

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, 22. Februar 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

Vorwort	2
1 Motivation und Grundlagen	6
1.1 Ziel des Projekts	6
1.2 Grundlagen	7
1.2.1 Hardwarebasis	7
1.2.2 Softwarebasis	11
2 Implementation des BruteForce-Algorithmus	13
2.1 Architektur	13
2.1.1 Nachrichtenstruktur	13
2.2 Brute Force-Algorithmus	16
2.3 Benutzeroberfläche	19
3 Fazit und Ausblick	20
3.1 Zusammenfassung des Projekts	20
3.2 Kritische Würdigung	20
3.3 Ausblick	20
Abbildungsverzeichnis	21
Tabellenverzeichnis	22
Glossar	23
Literaturverzeichnis	24

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 der Systemkomponenten 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 rezierten 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 18.12.2015) aktuellste Version des Betriebssystems El Capitan 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 Capitan

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 Capitan

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 Capitan

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 Capitan

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 Capitan

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 Capitan

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 Capitan

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 Capitan

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 Capitan

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 Capitan

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 Capitan) 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 wird¹ 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 vereinfachten dezentralen Entwickeln wird die Programmcode-Plattform www.github.com eingesetzt. Die auf dem Versionsverwaltungssystem *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.

¹<https://github.com/apple/swift>

²<https://github.com/Carthage/Carthage>

Durch die Auflösung von Abhängigkeiten, beispielsweise von bestimmten Frameworks, wird das asynchrone und dezentrale Entwickeln weiter optimiert.

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 wird im Anschluss die Implementation erfolgen.

2.1 Architektur

In diesem Kapitel wird die geplante Architektur des verteilten Systems beschrieben. Generell wird die Kommunikation nachrichtenbasiert umgesetzt. Dies bedeutet, dass die Komponenten des verteilten Systems kommunizieren, indem sie sich Nachrichten übermitteln können. Die Steuerung der Kommunikation wird primär von den als Master (siehe 3.3) ausgewählten Komponenten umgesetzt. Als Provider wird ein angepasster Webserver eingesetzt, die Verbindung der einzelnen Komponenten zueinander geschieht über WebSockets. Der Webserver basiert auf dem Javascript-Framework Node.js. Details der nachrichtenbasierten Kommunikation werden nun in den folgenden Kapiteln beschrieben.

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 dem Master und dem Worker oder den Workern. Zur Strukturierung fiel die Wahl auf das Nachrichtenaustauschformat „JavaScript Object Notation“ oder kurz *JSON*, da dieses Format sehr leichtgewichtig und sehr gut anpassbar ist.

Nachfolgend werden die entwickelten Nachrichtenstrukturen und deren Inhalt detailliert beschrieben. Die Nachrichten werden, je nach Senderichtung, in *Nachrichten zum Worker* und *Nachrichten zum Master* unterteilt.

Nachrichten zum Worker

Hier werden die Strukturen beschrieben, welche 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   }
8 }
```

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

```
1 {
2   "status" : "newWorkBlog",
3   "value" : {
4     "worker_id" : "#WORKER_ID"
5     "hashes" : ["#NEW_HASHES"]
6   }
7 }
```

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

stillAlive

```
1 {
2   "status" : "stillAlive",
3   "value" : ""
4 }
```

Mit Hilfe dieser Nachricht fragt der Master an, ob die angesprochenen Worker noch verfügbar sind. Wenn diese nicht antworten, werden sie aus dem Array der verfügbaren Worker entfernt.

Nachrichten zum Master

Folgende Nachrichten werden von den Workern an die Master gesendet.

newClientRegistration

```
1 {
2     "status" : "newClientRegistration",
3     "worker" : "#WORKER_ID"
4 }
```

Der Worker beantragt eine ID, um sich im Cluster identifizieren zu können.

hitTargetHash

