
# Verteilte Systeme - RSA Bruteforce

## Projektarbeit

des Studiengangs Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

**5320806, 4776192, 1646552**

17.12.2021

| | |
|---|---|
| **Bearbeitungszeitraum** | Wintersemester 2021/22 |
| **Matrikelnummer, Kurs** | 5320806, 4776192, 1646552, INF19B |
| **Dozent** | Patrick Jungk |

# Inhaltsverzeichnis

# 1 Analyse

Bei dem RSA Verfahren handelt es sich um ein asymmetrisches Verschlüsselungs- und Signaturverfahren. Verschlüsselt wird dabei mit einem öffentlichen Schlüssel, und entschlüsselt mit dem dazugehörigen privaten Schlüssel. Um bei RSA den privaten Schlüssel aus dem öffentlichen Schlüssel zu berechnen, müssen die beiden Primzahlen $p$ und $q$ gefunden werden, welche multipliziert das RSA Modul $n = p * q$ ergeben. Dabei handelt es sich um eine Primfaktorzerlegung. Um die richtigen $p$ und $q$ zu finden, müssen alle Möglichkeiten ausprobiert werden, zwei Primzahlen $p$ und $q$ aus einer bestimmten Liste von Primzahlen $P$ mit der Länge $N$ auszuwählen, eine sog. Exhaustive Key Search oder Bruteforce-Angriff.

## 1.1 Verteilte Aspekte

Da dafür viel Rechenleistung benötigt wird, soll ein verteiltes System erstellt werden, um die Arbeitslast auf mehrere Rechner zu verteilen. Ziel dabei ist es, dass ein Knoten nicht den ganzen Primzahlbereich berechnet, sondern nur ein Subset. Es soll ein Konzept entwickelt werden, wie der Primzahlbereich effizient verteilt werden kann. Zusätzlich soll das verteilte System in der Lage sein, Ausfälle mehrerer Knoten auszugleichen, sodass immer der ganze Primzahlbereich abgedeckt ist. Zum Test des verteilten Systems werden Listen mit möglichen Primzahlen der Größen 100, 1.000, 10.000 und 100.000 und zugehörige öffentliche Schlüssel sowie Chiffretexte bereitgestellt. Für jeden Primzahlbereich sollen gemessen werden, wie lange das verteilte System bestehend aus zwei, fünf und zehn Knoten braucht, um die Primzahlen des öffentlichen Schlüssels zu finden. Es soll eine Prognose gemacht werden, wie lange das verteilte System (mit 2,5,10 Knoten) benötigt, um 1 Billionen Primzahlen zu berechnen. Zusätzlich soll aufgezeigt werden, wie groß das Cluster sein müsste, um 1 Billionen Kombinationen in 1h zu berechnen.

## 1.2 Bruteforceoptimierung

Wenn man wie in Abschnitt 1.1 beschrieben alle Kombinationen von zwei Primzahlen aus $P$ mit der Länge $N$ bildet, so würde man $N^2$ Kombinationen ausprobieren müssen. Da aufgrund der Kommutativität der Multiplikation $n = q * p = p * q$ gilt, kann ca. die Hälfte der Kombinationen ausgelassen werden kann. Dies ist beispielhaft in Abbildung 1.1 zu sehen.

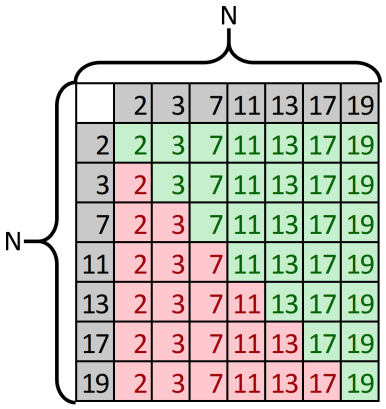


Abbildung 1.1: Kombinationen die bei Bruteforce getestet werden müssen (grün) und die ausgelassen werden können (rot)

Für die erste Primzahl (hier 2) ist es erforderlich, diese mit jeder anderen Primzahl in der Liste zu prüfen. Die zweite Zahl muss jedoch nur noch mit allen danach folgenden Zahlen größer 2 geprüft werden, da die Kombination der ersten beiden in anderer Reihenfolge bereits im ersten Schritt geprüft wurde. Für eine Primzahl $p_i$ müssen also nur noch die Primzahlen $q_i$ geprüft werden, für die $q_i \geq p_i$ gilt. Somit nimmt die Anzahl der notwendigen Überprüfungen mit zunehmendem Index der Primzahl linear ab.

# 2 Grobkonzept

## 2.1 Architektur

Bei der Architektur des Clusters wurde ein dezentraler Ansatz ohne Koordinator verfolgt. Das heißt, der Cluster besteht aus verschiedenen `Workern`, welche alle die gleichen Rechte und Aufgaben haben. Es handelt sich hierbei um einen nachrichtenbasiertes Cluster, da Informationen zwischen den `Workern` mit Nachrichten ausgetauscht werden. Um Synchronität zu erlangen, wird ein verteilter Algorithmus verwendet. Auf diesen wird später eingegangen. Ein `Worker` ist dabei als Thread realisiert und verwaltet im wesentlichen die Kommunikation mit anderen `Workern`, die Berechnung der Primzahlen sowie die Logik des Protokolls. Die Kommunikation der `Worker` untereinander ist mithilfe von Sockets implementiert. Jeder `Worker` verwaltet dafür einen Thread, indem ein `ServerSocket` auf einem spezifizierten Port neue Socketverbindungen annimmt. Zusätzlich verwaltet der `Worker` einen Thread zur Berechnung der Primzahlen, damit dieser trotz laufender Berechnungen auf Nachrichten antwortet. Dabei ist jeder `Worker` mit jedem `Worker` verbunden, es handelt sich um ein vollvermaschtes Netzwerk. Der konzeptuelle Aufbau ist in Abbildung 2.1 zu sehen. Der Client ist dabei ein Thread, der eine Verbindung mit einem `Worker` eingeht, und diesem eine Anfrage mit einem öffentlichen RSA Schlüssel schickt.

Der technische Aufbau des Clusters ist in Abbildung 2.2 zu sehen. Da ein `Worker` insgesamt nur drei Threads benötigt, können mehrere auf einem Rechner gleichzeitig laufen, mit besserer Hardware wird so eine vertikale Skalierung ermöglicht. Die `Worker` können jedoch auch auf mehreren Rechnern im gleichen Netz verteilt werden, um eine horizontale Skalierung zu ermöglichen.

