Testat - 4933832 & 2180238

Analyse

Ziel

Das Ziel der gegebenen Aufgabe ist der Aufbau eines verteilte Systems, welches in der Lage sein soll, einen privater Schlüssel bestimmen, um mit diesem einen mit dem RSA-Verfahren verschlüsselten Text zu entschlüsseln. Zu den Teilzielen gehört demnach das Erstellen eines Clusters, welches einen Master / Leader und mehrere Slaves / Follower beinhaltet.

Voraussetzung

Die Berechnung des privaten Schlüssels soll auf mindestens zehn Knoten aufgeteilt werden können, wobei mindestens einer der Knoten auf einer anderen Maschine arbeiten soll, als die anderen Knoten.

Hauptanforderungen

Kernaspekt des Projektes ist das effiziente Aufteilen von Rechenleistung für die Berechnung der richtigen Primzahlen für das Finden des passenden privaten Schlüssel. Dementsprechend gehört das Aufteilen der Rechenleistung auf mehrere Arbeiter, hier Slaves genannt, zu den Hauptanforderungen. Um eine passende Aufteilung der Arbeit zu bestimmen, wird ein Koordinator gebraucht, welcher die Arbeiter koordiniert und die zu erledigende Arbeit richtig aufteilt. Dieser wird hier Master genannt. Zusätzlich dazu muss es einen Client geben, der bestimmt, welche Arbeit geleistet werden muss. Schlussendlich muss die Kommunikation zwischen den vorher aufgezählten Akteuren ermöglicht werden.

Nebenanforderungen

Zu den Nebenanforderungen gehören Aspekte, durch welche das System zuverlässiger oder effizienter arbeiten kann. Diese sind allerdings nicht zwingend notwendig für das Funktionieren des Systems. Dazu gehören Anforderung wie:

- Ausfallsicherheit: Falls bei einem Slave ein unerwartetes Problem auftritt, müssen seine Aufgaben von anderen Slaves erledigt werden

- Dokumentation: Beinhaltet unter anderem das Erstellen einer Liste mit allen Akteuren im System, sowie ihren Aufgaben und Adressen

Abgrenzung

In diesem Projekt muss kein User Interface entwickelt werden. Das Programm soll lediglich über die Konsole mithilfe von Startparametern bedient werden können. Zusätzlich dazu sind die Möglichkeiten der Eingabe bezüglich der Anzahl an Primzahlen auf die vordefinierten beschränkt. Dementsprechend muss keine Übergabe der öffentlichen Schlüssel und der Chiffre stattfinden, da diese direkt mit der Anzahl an Primzahlen zusammenhängen.

Verteilung der Aufgaben

Am Projekt wurde teilweise alleine und teilweise gemeinsam gearbeitet.
Von Leon wurden die Vorlage für alle vorhandenen ConnectionHandler, das Lesen und Schreiben in Dateien, die RSA Entschlüsselung anhand einer Example Application implementiert sowie die Grundarchitektur für die vorhandenen Komponenten und Klassen visualisiert.

Von Michel wurde das Verschicken von Objekten, die Klasse MasterSlave und die NodeList implementiert. Zudem hat Michel auch die Algorithmik und Visualisierung für den Heartbeat und die Discovery implementiert.

Der Rest des Projekts wurde im sogenannten Extreme Programming bearbeitet. Dabei haben Gruppenmitglieder gemeinsam am Code gesessen, wobei einer die Rolle des Schreibenden übernommen hat und der andere die Rolle des Unterstützenden, welcher auf Ungereimtheiten im Code achtet oder Vorschläge für Verbesserung gibt. Auf diese Weise wurde grundsätzlich besserer Quellcode geschrieben und komplexe Probleme schneller gelöst. Die Rollen wurden dabei regelmäßig gewechselt. Das Vorgehen nach Extreme Programming wurde genutzt, da eine Aufteilung in separate Aufgaben, die nur geringe Überschneidungen beinhalten, ab einem gewissen Zeitpunkt nicht mehr möglich war.

Grobkonzept

Architektur

Die Architektur dieses Projektes ist in drei Schichten zu gliedern. Dabei ist die erste Schicht die des Masters, die zweite die seiner Slaves und die dritte die des Clients. Die Kommunikation zwischen der ersten und zweiten Schicht finden über sogenannte SlaveHandler statt, welche die Verbindungen von den Slaves zum Master über Sockets herstellen. Zwischen der zweiten und dritten Schicht findet die Kommunikation über einen sogenannten ConnectionThread statt, welcher die Verbindung zwischen dem Client und genau einem Slave über einen Socket herstellt.

Klassen

Master

Der Master ist die schlussendliche Anlaufstelle für Anfragen vom Client, das Verbreiten von Nachrichten im Cluster sowie das Verteilen von Aufgaben an die Slaves. Dementsprechend gibt der jeweilige Slave, mit dem der Client verbunden ist, alle Anfragen, welche er vom Client erhält an den Master weiter, wo diese dann verarbeitet werden. Dabei erzeugt der Master unterschiedliche Threads die spezialisierte Aufgaben erfüllen. Dazu gehören ConnectionChecker sowie SlaveHandler

SlaveHandler

Der SlaveHandler ist ein Thread der für jede angefragte Verbindung von einem Slave gestartet wird und den jeweiligen Socket, der zu Verbindung gehört, übergeben bekommt. Die SlaveHandler kümmern sich um die Kommunikation zwischen den Slaves und dem Master. Hier liegen unterschiedliche Funktionen für das Verarbeiten von erhaltenen Nachrichten und Versenden von neuen Nachrichten vor.

ConnectionChecker

Der ConnectionChecker ist ein Thread, der vom Master wiederholt aufgerufen wird, um die Verbindung zwischen den SlaveHandlern und den Slaves zu prüfen. Dieser Thread wird alle fünf Sekunden gestartet und weist jeden der SlaveHandler an einen Heartbeat-Prozess

auszuführen. Wird eine Heartbeat-Anfrage nicht bis zum Versenden der Nächsten beantwortet so schließt der ConnectionChecker den jeweiligen SlaveHandler.

Slave

Der Slave hat mehrere Aufgaben zu erfüllen. Dazu gehören das Weiterleiten von Nachrichten vom Client zum Master, vom Master zum Client und das Durchführen von Berechnungen für die Ermittlung des privaten Schlüssels des RSA-Verfahrens. Dafür erstellt der Slave zwei Threads, einen ConnectionThread sowie einen WorkingThread-Thread.