```
1 {
2     "status" : "hitTargetHash",
3     "value" : {
4         "hash" : "#HASH_VALUE",
5         "password" : "#PASSWORD"
6         "time_needed" : "#TIME"
7         "worker_id" : "#WORKER_ID"
8     }
9 }
```

Diese Nachricht wird vom Worker versendet, wenn der berechnete Hash dem Zielhash entspricht und somit das Passwort berechnet wurde. Es werden der berechnete Hash und das zugehörige Passwort übertragen. Zudem wird die Zeit, die die Berechnung in Anspruch genommen hat, übertragen. Die Zeit kann für spätere Erweiterungen des Projekts genutzt werden, beispielsweise zum Vergleich verschiedener Hash-Algorithmen.

finishedWork

```
1 {
2     "status" : "finishedWork",
3     "value" : "#WORKER_ID"
4 }
```

Mit dieser Nachricht teilt der Worker mit, dass alle möglichen Passworte des aktuellen Arbeitspakets berechnet worden sind. Falls bei der Berechnung der Zielhash bzw. das Zielpasswort berechnet worden ist, wird zusätzlich die Nachricht „finishedWork“ versandt. Ansonsten erhält der Worker ein neues Arbeitspaket aus dem Nachrichtenstrom.

HashesPerTime

```
1 {
2   "status" : "hashesPerTime"
3   "value" : {
4     "worker_id" : "#WORKER_ID"
5     "hash_count" : "#NUMBER_COMPUTED_HASHES"
6     "time_needed" : "#TIME"
7   }
8 }
```

Zum Auswerten der ausgeführten Tätigkeiten übermittelt der Worker zur Identifikation seine ID und zur statistischen Auswertung sowohl die Anzahl der berechneten Hashes, als auch die zu dieser Berechnung benötigten Zeit.

replyAlive

