Technology
Arts Sciences
TH Köln

Brute Force Angriff auf einem parallelem, verteilten System

Dokumentation Architektur verteilter Systeme

ausgearbeitet von

Sebastian Domke
Pascal Schönthier
Dennis Jaeger

TH Köln
Campus Gummersbach
Fakultät für Informatik

im Studiengang

Computer Science Master
Software Engineering

vorgelegt bei: Prof. Dr. Lutz Köhler
TH Köln

Gummersbach, 27. Januar 2016

Kurzbeschreibung

Diese Dokumentation wird im Rahmen des Moduls **Architektur verteilter Systeme** im Studiengang Computer Science Master, Fachrichtung Software Engineering, an der technischen Hochschule Köln am Campus Gummersbach erstellt.

Das Ziel ist die prototypische Implementierung eines Brute Force Algorithmus innerhalb einer verteilten Architektur.
Mit Hilfe des Brute Force-Angriffs soll ein vordefiniertes Passwort entschlüsselt werden. Um dies zu realisieren, werden mögliche Passwort-Hash-Kombinationen berechnet und mit einem Ziel-Hash verglichen. Sobald ein berechneter Hash mit dem Ziel-Hash übereinstimmt, ist der Angriff erfolgreich abgeschlossen.

Die Implementierung soll in der von Apple© entwickelten Programmiersprache Swift umgesetzt und den allgemeinen Prinzipien eines verteilten Systems gerecht werden.

Inhaltsverzeichnis

1 Motivation und Grundlagen

Das Projekt wird im Rahmen des Moduls „Architektur verteilter Systeme“ im Masterstudiengang Computer Science, Fachrichtung Software Engineering, durchgeführt. Das Ziel ist die Implementierung einer verteilten Architektur. Die notwendige Hardware wird von der Hochschule zur Verfügung gestellt, welche detailliert in Kapitel **??** beschrieben wird. Die Entwicklung der Software ist Kernbestandteil dieses Projekts.

1.1 Ziel des Projekts

Zu Beginn des Projektes wurde ein Problem gesucht, dass auf Basis einer einer verteilten Architektur gelöst oder berechnet werden kann. Das Projekt-Team legte sich fest, dass das Problem aus der Domäne der *IT-Sicherheit* stammen soll. Aufgrund des hohen Rechenaufwands entschied das Projektteam sich zu einem BruteForce-Angriff. Konkret bedeutet dies, dass durch Ausprobieren aller möglichen Kombinationen versucht wird ein Passwort zu entschlüsseln. Das zu entschlüsselnde Passwort wird zu Beginn eingegeben und in Form eines Hashes hinterlegt. Der Hash stellt die Zielbedingung für die geplante Anwendung dar. Nun soll die verteilte Architektur die möglichen Passworte bzw. deren Hashes berechnen. Sobald ein berechneter Hash mit dem Zielhash übereinstimmt, ist das vorgegebene Passwort entschlüsselt. Weitere Details dazu werden in Kapitel 2.2 erläutert.

Das geplante Projekt soll außerdem den allgemeinen Ansprüchen an ein verteiltes System genügen, die im folgenden beschrieben werden.
Eine mögliche Definition eines *verteilten Systems* lautet wie folgt:

> *„Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen.“* [Tanenbaum u. van Steen, 2003]

Aus diesem Zitat lässt sich unter anderem entnehmen, dass bei einem verteilten System mehrere Rechner eingesetzt werden, welche zwar unabhängig voneinander arbeiten können, aber nun als kohärentes System eingesetzt werden.
Nach [Tanenbaum u. van Steen, 2003] verfolgt ein verteiltes System zudem folgende Ziele:

- **Ein verteiltes System sollte Ressourcen leicht verfügbar machen.** Die einfache Verfügbarkeit von Ressourcen innerhalb ist nach Aussage des Autors das Hauptziel eines verteilten Systems. Als Ressource kann dabei alles angesehen werden, was sich innerhalb des verteilten Systems befindet. Dies kann beispielsweise eine Datei, ein Drucker, Speichergeräte oder ähnliches sein.