### 2.1.1 Kommunikationskonzept

Jeder `Worker` muss mit jedem `Worker` aus dem Cluster kommunizieren können, um die zu bearbeitenden Segmente auszuhandeln. Jeder `Worker` baut also eine Socketverbindung zu allen `Workern` des Clusters auf. Alle für eine Verbindung relevanten Informationen

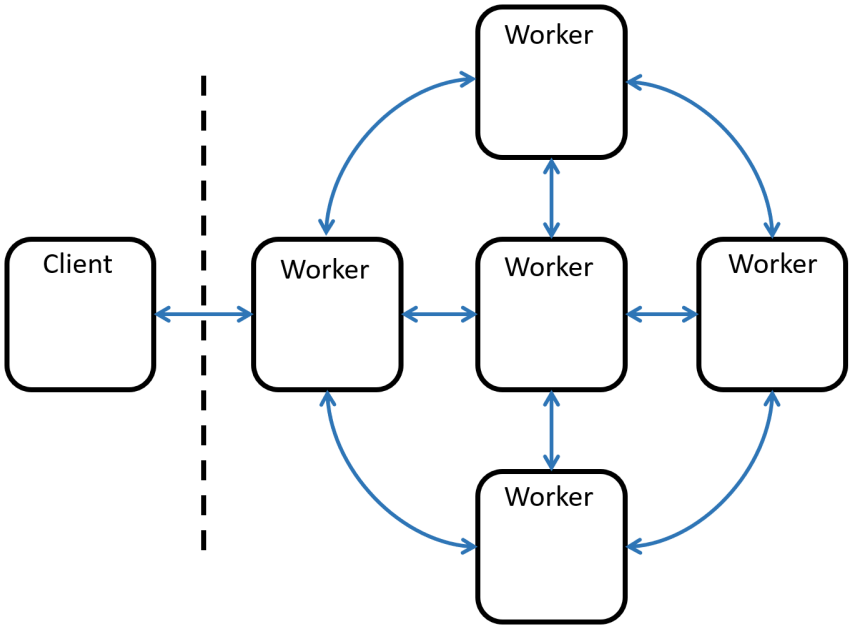


Abbildung 2.1: Konzeptioneller Aufbau des Clusters

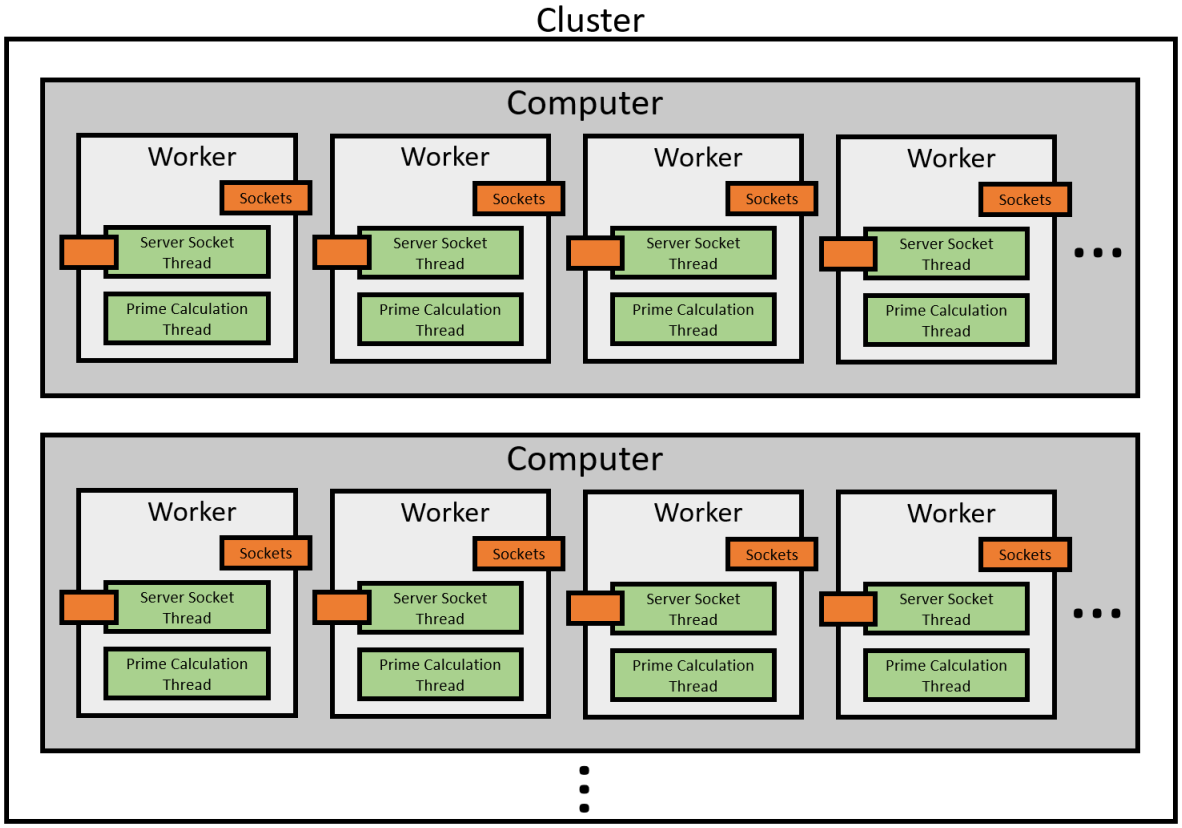


Abbildung 2.2: Schematischer Aufbau des Clusters

werden dabei in einem `Connection`-Objekt persistiert. Dieses enthält z.B. den Socket, ObjectInput/OutputStreams, die Rolle des Kommunikationsteilnehmers (Unknown, Client, Worker) sowie Funktionen für das Lesen und Schreiben von Nachrichten an den Socket. Um die Anzahl an verwendeten Threads zu begrenzen, wird nicht pro Verbindung mit einem Worker ein Thread erzeugt, alle Verbindungen zum Cluster werden sequentiell auf neue Nachrichten überprüft. Problematisch ist, dass die `read()`-Methode bei Socketstreams beim Aufruf blockiert, bis eine Nachricht gelesen wird. Mithilfe der `available()`-Methode wird geprüft, ob sich eine Nachricht im Puffer befindet.

Der ServerSocket, der im `ConnectionHandler` läuft, fügt beim Aufbau einer neuen Verbindung ein `Connection`-Object in die Liste `connections` eines `Workers` hinzu. Da diese alle Verbindungen verwaltet und mehrere Threads auf sie zugreifen, wurde eine `CopyOnWriteArrayList` verwendet um die Thread-Savety zu gewährleisten.

