Technology
Arts Sciences
TH Köln

Distributed Hashcracker

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, 20. Juli 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 BruteForce-Angriffs mit Hilfe einer verteilten Architektur.
Mit Hilfe des BruteForce-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 Prinzipien eines verteilten Systems gerecht werden.

Inhaltsverzeichnis

1 Einführung

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 2.1 beschrieben wird. Kernbestandteil dieses Projekts wird die Konzeption und Implementation des verteilten Systems sein.

1.1 Vorgehen

Um grundlegende Kenntnisse im Bereich der verteilten Systeme zu erhalten, soll das Projekt mit einer Literaturrecherche beginnen. Dabei soll eine Literaturauswahl getroffen werden, auf Basis derer genügend Fachkenntnisse gewonnen werden können.
Danach soll die Projektidee konkretisiert und anschließend definiert werden, welche in Abschnitt 1.2 ausführlicher beschrieben werden soll.
Nach der Definition der Projektidee soll die Konzeption des geplanten verteilten Systems geschehen. Das entstehende Konzept soll die Grundlagen für die folgende Implementation bereitstellen.
Der Kern des Projektes wird nach der Konzeption die nachfolgende Implementation darstellen, bei der die theoretischen Überlegungen in die Praxis umgesetzt werden.
Zum Abschluss der Dokumentation sollen die durchgeführten Aktionen zusammengefasst werden. Darüber hinaus sollen diese kritisch reflektiert werden, um so eine bessere Einordnung des Gesamtprojekts zu gewährleisten. Abschließend sollen in einem Ausblick mögliche Zukunftsperspektiven und Ansatzpunkte für die Fortsetzung dieses Projekts gegeben werden.

1.2 Projektidee

Zu Beginn des Projektes wurde ein Problem definiert, das auf Basis einer verteilten Architektur gelöst werden kann. Das Projektteam entschied sich für ein Problem aus dem Fachgebiet der IT-Sicherheit.
Die Wahl fiel auf die Implementation eines BruteForce-Angriffs. Konkret bedeutet dies,

dass durch Ausprobieren aller möglichen Zeichenkombinationen versucht wird ein Passwort zu entschlüsseln. Durch die hohe Rechenleistung eines verteilten Systems bietet dieses eine ideale Grundlage für die Durchführung eines BruteForce-Angriffs.
Das zu entschlüsselnde Passwort wird zu Beginn vom Benutzer eingetragen und in Form eines Hashes hinterlegt. Da das Projekt unter Laborbedingungen stattfinden soll, ist das Eintragen des Passwortes durch den Benutzer im gegebenen Kontext ausreichend. In der Praxis sollte das Passwort automatisiert übernommen werden. 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 3.1 erläutert.

1.3 Allgemeine Anforderungen

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 zitierten 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.

2 Grundlagen des verteilten Systems

Dieses Kapitel bietet Informationen zur verwendeten Hard- und Software. Damit werden die Umstände, unter denen das Projekt durchgeführt wird, verdeutlicht. Außerdem wird die Reproduzierbarkeit und damit der wissenschaftliche Anspruch an das Projekt realisiert.

2.1 Hardware

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 09.07.2016) aktuellste Version des Betriebssystems El Capitain und der Entwicklungsumgebung Xcode installiert. Details zu den Softwareversionen sind in Kapitel 2.2 zu finden.
In der folgenden Liste sind die Details zu den einzelnen Rechnern zu aufgeführt:

- **pip02-pip05: Mac Pro (Anfang 2008)**

  Seriennummer: CK8250EUXYL

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

  RAM: 8 GB 800 MHz DDR2 FB-DIMM

  Grafikkarte: NVIDIA GeForce 8800 GT (512MB)

  OS 10.11.5 El Capitain

  Xcode 7.3, Swift 2.1.1

- **pip06-pip11: Mac Pro (Anfang 2009)**

  Seriennummer: CK92608B20H

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

  RAM: 8 GB 1066 MHz DDR3 ECC

  Grafikkarte: NVIDIA GeForce GT 120 (512MB)

OS 10.11.5 El Capitain

Xcode 7.3, 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 1024 Megabit/s (Gigabit-LAN).

2.2 Software

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. Ziel ist es aber, möglichst nur auf Swift zurückzugreifen und Objective C da einzusetzen, wenn keine andere Möglichkeit besteht.

2.2.1 Versionsverwaltung

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.

2.2.2 Frameworks

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 leichtgewichtiger im Projekt, als es bei CocoaPods der Fall wäre. Dadurch entsteht eine weniger starke Abhängigkeit. Aus den genannten Gründen fiel die Wahl auf die Verwendung von Carthage.
Zudem wird die Plattform *Node.JS* benutzt, um einen Kommunikationsserver zur Verfügung zu stellen. Weitere Informationen dazu sind in Kapitel 3 zu finden.

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

3 Konzeption

Nachfolgend wird die Konzeption des Projekts beschrieben. Die Kommunikation wird nachrichtenbasiert umgesetzt. Dies bedeutet, dass die Komponenten des verteilten Systems kommunizieren, indem sich sich Nachrichten zusenden. Die Steuerung der Kommunikation wird primär von den als Master (siehe 5.3) ausgewählten Komponenten umgesetzt. Als Provider wird ein an das Projekt angepasster Webserver eingesetzt. Die Verbindung der einzelnen Komponenten zueinander geschieht über WebSockets.

3.1 Brute Force-Algorithmus

Primäres Ziel des Projektes ist das Entschlüsseln eines vorgegebenen Passwortes. Das zu entschlüsselnde Passwort wird vor der Berechnung vom Benutzer eingegeben. Das eingetragene Passwort wird dann durch eine Hashfunktion geleitet. Der erzeugte 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. Die eingetragenen Passwörter werden ebenfalls der Reihe nach gehasht und der berechnete Hash mit dem Zielhash verglichen.
Durch die vorangestellte Attacke auf Basis von häufig benutzten Passwörtern wird die Effizienz des verteilten Systems verbessert. Der folgende Auszug aus der Passwortliste verdeutlicht das Prinzip eines Dictionaries.

Dictionary:

```
        LOVE123
        LOVEME1
        Lamont1
        Leasowes2
        Lemon123
        Liberty
        Lindsay
        Lizard
        Love21
        MASTER
        MORIAH07
        MOSS18
        Madeline
        Margaret
        Master
        Matthew
        Maxwell
        Mellon
        Merlot
        Metallic
        Michael
```

Bleibt die Dictionary-Attack erfolglos, wir der BruteForce-Angriff durchgeführt.
Die erste Idee war es, dass der steuernde Rechner alle möglichen Passwörter berechnet und in einem Array ablegen wird. Dabei sollten die Passwörter eine statische Länge besitzen. Das Muster der möglichen Passwörter sollte wie folgt aufgebaut werden:

Muster der zu berechnenden Passwörter:

```
        Array passwordsUPPER =
                [A*****,
                B*****,
                C*****,
                D*****,
                ...
        ]