- **Es sollte die Tatsache vernünftig verbergen, dass Ressourcen über ein Netzwerk verteilt sind.** Das System soll sich bei der Benutzung so anfühlen, als würde es nur auf einem einzigen Rechner arbeiten. Die Tatsache, dass verschiedene Komponenten in einem Netz verteilt sind, soll für den Anwender nicht bemerkbar sein.

- **Es sollte offen sein.** Das bedeutet, dass ein verteiltes System „seine Dienste so anbietet, dass diese die Syntax und Semantik der Dienste beschreiben“. Die Benutzung der Dienste soll durch formalisierte Beschreibung für alle Komponenten einfach und effizient durchführbar sein. In der Regel geschieht die Benutzung mit Hilfe von Schnittstellen. Benutzt man für die Spezifizierung der Schnittstellen beispielsweise die Schnittstellendefinitionssprache *Interface Definition Language (IDL)*, ist eine standardisierte Benutzung der Dienste möglich-das System ist damit *offen*.

- **Es sollte skalierbar sein.** In der rezitierten Literatur werden drei Faktoren genannt, mit denen die Skalierbarkeit eines verteilten Systems angegeben werden kann. Dies ist zum einen die Größe des verteilten Systems. Können zum System einfach neue Benutzer und Geräte hinzugefügt werden, ist eine gute Größen-Skalierbarkeit gegeben. Als zweiter Faktor wird die geografische Größe genannt, also die mögliche Entfernung zwischen verschiedenen Ressourcen. Der dritte Faktor ist die administrative Skalierbarkeit. Die administrative Skalierbarkeit ist gegeben, wenn eine einfache Verwaltung von diversen Organisationen möglich ist.

1.2 Grundlagen

In diesem Abschnitt wird die soft- und hardwareseitigen Basis des Projekts vorgestellt.

1.2.1 Hardwarebasis

Das verteilte System wird auf einem Mac-Cluster implementiert, das von der Hochschule zur Verfügung gestellt wird. Es kann auf 10 Rechner zugegriffen werden, die mit *pip02* bis *pip11* gekennzeichnet sind. Auf allen Rechnern ist (Stand 18.12.2016) aktuellste Version des Betriebssystems El Capitain und der Entwicklungsumgebung Xcode

installiert. Details zu den Softwareversionen sind in Kapitel 1.2.2 zu finden.
In der folgenden Liste sind die Details zu den einzelnen Rechnern zu finden:

- **pip02: Mac Pro (Anfang 2008)**

  Seriennummer: CK8250EUXYL

  Prozessor: 2 x 2,8 GHz Quad-Core Intel Xeon

  RAM: 2 GB 800 MHz DDR2 FB-DIMM

  Grafikkarte: NVIDIA GeForce 8800 GT (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip03: Mac Pro (Anfang 2008)**

  Seriennummer: CK8250EUXYL

  Prozessor: 2 x 2,8 GHz Quad-Core Intel Xeon

  RAM: 2 GB 800 MHz DDR2 FB-DIMM

  Grafikkarte: NVIDIA GeForce 8800 GT (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip04: Mac Pro (Anfang 2008)**

  Seriennummer: CK8250EUXYL

  Prozessor: 2 x 2,8 GHz Quad-Core Intel Xeon

  RAM: 2 GB 800 MHz DDR2 FB-DIMM

  Grafikkarte: NVIDIA GeForce 8800 GT (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip05: Mac Pro (Anfang 2008)**

  Seriennummer: CK8250EUXYL

  Prozessor: 2 x 2,8 GHz Quad-Core Intel Xeon

  RAM: 2 GB 800 MHz DDR2 FB-DIMM

  Grafikkarte: NVIDIA GeForce 8800 GT (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip06: Mac Pro (Anfang 2009)**

  Seriennummer: CK92608B20H

  Prozessor: 2 x 2,26 GHz Quad-Core Intel Xeon

  RAM: 6 GB 1066 MHz DDR3 ECC

  Grafikkarte: NVIDIA GeForce GT 120 (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip07: Mac Pro (Anfang 2009)**