Ein Worker kann sequentiell über alle Verbindungen iterieren und wenn eine Nachricht ankommt wird diese gelesen und verarbeitet, ohne das der `Worker`-Thread blockiert. Nachrichten sind dabei `Message`-Objekte, die für die Übertragung serialisiert/deserialisiert werden. Sie haben einen Typ und ein frei wählbares Payload. Auf die verschiedenen Nachrichtentypen wird im Folgenden eingegangen.

### 2.1.2 Nachrichtentypen

Eine Nachricht hat dabei einen der folgenden aufgelisteten Nachrichtentypen, die mithilfe des Enums `MessageType` implementiert sind. Nachrichten enthalten basierend auf ihrem Typ verschiedene Payloads und werden unterschiedlich verarbeitet.

- `JOIN`: Initialnachricht, die von Worker während des Verbindungsaufbaus zum Cluster geschickt wird. Der Worker erwartet als Antwort eine `CLUSTER_INFO` Nachricht, um sich mit allen anderen zu verbinden.
- `CLUSTER_INFO`: Enthält eine Liste mit IP Adressen und Verbindungsports jedes Workers im Cluster als Payload.
- `CONNECT_CLUSTER`: Nachdem ein neuer Worker `CLUSTER_INFO` bekommen hat, baue Verbindung mit jedem Worker auf und schicke `CONNECT_CLUSTER` Nachricht mit. Diese enthält den eigenen Verbindungsport und schließt Verbindungsaufbau ab.
- `RSA`: Public Key wird von Client an den Cluster gesendet.

- `START`: Verteile Public Key im Cluster und starte Berechnung. Enthält Public key als Payload.
- `FREE`: Anfrage an alle Worker, ob Segment frei ist oder bereits bearbeitet wird. Enthält Segmentnummer als Payload.
- `OK`: Antwort auf `FREE`, Segment wird nicht bearbeitet. Enthält Segmentnummer als Payload.
- `NOK`: Antwort auf `FREE`, Segment wird bereits bearbeitet. Enthält Segmentnummer als Payload.
- `FINISHED`: Teile allen Workern mit, dass Segment fertig bearbeitet wurde. Enthält Segmentnummer als Payload.
- `DEAD_NODE`: Teile allen Workern mit, dass Worker nicht mehr antwortet. Enthält IP Adresse und Verbindungsport des Workers.

Auf den genauen Zusammenhang der einzelnen Nachrichten wird im folgenden eingegangen.

### 2.1.3 Aufgabenverteilung

Jeder `Worker` besitzt eine vollständige Liste (`primes`) aller Primzahlen $P$, die er zu Beginn aus der Textdatei einliest. Um diese Primfaktorzerlegung auf mehrere Clusterknoten zu verteilen, muss die Liste an Primzahlen $P$ in mehrere Segmente aufgeteilt werden, sodass jeder Knoten unabhängig von anderen Knoten Segmente berechnen kann. Eine Möglichkeit dafür wäre, dass man den Primzahlbereich $P$ in Segmente gleicher Größe $g$ unterteilt. Jede Primzahl eines Segments $[q_i, ..., q_{i+g}]$ muss dann mit allen Primzahlen $p_i$ aus $P$ durchprobiert werden. Wenn man das mit der in Abschnitt 1.2 vorgestellten Optimierung kombinieren will, so sieht man, dass bei Segmenten die mit höheren Primzahlen beginnen, immer weniger Überprüfungen durchgeführt werden müssen. Jedoch sollte ein Segment immer einen ähnlichen Berechnungaufwand haben, unabhängig vom Start des Segments. Zum einen kann mit einem konstanten Berechnungsaufwand besser der Overhead gesteuert werden, welcher beim Cluster durch die Kommunikation entsteht. Für den Cluster ist es am effizientesten, wenn er so viel wie möglich rechnet und wenig kommuniziert. Wenn vor und nach der Bearbeitung eines Segments kommuniziert wird, dann wird mit zunehmender Segmentzahl die Kommunikationsfrequenz und der damit verbundene Overhead aufgrund des sinkenden Berechnungsaufwandes immer größer. Damit jedes Segment also immer etwa

gleich viele Berechnungen mit sich bringt, muss die Größe eines Segments bei steigendem Index der Primzahl dynamisch vergrößert werden.

Aus der vorgeschlagenen Optimierung wird ersichtlich, dass eine Primzahl mit dem Index $i$ mit $P(i) = N - i$ Primzahlen kombiniert werden muss. Es ist nun also eine Funktion $S(i,c)$ gesucht, welche für einen Primzahlindex $i$ die Länge des Segments $n$ berechnet, ab der die Anzahl an Berechnungen/Kombinationen erstmals größer als $c$ sind:

$$S(i,c) := n,\ n \text{ ist kleinste Ganzzahl, für die gilt: } \sum_{k=0}^{n-1} P(i+k) \geq c$$

Wie im folgenden zu sehen ist, kann diese Bedingung umgeformt werden, um $n$ zu isolieren.

$$\sum_{k=0}^{n-1} N - (i+k) = \sum_{k=0}^{n-1} N - i - k = \sum_{k=0}^{n-1} (N-i) - \sum_{k=0}^{n-1} k = n(N-i) - \frac{n(n-1)}{2} \geq c$$

Die Suche nach der Segmentgröße $n$ reduziert sich so auf die Lösung der folgenden quadratischen Gleichung, die mit der Mitternachtsformel gelöst werden kann.

$$-\frac{1}{2}n^2 + n(N - i + \frac{1}{2}) - c \geq 0$$

$$n_{1/2} = \frac{-b \pm \sqrt{b^2 - 4(-\frac{1}{2})(-c)}}{2 - \frac{1}{2}} = b \pm \sqrt{b^2 - 2c}$$

Für einen Primzahlindex $i$, die Anzahl der Primzahlen $N$ und die Mindestanzahl an Berechnungen $c$ ergibt sich die Segmentgröße über die Funktion $S$:

$$S(N,c,i) = \left\lceil b(N,i) - \sqrt{b(N,i)^2 - 2c} \right\rceil$$

wobei

$$b(N,i) = N - i + \frac{1}{2}$$

Das erste Fenster berechnet sich durch Errechnen von $S$ mit $i = 0$. Die berechnete Fenstergröße ist anschließend das neue $i$. Danach wird $i$ auf die Summe der bisher berechneten Fenstergrößen gesetzt. Das setzt sich fort, bis $i > N$. Dann wird das letzte Fenster so zugeschnitten, dass es mit der Liste an Primzahlen abschließt. Diese Berechnungen werden auf jedem `Worker` ausgeführt, sodass ein Segment nur mit seinem Index bestimmt werden muss.

