

Verteilte Systeme

# RSA Entschlüsselung

Matrikelnummern:  
4716336, 3552114, 8090097

Abgabe: 17.12.21

# Dokumentation

<b>1. Analyse</b>	<b>2</b>
Aufgabenanalyse	2
Ziel der Aufgabe	2
Voraussetzungen	3
Abgrenzung	3
<b>2. Grobkonzept</b>	<b>3</b>
Architektur	3
Aspekte	5
Kernabläufe	5
Schnittstellen	6
Anpassungen(nur wenn wir welche haben)	7
<b>3. Programm</b>	<b>7</b>
Umsetzung der Forderungen	7
Umsetzung der Architektur	7
Application	7
ApplicationClient, ApplicationMaster, ApplicationSlave	8
FileEditor	8
Message	8
Node	8
RSAPayload	8
Client	8
Master	8
Slave	9
ClientHandler	9
RequestHandler	9
DecryptionHandler	9
RunnableClassClient, RunnableClassMaster &RunnableClassSlave	9
Make-Anweisung	9
Startanweisungen	10
<b>4. Testplan</b>	<b>10</b>
<b>5. Nachbetrachtung</b>	<b>11</b>
Zusammenfassung der Ergebnisse	11
Zur Verbesserung	12

# 1. Analyse

## Aufgabenanalyse

### Ziel der Aufgabe

Das Ziel der Aufgabe ist es, das RSA Verfahren mithilfe eines verteilten Systems zu lösen. Das RSA-Verfahren baut, wie andere Verschlüsselungs- und Signaturverfahren, auf dem Prinzip, dass die Primfaktoren eine lange Zeit brauchen, um durch ein Brute-Force-Verfahren gefunden zu werden. In diesem Projekt werden bereits mögliche Primzahlen zur Entschlüsselung gegeben. Sonst würden die Möglichkeiten der Entschlüsselung gegen Null stehen.

Die **Verteilung der Aufgaben** gelingt parallel zum Erkennen der Anforderungen:

Die Arbeitsteilung des Projektes sieht folgendermaßen aus:

4716336: Dokumentation

8090097: Dokumentation

3552114: Programmcode, Dokumentation (Programm: Make-Anweisungen, Startanweisungen)

Nähere Informationen zur Arbeitsteilung bezüglich des Programmcodes siehe




GitHub-Repository: <https://github.com/tillmbecker/verteilte-systeme/commits/main>.




































### Voraussetzungen

Als Voraussetzung nutzen wir den Quellcode, den wir im Laufe der Vorlesung "Verteilte Systeme" erstellt und erweitert haben.

Außerdem bereitgestellt sind Funktionen zur Entschlüsselung des RSA-Verfahrens und eine Liste der zu verwendenden Primzahlen.

# Abgrenzung

 Starker Zusammenhang = 1    
  Mäßiger Zusammenhang=0,5    
  schwacher Zusammenhang = 0,25

Anforderungen / Prozesse	Logging mit Datei	Multithreading Nebenläufigkeit	Client – vert. System Kommunikation	Cluster	RSA Funktionalität	SUM = Priorität
Dateizugriffe und Multithreading						4,5
Socket und Streams						4
Client-Verbindung über ClientHandler						3,75
Master-Slave Rollenverteilung + Kommunikation						3,75
Größere Anzahl von Slaves und Clients						3,75
Senden von RSA Anfragen						3,75
DecryptionHandler						3,25

Die Ausfallsicherheit wird nur in begrenztem Maße umgesetzt.

## 2. Grobkonzept

### Architektur

Die Gesamtarchitektur eines verteilten Systems besteht aus homogenen oder auch heterogenen Komponenten, welche die Rechenlast gemeinsam unter sich aufteilen. Das geregelte Aufteilen und das Verwalten von Rechenaufgaben erfordert eine ausgeprägte und effiziente Kommunikation. Die Kommunikation kann durch verschiedene Protokolle, meist auf Schicht 5 des ISO/OSI-Modells, umgesetzt werden. Der Benutzer eines verteilten Systems muss somit nur mit einer Komponente kommunizieren und alle Komponenten erhalten ebenfalls diese Informationen.

In unserem Beispiel nennen wir jeden Prozess, welcher ein Teil der Rechenlast übernimmt, einen Knoten. Der Knoten, der die Kommunikationszentrale darstellt und bei welchem alle Informationen ankommen, wird der Masterknoten genannt.