  Seriennummer: CK92608B20H

  Prozessor: 2 x 2,26 GHz Quad-Core Intel Xeon

  RAM: 6 GB 1066 MHz DDR3 ECC

  Grafikkarte: NVIDIA GeForce GT 120 (512MB)

  OS 10.11.2 El Capitain

Xcode 7.2, Swift 2.1.1

- pip08: Mac Pro (Anfang 2009)

  Seriennummer: CK92608B20H

  Prozessor: 2 x 2,26 GHz Quad-Core Intel Xeon

  RAM: 6 GB 1066 MHz DDR3 ECC

  Grafikkarte: NVIDIA GeForce GT 120 (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip09: Mac Pro (Anfang 2009)**

  Seriennummer: CK92608B20H

  Prozessor: 2 x 2,26 GHz Quad-Core Intel Xeon

  RAM: 6 GB 1066 MHz DDR3 ECC

  Grafikkarte: NVIDIA GeForce GT 120 (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip10: Mac Pro (Anfang 2009)**

  Seriennummer: CK92608B20H

  Prozessor: 2 x 2,26 GHz Quad-Core Intel Xeon

  RAM: 6 GB 1066 MHz DDR3 ECC

  Grafikkarte: NVIDIA GeForce GT 120 (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

- **pip11: Mac Pro (Anfang 2009)**

  Seriennummer: CK92608B20H

  Prozessor: 2 x 2,26 GHz Quad-Core Intel Xeon

  RAM: 6 GB 1066 MHz DDR3 ECC

  Grafikkarte: NVIDIA GeForce GT 120 (512MB)

  OS 10.11.2 El Capitain

  Xcode 7.2, Swift 2.1.1

Zur Netzverbindung wird ein Switch des Herstellers *Netgear* eingesetzt. Die Modellbezeichnung lautet *Netgear GS116*. Der Switch hat 16 Ports und unterstützt bis zu 1000 Megabit/s (Gigabit-LAN).

1.2.2 Softwarebasis

Da Rechner des Herstellers Apple eingesetzt werden, sind die Programmiersprachen *Objective C* oder *Swift* effizient einsetzbar, da Apple diese vorrangig unterstützt. Das eingesetzte Betriebssystem Mac OS X 10.11.2 (El Capitain) und die native Entwicklungsumgebung Xcode 7.2 weisen eine hohe Kompatibilität zu den genannten Programmiersprachen auf.
Da die Programmiersprache *Swift* seit Version 2.0 quelloffen angeboten wird[1] und zudem die aktuellere der beiden genannten Sprachen ist, möchte das Projektteam primär auf Swift zurückgreifen. Da aktuell der Einsatz von Objective C noch Bestandteil von Swift ist, werden beide genannten Programmiersprachen zum Einsatz kommen.

Zur Versionierung des Programmcodes und zum vereinfachtem dezentralen Entwickeln wird die Programmcode-Plattform `www.github.com` eingesetzt. Die auf dem Versionsverwaltungs-System *Git* basierende Plattform ermöglicht ein flexibles und kollaboratives Arbeiten am Projekt sowie der Dokumentation.

Damit im Projekt weitere Frameworks mit wenig Aufwand eingesetzt werden können, hat das Projektteam sich entschieden *Carthage*[2] einzusetzen. Carthage ist ein „einfacher, dezentraler Dependency-Manager“ und wird quelloffen zur Verfügung gestellt. Durch die Auflösung von Abhängigkeiten, beispielsweise von bestimmten Frameworks, wird das asynchrone und dezentrale Entwickeln weiter optimiert.