Die `Worker`-Klasse verwaltet die hier genannten Informationen. Um festzuhalten, welche Segmente bereits erledigt wurden, wird eine Bitmap geführt (`calculatedSegments`). Zunächst ist diese mit einer 0 für jedes Segment, dass sich bei der obigen Berechnungsvorschrift ergeben hat, gefüllt. Anschließend wird bei jeder Beendigung eines Paketes das entsprechende Bit auf 1 gesetzt. Zusätzlich gibt es eine Liste, die jedem Segmentindex den Primzahlindex zuordnet, bei dem mit der Berechnung begonnen werden muss (`segmentStartIndex`) sowie eine Liste, welche die berechneten Segmentgrößen speichert (`segmentSizes`).

## 2.2 Verbindungsaufbau

Um das Cluster zu starten, wird zunächst ein Worker gestartet. Sollen weitere Worker in das Cluster kommen, so müssen diese einen Handshake mit einem sich bereits im Cluster befindenden Worker durchführen. Als erstes wird eine Socketverbindung mit einem Worker gestartet. Dazu verbindet man sich mit dem Verbindungport (also der Port des Serversockets) und der IP eines Workers. Danach wird ein Handshake durchgeführt, der in Abbildung 2.3 zu sehen ist. Dabei wurden die Socketverbindungen nicht dargestellt.

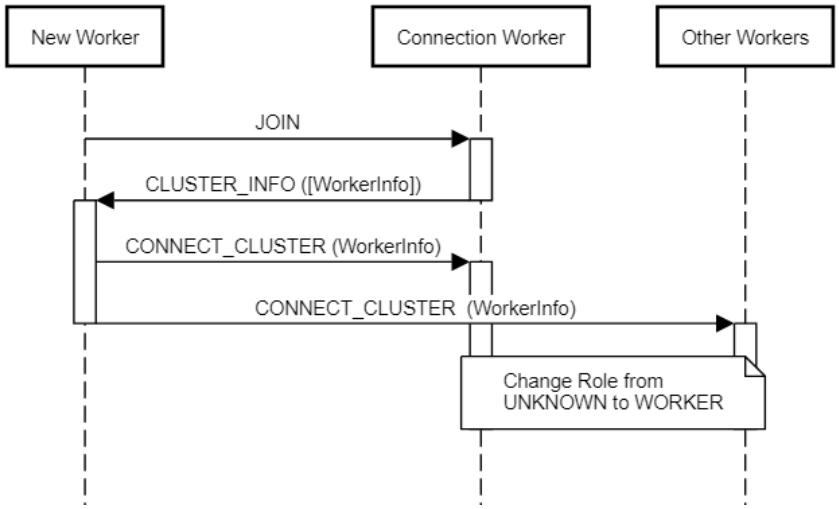


Abbildung 2.3: Verbindungsaufbau zum Cluster

Vom neuen Worker wird als erstes eine `JOIN` Nachricht gesendet. Diese fordert Verbindungsinformationen(Port, Adresse) für alle weiteren `Worker` im Cluster an, worauf mit einer `CLUSTER_INFO` Nachricht geantwortet wird. Zu allen Workern, die sich darin befinden, wird eine Socketverbindung aufgebaut. Jeder Worker hat ein `Connection`-Objekt für die neue Verbindung angelegt. Um den Verbindungport des neuen Workers einzutragen und den Verbindungsaufbau abzuschließen, schickt der neue Worker eine `CONNECT_CLUSTER` Nachricht an alle, die den Port seines Serversockets enthält. Das ist nötig, damit diese Information

weiteren neuen Workern weitergegeben werden kann. Ist der Handshake korrekt abgelaufen, so setzt jeder Worker die Rolle des neuen Workes von `UNKNOWN` auf `WORKER`, und sein Verbindungsaufbau ist abgeschlossen.

## 2.3 Berechnungsphase

### 2.3.1 Startphase

Die Berechnungsphase beginnt mit dem Verbindungsaufbau eines Clients zu einem `Worker` . Der Client schickt direkt nach Verbindungsaufbau eine `RSA`-Nachricht. Diese enthält ein `RSAPayload` mit dem Public Key. Um die Verbindung zum Client von den `Workern` zu unterscheiden, wird bei Erhalt der `RSA`-Nachricht die Rolle des Clients auf `Client` gesetzt. Zusätzlich wird der öffentliche Schlüssel mit der `START`-Nachricht im Cluster verteilt. Der öffentliche Schlüssel wird gespeichert und die Berechnung begonnen.

### 2.3.2 Aufgabenverteilung

Wie bereits in Unterabschnitt 2.1.3 erklärt wurde, wird zur Aufgabenverteilung der Primzahlbereich in Segmente unterteilt. Nun gilt es auszuhandeln, welcher `Worker` welches Segment bearbeitet. Damit die Lösung sicher und effizient gefunden wird, müssen alle Segmente genau einmal berechnet werden. Um das zu gewährleisten, wird die Berechnung eines Segments als verteilter wechselseitiger Zugriff auf ein Segment mit einer ID interpretiert. Dabei ist die gemeinsam verwaltete Ressource eine Liste mit berechneten und nicht berechneten Segmenten (`calculatedSegments`). Zum wechselseitigen Aufschluss wurde ein dezentraler Algorithmus gewählt, alle `Worker` bestimmen, ob ein `Worker` Zugriff auf ein Segment bekommt.

Zu Beginn einer Berechnung sucht sich ein `Worker` mit der Funktion `selectPrimeRange()` ein zufälliges noch nicht berechnetes Segment aus `calculatedSegments` aus. Da soll die Wahrscheinlichkeit minimieren, dass sich zwei `Worker` das gleiche Segment aussuchen. Nun wird an alle `Worker` eine `FREE(ID)`-Nachricht geschickt, die die ID des Segments als Payload enthält. Wird das Segment gerade berechnet, so wird mit `NOK(ID)` geantwortet, anderenfalls mit `OK(ID)`. Erhält ein `Worker` so viele `OK`‘s wie es `Worker` im Cluster gibt, so beginnt er die Berechnung. Andernfalls fragt er wieder ein zufälliges Segment an. Dieser Ablauf ist beispielhaft in Abbildung 2.4 zwischen zwei Workern dargestellt.