```

```
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 Beispiel.

In der Praxis werden alle möglichen Passwörter probiert, begonnen mit Passwörtern der Länge = 1. Wird kein Passwort ermittelt, so werden Passwörter der Länge n+1 erzeugt. Abbildung 3.1 stellt das Konzept zur Berechnung aller möglichen Passwörter dar. Der genaue Algorithmus dazu wird in Kapitel 4 ausführlicher beschrieben.

Die berechneten Passwörter werden dann in eine Warteschlange geschrieben, sodass sich die Worker des verteilten Systems (Beschreibung siehe 5.3) „Arbeitspakete“ beziehen können. Ein solches Arbeitspaket beinhaltet eine Sammlung von Passwörtern. Die Worker berechnen dann die Hashes der Passwörter und vergleichen diese mit dem Zielhash.
Berechnet ein Worker einen Hash, der mit dem Zielhash überein stimmt, so ist das gesuchte Passwort identifiziert.

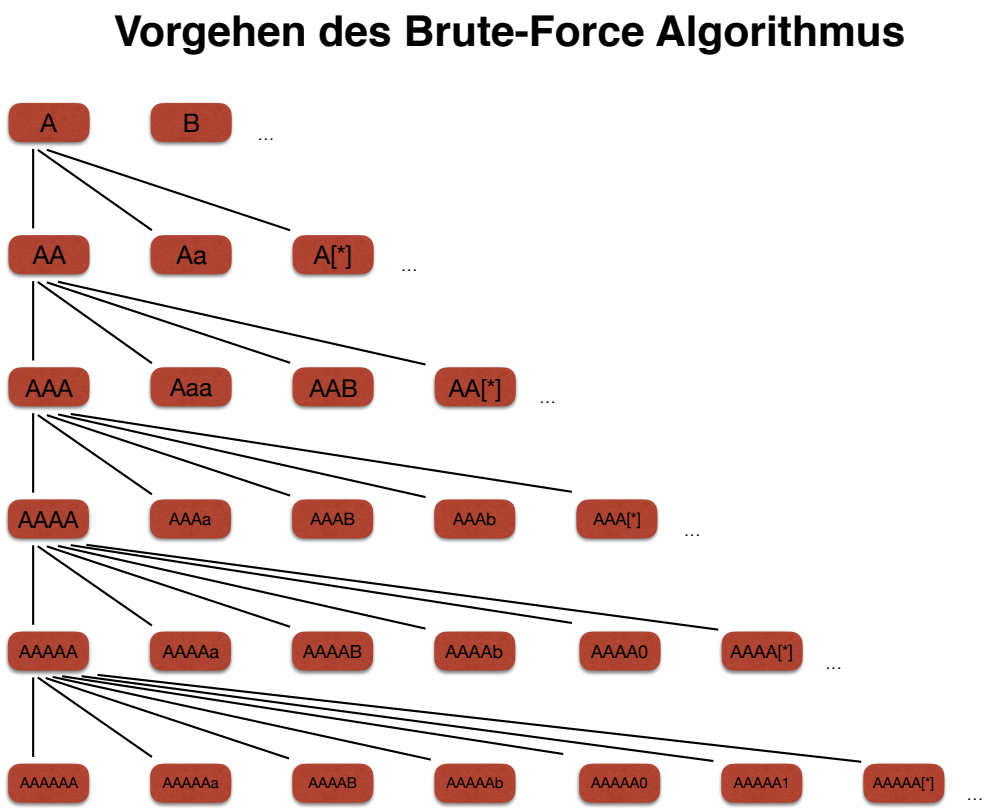
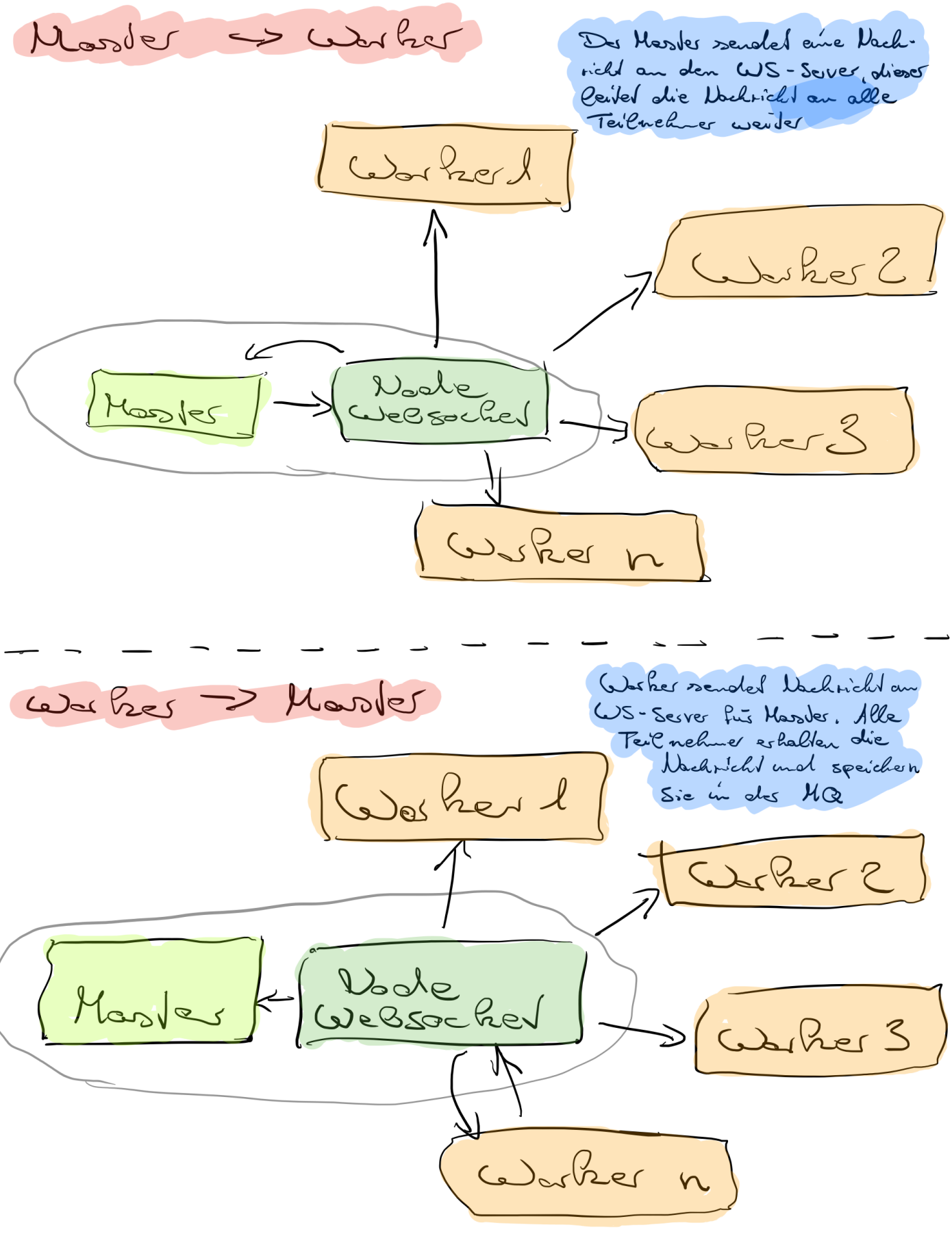


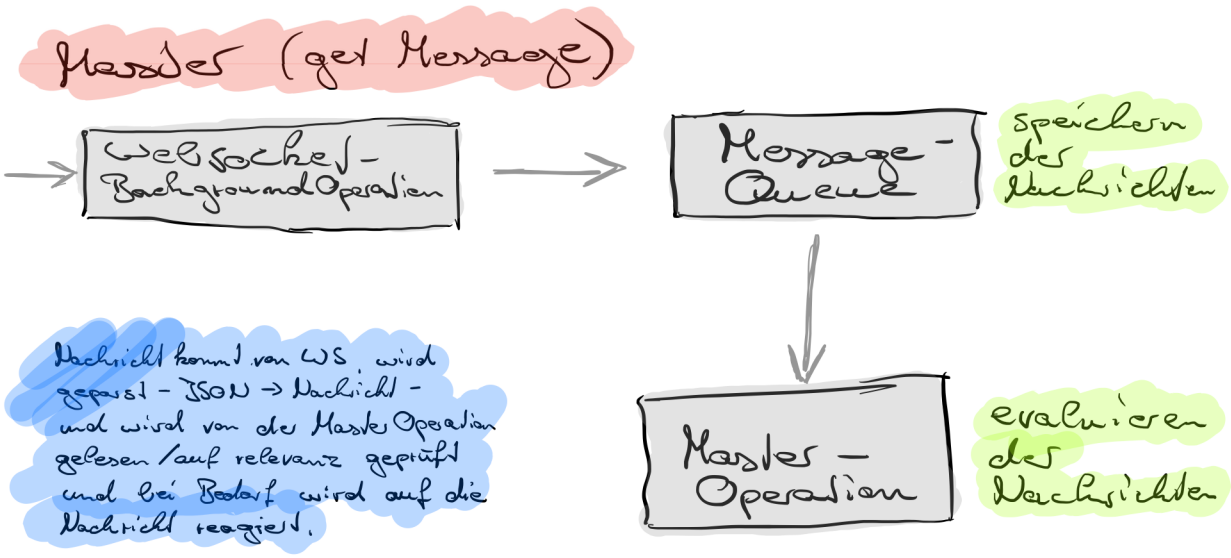
Abbildung 3.1: Darstellung der Suchstrategie, die der Brute-Force Algorithmus zum Ermitteln des Passwortes benutzt.

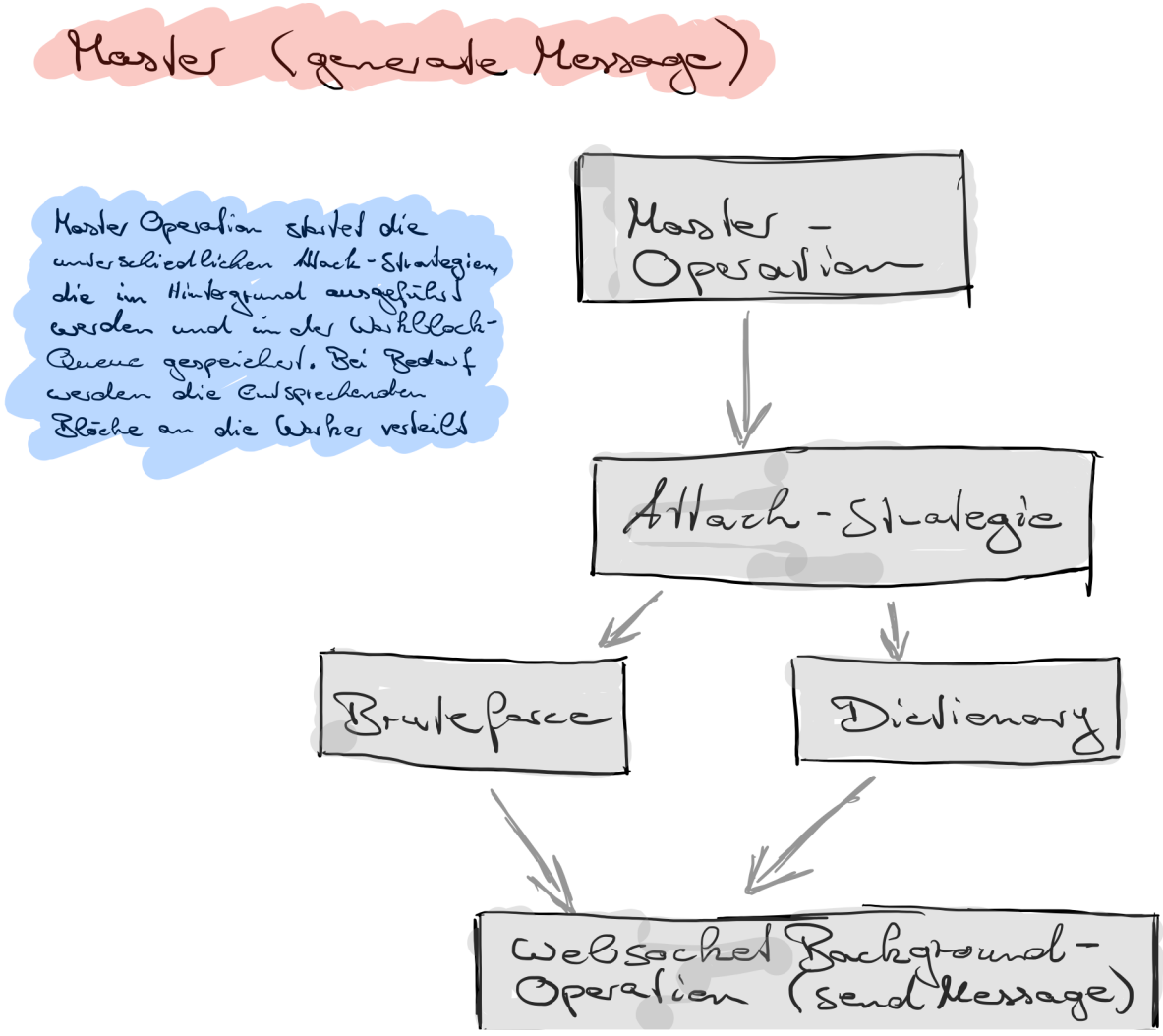
3.2 Architektur des verteilten Systems

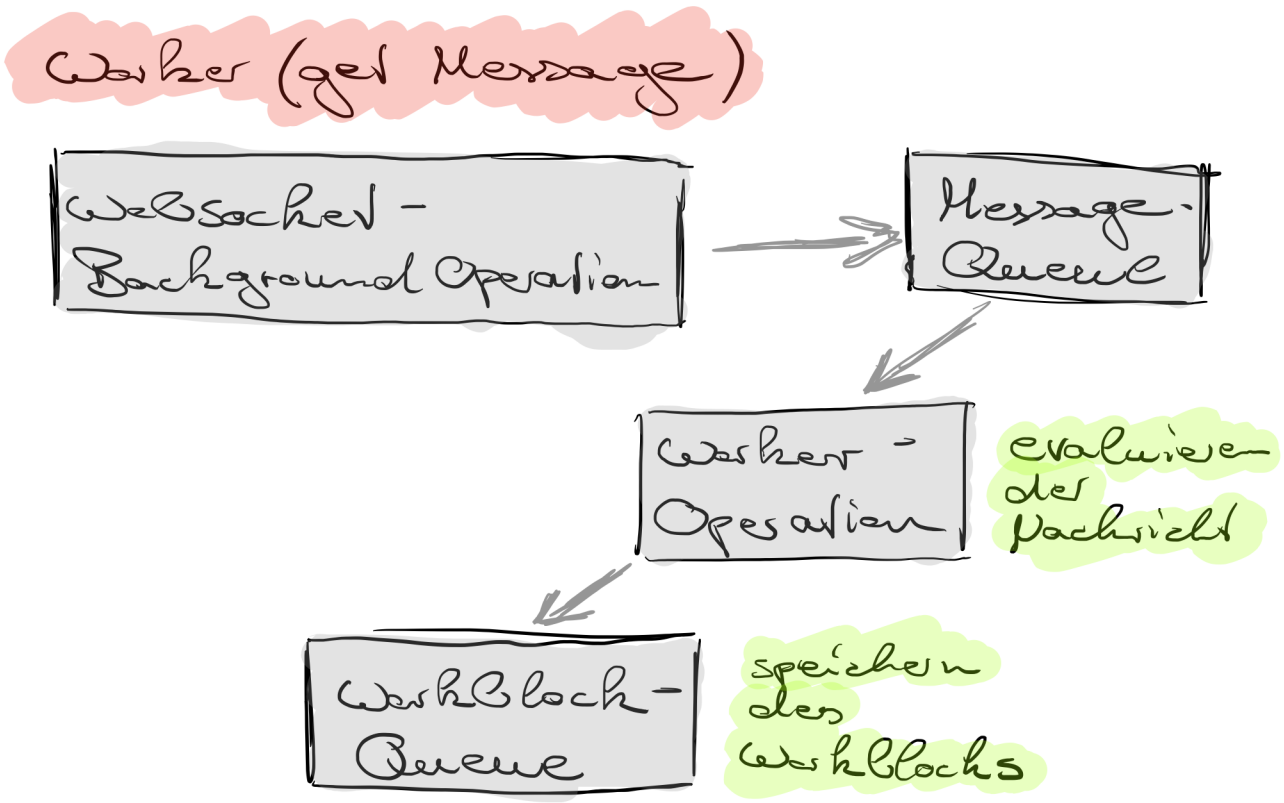
Das verteilte System wird so konzipiert, dass die vier in Kapitel 1.3 genannten Anforderungen umgesetzt werden können. Um dies zu realisieren, wird mit einer nachrichtenbasierten Architektur geplant. Die Rechner des verteilten Systems können so miteinander kommunizieren, ohne dass eine restriktive Anordnung notwendig ist. Dadurch wird die Skalierbarkeit sichergestellt.
Außerdem realisiert die nachrichtenbasierte Architektur, dass die Ressourcen innerhalb des verteilten Systems leicht verfügbar sind.
Im verteilten System übernimmt der Master die Verantwortlichkeit für die Kommunikation, sodass die Skalierbarkeit weiter verbessert wird. Es wird lediglich ein Master im System benötigt, die Menge der Worker ist irrelevant. Konkret wird der Master die Funktion des Webservers beinhalten, welcher für die Steuerung des Nachrichtenversands innerhalb des verteilten Systems verantwortlich ist. Die Verbindungen werden mit Hilfe von Websockets aufgebaut.
Die Nachrichten werden versandt und bei jedem Rechner lokal in einer Warteschlange