[1] `https://github.com/apple/swift`
[2] `https://github.com/Carthage/Carthage`

Als alternativer Dependency-Manager hätte sich das Werkzeug *CocoaPods*[3] angeboten. Einer der großen Unterschiede in der Arbeitsweise der beiden Werkzeuge liegt in der Verwaltung der Dependencies. Während CocoaPods auf eine zentrale Verwaltung setzt, werden die Abhängigkeiten bei Carthage dezentral verwaltet. Durch die dezentrale Verwaltung wird unser Primärziel, das effektive kollaborative Arbeiten, besser abgedeckt. Zudem wird Carthage nicht so tief in das Entwicklungsprojekt in der Entwicklungsumgebung Xcode verwurzelt, als es bei CocoaPods der Fall wäre. Dadurch entsteht eine weniger starke Abhängigkeit von dem Werkzeug. Aus den genannten Gründen entschied das Projektteam sich gegen die Verwendung von CocoaPods.

[3] `https://github.com/CocoaPods/CocoaPods`

2 Implementation des BruteForce-Algorithmus

In diesem Kapitel wird die Konzeption des Brute Force-Angriffs beschrieben. Auf Basis der beschriebenen Konzeption soll im Anschluss die Implementation erfolgen können.

2.1 Architektur

2.1.1 Nachrichtenstruktur

Da unsere Implementation auf einer nachrichtenbasierten Kommunikation basieren wird, ist der Entwurf eigener Nachrichtenstrukturen notwendig. Die Nachrichtenstrukturen übertragen alle projektrelevanten Informationen zwischen den Mastern und den Workern. Zur Strukturierung fiel die Wahl auf das Nachrichtenaustauschformat „JavaScript Object Notation“ oder kurz *JSON*, da dieses Format sehr leichtgewichtig und einfach anpassbar ist.
Nachfolgend werden die entwickelten Nachrichtenstrukturen und deren Inhalt detailliert beschrieben. Die Nachrichten werden in drei Kategorien unterteilt: Nachrichten zum Worker, Nachrichten zum Master und allgemeine Nachrichten.

Nachrichten zum Worker

Hier werden die Strukturen beschrieben, welche primär vom steuernden Rechner (Master) zu einem der Worker gesendet werden.

`SetupAndConfig`

```
1 {
2   "status" : "setupConfig",
3   "value" : {
4     "algorithm" : "#HASH_ID",
5     "target" : "#TARGET_HASH",
6     "worker_id" : "#WORKER_ID"
7   }
```

```
}
```

Ein im Cluster neu hinzugefügter Worker erhält seine Konfigurationsparameter, damit dieser mit dem Berechnen beginnen kann. Der Wert **algorithm** übergibt die ID des Hash-Algorithmus, welcher in der aktuellen Passwortberechnung benutzt wird. **Target** übermittelt den Hash des Zielpasswortes. Anhand des Hashes kann ein Worker bestimmen, ob das Zielpasswort berechnet wurde. Die **workerID** ist eine vom Master vergebene, fortlaufende Nummer und dient der Identifizierung der Worker.

`getWork`

```
{
        "status" : "newWorkBlog",
        "value" : "#NEW_HASH(ES)"
}
```

Diese Nachricht übermittelt dem Worker eine Anzahl neuer Passwörter, von denen dieser die Hashes berechnen wird.

Nachrichten zum Master

Folgende Nachrichten werden primär von den Workern an die Master gesendet.

`newClientRegistration`

```
{
  "status" : "newClientRegistration",
  "value" : {
    "worker" : "#WORKER_ID"
  }
}
```

Beschreibung hier.

`hitTargetHash`

```
{
  "status" : "hitTargetHash",
  "value" : {
    "hash" : "#HASH_VALUE",
    "password" : "#PASSWORD"
    "time_needed" : "#TIME"
```

```
  }
}
```

Beschreibung hier.

`finishedWork`

```
{
  "status" : "finishedWork",
  "value" : "#WORKER_ID"
}
```

Beschreibung hier.

Allgemeine Nachrichten

`pingRequest`

```
{
  "status" : "stillAlive",
  "value" : ""
}
```

Beschreibung hier.

`Ping reply`

```
{
  "status" : "alive",
  "value" : "true/false"
}
```

Beschreibung hier.

2.2 Brute Force-Algorithmus

Grundlegend ist das Ziel des Projektes das Entschlüsseln eines vorgegebenen Passwortes. Das zu entschlüsselnde Passwort wird vor der Berechnung vom Benutzer eingetragen. Das eingetragene Passwort wird dann durch eine Hashfunktion geleitet. Der entstandene Hash wird gespeichert und dient als Zielbedingung der folgenden Berechnung.