```
1 {
2   "status" : "alive",
3   "value" : "#WORKER_ID"
4 }
```

Der Worker meldet mit dieser Nachricht, dass er dem verteilten System weiterhin zur Verfügung steht. Zur Identifikation antwortet der Worker auf die Nachricht *stillAlive* mit seiner Worker-ID. Erfolgt auf die genannte Anfrage keine Antwort, dann entfernt der Master den nicht antwortenden Worker aus dem Array verfügbarer Komponenten.

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 Entschlüsselns des gesuchten Passworts erhöht.

Bleibt die Dictionary-Attack erfolglos, werden alle möglichen Zeichenkombinationen untersucht.

Die erste Idee war es, dass der steuernde Rechner alle möglichen Passwörter in einem Array ablegen wird. Das Muster der möglichen Passwörter sollte 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 geplante Aufteilung 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.

Vorgehen des Brute-Force Algorithmus

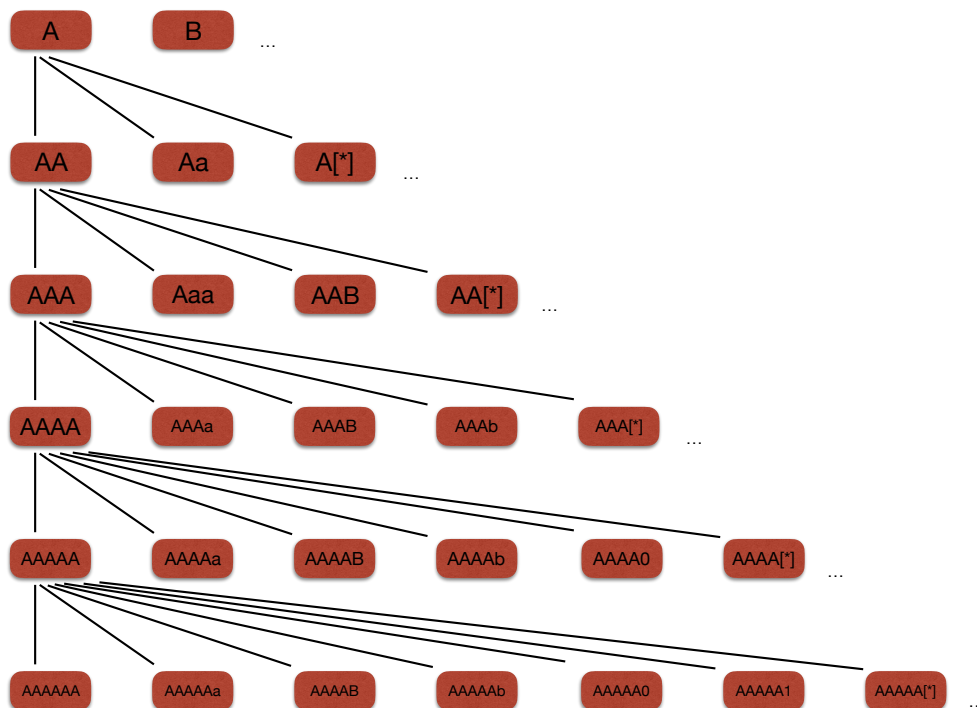


Abbildung 2.1: Darstellung der Suchstrategie, die der Brute-Force Algorithmus zum Ermitteln des Passwortes benutzt.

2.3 Benutzeroberfläche

Um Rückschlüsse auf die Geschwindigkeit verschiedener Hash-Algorithmen schließen zu können, kann bei Start der Applikation zwischen verschiedenen Hash-Algorithmen gewählt werden. Zur Auswahl stehen die Hash-Algorithmen *MD5*, *SHA 128* sowie *SHA 256*. Die Geschwindigkeitsunterschiede beruhen primär auf der unterschiedlichen Schlüssellänge der jeweiligen Algorithmen.

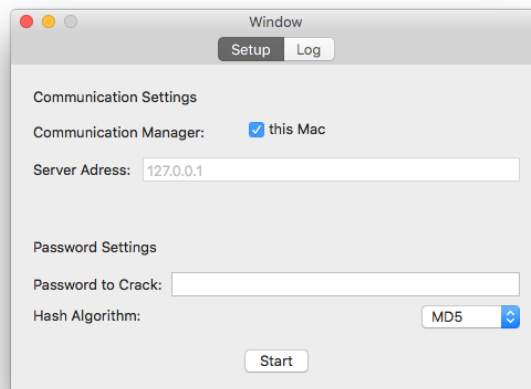


Abbildung 2.2: Benutzeroberfläche der implementierten Anwendung als Master der verteilten Anwendung

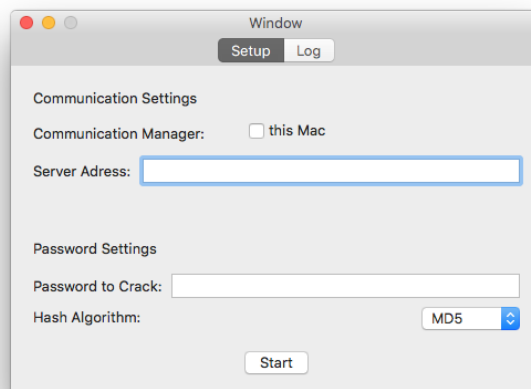


Abbildung 2.3: Benutzeroberfläche der implementierten Anwendung als Worker der verteilten Anwendung

3 Fazit und Ausblick

3.1 Zusammenfassung des Projekts

3.2 Kritische Würdigung

3.3 Ausblick

Abbildungsverzeichnis

2.1	Darstellung der Suchstrategie, die der Brute-Force Algorithmus zum Ermitteln des Passwortes benutzt.	18
2.2	Benutzeroberfläche der implementierten Anwendung als Master der verteilten Anwendung	19
2.3	Benutzeroberfläche der implementierten Anwendung als Worker der verteilten Anwendung	19

Tabellenverzeichnis

Glossar

Master	Instanz innerhalb des verteilten Systems, welche für die Verteilung der anfallenden Tätigkeiten verantwortlich ist.
Provider	Synonym für den <i>Master</i> .
Worker	Instanz innerhalb des verteilten Systems, welche für die Bearbeitung der anfallenden Tätigkeiten verantwortlich ist.

Literaturverzeichnis

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