(MessageQueue) gespeichert. Somit besteht für das gesamte verteilten Systems jederzeit Zugriff auf alle Nachrichten. Je nach Inhalt der Nachricht (z.B. „Nachricht an Worker“) wird entschieden, wer die jeweiligen Nachrichten erhalten oder verarbeiten soll.
In den folgenden Skizzen sind die Planungen auf konzeptioneller Ebene zusammengefasst.

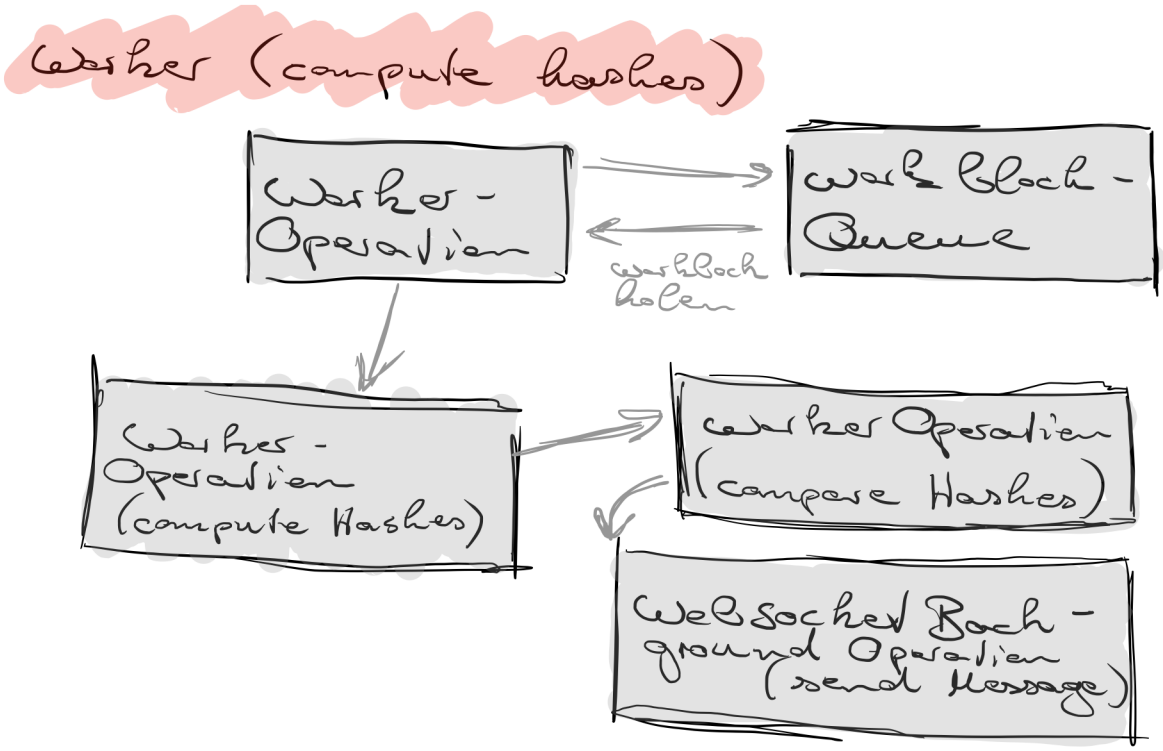

Master → Worker
Der Master sendet eine Nachricht an den WS-Server, dieser leitet die Nachricht an alle Teilnehmer weiter
Worker 1
Worker 2
Master
Node Websocket
Worker 3
Worker n
Worker → Master
Worker sendet Nachricht an WS-Server für Master. Alle Teilnehmer erhalten die Nachricht und speichern sie in das MQ
Worker 1
Worker 2
Master
Node Websocket
Worker 3
Worker n


Master (get Message)
Websocket - Background Operation
Message - Queue
speichern der Nachrichten
Nachricht kommt von WS, wird geparst - JSON → Nachricht - und wird von der Master Operation gelesen / auf relevanz geprüft und bei Bedarf wird auf die Nachricht reagiert.
Master - Operation
evaluieren der Nachrichten


Master (generate Message)
Master Operation startet die unterschiedlichen Hash-Strategien, die im Hintergrund ausgeführt werden und in der Workblock-Queue gespeichert. Bei Bedarf werden die entsprechenden Blöcke an die Worker verteilt
Master - Operation
Attack - Strategie
Bruteforce
Dictionary
Websocket Background - Operation (send Message)



Nachricht wird von WS Operation in der MessageQueue gespeichert. Aus der MessageQueue holt sich die WorkerOperation die Nachricht und evaluiert sie auf relevanz (Für Worker gedacht und für diesen Worker gedacht) Handelt es sich hierbei um einen WorkBlock, wird dieser in die WorkBlock-Queue gespeichert



Die WorkerOperation holt sich im Hintergrund laufend Work-Blöcke und berechnet die Hashwerte für alle Elemente innerhalb des Blocks. Nach der Hash-Berechnung erfolgt der Vergleich mit dem Ziel-Hash auf Basis dessen wird ein neuer Work Block angefordert oder die Verarbeitung Clusterweit beendet (Hash gefunden)

4 Implementation

In diesem Kapitel wird die Implementation des Projekts beschrieben. Auf Basis der beschriebenen Konzeption wird im Anschluss die Implementation erfolgen.

4.1 Kommunikation

Da die Kommunikation des verteilten Systems nachrichtenbasiert geschieht, ist der Entwurf eigener Nachrichtenstrukturen notwendig. Die Kommunikation zwischen dem Master den Workern findet mit Hilfe der Nachrichten statt. Konkret handelt es sich nach [Tanenbaum u. van Steen, 2003] bei der gewählten Architektur um eine *Message-Queue-Architektur*.