Die weiteren Knoten werden als Slaves (Arbeiterknoten) behandelt. Ihre Hauptaufgabe ist es, die Rechenlast zu verteilen.

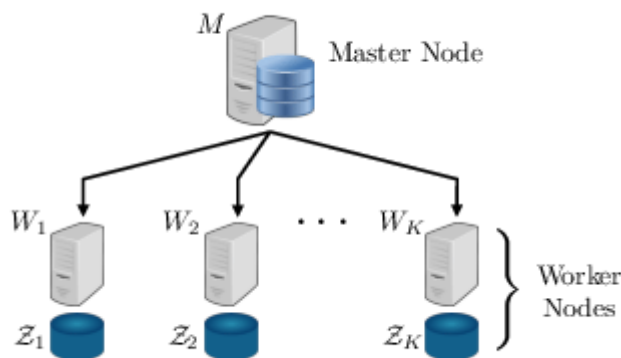


Abbildung: Zusammenspiel Masterknoten und Slaves (Arbeiterknoten)

Die einzelnen Teilschichten werden durch die verschiedenen Klassen repräsentiert. Dabei sind augenscheinlich die Klassen "Master", "Slave", "Client", sowie "ClientHandler" und "RequestHandler" von größerer Bedeutung.

Der Master spielt insofern eine zentrale Rolle, da er jede Socket-Verbindung zwischen ihm und den Slaves bereitstellt. Er speichert diese Verbindungsinformationen zusätzlich in einer HashMap, um auch über den aktuellen Status der Slaves informiert zu sein. Dabei aktualisiert sich die Map, falls ein Slave sich entfernt oder sich ein neuer Slave dazugeschaltet hat. Der Master implementiert außerdem die Funktionalität, dass sich ein eigener Thread abspaltet, welcher sich um die eingehenden Socket-Verbindungen kümmert. So wird bei jeder neuen Anfrage an den Master, bei welcher sich ein Slave an ein Socket verbinden will, wird ein eigener Thread gestartet. Dieser Thread wird durch die Klasse "RequestHandler" aufgerufen und erhält von dem Master Zugriff auf das Socket und auf die ObjectOutputStream- und ObjectInputStreams.

Aufgaben des RequestHandlers bestehen zum einen darin, dass er auf eingehende Anfragen wartet und über die ein- und ausgehenden Streams Nachrichten empfängt oder sendet. Der RequestHandler entscheidet aufgrund des Typs, welcher in der Nachricht gespeichert wird, was er mit dieser Nachricht anfangen kann.

Es können folgende Typen in der Message vorkommen:

join	ein neuer Eintrag in die Map + Senden einer Bestätigung
write	Schreibzugriff in eine Datei + Senden einer Bestätigung
read	Senden der letzten Message
rsa	Neue Liste mit Primzahlen erstellen
rsa-success	Beenden des RSA Prozesses
leave	Löschen des Slaves aus der Map

Der Client initiiert die Kommunikation und schickt die Aufgabenstellung an das verteilte System. In diesem Projekt ist es die Entschlüsselung des RSA-Verfahrens, wobei der Client einen public key und eine Chiffre sendet.