WorkingThread

Der WorkingThread ist ein Thread vom Slave und dafür zuständig, seinen zugeteilten Bereich einer Liste von Primzahlen zu überprüfen, um den privaten Schlüssel für eine verschlüsselte Nachricht zu berechnen. Die Größe des zugeteilten Abschnitts von Primzahlen hängt von der Anzahl der anderen Slaves im Cluster und natürlich der Größe der Liste der Primzahlen ab. Hat der WorkingThread seine Berechnungen beendet gibt dieser Rückmeldung an den Slave.

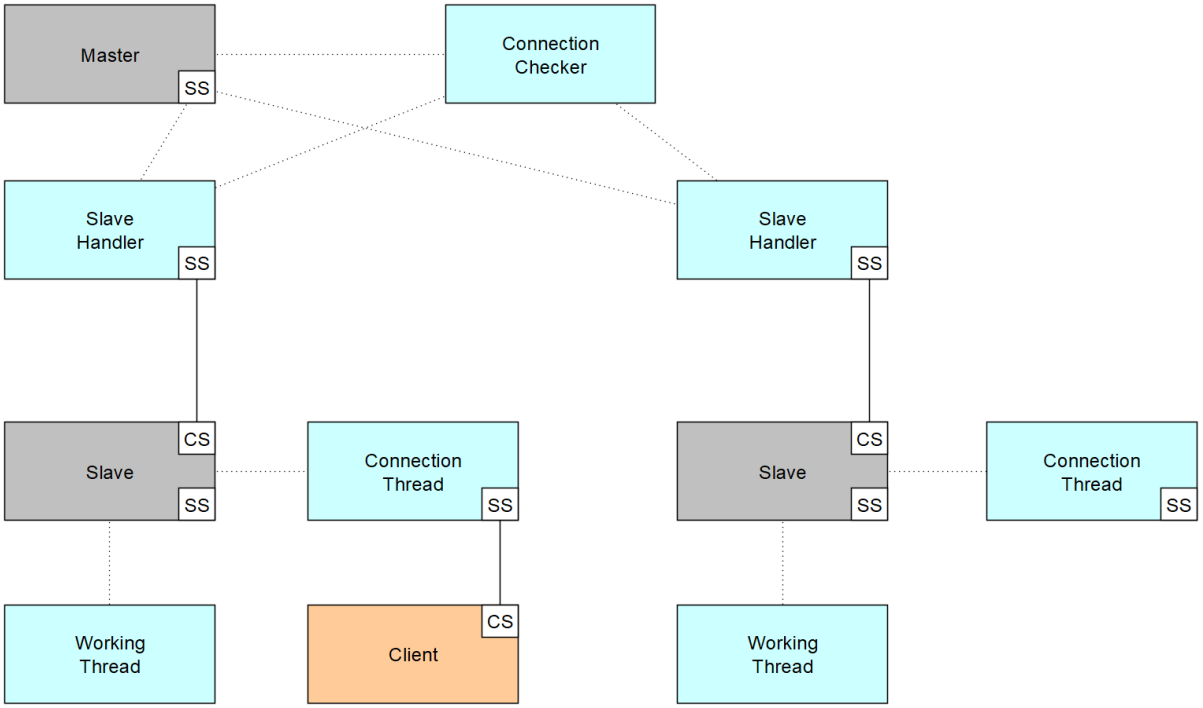
ConnectionThread

Der ConnectionThread ist ein Thread vom Slave und kümmert sich um die Kommunikation zwischen Client und Slave. Der Connection Thread empfängt Nachrichten vom Client und leitet diese an den Slave weiter, welcher die Nachricht an den Master weiterleitet.

Client

Der Client sendet kann unterschiedliche Anfragen an den Master senden. Dazu gehören:

- WRITE (*Text*): Schreibe *Text* in die Datei
- READ (*Anzahl*): Schicke mir eine bestimmte *Anzahl* der zuletzt erhaltenen Nachrichten
- RSA(*publicKey*, *chiffre*, *amountOfPrimes*) : Entschlüssel ein *Chiffre*, nutze dafür einen *publicKey* und eine bestimmte *Anzahl von Primzahlen* (*amountOfPrimes*)



Legende

- Grau: Hauptprozess
- Blau: Thread
- Gepunktete Linie: Verbindung innerhalb eines Prozesses
- Durchgezogene Linie: Verbindung durch Sockets
- Orange: Client
- SS: Serversocket
- CS: Clientsocket

Start

Bei dieser Klasse handelt es sich um die Hauptfunktionalität, was das Starten von Komponenten des Clusters angeht. Dabei können Startparameter auf drei verschiedene Weisen angeben werden, um Master, Slave beziehungsweise Slaves und Clients zu starten:

1. Master <Port des Masters>
2. Slave <Port des Masters> <DNS des Masters> <Port des 1. Slaves> ... <Port des letzten Slaves>
3. Client <Port des Slaves, mit dem sich verbunden werdens soll> <DNS dieses Slaves> <Anzahl an Primzahlen>

Wichtig zu erwähnen ist, dass beim Starten von Slaves am Ende der Startparameter beliebig viele Ports angegeben werden können. Für jeden dieser Ports wird dann ein Slave gestartet. Zudem sollten die Komponenten in der hier beschriebenen Reihenfolge gestartet werden.