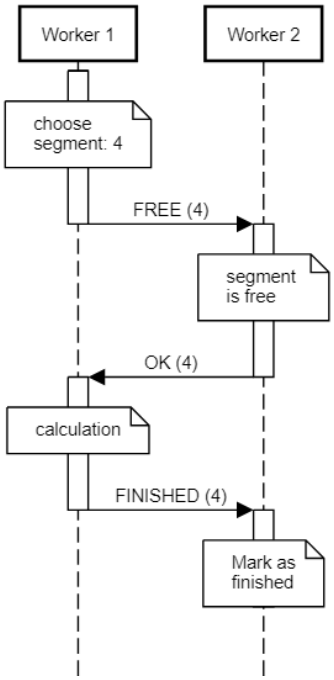


Abbildung 2.4: Sequenzdiagramm von wechselseitigem Aufschluss bei der Aufgabenverteilung

Die Daten über die bereits berechneten Segmente `calculatedSegments` sind auf N Workern komplett repliziert. Gründe für die vollständige Replikation sind:

- Zuverlässigkeit bei Verlust eines Workers
- Leistung durch lokales Vorhandensein
- Geringe Größe der Liste

Um die Datenkonsistenz im System zu erhalten, müssen die Daten synchronisiert werden. Dies ist der Fall, wenn ein Worker schreibend auf die Daten zugreift, also wenn er das Beenden der Berechnung eines Segmentes markiert. Wie in Abbildung 2.4 zu sehen ist, broadcasted dieser dann eine `FINISHED(ID)`-Nachricht mit der ID des berechneten Segments, was von allen anderen `Workern` festgeschrieben wird. Fällt ein `Worker` während der Berechnung aus, so wird für sein Segment keine `FINISHED(ID)`-Nachricht gesendet und es kann von einem anderen `Worker` berechnet werden.

### 2.3.3 Endphase

Findet ein `Worker` die Lösung, dann broadcasted er eine `ANSWER_FOUND`-Nachricht an alle. Diese enthält ein `PrimeCalculationResult`-Payload mit den berechneten Primzahlen $p$ und $q$. Bei Erhalt dieser Nachricht bricht ein `Worker` seine laufende Berechnung ab. Ist er mit

dem Client verbunden, so leitet er ihm sie weiter. Der Client entschlüsselt den Chiffretext und gibt diesen sowie die benötigte Zeit aus.

## 2.4 Ausfallsicherheit

### 2.4.1 Ausfall eines Workers

Die allgemeine Vorgehensweise bei Ausfall eines `Workers` ist der Ausschluss des `Workers` aus dem Informationsfluss, sobald der Ausfall bemerkt wurde. Es wird also kein Rejoin versucht, da davon ausgegangen wird, dass der `Worker` nicht mehr funktioniert. Dies wäre beispielsweise bei Abbruch des Programms auf einem anderen Rechner der Fall. Dazu wird beim Versuch, ein beliebiges `Message`-Objekt an einen anderen Worker zu versenden, überprüft, ob durch den dafür verwendeten OutputStream eine Exception geworfen wird. Im Anschluss wird in dem mit dem ausgefallenen `Worker` asssoziierten `Connection`-Objekt markiert, dass es sich bei dieser Verbindung um eine fehlerhafte handelt. Dies hat zur Folge, dass diese `Connection` aus der eigenen Liste entfernt wird. Es wäre nun prinzipiell möglich, den sonstigen Ablauf weiterzuverfolgen, da mit diesem Mechanismus jeder `Worker` beim Versuch, mit dem ausgefallenen `Worker` zu kommunizieren, die damit assoziierte `Connection` entfernen würde. Wie jedoch im Unterabschnitt 2.1.3 erwähnt, ist es zielführend, die Zeit, welche jeder `Worker` mit Kommunikationen verbringt, möglichst gering zu halten. Andernfalls würde jeder `Worker` für sich selbst herausfinden müssen, dass ein `Worker` ausgefallen ist, obwohl diese Information bereits einem `Worker` bekannt ist. Dies wäre zusätzliche Zeit, welche nicht für weitere Berechnungen genutzt wird. Aus diesem Grund broadcasted der erste `Worker` eine `DEAD_NODE`-Nachricht, deren Payload Informationen zur assoziierten `Connection`, also Port und IP-Adresse des ausgefallenen `Workers`, beinhaltet. Erhält ein `Worker` eine solche Nachricht, entfernt dieser die `Connection` auch aus seiner eigenen Liste, wodurch er zu einem späteren Zeitpunkt keine weitere Zeit durch eine fehlerhafte Kommunikation mit dem ausgefallenen `Worker` verliert. Dieser Mechanismus wird in Abbildung 2.5 dargestellt.