Zur Strukturierung fiel die Wahl auf das Nachrichtenaustauschformat „JavaScript Object Notation“ oder kurz *JSON*, da dieses Format sehr leichtgewichtig und vollständig anpassbar ist. Zudem ist die Maschinenlesbarkeit im Vergleich zu XML einfacher umzusetzen, da zur Interpretation von XML eigene Schemata entwickelt werden müssten. Als Kommunikationsplattform wird ein Node-Server benutzt, da dieser für den vorgesehenen Bereich schnell einsetzbar und mit dem JSON-Format kompatibel ist. Der Webserver basiert auf dem Javascript-Framework Node.js und ist für die Verteilung der Nachrichten zwischen den Rechnern des verteilten Systems verantwortlich.

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.

Nachfolgend werden die entwickelten Nachrichtenstrukturen und deren Inhalt beschrieben. Zum besseren Verständnis werden die Nachrichten in die Senderichtungen zum Worker oder zum Master strukturiert.
Auf programmatischer Ebene wird zudem zwischen *Basic Messages* und *Extended Messages* unterschieden. Basic Messages beinhalten den Status des sendenden Rechners und

maximal einen weiteren Wert, während Extended Messages neben dem Status mehrere Werte beinhalten. Durch diese Strukturierung ist ein effektives Messageparsing möglich.

4.1.1 Asynchronität

Die Kommunikation innerhalb des verteilten Systems wird durch den Master sichergestellt. Konkret werden die Nachrichten durch den NODE-Server an alle Rechner des verteilten Systems (Broadcast) versendet. Dies bedeutet, dass jeder Rechner alle Nachrichten erhält und lokal zur Verfügung stehen. Damit die jeweiligen Rechner die Nachrichten zuordnen können, werden die Funktionen *decideWhatToDoBasicMessage* und *decideWhatToDoExtendedMessage* implementiert. Diese unterscheiden, wie der Name bereits vermuten lässt, zwischen *Basic Messages* und *Extended Messages*. Aber auch bei Master und Worker sind die Funktionen unterschiedlich implementiert, da diese unterschiedlich auf die Nachrichten reagieren.
Die Funktionen treffen ihre Entscheidungen auf Basis des Nachrichtenheaders sowie des Ziels der Nachricht, wie in den folgenden Codefragmenten zu erkennen ist.

Klasse MasterOperation.swift