Nun beginnt der eigentliche Angriff. Zu Beginn wird eine sogenannte „Dictionary-Attack“ vorgenommen. Dies bedeutet, dass ein Wörterbuch mit häufig genutzten Passwörtern als Basis des Angriffs genutzt wird. Durch die vorangestellte Attacke auf Basis von häufig benutzten Passwörtern wird die Wahrscheinlichkeit des effizienten Entschlüsseln des gesuchten Passworts erhöht.
Bleibt die Dictionary-Attack erfolglos, werden alle möglichen Zeichenkombinationen untersucht. Der steuernde Rechner wird alle möglichen Passwörter in einem Array ablegen. Das Muster der möglichen Passwörter soll wie folgt aufgebaut werden:
`Muster der zu berechnenden Passwörter:`

```
1         Array passwordsUPPER =
2                 [A*****,
3                 B*****,
4                 C*****,
5                 D*****,
6                 ...
7         ]
8
9
10        Array passwordsLOWER =
11                [a*****,
12                b*****,
13                c*****,
14                d*****,
15                ...
16        ]
17
18
19
20        Array passwordsNUM =
21                [1*****,
22                2*****,
23                3*****,
24                4*****,
25                ...
26        ]
```

Die exemplarische Darstellung soll die Aufteilung der Aufgaben verdeutlichen. Die hier dargestellte feste Länge der Passwörter auf 6 Zeichen dient als Proof Of Concept. Wenn

dieses Proof of Concept erfolgreich umgesetzt werden kann, wird in der nächsten Iteration eine variable Passwortlänge ermöglicht. Im ersten Schritt soll die Passwortlänge noch ermittelt werden, bevor die Berechnung der möglichen Passwortkombinationen beginnt. Dadurch wird die Berechnung der Aufgabenverteilung vereinfacht. Wenn auch dieser Meilenstein erfolgreich implementiert werden kann, soll in der nächsten Iteration die Berechnung ohne bekannte Passwortlänge durchgeführt werden. Dies bedeutet implizit, dass die Berechnungsdauer durch die gewachsene Anzahl an möglichen Passwortkombinationen stark ansteigt. Dadurch dann die Robustheit der konzipierten verteilten Architektur geprüft werden.

Das Befüllen des Arrays mit den Passwort-Mustern wird dynamisch durch eine Schleife geschehen. Die Schleife wird in Abhängigkeit angepasst, je nachdem, ob die Passwortlänge bekannt oder nicht bekannt ist.
Die Rechner innerhalb der verteilten Architektur (Worker) holen sich nun „ihre“ Aufgaben aus dem Array und beginnen mit der Berechnung. Die Worker berechnen nun, in Abhängigkeit der vorher bezogenen Aufgabe, alle möglichen Passworte und die zugehörigen Hashes. Die berechneten Hashes werden nun mit dem hinterlegten Hash des zu entschlüsselnden Passwortes verglichen. Sind berechneter Hash und Zielhash gleich, gilt das Passwort als entschlüsselt.

`Pseudoalgorithmus der Passwortberechnung:`

```
1         func calculateHashes([u******])
2         {
3                 while(![u******].isempty);
4                         calculateNextPassword([u******]);
5                         calculateHash(calculatedPassword);
6
7                         if(compareHashWithTargetHash
8                         (calculatedHash)==true)
9                                 print("Password encrypted")
10        }
```

Dieser stark gekürzte Pseudoalgorithmus verdeutlicht die geplante Vorgehensweise bei der Passwortentschlüsselung.

3 Fazit und Ausblick

3.1 Zusammenfassung des Projekts

3.2 Kritische Würdigung

3.3 Ausblick

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Tanenbaum u. van Steen 2003] Tanenbaum, Andrew S. ; Steen, Maarten van: *Verteilte Systeme: Grundlagen und Paradigmen*. Bd. 1. Pearson Studium, 2003