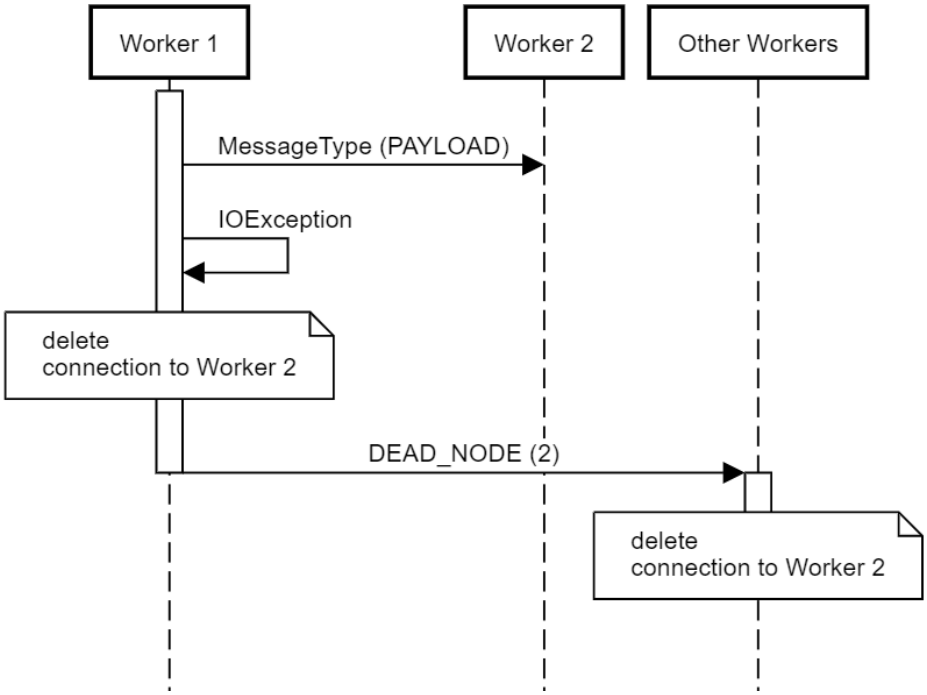


Abbildung 2.5: Sequenzdiagramm von Handhabung eines Ausfalls

### 2.4.2 Reconnect von Client

Da der Client über einen `Worker` mit dem Cluster verbunden ist (Abbildung 2.1), kann der Ausfall eines `Workers` zur Folge haben, dass der Client nicht mehr mit dem Cluster kommunizieren kann. Um sicher zu sein, dass eine Verbindung zum Cluster weiterhin besteht, sendet der Client regelmäßig eine Herzschlag-`Message` des Typs `CLUSTER_INFO`. Antwortet der `Worker` wie erwartet, ist davon auszugehen, dass er lediglich längere Zeit keine Nachricht entweder gebroadcasted oder direkt an den Client gesendet hat. Wird jedoch eine Exception geworfen, führt der Client einen Reconnect durch, bei welchem er iterativ versucht, sich mit einem der übriggebliebenen `Worker` zu verbinden. Dies geschieht nach dem gleichen Ablauf wie jener, der in Abschnitt 2.1 beschrieben ist.

## 2.5 Verteilte Aspekte und Umsetzung

### 2.5.1 Synchronisation

Das Wissen, welche Primzahlsegmente bereits abgearbeitet wurden, wird in allen `Workern` synchron gehalten.

Das geschieht über die `FINISHED`-Messages, welche signalisieren, dass der gesendete Primzahlbereich abgearbeitet wurde.

### 2.5.2 Nebenläufigkeit

Die Nebenläufigkeit wird dadurch umgesetzt, dass jeder Worker stets nur für einen kleinen Teil der gesamten Primzahlen zuständig ist und jeder Worker aktiv nach neuen Aufgabe sucht. Somit ist sichergestellt, dass, wenn genug Segmente verfügbar sind, auch jeder Worker eine Aufgabe hat. Besteht jeder Worker aus drei Threads, die miteinander kommunizieren.

### 2.5.3 Kommunikation

Die Kommunikation der einzelnen Worker findet über ein definiertes `Message`-Objekt mit verschiedenen Typen und in Zusammenhang damit verschiedenen `Payloads` statt. Diese `Message`-Objekt werden serialisiert und über Socketverbindungen versendet beziehungsweise empfangen.

## 2.6 Beschreibung der Anpassung

Als alternatives Konzept für die Kommunikation wurde der Ansatz verfolgt, dass jede Verbindung zwischen zwei `Worker` einen eigenen Thread bekommt. Das wurde aber schnell wieder verworfen, da `Worker` so zu viele Threads geöffnet wurden, und das System zu langsam wurde. Wenn z.B. pro Rechner fünf `Worker` gestartet wurden, die für jeden Knoten in einem vollvermaschten Netzwerk einen Thread geöffnet haben, so wurde dass schnell zu viel und war nicht skalierbar.

# 3 Testplan

Ein verteiltes System, welches zudem mit nebenläufiger Programmierung arbeitet, ist sehr fehleranfällig. Um diese Fehler bei der Entwicklung zu entdecken und das System robuster zu machen, muss ausgiebig getestet werden. Dabei wurden folgende Tests durchgeführt:

- Zum Erleichtern der Tests wurde ein Loggingsystem eingerichtet. Dafür wurde die Klasse `Logger` geschrieben. Mit der statischen Methode `Logger.log()` kann ein beliebiger String in die Konsole ausgegeben werden. Beim erzeugen jedes `Worker`-Thread zu Beginn des Programms werden diese mit `Thread.setName()` zu *Worker1*, *Worker2* usw. umbenannt. So kann man bei den Loggingausgaben den Überblick behalten, aus welchem Thread sie kommen.

- Zudem sollte getestet werden, ob sich ein neuer `Worker` zum Cluster verbinden kann. Dazu wurde eine Testumgebung mit vier `Workern` geschaffen, zu der sich dann ein neuer `Worker` verbinden sollte. Ziel ist es, dass der neue `Worker` nach dem Test mit allen eine Verbindung aufgebaut hat und der neue `Worker` bereit zur Berechnung ist. Dieses Ergebnis wurde erreicht.

- In der `Main` Klasse wurde getestet, ob die Applikation in mehreren Threads lauffähig ist. Dies gilt als Vorstufe zum Testen mit mehreren Geräten über das Netzwerk. Über localhost konnten alle Funktionen des Clusters verwendet werden.

- Danach wurde getestet, ob die Applikation verteilt auf mehrere Rechner im lokalen Netzwerk läuft. Dazu wurde sie auf zwei Rechnern gestartet, wobei der Fokus dabei auf der Kommunikation im Netzwerk lag. Ziel des Tests war, das eine Socketverbindung aufgebaut werden kann, vor allem ob dies im Bezug auf die Firewall ohne weiteres möglich ist. Zum Test wurden jeweils zwei `Worker` auf zwei Rechnern verteilt, wobei die IP Adresse des anderen Rechners zur Verbindung angegeben wurde. Ein Verbindungsaufbau war ohne Änderung der Firewalleinstellungen möglich.

- Um die wichtigste Methode der Kommunikation zu testen, die `broadcast()`-Methode, wurde das gleiche Setup wie in 2. verwendet. Ziel des Tests war es, dass eine `FREE`-Nachricht gebroadcastet wird, und alle im System darauf mit `OK` oder `NOK` antworten. Dieser Ergebnis wurde erreicht.

- Es wurde getestet, ob trotz eines Ausfalls im System der ganze Primzahlbereich abgearbeitet wird und keine Lücken entstehen. Dafür wurde temporär ein Stück Code in die Klasse zur Berechnung der Primzahlkombinationen (`PrimeCalculation`) eingfügt, dass den Worker herunterfährt, der das Segment mit der Lösung bearbeitet. Das Ergebnis des Tests war, dass trotz Ausfall das Segment von einem anderen `Worker` bearbeitet wurde und so die Lösung gefunden werden konnte.