```
func decideWhatToDoBasicMessage(message:BasicMessage){
        let messageHeader = message.status

    switch messageHeader {
        case MessagesHeader.newClientRegistration:
            newClientRegistration(message)
            break
        case MessagesHeader.alive:
            alive(message)
            break
        default:
            break
    }
}


func decideWhatToDoExtendedMessage(message:ExtendedMessage){
    let messageHeader = message.status

    switch messageHeader {
        case MessagesHeader.hitTargetHash:
            hitTargetHash(message)
            break
        case MessagesHeader.hashesPerTime:
            hashesPerTime(message)
            break
        case MessagesHeader.finishedWork:
            finishedWork(message)
            break
        default:
            break
    }
}
```

Klasse WorkerOperation.swift

```
func decideWhatToDoBasicMessage(message:BasicMessage){
   let messageHeader = message.status
        switch messageHeader {
        case MessagesHeader.stillAlive:
            stillAlive(message)
            break
        case MessagesHeader.stopWork:
            stopWork()
            break
        default:
            break
    }
}

func decideWhatToDoExtendedMessage(message:ExtendedMessage){
    let messageHeader = message.status

    switch messageHeader {
        case MessagesHeader.setupConfig:
            setupConfig(message)
            break
        case MessagesHeader.newWorkBlog:
            newWorkBlog(message)
            break
        default:
            break
     }
}
```

Dieses Vorgehen hat den Vorteil, dass die Verarbeitung der Nachrichten asynchron geschehen kann. Durch die asynchrone Verarbeitung kann die Performance des verteilten Systems deutlich gesteigert werden, da die Rechner unabhängig voneinander arbeiten können.

Die Nachrichten werden in Warteschlangen (Queues) gespeichert. Durch den Einsatz von *GrandCentralDispatch (GCD)*[1] wird die Verteilung von Threads auf verschiede-

[1] https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html

ne Prozessorkerne vom Betriebssystem übernommen. Durch diese Abstraktion ist für den Entwickler eine effektivere Verteilung von Threads möglich. Da die Erstellung von Queues durch das Grand Central Dispatch realisiert wird, ist die Verteilung im verteilten System effizient möglich. Das GCD legt die Queues in Threads ab und übernimmt deren Verteilung. Zudem unterstützt GCD die Prozessverteilung auf Ebene des Betriebssystems.
Um die Performance weiter zu steigen, sollen die erstellten Threads eine hohe Priorität erhalten, um vom Betriebssystem bevorzugt bearbeitet zu werden. Die Zuweisung einer hohen Priorität unter GCD ist allerdings nur über einen Umweg möglich, da hohe Prioritäten nur für bestimmte Systemdienste vorgesehen sind. Mit folgendem Programmcode können die Nachrichten-Warteschlangen dennoch mit einer hohen Priorität versehen werden:

`Klasse WorkerOperation.swift`

```
1   let queue = dispatch_queue_create
2   ("\(Constants.queueID).compareHashes", nil)
3
4   let highPriority =
5   dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
6
7   dispatch_set_target_queue(queue, highPriority)
8
9
10
11  dispatch_async(queue) {
12     _ = passwordArray.map { password in
13
14     let passwordQueue =
15     dispatch_queue_create("\(Constants.queueID).
16     \(password)", nil)
```

Des weiteren wird die Parallelisierung weiter ausgebaut, indem die Queues, in denen sich die Passwortblöcke befinden, auf den einzelnen Workern weiter aufgeteilt werden. Ein Beispiel dazu lässt sich ebenfalls aus dem Programmcode entnehmen.

Grafik 4.1 zeigt, dass die zwei verfügbaren Prozessorkerne (ein Kern entspricht einer Auslastung von 100%) mit der implementierten Verteilungsstrategie effektiv genutzt werden. Zur Messung wurde die Entwicklungsumgebung *Xcode* genutzt.

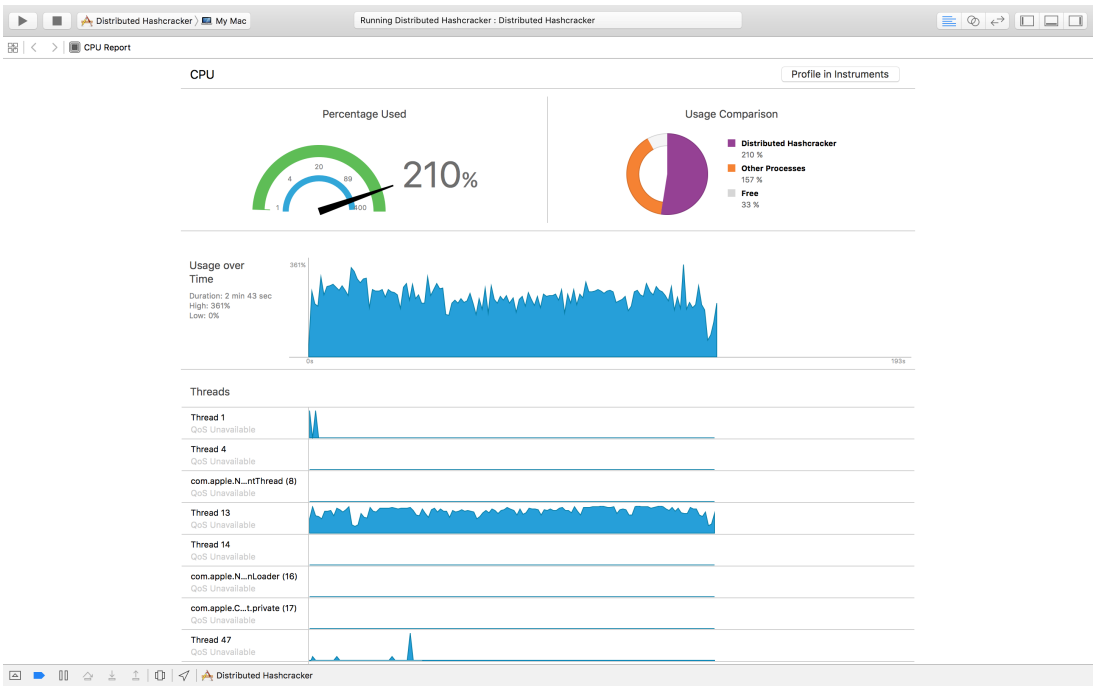


Abbildung 4.1: Verlauf der CPU-Auslastung während der Laufzeit, gemessen in Xcode mit zwei Prozessorkernen.

4.1.2 Lenses

Threadsicherheit bedeutet, dass Softwarekomponenten problemlos von verschiedenen Bereichen genutzt werden können, ohne dass dabei Nebeneffekte auftreten. Damit ist Threadsicherheit implizit wichtiger Bestandteil eines verteilten Systems.
Die Arbeitspakete (*Work Blogs*) werden in *WorkBlogQueues* gespeichert. Diese Queues beinhalten Arrays. Da Arrays in der Programmiersprache *Swift* standardmäßig nicht threadsicher sind, werden sogenannte *Lenses* eingesetzt.
Lenses stellen Getter- und Setter-Methoden zur Verfügung.
`Klasse Lens`