BouncyCastle und Helferklassen

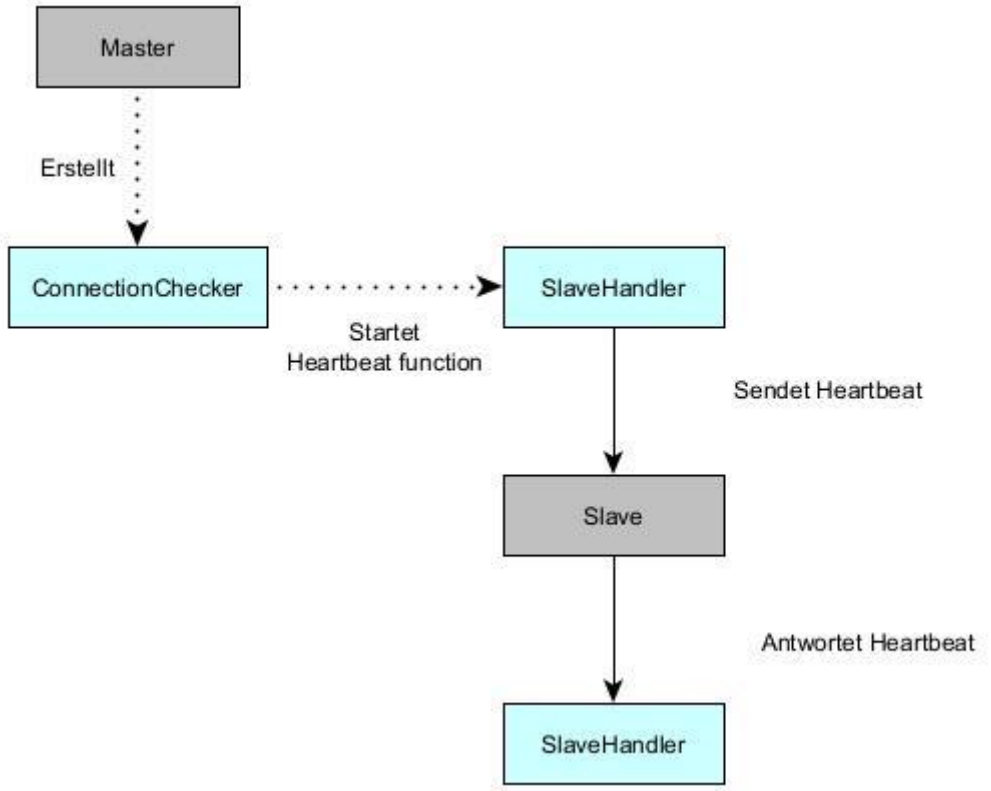
Im Projekt werden einige Helferklassen verwendet, welche von unserem Dozenten vorgegeben wurden. Diese werden im WorkingThread verwendet, um berechnete Werte auf ihre Richtigkeit zu prüfen, sowie im Master, um ein gegebenes Chiffre zu entschlüsseln. Die erwähnten Helferklassen greifen auf eine externe Bibliothek namens BouncyCastle zu, welche unter anderem Funktionalitäten zum Generieren von Schlüsselpaaren bereitstellt.

Verteilte Aspekte

Transparenz

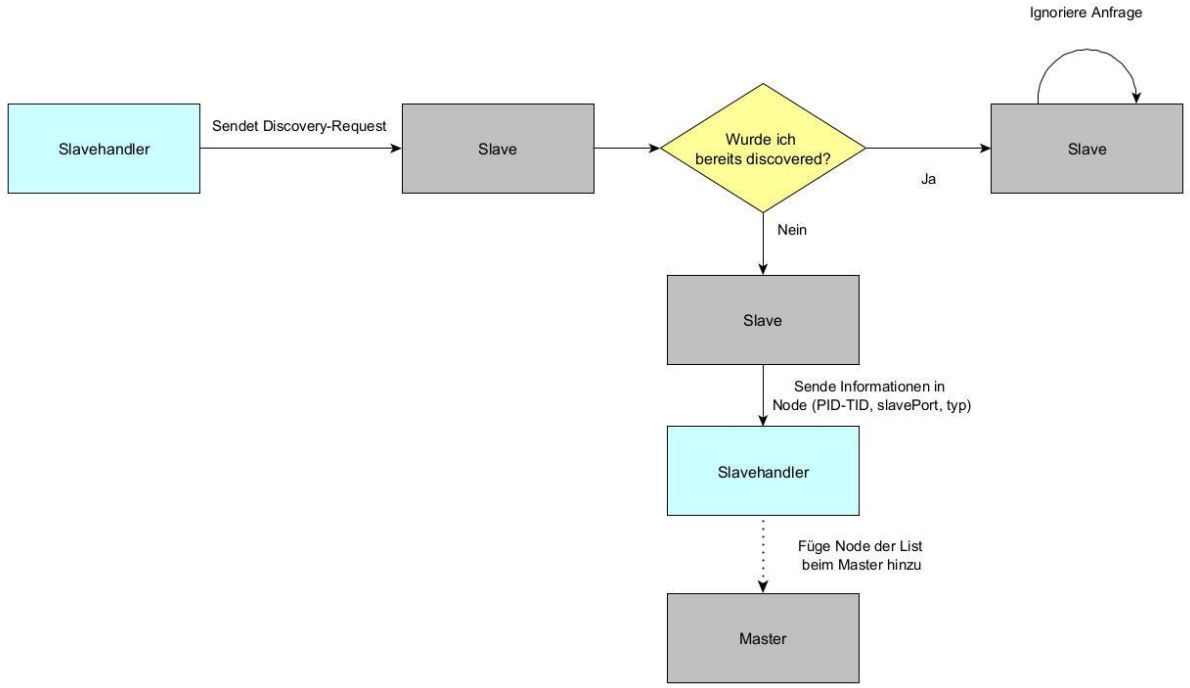
Heartbeat

Beim Heartbeat-Prozess wird alle fünf Sekunden mit einer anfänglichen Verzögerung von zehn Sekunden einen neuer Thread vom Master aus gestartet, welcher ConnectionChecker heißt. Dieser Thread weist jeden der vorhandenen SlaveHandler an, einen Heartbeat auszuführen. Dabei senden die SlaveHandler ihrem jeweiligen Slave eine Nachricht vom Typ “HEARTBEAT”. Antwortet der Slave seinem SlaveHandler nach drei weiteren Heartbeat Requests nicht mit einer Nachricht vom Typ “HEARTBEAT-RESPONSE”, so wird der jeweilige SlaveHandler terminiert. Antwortet der Slave rechtzeitig passiert nichts.



Discovery

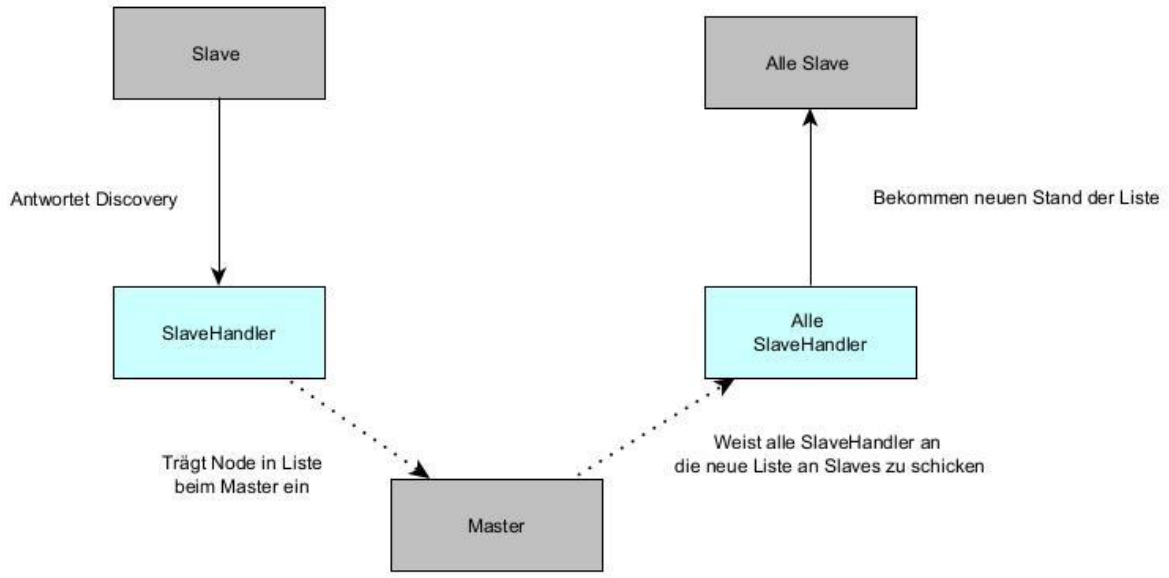
Bei der Initialisierung eines SlaveHandlers wird ein Discovery-Prozess durchgeführt. Bei diesem Prozess senden die SlaveHandler eine Nachricht an den jeweils zugehörigen Slave mit dem Nachrichtentypen “DISCOVERY”. Daraufhin prüft der Slave, ob dieser selbst bereits Discovered wurde, also ob er bereits eine Discovery-Request erhalten hat. Ist dies nicht der Fall, so erstellt der Slave ein Objekt der Klasse Node, welches er mit seiner Thread-ID, Prozess-ID und seinem Port und Typen füllt. Dieses Objekt sendet der Slave im Anschluss an den SlaveHandler zurück, welcher den erhaltenen Node zu der Liste von Nodes beim Master hinzufügt.