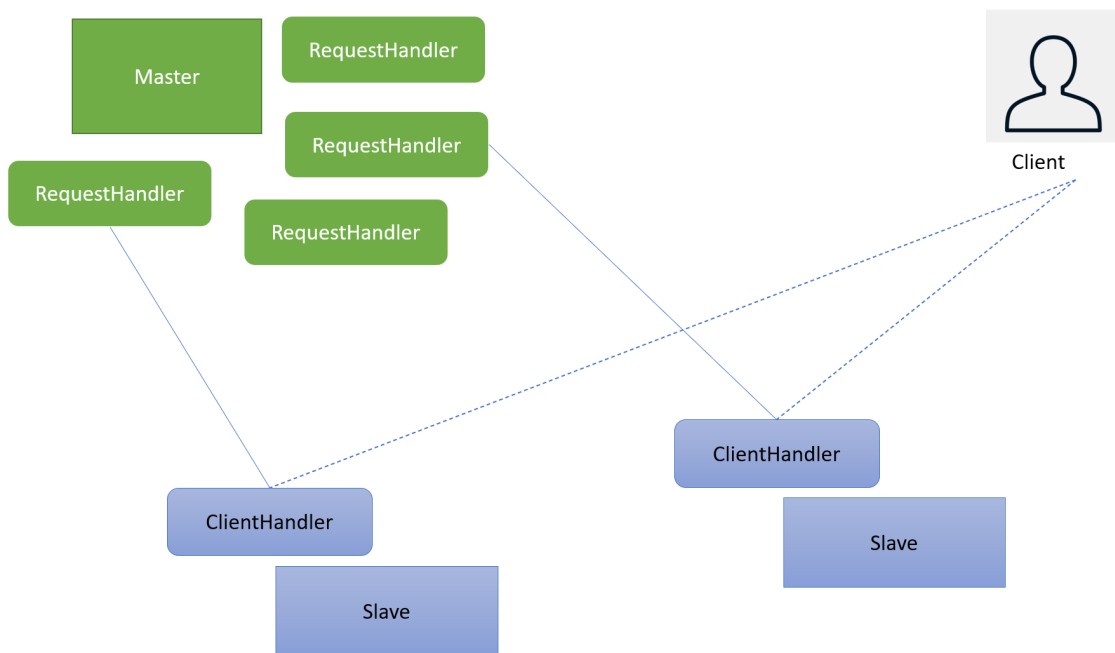


Abbildung: Schematischer Aufbau des verteilten Systems

Der Client könnte auch mit dem Master direkt in Verbindung treten, dieser Fall ist aus Gründen der Übersichtlichkeit herausgenommen worden. Bei der Kommunikation mit dem Slave, wird ein eigener Thread, der ClientHandler gestartet. Dieser kann gleichzeitig mit dem Client, sowie mit dem Master kommunizieren. Die Funktionalität des ClientHandlers ist ähnlich wie die des RequestHandlers.

Ein Slave ist für das Verteilen der Rechenlast zuständig. Er verständigt sich mit dem Master über den ClientHandler und mit dem Slave startet auch der DecryptionHandler.

## Aspekte

Das verteilte System ist für den Benutzer zum größten Teil transparent, was für ihn bedeutet, dass er keinen Einblick in die Abläufe oder in die Programmierung hat. Das hat zur Folge, dass das verteilte System immer für einen Client aktiv sein muss. Ein Client der keinen Einblick in das System hat, sieht beispielsweise nicht, dass das System gerade an Schreibzugriffen in Dateien arbeitet und fordert nun einen weiteren Dateizugriff an. In diesen Fällen muss ein verteiltes System Nebenläufigkeit unterstützen, was das verteilte System in diesem Projekt unterstützt. Die Klassen "ClientHandler", "RequestHandler" oder auch die Klasse DecryptionHandler arbeiten jedesmal in einem neuen Thread. Die Synchronisation beruht auf ständiger Kommunikation zwischen dem Slave oder dem Master, dabei speichert der Master die gegenwärtigen Teilnehmer ab.