- Es wurde getestet, ob der Cluster das Abbrechen der Netzwerkverbindung zwischen zwei Knoten aushält. Dafür wurde das Programm auf zwei Rechnern getestet, und das Ethernet Kabel ausgesteckt. Ziel war es, dass der Cluster trotzdem noch funktionstüchtig ist und die Lösung findet, in dieser Hinsicht wurde der Test bestanden. Jedoch kam es hier zu einem *Split-Brain*, das heißt beide Teile des Clusters haben autark weitergearbeitet.

- Eine weiterer Fehlerfall ist, dass der `Worker` ausfällt, mit dem der Client verbunden ist. In diesem Fall soll sich der Client zu einem anderen `Worker` im Netzwerk verbinden und die Berechnung soll nicht unterbrochen werden. Um dies zu testen, wurde der `Worker` heruntergefahren, mit dem der Client verbunden ist. Der Client hat sich ordnungsgemäß mit einem anderen `Worker` verbunden und bekommt wie erwartet die Lösung.

# 4 Performanceauswertung

## 4.1 Performancetest

In einem realen Szenario ist der Bereich $P$ der möglichen Primzahlen nicht bekannt. Um abschätzen zu können, wie lange es dauert, um eine Billionen Primzahlen zu berechnen, werden die gegebenen Bereiche von 100, 1.000, 10.000 und 100.000 Primzahlen mit verschiedenen Clustergrößen durch gerechnet. Dabei sollen je Cluster mit 2, 5 und 10 Workern verwendet werden. Zusätzlich wird noch ein Cluster der Größe 20 betrachtet, der jedoch nicht in die Prognose einfließt. Um die Genauigkeit der Vorhersage zu erhöhen, wird jedes Szenario drei Mal durchgeführt. Somit werden Schwankungen ausgeglichen. Solche Schwankungen können etwa durch erhöhte Aktivität des jeweiligen Computers unabhängig der Primzahlberechnung oder durch kurzzeitige Einbrüche des Netzwerkes hervorgerufen werden.

Natürlich hängt die Geschwindigkeit der Berechnung auch wesentlich von dem Rechner ab, auf dem sie durchgeführt wird. Folgende Rechner wurden zur Erprobung verwendet:

- Desktop PC
  AMD Ryzen 7 3700X 8C/16TH @ 4.2 GHz
  Windows 10

- Microsoft Surface Pro 6
  Intel Core i5-8250U 4C/8TH @ 3.2 Ghz
  Windows 11

Dabei wurde der Client immer auf dem Desktop PC gestartet, da dieser ressourcenstärker ist. Im Falle von 5 Workern wurden 3 davon auf dem Desktop PC gestartet und 2 auf dem Surface. Da das verteilte System zufällig Segmente zum Bearbeiten auswählt, ist die tatsächliche Bearbeitungszeit bei mehrmaliger Ausführung unterschiedlich. Da jedoch alle Segmente ca. die gleiche Anzahl an Berechnungen benötigen, kann mit dem

Anteil der berechneten Segmente sowie der dafür benötigten Zeit die Dauer für jeden Anteil an berechneten Segmente berechnet werden. Da das System zufällig Segmente auswählt, müssen im Mittel 50% der Segmente berechnet werden, um die Lösung zu finden. Um die Messwerte vergleichbar zu machen, wurde die Berechnungsdauer für 50% des Primzahlbereichs hochgerechnet und der Durchschnitt aller drei Durchläufe berechnet. Die Ergebnisse sind in folgender Tabelle zu sehen:

| **Primzahlanzahl** | **2 Nodes** | **5 Nodes** | **10 Nodes** | **20 Nodes** |
|---|---|---|---|---|
| 100 | 0.60 s | 0.54 s | 0.65 s | 0.75 s |
| 1.000 | 3.75 s | 2.08 s | 1.67 s | 1.04 s |
| 10.000 | 00:03:35 h | 00:01:41 h | 00:01:13 h | 00:01:05 h |
| 100.000 | 03:26:45 h | 02:20:12 h | 01:46:21 h | 01:26:35 h |

Hierbei ist ein Trend zu erkennen, welcher zeigt, dass die Erhöhung der Clustergröße zu einer Erhöhung der Rechengeschwindigkeit führt. Dieser ist wesentlich stärker ausgeprägt für die großen Primzahlbereiche wie 10.000 und 100.000. In den kleinen Primzahlbereichen ist die Performanceverbesserung zwar merklich, bei 100 Primzahlen scheint aber der Overhead der Kommunikation zu dominieren.

## 4.2 Performanceprognose

Bei einer Erhöhung der möglichen Primzahlen um das 10-fache erhöht sich die Anzahl der nötigen Berechnungen etwa um das 100-fache. Aufgrund der Optimierung halbiert sich dieser Wert auf das 50-fache. Da eine Billion das Hundertmillionen-fache von 10.000 ist, sollte die Berechnung also etwa Zehnmilliarden mal so lange dauern.

### 4.2.1 Dauer für eine Billion Primzahlen

Es soll vorhergesagt werden, wie lange ein Cluster der Größe 2,5 und 10 für eine Billionen ($10^{12}$) Primzahlen brauchen würde. Für jede Clustergröße wird eine lineare Regression durchgeführt, die entstehenden Regressionsgeraden sollen die Anzahl der Primzahlen (x) auf die Berechnungsdauer (y) in Sekunden (Spalte der jeweiligen Clustergröße) abbilden. Die Gleichung der Regressionsgerade sowie eine Vorhersage für die Berechnungsdauer von $10^{12}$ Primzahlen in Sekunden und in Jahren ist in der folgenden Tabelle dargestellt. Dabei handelt es sich um die erwartete Berechnungsdauer, die tatsächliche kann bis zu doppelt so groß sein.

| Anzahl Nodes | Regressionsgerade | Vorhersage |
|---|---|---|
| 2 | $y = 0.1271549x - 375$ | 127154901849 s $\approx$ 4032 Jahre |
| 5 | $y = 0.0863396x - 269$ | 86339682271 s $\approx$ 2737 Jahre |
| 10 | $y = 0.0655018x - 205$ | 65501618528 s $\approx$ 2077 Jahre |

### 4.2.2 Benötigte Clustergröße für eine Stunde Berechnungszeit

Zudem soll berechnet werden, wie groß der Cluster sein müsste, um eine Billionen Primzahlen in einer Stunde zu berechnen. Zunächst wurde dafür ein Curve-Fitting auf der in Unterabschnitt 4.2.1 gezeigten Tabelle ausgeführt, die einen Zusammenhang zwischen der Clustergröße und der Berechnungsdauer in Sekunden für eine Billion Primzahlen herstellt. Es hat sich gezeigt, dass sich die Funktion $t = as^b$ dafür sehr gut eignet, mit einem $R^2$-Wert von 0.9998. Durch Bestimmung der Parameter a und b ergibt sich die Funktion:

$$t = 169342300000s^{-0.415}$$

Dabei ist t die Berechnungszeit in Sekunden und s die Größe des Clusters. Setzt man nun $t = 3600$, also eine Stunde in die Gleichung ein, so ergibt sich nach Umformung eine Clustergröße $s$ von ca. $3.0729 * 10^{18}$ Workern. Dieser Wert ist aufgrund der geringen Anzahl an Datenpunkten fehlerbehaftet, er sollte jedoch eine hinreichende Einschätzung für die Größenordnung der benötigten Knoten liefern.