Synchronisierung

New List

Bei jeder neuen Verbindung bei dem der Slave die Anfrage des SlaveHandlers beantwortet und der dabei entstehende Node in der Liste des Masters eingetragen wird, weist der Master alle vorhandenen SlaveHandler an, den neuen Stand der Liste an die Slaves zu verteilen. Dies geschieht in der Methode “sendNewList”, welche als Eingabeparameter die aktuelle Liste des Masters nimmt. Dadurch hat jeder der Akteure im Cluster eine Liste mit den vorhandenen Slaves.



Nebenläufigkeit

Die Klassen Slave und Master haben unterschiedliche Aufgaben zu erfüllen. Beide erstellen deshalb Threads, die diese Aufgaben abarbeiten.

Der Master erstellt beispielsweise SlaveHandler für jede eingehende Verbindung eines Slaves. Dieser Slavehandler kümmert sich dann um die Kommunikation zwischen dem jeweiligen Slave und dem Master. Neben dem SlaveHandler erstellt der Master wiederholt einen ConnectionChecker, welcher sich um den Heartbeat-Prozess kümmert.

Der Slave hingegen erstellt einen ConnectionThread für eingehende Verbindungen von Clients und leitet diese über den Slave selbst an den Master weiter. Zudem erstellt der Slave auch einen WorkingThread, welcher zugeteilte Berechnungen vom Master bearbeitet.

Kommunikation

Für die Kommunikation zwischen einzelnen Akteuren, also Master, Slave und Client werden Sockets für die Kommunikation genutzt. Dabei haben sowohl der Master als auch jeder Slave einen Serversocket. Der Master nutzt diesen, um mit den Slaves des Clusters kommunizieren können. Der Slave hingegen verwendet seinen Serversocket, um eine Kommunikation zu einem Client herstellen zu können, falls sich ein solcher verbinden sollte. Für die Kommunikation innerhalb von Prozessen, wie beispielsweise zwischen Master und SlaveHandler, werden Funktionen genutzt. Erhält ein SlaveHandler beispielsweise eine Discovery-Response, so fügt dieser den neuen Node in die Liste des Masters ein und führt

eine Methode beim Master aus, welche jeden der SlaveHandler anweist, die neue Liste zu verschicken. Bei der Kommunikation zwischen ConnectionChecker und SlaveHandler wird der Heartbeat auf die gleiche Weise hergestellt und verwendet. Dies gilt auch für ConnectionThreads und ihre Slaves beim Weiterleiten von Nachrichten.

Konzeptioneller Ablauf

Vom Client zum Master

Der Prozess des Entschlüsselns des RSA-Verfahrens beginnt damit, dass sich ein Client mit dem ConnectionThread eines Slaves verbindet und diesem eine Nachricht vom Typ “RSA” schickt, welche den öffentlichen Schlüssel, die zu entschlüsselnde Chiffre und die dazu zu verwendende Anzahl an Primzahlen enthält. Der ConnectionThread empfängt diese Nachricht und leitet sie durch die Funktion “forward” des Slaves, zu welchem er gehört, an den dazu gehörigen SlaveHandler weiter. Dieser SlaveHandler empfängt die Nachricht und überprüft ihren Typ.

Vom Master zu den Slaves

Daraufhin wird der Inhalt entpackt und eine Funktion des Masters namens “findRSASolution” aufgerufen. Innerhalb dieser Funktion wird zunächst die passende Liste an Primzahlen geladen. Als Nächstes wird in jedem SlaveHandler die Funktion “sendToSlave” aufgerufen und dabei der öffentliche Schlüssel, der Umfang der zu überprüfenden Primzahlen und die vollständige Liste der Primzahlen übergeben. Jeder SlaveHandler schickt daraufhin eine Nachricht mit dem Typ “RSA-INFORMATION” an seinen jeweiligen Slave, welche die zuvor aufgezählten Informationen beinhaltet. Jeder Slave empfängt nun diese Nachricht mit seinem individuell ausgewähltem Bereich und überprüft zunächst ihren Typ. Daraufhin wird der Inhalt entpackt und ein neuer Thread namens WorkingThread gestartet, welchem die für die Entschlüsselung relevanten Informationen übergeben werden.

Knacken des RSA-Verfahrens

Jeder WorkingThread bearbeitet nun seinen Teil der Primzahlen und sucht nach einer Lösung. Falls ein WorkingThread in seinem Teilbereich keine Lösung findet oder die Lösung bereits gefunden hat, so ruft dieser die Funktion “closeWorkingThread” im dazugehörigen Slave auf, durch welche sein Thread beendet wird. Findet ein WorkingThread die Lösung ruft er die Funktion “shareSolution” im dazugehörigen Slave auf, wodurch eine Nachricht an

den jeweiligen SlaveHandler geschickt wird. Diese ist vom Typ “RSA-SOLUTION” und beinhaltet die beiden Primzahlen, mit denen die RSA-Verschlüsselung geknackt wurde.

Vom Slave zum Master

Nachdem der WorkingThread also über den Slave eine Nachricht an den SlaveHandler verschickt hat, empfängt diese die Nachricht und überprüft wieder ihren Typ. Daraufhin wird der Inhalt entpackt und die Funktion “decrypt” aus dem Master aufgerufen, welcher als Parameter die beiden relevanten Primzahlen übergeben werden. Die Funktion gibt darauffolgend die entschlüsselte Chiffre zurück.