## Kernabläufe

Der Kernablauf wird in diesem Diagramm dargestellt

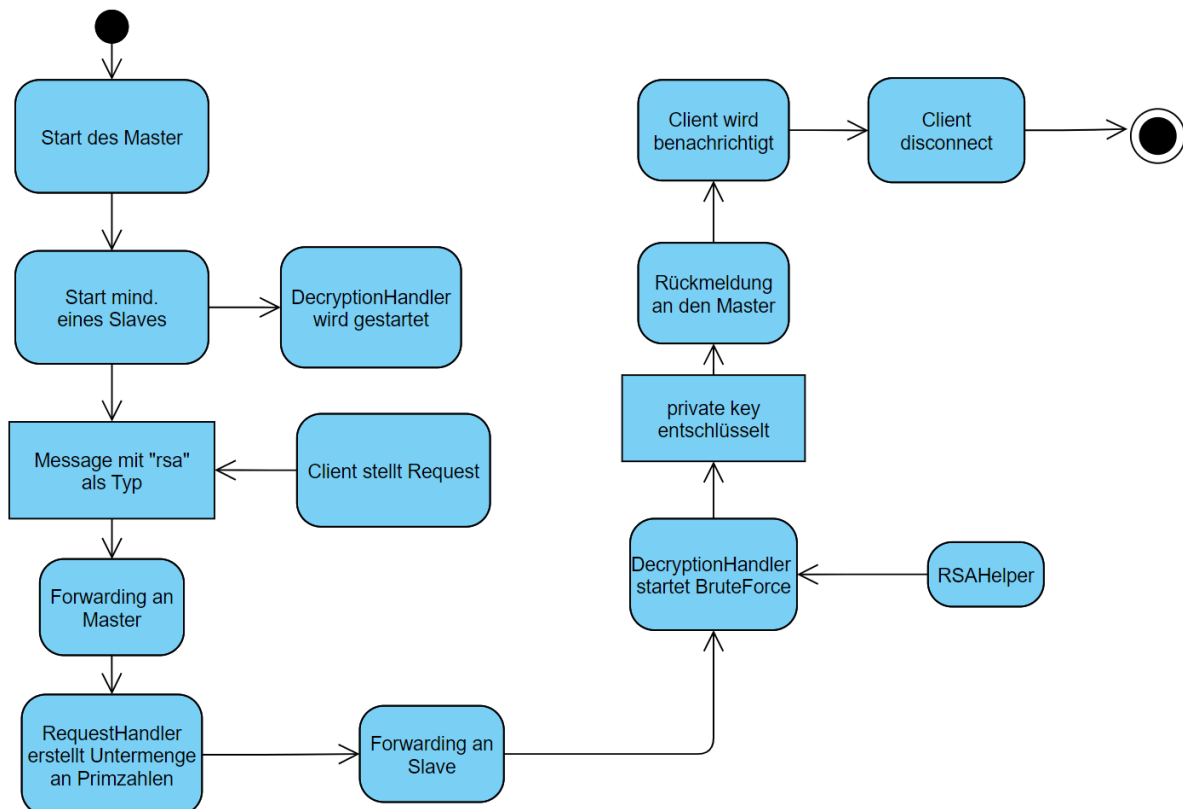


Abbildung: Kernablauf

## Schnittstellen

Schnittstelle	Partner	Parameter	Fehlerbehandlung
Client - Knoten	Client - ClientHandler	Socket + Stream + Message	
Master - Slave	ClientHandler - RequestHandler	Socket + Stream + Message	
Slave - Decryption	ClientHandler - DecryptionHandler	Socket + Stream + RSA-Payload	

Die Schnittstelle zwischen dem verteilten System und einem Endnutzer wird über die Client-Knoten-Verbindung symbolisiert. Ein Client verbindet sich über ein Socket und belegt dabei einen Port des verbundenen Knotens. Nach Abschluss der Anfrage trennt der Client seine Socket-Verbindung und der Port des ehemals verbundenen Knotens wird wieder freigegeben.

Die Schnittstelle zwischen dem Master und den Slaves bilden ebenfalls Socket-Verbindungen, welche durch Output- sowie InputStreams eine moderate Performance im Übertragen von Nachrichten aufweisen.

Die Nachrichten, welche über die über die Output- sowie InputStreams geschickt werden sind als Message-Objekte definiert. Diese beinhalten die Information, von wem das Message-Objekt gesendet wird, sowie wer es erhält. Außerdem wird ein Zeitstempel, eine Sequenznummer und der Typ gespeichert.

## Anpassungen

In der Konzeptentwicklung wurde der Fokus stark auf die funktionierende Kommunikation gelegt. Anschließend wurde das Projekt erst in der Umsetzung um die Berechnungs-Funktionalitäten erweitert.

Hierzu gehört die RSAPayload, eine Objektstruktur, welche es einfacher macht, die Ressourcen zur Berechnung bereitzustellen und zu versenden.

Der DecryptionHandler hat die Funktion, die Berechnung der vom Master an den Slave delegierten Payload gethreadet durchzuführen.



## 3. Programm

### Umsetzung der Forderungen

Text wird entschlüsselt:

Das verteilte System ist in der Lage, einen Chiffretext mit dem gegebenen Verfahren zu entschlüsseln

System besteht aus Client und Cluster:

Es gibt sowohl einen lauffähigen Client, als auch ein Cluster. Dieses besteht aus einem Master und mehreren Slaves.

Aufgabenteilung im Cluster

Der Master unterteilt die zu berechnenden Primzahlen und sendet an verschiedene Indizes an die Slaves. So berechnen alle Slaves unterschiedliche Bereiche der Primzahlen.

Ausfälle ausgleichen

Das verteilte System ist leider noch nicht in der Lage Ausfälle auszugleichen. Hierzu müsste man Ansätze wie das Heartbeat Prinzip einbinden.