```
1 struct Lens<Whole, Part>
2 {
3         let get : Whole -> Part
4         let set: (Part, Whole) -> Whole
5 }
```

Die Getter-Methode ruft dabei einen Teil (Part) „des Ganzen (Whole)“ ab, während die Setter-Methode aus einem Part und einem Whole ein „neues Ganzes (Whole)“ erzeugt. Mit Hilfe dieser Methode kann mit konstanten Arrays gearbeitet werden, um die Thread-

sicherheit herzustellen. Dabei werden die Daten in konstanten Arrays abgelegt. Muss ein Eintrag bearbeitet werden, wird mit Hilfe der Getter-Methode ein konstantes Array aufgerufen. Die Inhalte können nun über die Setter-Methode bearbeitet und in ein neues, konstantes Array geschrieben.
Folgender Programmcode zeigt eine konkrete Implementierung von Lenses:
`Klasse MessageQueue`

```
func get() -> Message? {
        dispatch_semaphore_wait
        (semaphore, DISPATCH_TIME_FOREVER)
        guard let firstElement = messagesLens.
        get(messages).first else {
                dispatch_semaphore_signal(semaphore)
        return nil
        }

        messages = messagesLens.set([],
        messages.dropFirst().map{ $0 })
        dispatch_semaphore_signal(semaphore)

        return firstElement
    }
```

4.1.3 Nachrichten zum Worker

In diesem Abschnitt werden die Strukturen beschrieben, welche vom steuernden Rechner (Master) zu einem der Worker gesendet werden.

`SetupAndConfig(Extended Message)`

```
{
  "status" : "setupConfig",
  "value" : {
    "algorithm" : "#HASH_ID",
    "target" : "#TARGET_HASH",
    "worker_id" : "#WORKER_ID"
  }
}
```

Ein im Cluster neu hinzugefügter Worker erhält seine Konfigurationsparameter, damit dieser als Worker im verteilten System genutzt werden kann. Der Wert **algorithm** übergibt die ID des Hash-Algorithmus, welcher in der aktuellen Passwortberechnung benutzt wird. **Target** übermittelt den Hash des Zielpasswortes. Die **workerID** ist der vom Master vorgegebene Hostname oder die IP-Adresse, die den Workern mitgeteilt werden muss.

`getWork (Extended Message)`

```
{
  "status" : "newWorkBlog",
  "value" : {
    "worker_id" : "#WORKER_ID"
    "hashes" : ["#NEW_HASHES"]
  }
}
```

Der Worker erhält durch einen neuen WorkBlog, den dieser abarbeiten wird.

`Stop Work (Basic Message)`

```
{
  "status" : "stopWorking",
  "value" : ""]
  }
}
```

Durch „stopWork“ wird den Workern mitgeteilt, dass diese die Berechnungen stoppen können. Dies ist in der Regel der Fall, wenn das Zielpasswort identifiziert wurde.

`stillAlive (Basic Message)`

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

Mit Hilfe dieser Nachricht fragt der Master an, ob die angesprochenen Worker noch verfügbar sind. Wenn diese nicht antworten, werden sie der Liste der verfügbaren Worker (Worker Queue) entfernt.

4.1.4 Nachrichten zum Master

Folgende Nachrichten werden von den Workern an die Master gesendet.

`newClientRegistration (Basic Message)`

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

Mit Hilfe dieser Nachricht kann sich ein hinzugefügter Worker beim Master registrieren. Dieser wird dann vom Master in die Liste verfügbarer Worker hinzugefügt.

`hitTargetHash (Extended Message)`

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

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 der Worker für die Berechnung in Anspruch genommen hat, übertragen. Die Zeit kann für spätere Erweiterungen des Projekts genutzt werden, beispielsweise zum Vergleich verschiedener Hash-Algorithmen. Allerdings gilt es zu beachten, dass sich die Zeit nur auf einen Worker bezieht. Eine globale Betrachtung der Berechnungszeit kann damit ungenau sein.

`finishedWork (Basic Message)`

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

Der Worker teilt 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. Ist das Zielpasswort noch nicht ermittelt worden, erhält der Worker ein neues Arbeitspaket, da der Master nach Erhalt der Nachricht „finished work“ den Worker in den *idle*-Zustand versetzt.

`HashesPerTime (Extended Message)`

```
{
  "status" : "hashesPerTime"
  "value" : {
    "worker_id" : "#WORKER_ID"
    "hash_count" : "#NUMBER_COMPUTED_HASHES"
    "time_needed" : "#TIME"
  }
}
```

Zum Auswerten der Berechnungen übermittelt der Worker zur Identifikation seine ID und zur Auswertung sowohl die Anzahl der berechneten Hashes, als auch die Berechnungszeit mit.

`replyAlive (Basic Message)`

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

Der Worker teilt mit, dass dieser 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 Worker.

4.2 Speichermanagement

Das verteilte System muss sicherstellen, dass die Speichernutzung fehlerfrei und effektiv geschieht. Ein Beispiel für ein mögliches Problem ist die Erstellung der Arbeitspakete (*Work Packages*). Geschieht die Erstellung zu schnell, würde der Speicherbedarf stark anwachsen. Dies kann zu einem Speicherüberlauf führen. Werden die Arbeitspakete zu langsam erstellt, würden Worker auf die Erstellung warten müssen. Durch die entstehende Wartezeit würde das System verlangsamt werden.

Um diese Probleme auszuschließen, wird das verteilte System immer die doppelte Anzahl an Work Packages bereitstellen, wie aktuelle Worker vorhanden sind. Es gilt also das Verhältnis:

$$Arbeitspakete = AnzahlWorker \cdot 2$$

`Funktion generateWorkBlogs`