Vom Master zum Client

Der SlaveHandler ruft im Folgenden die Funktion “distributeSolution” aus dem Master auf. Diese ruft wiederum in jedem SlaveHandler die Funktion “sendRSASolution” auf, durch welche eine Nachricht an den jeweiligen Slave gesendet wird. Diese hat den Typ “RSA-SOLUTION” und enthält die entschlüsselte Chiffre. Jeder Slave leitet diese Nachricht automatisch per Funktionsaufruf der Methode “forward” an seinen ConnectionThread weiter. Jeder ConnectionThread überprüft nun, ob ein Client mit ihm verbunden ist. Falls dies der Fall ist, so leitet er die Nachricht an den Client weiter. Zu guter Letzt empfängt der Client nun die Nachricht, entpackt den Inhalt und gibt die entschlüsselte Chiffre aus.

Schnittstellen

Master

Funktionsname: addNode
Übergabeparameter: Node node
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im Master enthalten und wird von einem SlaveHandler aufgerufen, wenn er eine Discovery Response von seinem Slave erhalten hat. Daraufhin wird der übergebene Node zur NodeList im Master hinzugefügt, welche alle Knoten des Clusters und wichtige Informationen zu ihnen enthält.
Fehlerbehandlung: Wirft eine IOException.

Funktionsname: findRSASolution
Übergabeparameter: String publicKey, String chiffre, String amountOfPrimes
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im Master enthalten und wird von einem SlaveHandler aufgerufen, wenn dieser eine Nachricht vom Typ “RSA-INFORMATION” erhält. Die Methode ist für das Laden der Primzahlen und das Verteilen der Aufgabenbereiche an die Slaves zuständig.
Fehlerbehandlung: Wirft eine IOException.

Funktionsname: decrypt
Übergabeparameter: String p, String q
Rückgabeparameter: String
Beschreibung: Diese Funktion ist im Master enthalten und wird von einem SlaveHandler aufgerufen, wenn dieser eine Nachricht vom Typ “RSA-SOLUTION” erhält. Die Methode gibt die entschlüsselte Chiffre zurück.
Fehlerbehandlung: -

Funktionsname: distributeSolution
Übergabeparameter: String chiffre
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im Master enthalten und wird von einem SlaveHandler aufgerufen, wenn dieser eine Nachricht vom Typ “RSA-SOLUTION” erhält. Die Methode kümmert sich um das Verteilen der entschlüsselten Chiffre an alle Slaves und somit schlussendlich auch an den Client.
Fehlerbehandlung: Wirft eine IOException.

SlaveHandler

Funktionsname: sendNewList
Übergabeparameter: ArrayList NodeList
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im SlaveHandler enthalten und wird vom Master aufgerufen, um die neue NodeList über die SlaveHandler an die Slaves zu verteilen.
Fehlerbehandlung: Wirft eine IOException.

Funktionsname: sendToSlave
Übergabeparameter: ArrayList<String> rsaInformation
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im SlaveHandler enthalten und wird vom Master aufgerufen, um die RSA Informationen über den SlaveHandler an den verbundenen Slave zu schicken.
Fehlerbehandlung: Wirft eine IOException.

Funktionsname: sendRSASolution
Übergabeparameter: String chiffreText
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im SlaveHandler enthalten und wird vom Master aufgerufen, um die entschlüsselte Chiffre über den SlaveHandler an den verbundenen Slave zu schicken.
Fehlerbehandlung: Wirft eine IOException.

Funktionsname: heartbeat
Übergabeparameter: -
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im SlaveHandler enthalten und wird vom ConnectionChecker aufgerufen, um eine Heartbeat Request an den verbundenen Slave zu schicken.
Fehlerbehandlung: -

Funktionsname: getSlaveAnsweredHeartbeat
Übergabeparameter: -
Rückgabeparameter: int
Beschreibung: Diese Funktion ist im SlaveHandler enthalten und wird vom ConnectionChecker aufgerufen, um zu erfahren, vor wie vielen Heartbeat Zyklen der Slave das letzte Mal eine Heartbeat Response zurückgeschickt hat.
Fehlerbehandlung: -

Slave

Funktionsname: forward
Übergabeparameter: Message message
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im Slave enthalten und wird vom ConnectionThread aufgerufen, um Nachrichten vom Client über den Slave an den Master weiterzuleiten.
Fehlerbehandlung: Wirft eine IOException.

Funktionsname: closeWorkingThread
Übergabeparameter: -
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im Slave enthalten und wird vom WorkingThread aufgerufen, um diesen WorkingThread zu schließen.
Fehlerbehandlung: -

Funktionsname: shareSolution
Übergabeparameter: String p, String q
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im Slave enthalten und wird vom WorkingThread aufgerufen, um die Primzahlen an den Master zu schicken, durch die eine Lösung gefunden wurde.
Fehlerbehandlung: Wirft eine InterruptedException.

ConnectionThread

Funktionsname: forward
Übergabeparameter: Message message
Rückgabeparameter: void
Beschreibung: Diese Funktion ist im ConnectionThread enthalten und wird vom Slave aufgerufen, um Nachrichten vom Master über den ConnectionThread an den Client weiterzuleiten.
Fehlerbehandlung: Wirft eine IOException.

Beschreibung der Anpassung

Typ: Funktion
Name: sendMultiple
Beschreibung: Diese Funktion ist im Client enthalten und dafür zuständig mehrere “WRITE” Befehle hintereinander zu schicken und schlussendlich einen “READ” Befehl zu versenden.
Anpassung: Nicht verwendet
Grund: Die Funktion wurde im Zuge der Bearbeitung der Übungen implementiert und weiterhin für Tests verwendet. Für die letztendliche Aufgabe wird sie allerdings nicht mehr benötigt.

Typ: Klasse
Name: main
Beschreibung: Diese Klasse ist dafür zuständig einen Master, mehrere Slaves und einen Client nacheinander zu starten.
Anpassung: Nicht verwendet
Grund: Die Klasse wurde zu Beginn zu Testzwecken impementiert und wurde von der Klasse “start” abgelöst, da “main” keine Startparameter übernimmt und alle Komponenten zusammen startet, was nicht den Anforderungen des Projektes entspricht.