### Umsetzung der Architektur

Die Umsetzung der Architektur erfolgte in Form von Klassen, welche die Knoten repräsentieren, als auch in den sogenannten Handlern, welche eingegangene Nachrichten gethreadet verarbeiten.

#### Application

Die Application ist die Klasse, welche die Initialisierung des verteilten Systems beinhaltet. In dieser werden zunächst die Ports der verschiedenen Komponenten und die Anzahl der Slaves festgelegt. Anschließend werden ein Master, die festgelegte Anzahl an Slaves und ein Client gestartet. Diese werden als Threads mithilfe der Runnable Klassen gestartet.

#### ApplicationClient, ApplicationMaster, ApplicationSlave

Über diese Klassen lassen sich Master, Slave und Client mit Programmanweisungen starten. Sie machen sich die Runnable Classes zunutze, um die Prozesse mit den Programmanweisungen zu konfigurieren und zu starten.

Des Weiteren wurden sie zur Generierung der JAR-Dateien Master.jar, Slave.jar und Client.jar verwendet.

#### FileEditor

Der FileEditor dient dazu, Dateien zu erstellen, zu beschreiben und auszulesen. Er wird zum einen vom RequestHandler dazu genutzt, die Kommunikation zu protokollieren und erhaltene Nachrichten mitsamt eines Zeitstempels abzuspeichern, zum anderen wird er dazu verwendet, die zu berechnenden Primzahlen aus den Dateien auszulesen

## Message

Message ist die Objektstruktur der Nachrichten, welche sich die Knoten gegenseitig senden. Diese enthält einen Sender, einen Empfänger und eine Payload, den Inhalt der Nachricht, sowie einen Zeitstempel. Zur Klassifizierung der Nachricht enthält diese außerdem einen Typ. Anhand des Typs können die Handler bestimmen, für welchen Zweck die Nachricht bestimmt ist und die dementsprechend weiterverarbeiten. Das Attribut SequenceNo soll die richtige Reihenfolge der Nachrichten gewährleisten und ist für die Ausfallsicherheit geeignet.

## Node

Node ist die Objektstruktur in welcher Informationen zu einer Verbindung gespeichert sind. Sie enthält den Port des Clients, einen Boolean, welcher Auskunft darüber gibt, ob der gespeicherte Knoten ein Master ist und enthält die Socketreferenz der Verbindung.

## RSAPayload

Die RSA Payload enthält die nötigen Informationen zur Berechnung. Hierzu gehören der Chiffretext, der öffentliche Schlüssel, die Liste der Primzahlen, sowie ein Start- und ein Endindex, damit der Slave weiß, welchen Bereich der Primzahlen er berechnen soll.

## Client

Der Client ist in der Lage, sich zu einem Port zu verbinden und diesem Nachrichten zu senden als Nachrichten von ihm zu erhalten. Er verfügt über die Funktion createRSA, in welcher er eine RSA Entschlüsselungsaufgabe mit verschiedener Anzahl zu berechnender Primzahlen basierend auf dem Eingabeparameter erstellt, welche er anschließend verschicken kann.

## Master

Der Master ist dafür zuständig, das Netz zu verwalten, Nachrichten, welche von einem Client über den Slave empfangen werden zu verarbeiten und die Aufgabe der Entschlüsselung aufzuteilen und an die vorhandenen Slaves zu verteilen. Sobald der Master eine neue Nachricht erhält, startet er einen gethreadeten Requesthandler, welcher die Nachricht verarbeitet. Der Master verfügt über eine Liste, in welcher alle laufenden RequestHandler gespeichert sind.

## Slave

Der verfügt über eine ähnliche Funktion, Nachrichten zu verarbeiten wie der Master. Er kann Nachrichten von einem Client empfangen, in welchem Fall er einen ClientHandler startet, welcher die Nachricht weiterverarbeitet und an den Master weiterschickt. Wenn er Nachrichten vom Master erhält, handelt es sich bei diesen entweder um eine Response an den Client, welche mithilfe des bestehenden Clienthandlers an den Client weiter geleitet wird oder eine RSA bezogene Nachricht. In diesem Fall erhält der Slave entweder eine Aufgabe vom Master, in welchem Fall er einen DecryptionHandler erstellt, welcher die Berechnung durchführt, oder eine Anweisung, die RSA-Berechnung abzuberechnen.