# 5 Installation

Dieses Kapitel beschreibt das Vorgehen zur Verwendung des Programms.

1. Installiere OpenJDK 16. Eine Anleitung dazu findet sich zum Beispiel hier.
2. Klone das GitHub-Repository.. Alle erforderlichen Dateien zum Ausführen finden sich im Ordner *toPi*. Falls das Repository direkt auf dem Raspberry Pi geklont wurde, muss in diesen Ordner gewechselt werden. Falls es zunächst auf einem anderen Rechner geklont wurde, reicht es diesen Ordner z.B. per *scp* an den Raspberry Pi zu schicken. Auch dann muss in den Ordner gewechselt werden.
3. Nun muss die Konfigurationsdatei *VerteiltesKnacken.conf* angepasst werden. Sie besitzt folgende Struktur.

```
client=no
workerThreads=2
myPort=25000
connectionPort=25000
connectionAddress=localhost
primeRange=1000
```

Dabei haben die einzelnen Parameter folgende Bedeutungen:

**client**
mögliche Werte: yes no
Hiermit wird angegeben, ob auf dem Host ein Client erstellt werden soll, oder nicht. Dabei sollte im Cluster immer nur ein Client gestartet werden.

**workerThreads**
Hiermit wird angegeben, wie viele `Worker` auf dem Host erstellt werden soll. Dabei muss mindestens einer angegeben werden. Falls der aktuelle Host der einzige Host im Cluster ist, müssen mindestens zwei angegeben werden.

**myPort**
Hiermit wird angegeben, auf welchem Port des Hosts der erste erstellte `Worker` seine Socketverbindung öffnen soll. Die darauffolgenden `Worker` erhöhen ihre Portnummer jeweils um eins.

**connectionPort**
Hiermit wird angegeben, mit welchem Port sich die `Worker` verbinden sollen. Jeder `Worker` und auch der Client werden versuchen, sich zunächst mit diesem Port auf der *connectionAddres* zu verbinden. Soll nur ein Host verwendet werden, ist *connectionPort* gleich *myPort* zu setzen.

**connectionAddress**
Hiermit wird angegeben, mit welchem anderen Host sich die `Worker` verbinden sollen. Das kann entweder eine IP-Adresse oder auch ein Hostname sein. Sollen sich die `Worker` mit einem anderen `Worker` auf dem eigenen Host verbinden, kann man hier *localhost* eintragen.

**primeRange**
mögliche Werte: 100 1000 10000 100000
Hiermit wird angegeben, welches Primzahlpaket verwendet werden soll. Chiffrat und Public-Key werden dann automatisch an den angegebenen Bereich angepasst. Jede Konfigurationsdatei auf jedem Host sollte die gleiche *primeRange* besitzen.

4. Anschließend kann das Programm mit *java –jar VerteiltesKnacken.jar* gestartet werden. Dabei ist darauf zu achten, dass der **Host mit Client zuletzt gestartet** wird.

Zum einfachen Reproduzieren folgen nun zwei gültige Konfigurationsdateien, mit welchen 10.000 Primzahlen auf zwei Hosts, mit je 5 `Workern` getestet werden können.

```
client=yes
workerThreads=5
myPort=22000
connectionPort=25000
connectionAddress=192.168.2.76
primeRange=10000
```

```
client=no
workerThreads=5
myPort=25000
connectionPort=25000
connectionAddress=localhost
primeRange=10000
```

Die IP-Adresse der oberen Datei muss noch zur tatsächlichen IP des Hosts, auf welchem die untere Datei befindlich ist, geändert werden. Dann muss der untere Host zuerst gestartet werden, anschließend der obere.

# 6 Fazit

Ziel war es mittels eines verteilten Systemes das Herausfinden eines Private-Keys auf Basis eines Public-Keys einer RSA-Nachricht zu ermöglichen.

Das Verteilen der komplexen Berechnung auf mehrere Rechner konnte erfolgreich implementiert werden. Dazu wurde sowohl ein Konzept zum Aufbau des Clusters, zur Kommunikation innerhalb des Clusters und zur effektiven Verteilung der vorhandenen Primzahlen erstellt.

Ein `Client` verbindet sich mit dem Cluster, sendet den Public-Key und wartet anschließend auf den Private-Key, welcher vom Cluster per Brute-Force herausgefunden werden soll. Das Cluster besteht aus gleichgestellten `Workern`, welche sich selbstständig einen zu prüfenden Primzahlbereich zufällig auswählen. Die Primzahlbereiche sind dabei so gestaltet, dass jeder Bereich etwa gleich viele Berechnungen beinhaltet. Anschließend stellen die `Worker` durch Broadcast-Nachrichten sicher, dass kein anderer `Worker` diesen Bereich bearbeitet. Haben sie von allen anderen ein *OK* bekommen, prüfen sie den ausgewählten Bereich. Entweder finden sie die Lösung und schicken diese umher oder teilen den anderen `Workern` mit, dass der Bereich abgearbeitet ist und nicht erneut geprüft werden muss. Fallen ein oder mehrere `Worker` aus, so bemerken andere noch aktive `Worker`, dass die Kommunikation zu diesen nicht mehr möglich ist und sie werden aus dem Cluster entfernt.

Die Effektivität der Berechnung wurde in einer Performance-Analyse bestätigt, wobei das Prüfen von 100.000 Primzahlen auf zwei Hosts mit je 10 `Workern` etwa 1,5 Stunden dauerte. Auch die Ausfallsicherheit wurde in Testfällen bestätigt.

Eine Verbesserung wäre etwa durch das Ermöglichen von einer erneuten Verbindung nach dem Ausfall von Workern erreichbar.