```
1 waitLoop: while(WorkBlogQueue.sharedInstance.
2 getWorkBlogQueueCount() >
3 (WorkerQueue.sharedInstance.getWorkerQueueCount()*2))
4         {
5         guard generateLoopRun == true else{
6                 break waitLoop
7                 }
8         }
```

Dadurch wird ein gleichbleibender Speicherbedarfs sichergestellt, wie sich anhand von Abbildung 4.2 erkennen lässt. Die Messung wurde mit *Xcode* während der Laufzeit des Angriffs durchgeführt. Dabei ist zu erkennen, dass sich der aktuelle Speicherbedarf des verteilten Systems zur Laufzeit bei ca. 73 MB einstellt. Dieser Wert ist, wie sich aus dem genannten Verhältnis ableiten lässt, abhängig von der Anzahl der zur Verfügung stehenden Worker.
Durch die Messung wird verdeutlicht, dass sich der Speicherbedarf nach einer gewissen Laufzeit einem gleichbleibendem Niveau annähert und dieses hält.

4.3 Benutzeroberfläche

Nachfolgend wird die Benutzeroberfläche sowie die Interaktion mit dem verteilten System beschrieben. Da primär die Funktionalität der Anwendung im Fokus des Projekts steht, werden nur wenig Interaktionselemente eingesetzt.
Die Anwendung wird auf allen Rechnern gestartet, die zum verteilten System zusammengefasst werden. Je nachdem, ob der Rechner als Master oder Worker verwendet werden soll, unterscheidet sich die weitere Interaktion. Die unterschiedlichen Interaktionsmöglichkeiten werden in den folgenden Abschnitten beschrieben.

4.3.1 Master

Durch Aktivierung des Feldes „this Mac“ im Bereich *Communication Manager* delegiert man den aktuellen Rechner als Master des verteilten Systems. Es gilt zu beachten, dass

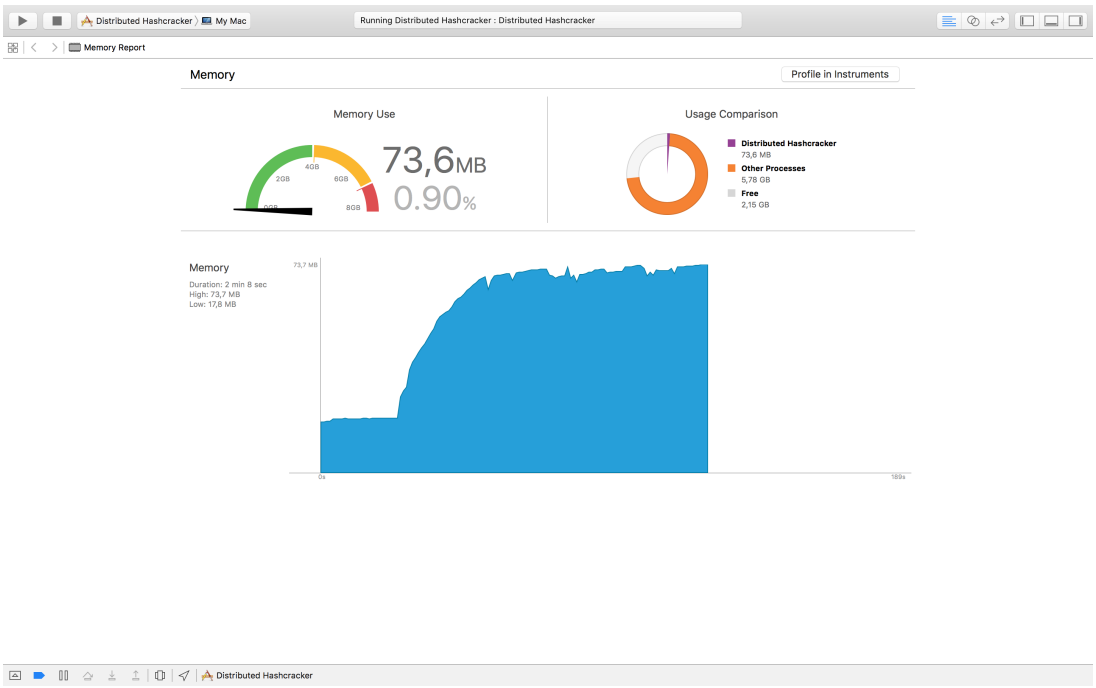


Abbildung 4.2: Verlauf der Arbeitsspeicher-Auslastung während der Laufzeit, gemessen in Xcode.

nur ein Master im verteilten System bestimmt wird. Nachdem diese Option gewählt wurde, wird im Feld *Server Adress* die Adresse des aktuellen Rechners angezeigt. Diese Adresse wird den Workern mitgeteilt (siehe Abschnitt 4.3.2). Auf Abbildung 4.3 lautet die Adresse beispielsweise *127.0.0.1*, da auf dem Rechner kein Domain-Name hinterlegt worden ist.

Nach der Konfiguration als Master wird das Passwort festgelegt, welches entschlüsselt werden soll. Dazu wird im Feld *Password to Crack* das gewünschte Passwort eingetragen. Abschließend kann der Hash-Algorithmus ausgewählt werden, mit dem die Hashes erzeugt werden. Die Vorgehensweise zur Entschlüsselung ändert sich dadurch nicht, die Option zur Wahl verschiedener Hash-Algorithmen dient primär zur Messung von Geschwindigkeitsunterschieden.

Nach Aktivierung des Start-Buttons steht der Master dem verteilten System zur Verfügung.

4.3.2 Worker

Nachdem ein Master festgelegt wurde, können beliebig viele Rechner als Worker benutzt werden. Die Rechner müssen sich lediglich im gleichen Netz befinden.

Abbildung 4.3: Benutzeroberfläche der implementierten Anwendung als Master der verteilten Anwendung

Standardmäßig ist die Anwendung nach dem Start als Worker konfiguriert. Dies ist beispielsweise daran zu erkennen, dass die Möglichkeiten zum Eintragen eines Passwortes oder die Auswahl eines Hash-Algorithmus deaktiviert sind (zu erkennen auf Abbildung 4.4).

Damit der Worker mit der Berechnung beginnen kann, muss diesem die Adresse des

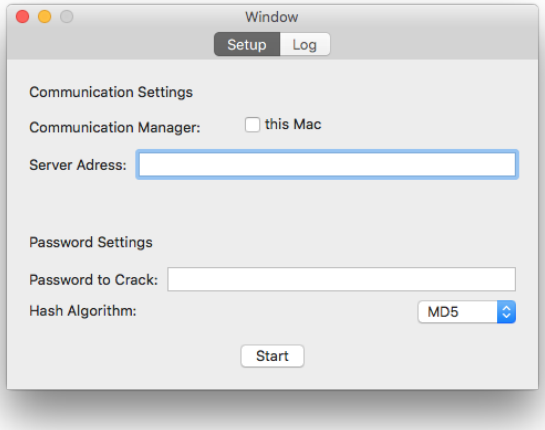


Abbildung 4.4: Benutzeroberfläche der implementierten Anwendung als Worker der verteilten Anwendung

Masters mitgeteilt werden. Diese kann aus der Anwendung des Masters entnommen werden. Im aktuellen Beispiel würde die Adresse *127.0.0.1*, entnommen aus Abbildung

4.3, eingetragen werden.
Nach der Eintragung der Adresse des Masters kann die Passwortberechnung durch Betätigung des *Start*-Buttons gestartet werden.

4.3.3 Auswertung des Angriffs

Nach dem Start des Angriffs sind mehrere Beobachtungsmöglichkeiten gegeben. Die erste Möglichkeit ist das Beobachten des Logs. Dieses ist erreichbar, indem das Feld *Log* ausgewählt wird.
Das Log stellt Informationen zu verschiedenen Ereignissen bereit. Im aktuellen Beispiel kann aus Abbildung 4.5 entnommen werden, dass die Anwendung gestartet und der MD5-Hash-Algorithmus gewählt wurde. Zudem wird der Hash des eingegebenen Passwortes angegeben. Die Information „showing ChartView“ bedeutet, dass die grafische Auswertung des Angriffs aufgerufen wurde.

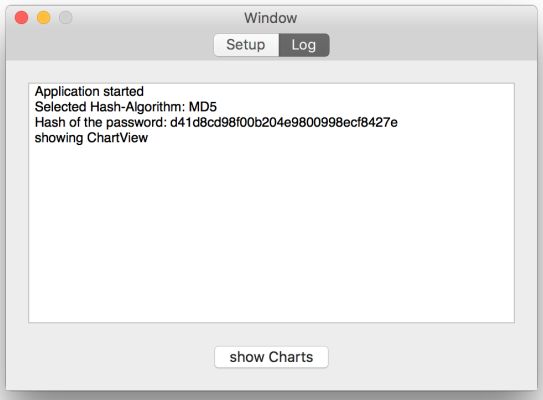


Abbildung 4.5: Log-Ansicht nach dem Start des BruteForce-Angriffs

Die grafische Auswertung des Angriffs ist auf Abbildung 4.6 dargestellt. Die Tabelle stellt in Echtzeit Informationen darüber bereit, wieviele Hashes pro Sekunde durch die jeweiligen Worker erzeugt werden. Auf Grafik 4.6 berechnet aktuell ein Worker 14.012 Hashes pro Sekunde. Die Diagramme zeigen den aktuellen Status der jeweiligen Worker an. Das bedeutet, dass bei Hinzufügen weiterer Worker zum System analog dazu weitere Diagramme angezeigt werden würden.

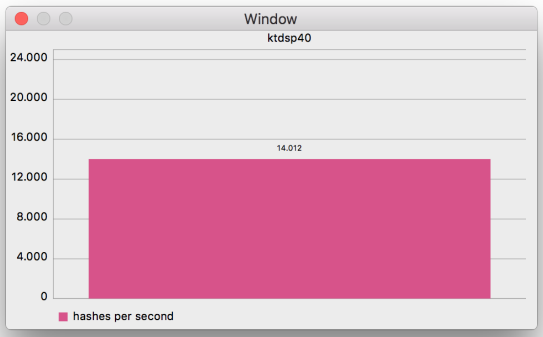


Abbildung 4.6: Grafische Darstellung der Quantität an erzeugten Hashes pro Sekunde

5 Fazit und Ausblick

Das abschließende Kapitel fasst das Projekt zusammen und gibt somit einen Überblick der durchgeführten Tätigkeiten.
Außerdem wird ein Ausblick gegeben, der Ideen zu möglichen Fortsetzungen dieses Projekts bietet.
Abschließend wird das durchgeführte Projekt kritisch reflektiert. Dadurch soll die Einordnung des Gesamtprojekts verbessert werden.

Das verteilte System wird auf der Entwickler-Plattform *Github* zur Verfügung gestellt und ist unter der Adresse `https://github.com/pixelskull/AVS-Project` aufrufbar.

5.1 Zusammenfassung des Projekts

