiden.

Ferner ist es eine Betrachtung Wert, ob sich die Berechnung der Synchronisationszuverlässigkeit weiter beschleunigen oder gar vermeiden lässt indem diese durch geeignete Heuristiken abgeschätzt wird und es dennoch möglich ist gute Ausführungspläne zu ermitteln.

Literaturverzeichnis

[Cha82] E. J. H. Chang. „Echo Algorithms: Depth Parallel Operations on General Graphs". In: *IEEE Transactions on Software Engineering* SE-8.4 (Juli 1982), S. 391–401. ISSN: 0098-5589. DOI: 10.1109/TSE.1982.235573 (zitiert auf S. 27).

[KLL09] R. K. Ko, S. S. Lee, E. W. Lee. „Business process management (BPM) standards: a survey". In: *Business Process Management Journal* 15.5 (2009), S. 744–791. DOI: 10.1108/14637150910987937. eprint: http://dx.doi.org/10.1108/14637150910987937. URL: http://dx.doi.org/10.1108/14637150910987937 (zitiert auf S. 11).

[MWV00] N. Malpani, J. L. Welch, N. Vaidya. „Leader Election Algorithms for Mobile Ad Hoc Networks". In: *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications.* DIALM '00. Boston, Massachusetts, USA: ACM, 2000, S. 96–103. ISBN: 1-58113-301-4. DOI: 10.1145/345848.345871. URL: http://doi.acm.org/10.1145/345848.345871 (zitiert auf S. 28).

[SBTR14] D. R. Schäfer, T. Bach, M. A. Tariq, K. Rothermel. „Increasing Availability of Workflows Executing in a Pervasive Environment". In: *2014 IEEE International Conference on Services Computing.* Juni 2014, S. 717–724. DOI: 10.1109/SCC.2014.98 (zitiert auf S. 11, 61).

[Sch] D. R. Schäfer. „Execution Planning: Improving the Reliability of Replicated Workflow Executions". Internal Document (zitiert auf S. 13).

[SSB+14] D. R. Schäfer, S. G. Sáez, T. Bach, V. Andrikopoulos, M. A. Tariq. „Towards Ensuring High Availability in Collective Adaptive Systems". In: *International Conference on Business Process Management.* Springer. 2014, S. 165–171 (zitiert auf S. 11).

[STR16] D. R. Schäfer, M. A. Tariq, K. Rothermel. *Highly Available Process Executions.* Englisch. Technischer Bericht Informatik 2016/02. Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Verteilte Systeme: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, März 2016, S. 11. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2016-02&engl=0 (zitiert auf S. 11).

[SWT+16] D. R. Schäfer, A. Weiß, M. A. Tariq, V. Andrikopoulos, S. G. Säez, L. Krawczyk, K. Rothermel. „HAWKS: A System for Highly Available Executions of Workflows". In: *2016 IEEE International Conference on Services Computing (SCC).* Juni 2016, S. 130–137. DOI: 10.1109/SCC.2016.24 (zitiert auf S. 11, 12).

[Tel00] G. Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000 (zitiert auf S. 27).

Alle URLs wurden zuletzt am 30. 03. 2017 geprüft.

Erklärung

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

Ort, Datum, Unterschrift