Testplan

Vorwort

Die verschiedenen Komponenten werden mithilfe der Klasse “start” verwendet, welcher die jeweils angegebenen Startparameter übergeben werden müssen. Um das Programm über die Konsole zu starten, muss zunächst der Ordner mit dem Pfad “MasterOfTheSlaves\out\artifacts\MasterOfTheSlaves_jar” geöffnet werden. Hier wird daraufhin der Befehl “java -jar MasterOfTheSlaves.jar” ausgeführt, gefolgt von den jeweiligen Startparametern. Dies sieht dann beispielsweise folgendermaßen aus: “java -jar MasterOfTheSlaves.jar Master 8000”. Auf diese Weise können schlussendlich in mehreren Terminals der Master, mehrere Slaves und ein Client gestartet werden. Hierbei ist wichtig zu erwähnen, dass für die Entwicklung dieses Projektes die JDK der Version 17.0.1 verwendet und damit auch kompiliert wurde.

Zudem sollte die verwendbare Größe des Heaps beim Starten des Programms begrenzt werden. Dies ist durch die Parameter “Xms” und “Xmx” möglich, wobei ersteres die anfänglich zugewiesene Größe beschreibt und letzteres die maximale Größe. In Betrachtung der Hardwarelimitationen des Raspberry Pis können diese Parameter auf 128m und 256m gesetzt werden. Schlussendlich sieht ein beispielhafter vollständiger Startbefehl für einen Master dann folgendermaßen aus: “java -Xms128m -Xmx256m -jar MasterOfTheSlaves.jar Master 8000”. Falls dies die Möglichkeiten des Raspberry Pis überschreitet, so lassen sich auch kleinere Werte angeben.

Test Nr. 1

Testbezeichnung: Minimaltest bezüglich der Anzahl an Slaves
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Clusters bei einer Minimalbesetzung von einem Slave zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. Einen Slave starten
    a. Startparameter: [Slave 8000 localhost 8001]
3. Client starten und Anfrage mit 1000 Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 1000]

Erwartetes Ergebnis: Heartbeat, Discovery und Berechnung des privaten Schlüssels erfolgen problemlos
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 2

Testbezeichnung: Maximaltest bezüglich der Anzahl an Slaves
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Clusters bei einer Maximalbesetzung von 30 Slaves zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. 30 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002 8003 8004 8005 8006 8007 8008 8009 8010 8011 8012 8013 8014 8015 8016 8017 8018 8019 8020 8021 8022 8023 8024 8025 8026 8027 8028 8029 8030]
3. Client starten und Anfrage mit 1000 Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 1000]

Erwartetes Ergebnis: Heartbeat, Discovery und Berechnung des privaten Schlüssels erfolgen problemlos
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 3

Testbezeichnung: Hardwarefehler eines Slaves vor der Bearbeitung der Primzahlen
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Clusters beim Absturz eines Slaves zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. 2 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002]
3. Einen weiteren Slave starten
    a. Startparameter: [Slave 8000 localhost 8003]
4. Den weiteren Slave beenden
5. Client starten und Anfrage mit 1000 Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 1000]

Erwartetes Ergebnis: Bearbeitung der Primzahlen erfolgt vollständig und problemlos
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 4

Testbezeichnung: Hardwarefehler eines Slaves während der Bearbeitung der Primzahlen
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Clusters beim Absturz eines Slaves zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. Einen Slave starten
    a. Startparameter: [Slave 8000 localhost 8001]
3. Einen weiteren Slave starten
    a. Startparameter: [Slave 8000 localhost 8002]
4. Client starten und Anfrage mit 10000 Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 10000]
5. Den weiteren Slave nach 10 Sekunden beenden, während die Berechnung noch stattfindet

Erwartetes Ergebnis: Bearbeitung der Primzahlen erfolgt vollständig und problemlos
Tatsächliches Ergebnis: Bearbeitung ist nicht vollständig erfolgt, da die Primzahlen des beendeten Slaves nicht bearbeitet wurden
Nachbedingung: Ausgleich bei Ausfall eines Slaves muss sichergestellt werden. Dazu müssen die Primzahlen, die sich in dem Bereich befinden, der dem ausgefallenen Slave zugeteilt wurde, auf die restlichen Slaves im Cluster verteilt werden.
Testergebnis: Nicht Akzeptiert

Test Nr. 5

Testbezeichnung: Hardwarefehler eines Slaves nach der Bearbeitung der Primzahlen
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Clusters beim Absturz eines Slaves zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. 2 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002]
3. Einen weiteren Slave beenden
    a. Startparameter: [Slave 8000 localhost 8003]
4. Client starten und Anfrage mit 1000 Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 1000]
5. Den weiteren Slave beenden. nachdem die Bearbeitung beendet wurde

Erwartetes Ergebnis: Bearbeitung der Primzahlen erfolgt vollständig und problemlos
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 6

Testbezeichnung: Langzeittest bezüglich des Heartbeats
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Heartbeat Prozesses über längere Zeit zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. 3 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002 8003]
3. 60 Minuten warten

Erwartetes Ergebnis: Heartbeat erfolgt über die Zeitspanne regelmäßig und vollständig
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 7

Testbezeichnung: Lasttest bezüglich des Heartbeats
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Heartbeat Prozesses unter starker Belastung zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. 30 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002 8003 8004 8005 8006 8007 8008 8009 8010 8011 8012 8013 8014 8015 8016 8017 8018 8019 8020 8021 8022 8023 8024 8025 8026 8027 8028 8029 8030]
3. 5 Minuten warten

Erwartetes Ergebnis: Heartbeat erfolgt über die Zeitspanne regelmäßig und vollständig
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 8

Testbezeichnung: Stabilitätstest bezüglich der Eingabe falscher Werte vom Client
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Programms bei inkorrekten Eingaben zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. 3 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002 8003]
3. Client starten und Anfrage mit falscher Anzahl an Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 42]

Erwartetes Ergebnis: Die Anfrage des Clients wird nicht versendet, da die Anzahl an Primzahlen nicht mit den vorgegebenen Möglichkeiten übereinstimmt
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 9

Testbezeichnung: Hardwarefehler eines Clients während der Bearbeitung der Primzahlen
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Programms beim Absturz des Clients zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

1. Master starten
    a. Startparameter: [Master 8000]
2. 10 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002 8003 8004 8005 8006 8007 8008 8009 8010]
3. Client starten und Anfrage mit 10000 Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 10000]
4. Den Client nach 10 Sekunden beenden, während die Berechnung noch stattfindet

Erwartetes Ergebnis: Bearbeitung der Primzahlen erfolgt vollständig und problemlos, der Client erhält aufgrund seiner Beendigung allerdings keine Nachricht
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Test Nr. 10

Testbezeichnung: Hardwarefehler eines Clients nach der Bearbeitung der Primzahlen
Referenz auf Forderung: Dieser Test wird benötigt, um die Stabilität des Programms beim Absturz des Clients zu testen
Vorbedingungen: Passende Java-Version ist installiert, Terminal befindet sich im korrekten Pfad
Testschritte:

5. Master starten
    a. Startparameter: [Master 8000]
6. 10 Slaves starten
    a. Startparameter: [Slave 8000 localhost 8001 8002 8003 8004 8005 8006 8007 8008 8009 8010]
7. Client starten und Anfrage mit 1000 Primzahlen schicken
    a. Startparameter: [Client 8001 localhost 1000]
8. Den Client nach der Bearbeitung der Primzahlen beenden

Erwartetes Ergebnis: Bearbeitung der Primzahlen erfolgt vollständig und problemlos
Tatsächliches Ergebnis: Wie erwartet
Nachbedingung: -
Testergebnis: Akzeptiert

Nachbetrachtung

Auswertung der Performance

	100 Primzahlen	1000 Primzahlen	10000 Primzahlen	100000 Primzahlen	1000000 Primzahlen (Prognose)	1000000000 Primzahlen (Prognose)
2 Knoten /Prozesse	223 ms	2196 ms	363987 ms	7638503 ms	487500000 ms	3168750000000 ms
5 Knoten /Prozesse	170 ms	2865 ms	179586 ms	6639681 ms	250800000 ms	978120000000 ms
10 Knoten /Prozesse	172 ms	1320 ms	60579 ms	8864292 ms	195000000 ms	1072500000000 ms

Bei 100 Primzahlen fallen die Zeiten für zwei, fünf und zehn Knoten recht ähnlich aus, da die Berechnungsdauer im Vergleich extremst gering ist. Ausschlaggebend für die Dauer bis zum Fund des richtigen Ergebnis ist die Position der richtigen Primzahlen, sowie die Zeit für das Starten der Threads selbst.

Beim Durchrechnen von 1000 Primzahlen gehen die Ergebnisse stärker auseinander. Überraschend ist hier, dass fünf Knoten langsamer sind als zwei. Dies ist wahrscheinlich auf die Position der Primzahlen zurückzuführen, welche bei der Aufteilung auf zwei Knoten schneller zu finden sind. Grundsätzlich sollte mit größerer Anzahl von Primzahlen die Relevanz der Position der Primzahlen schwächer werden.

Beim Durchrechnen der 10000 entsprechen die Ergebnisse den Erwartungen. Bei einer Verdopplung beziehungsweise 2,5-Fachung der Rechenknoten kommt ungefähr eine Halbierung der Zeit zustande.

Bei 100000 Primzahlen entsprechen die Ergebnisse für zwei und fünf Knoten den Erwartungen, da zwei Knoten länger brauchen als fünf. Das Ergebnis für zehn Knoten ist jedoch unerwartet, da die Berechnung hier am längsten dauert Eine mögliche Erklärung dafür wäre, dass die Position der gesuchten Primzahlen ungünstig für eine Aufteilung in den zehn Abschnitten liegt.

Prognose

Bei der Prognose für eine Million Primzahlen liegt eine Verzehnfachung der zu berechnenden Primzahlen vor. Dieser Faktor von zehn pro Steigerung an Primzahlen ist konsistent. Deshalb könnte man davon ausgehen, dass dies die benötigte Rechenzeit ebenfalls betrifft. Schaut man sich die Faktoren für die erhöhte Berechnungsdauer für zwei Knoten an:

- 100 - 1000: Faktor von ca. 10
- 1000 - 10000: Faktor von ca. 165
- 10000 - 100000: Faktor von ca. 21

Würde man den Durchschnitt dieser Faktoren nehmen wäre ein möglicher Faktor 65. Dementsprechend wäre eine mögliche Prognose für die Berechnungsdauer ca. 65 * 7500000 ms = 487500000 ms für eine Million Primzahlen bei zwei Knoten. Macht man das Gleiche für fünf Knoten ergibt sich eine mögliche Berechnungszeit von 250800000 ms. Für zehn Knoten läge sie bei 585000000 ms. Da die Prognose für die lange Berechnungszeit für 10 Knoten stark der schlechten Performance für zehn Knoten bei 100000 Primzahlen geschuldet ist, wäre es hier sinnvoll die Prognose anzupassen. Möglich wäre das Teilen der 585000000 ms durch drei wodurch die Prognose stärker den Erwartungen entsprechen würde, unter der Bedingung, dass die Position der gesuchten Primzahlen keinen starken Einfluss hat.

Geht man nach dem gleichen Vorgehen für eine Milliarde Primzahlen vor, so ergeben sich folgende Prognosen:

- 2 Knoten:
    - ((2200/220) + (360000/2200) + (7600000/360000) + (487500000/7600000)) / 4 = 65 → Faktor für Verzehnfachung der Primzahlen
    - 65 * 100 = 6500 Faktor für Vertausendfachung der Primzahlen
    - Prognose: 6500 * 487500000 ms = 3168750000000 ms
- 5 Knoten:
    - ((2800/170) + (180000/2800) + (6640000/180000) + (250800000/6640000)) / 4 = 39 → Faktor für Verzehnfachung der Primzahlen
    - 39 * 100 = 3900 Faktor für Vertausendfachung der Primzahlen
    - Prognose: 3900 * 250800000 ms = 978120000000 ms
- 10 Knoten:
    - ((1300/170) + (60000/1300) + (8860000/60000) + (195000000/8860000)) / 4 = 55 → Faktor für Verzehnfachung der Primzahlen

    - 55 * 100 = 5500 Faktor für Vertausendfachung der Primzahlen
    - Prognose: 5500 * 195000000 ms = 1072500000000 ms
- Lösung in einer 1h (3600000 ms):
    - Rate von der Verbesserung von 2 auf 5 Knoten: 3.2 (3168750000000 / 978120000000)
    - Benötige Rate der Verbesserung von 3168750000000 ms auf 3600000 ms: 880208.3
    - LGS:
        - x: Rate der Verbesserung für Vervielfachung der Knoten
        - y: Benötigte Vervielfachung
        - I: 2 * 2,5x = 3,2
        - II: 2 * xy = 880208.3
        - Ia: 4,5x = 3,2 ⇔ x = 0,64
        - IIa: 2 * 0,64y = 880208.3
        - IIb: 1,28y = 880208.3 ⇔ y = 687662,7
        - Ergebnis: Es werden mindestens 687663 mehr Knoten benötigt als im Vergleich zu zwei Knoten. Dementsprechend werden mindestens 687663 * 2 = 1375326 Knoten benötigt, um eine Milliarde Primzahlen in einer Stunde zu berechnen

Dies lässt sich schlussendlich auch auf eine Billion Primzahlen übertragen.
Im Kern wird dieselbe Rechnung wie vorher verfolgt, allerdings ändert sich der Faktor von Tausend zu 1000000, wodurch sich folgende Werte ergeben:

- 2 Knoten:
    - 4 = 65 → Faktor für Verzehnfachung der Primzahlen
    - 65 * 100000 = 6500000 Faktor für Vermillionenfachung der Primzahlen
    - Prognose: 6500000 * 487500000 = 3168750000000000 ms
- 5 Knoten:
    - 39 → Faktor für Verzehnfachung der Primzahlen
    - 39 * 100000 = 3900000 Faktor für Vermillionenfachung der Primzahlen
    - Prognose: 3900000 * 250800000 = 978120000000000 ms
- 10 Knoten:
    - 55 → Faktor für Verzehnfachung der Primzahlen
    - 55 * 100000 = 5500000 Faktor für Vermillionenfachung der Primzahlen
    - Prognose: 5500000 * 195000000 = 1072500000000000 ms
- Lösung in einer 1h (3600000 ms):
    - Rate von der Verbesserung von 2 auf 5 Knoten: 3.2

(3168750000000000 / 978120000000000)

- Benötige Rate der Verbesserung von 3168750000000000 ms auf 3600000 ms: 880208333.3
- LGS:
  - x: Rate der Verbesserung für Vervielfachung der Knoten
  - y: Benötigte Vervielfachung
  - I: 2 * 2,5x = 3,2
  - II: 2 * xy = 880208333.3
  - Ia: 4,5x = 3,2 ⇔ x = 0,64
  - IIa: 2 * 0,64y = 880208333.3
  - IIb: 1,28y = 880208333.3 ⇔ y = 687662760,4
  - Ergebnis: Es werden mindestens 687662760,4 mehr Knoten benötigt als im Vergleich zu zwei Knoten. Dementsprechend werden mindestens 687662760,4 * 2 = 1375325520,8 Knoten benötigt, um eine Billion Primzahlen in einer Stunde zu berechnen

Fazit

Die Durchführung des Projektes wurde stark geleitet durch die erklärenden Vorlesungen und Übungen. Deshalb wurde das Projekt nach und nach um die einzelnen Funktionalitäten erweitert, die in der Vorlesung vorgestellt wurden. Dadurch hat sich das Projekt iterativ weiterentwickelt, wobei Anforderungen, die nicht in der Vorlesung vorgestellt wurden auch grundsätzlich nicht in der Zeit der Vorlesung bearbeitet wurden.

Beim Durchführen des Projekts wurde viel Neues gelernt. Dazu gehören das richtige Erstellen und Einsetzen von Threads, das Nutzen von Sockets für Kommunikation, sowie generell das Verteilen von Rechenleistung für das effizientere Erfüllen von Aufgaben.
Das Ergebnis des Projekts ist ein verteiltes System, welches in der Lage ist, Berechnungen für das RSA Verfahren effizient zu verteilen, über mehrere Maschinen hinweg zu arbeiten und die Ressourcen der Maschinen zu nutzen. Leider ist das System noch anfällig für Ausfälle des Masters. Es können jedoch bestimmte Ausfälle vom Client oder den Slaves abgefangen werden, sodass das System nicht abstürzt, wodurch das System in seiner Grundfähigkeit erfolgreich arbeiten kann.

Das Arbeiten im Team lief sehr gut, da weniger komplexe Aufgaben fair zwischen Teammitgliedern aufgeteilt wurden, während komplexe Probleme und Aufgaben gemeinsam gelöst wurden. Die Kommunikation lief auch reibungslos und erfolgte zum großen Teil über Discord.

Ein nicht behandelter Aspekt ist die Ausfallsicherheit des Masters, welcher der Kern des Systems ist und als Endpunkt für Anfragen sowie zum Verteilen von Aufgaben dient. Es wurden zwar die Grundbausteine gelegt, um einen der Slaves in einen Master zu konvertieren, jedoch wurde kein Algorithmus für die Auswahl des Slaves oder den Prozess der sauberen Umwandlung implementiert. Dementsprechend wurden bisher der Slave und Master in einer gemeinsamen Klasse vereint, sowie bei beiden eine Liste mit allen Nodes, also Knoten des Clusters, hinterlegt.

Das System ist in seiner Skalierbarkeit begrenzt. Das Erstellen von zu vielen Slaves führt beispielsweise zu Timing Problemen, wodurch einzelne Funktionalitäten aussetzen. Darunter fällt unter anderem der Heartbeat-Prozess. Dieser iteriert über alle SlaveHandler und weist sie an einen Heartbeat auszuführen. Bei zu vielen SlaveHandlern führt dies allerdings dazu, dass die nächste Heartbeat-Anfrage gestellt wird, während die vorherige noch nicht beendet wurde.

Das System kann ohne Probleme die jeweiligen privaten Schlüssel für die unterschiedlichen Anzahlen an Primzahlen berechnen, insofern keiner der Arbeiter unerwartet beendet wird oder es Probleme bei der Übertragung von Nachrichten gibt.

Mögliche Erweiterungen des Systems wären die Einführung von mehreren Clients, sein die gleichzeitig Anfragen an den Master senden können, grundsätzlich mehr Cluster zu haben mit den sich ein einzelner Client verbinden kann oder das Verwenden von mehreren Mastern, die Wahl von diesem von seiten des Clients.

Im vorgestellten System dient der Master als zentraler Endpunkt für Anfragen, während die Slaves die Arbeiter darstellen. Fraglich ist jedoch, warum die Slaves die Kommunikationsbrücke von Client zu Master sind und gleichzeitig für die Berechnungen verantwortlich sind. In der Stelle wäre ein Proxy sinnvoll, welcher als Anlaufstelle für das Cluster dienen soll und Anfragen vom Client vorverarbeitet, bevor sie an den Master gesendet werden. Durch diese Trennung der beiden Aufgaben können sich die Slaves auf Berechnungen konzentrieren und dem Client entfällt die Entscheidung welchen der Slaves er ansprechen muss.

Ein Problem des Systems ist beispielsweise gegeben, wenn sich ein Client mit einem Slave verbindet und der Slave in dem Moment abstürzt, wenn der Master eine Nachricht an den Client über diesen Slave senden will. Die Nachricht geht in diesem Fall verloren. Verbindet sich der Client jedoch rechtzeitig mit einem anderen Slave, so erhält er die Nachricht dennoch. Zudem wurde nicht beachtete, dass auf dem Raspberry Pi nicht die neueste Version von Java beziehungsweise OpenJDK installierbar ist, wodurch der vorhandene Code dort nicht ohne Probleme ausgeführt werden kann. Dementsprechend wäre es nötig gewesen, dies vor Beginn des Programmierens zu prüfen und entsprechende Anpassungen an der Entwicklungsumgebung vorzunehmen.