Vor der eigentlichen Durchführung des Projekts wurde eine Literaturrecherche durchgeführt. Die Recherche hatte zwei Ziele: Zum einen wurde Fachliteratur ermittelt, um notwendige Kenntnisse zu verteilten Systemen zu gewinnen. Zum anderen brachte die Recherche den Vorteil, dass der Prozess der Ideenfindung verbessert wurde. Bei der Literaturrecherche stellte sich heraus, dass das Buch [Tanenbaum u. van Steen, 2003] eines der Standardwerke im Bereich der verteilten Systeme darstellt. Aus diesem Grund wurde das Buch primäre Wissensquelle für dieses Projekt ausgewählt.
Nach dem Studium von [Tanenbaum u. van Steen, 2003] wurden in Kapitel 1.3 die Faktoren abgeleitet, die das zu implementierende verteilte System erfüllen soll.
Nachdem die allgemeinen Anforderungen formuliert worden sind, wurden die Hard- und Softwaregrundlagen in Kapitel 2 definiert. In diesem Kapitel wird beispielsweise die zur Verfügung stehende Hardware beschrieben. Damit wird sichergestellt, dass die Reproduzierbarkeit des Projekts gegeben ist.
In Kapitel 3 wurden die theoretischen Überlegungen für die Implementation beschrieben. Dazu gehört die Planung der Architektur oder der Kommunikation. Bei der Konzeption wurde besonderer Wert darauf gelegt, dass die in Kapitel 1.3 zusammengefassten Aspekte bei der Implementation beachtet werden.
Danach werden in Kapitel 4 Details zur Implementation beschrieben. Der Fokus lag dabei nicht auf einer vollständigen Beschreibung des Programmcodes, sondern auf der

Verdeutlichung einiger Besonderheiten der Implementation. Auf die vollständige Beschreibung wurde verzichtet, da diese den Rahmen dieser Dokumentation sprengen würde. Stattdessen wurde der Programmcode kommentiert. Dies hat den Vorteil, dass eine semantische Trennung zwischen Implementationsdetails und Programmcodebeschreibung geschaffen wird. Dadurch wird das Leseverständnis dieser Dokumentation weiter erhöht.

5.2 Ausblick

Verteilte Systeme unterliegen dank ständiger Weiterentwicklung von Konzepten und Techniken einem gewissen Alterungsprozess. Aus diesem Grund bietet auch die implementierte BruteForce-Attacke einige Ansatzpunkte zur Weiterentwicklung.
Eine mögliche Optimierung ist die Anpassung an neue Features der genutzten Programmiersprache *Swift*. Swift ist quelloffen verfügbar und besitzt somit implizit eine große Gruppe an Entwicklern. Die Weiterentwicklung der Programmiersprache kann damit auch Optimierungspotenzial für den Quellcode dieses Projekts bieten.
Ein weiterer Aspekt für eine Fortsetzung des Projekts ist die Nutzung von maschinenbasiertem Lernen. Mit Hilfe von maschinenbasiertem Lernen kann beispielsweise das benutzte Wörterbuch der Dictionary Attack mit weiteren häufig genutzten Passwörtern erweitert werden. Ergänzend könnten Passwörter entfernt werden, welche aktuell seltener eingesetzt werden.

Aber auch der BruteForce-Angriff könnte von maschinenbasiertem Lernen profitieren. Durch eine Analyse, welche der erzeugten Passwörter besonders häufig zum Erfolg führen, kann der Algorithmus diese Passwörter priorisiert einsetzen. Dadurch würde eine Effektivitätssteigerung des Algorithmus erzielt werden.

Auch eine hybride Lösung wäre denkbar. Der Angriff könnte optimiert werden, indem ein Wörterbuch mit den Passwörtern erstellt wird, die bei einem BruteForce-Angriff häufigen Erfolg erzielen. Damit würde sich der Angriff durch maschinenbasiertes Lernen ständig an aktuelle „Trends“ im Bezug auf häufig gewählte Passwörter anpassen können.
Zudem könnte eine automatisierte Auswertung Informationen zur Untersuchung der Angriffsperformance bieten. Dabei könnte beispielsweise untersucht werden, wie stark sich die genutzten Hash-Algorithmen in der Geschwindigkeit unterscheiden. Mit einer Auswertung könnten indirekte Schlüsse auf die Sicherheit bestimmter Passwörter gezogen werden.

5.3 Kritische Würdigung

Primär werden verschiedene Details der Implementierung als verbesserungswürdig angesehen.
Ein Beispiel dazu ist die Nutzung der Methode *.map* bei der Ausgabe von Arrays. Normalerweise wird diese Methode in der funktionalen Programmierung benutzt, um über ein Clousure Werte aus einem Array abzurufen. Bei der Implementierung wurde die Methode zweckentfremdet, um Werte aus einem Array aufzurufen. Die Entscheidung für diese missbräuchliche Nutzung ist gefallen, weil der Zugriff auf Arrays damit performanter erfolgen kann, als es mit alternativen Methoden der Fall wäre.

Weitere Verbesserungsmöglichkeiten sehen die Projektteilnehmer in der Auswahl des Kommunikations-Servers. Der ausgewählte Node-Server liefert zwar die für dieses Projekt notwendige Funktionalität, allerdings läuft dieser nur in einem Thread. Ein multithreadfähiger Kommunikationsserver könnte besonders innerhalb eines verteilten Systems sinnvoll eingesetzt werden. Somit könnte durch Einsatz eines alternativen Servers die Performance des verteilten Systems weiter gesteigert werden.

Als letzten verbesserungswürdigen Punkt wird angemerkt, dass keine Tests implementiert worden sind. Durch automatisierte Tests können Fehler schnell entdeckt und somit die Qualität des Codes verbessert werden. Allerdings ist das Testen eines verteilten Systems schwieriger zu realisieren, als es bei einer lokalen Anwendung der Fall ist. Aus diesem Grund und dem Mangel an Kenntnissen wurde in diesem Projekt auf die Implementation von Tests verzichtet.

Abbildungsverzeichnis

Glossar

BruteForce-Angriff . Entschlüsselung eines unbekannten Passwortes, indem alle möglichen Zeichenkombinationen durchprobiert werden, bis das korrekte Passwort ermittelt ist.

Dictionary-Angriff .. Effiziente Methode zum Entschlüsseln eines unbekannten Passwortes. Basis dazu ist ein Wörterbuch mit häufig benutzten Passwörtern, die nach einer festgelegten Strategie durchprobiert werden.

Master ..... Instanz innerhalb des verteilten Systems, welche für die Verteilung der anfallenden Tätigkeiten verantwortlich ist.

Provider ..... Synonym für den *Master*.

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