## ClientHandler

Der ClientHandler managed die Verbindung zum Client. Er verarbeitet die Nachrichten, welche vom Client an den Slave gesendet werden und leitet diese an den Master weiter. Er ist auch dafür zuständig, die Antworten des Masters an den Client zu senden, die Verbindung zum Client aufzubauen und diese auch wieder zu beenden.

## RequestHandler

Der RequestHandler wird vom Master beim Eintreffen einer neuen Nachricht in einem Thread gestartet, damit der Master mehrere Anfragen simultan bearbeiten kann. Er entscheidet wie oben beschrieben anhand des Typs der Nachricht, wie diese weiterverarbeitet wird.

## DecryptionHandler

Der DecryptionHandler wird vom Slave gestartet, wenn er eine RSAPayload vom RequestHandler erhält. Innerhalb des DecryptionHandlers wird der angegebene Bereich der Primzahlen an allen anderen Primzahl ausprobiert und mit dem öffentlichen Schlüssel verglichen. Ist die Lösung gefunden, so schickt der DecryptionHandler eine Nachricht an den Master, welcher daraufhin den anderen Slaves mitteilt, die Berechnung abubrechen.

## RunnableClassClient, RunnableClassMaster & RunnableClassSlave

Diese Klassen sorgen dafür, dass Client, Master und Slave gethreadet ausgeführt werden können.

Die Funktion `delegateConnections` findet sowohl im Slave als auch im Master Anwendung. Prinzip der Funktion ist es, eine Nachricht aus dem Verbindungskanal aufzunehmen, zu klassifizieren und einen gethreadeten Handler zu starten, welcher die Nachricht verarbeitet.

## Make-Anweisung

Systemvoraussetzungen: JDK 11 zum kompilieren und builden des Projekts.

Das Projekt benötigt die Dependency `bouncyCastle`, deren JAR-Datei `bouncyCastle.jar` GitHub- Repository oder in der Zip-Datei unter "jars" zu finden. Diese muss in das Projekt eingebunden werden. Sie wird benötigt, um die RSA Helper Klassen zu verwenden, welche zur Entschlüsselung des Chiffres dienen

Durch die Ausführung der Klasse `Application` lässt sich das Projekt in der IDE ausführen. `masterPort`, `startSlavePort`, `numberOfSlaves` und `amountOfPrimes` am Anfang der `main` Methode bestimmen die Parameter des Clusters und der zu berechnenden Parameter.

Des Weiteren lässt sich das Cluster über die `main` Methoden der Klassen `Master`, `Slave` und `Client` - in dieser Reihenfolge - starten. Die Methoden sind zu Demozwecken eingerichtet.

# Startanweisungen

Systemvoraussetzungen: JRE 11 auf dem Gerät installiert, welches die JAR-Dateien ausführt.

Die JAR-Dateien sind in der Zip-Datei oder im GitHub-Repository im Ordner "jars" zu finden.

## 1. Start des Masters:

Der Master ist über die *Master.jar* zu starten.

Programmargumente:

1. Port

Beispielaufruf per CMD: "java -jar Master.jar 8000"

## 2. Start der Slaves:

Der/die Slaves sind über die *Slave.jar* zu starten:

Programmargumente:

1. Anzahl an zu startenden Slaves
2. Slave Port (Bei mehreren Slaves werden alle folgenden Ports inkrementiert)
3. Master Host-Adresse
4. Master Port

Beispielaufruf per CMD: "java -jar Slave.jar 5 9000 localhost 8000"

## 3. Client starten:

Der Client ist über *Client.jar* zu starten.

Programmargumente:

1. Host-Adresse des Slaves
2. Slave Port
3. Anzahl der zu berechnenden Primzahlen (akzeptierte Zahlen sind 100, 1000, 10000, 100000)

Beispielaufruf per CMD: "java -jar Client.jar localhost 9000 1000"

## 4. Testplan

Eine implementierte Testklasse hat es nicht mehr in das finale Produkt geschafft, dennoch gibt es einige Testfälle, welche in diesem Projekt sehr nützlich sind und die im Folgenden erläutert werden.

### Extremwerttest:

Dieser Test überprüft die Belastbarkeit vom System. Denkbare Szenarien wären etwa eine sehr große Menge an Slaves zu generieren, oder eine sehr große Menge an Primzahlen zu berechnen.

### HW Fehler:

Ein HW Fehler, wie er etwa durch das durchtrennen eines Kabels verursacht wird, würde überprüfen, ob das verfahren automatisch zwischengespeichert oder ob es mit dem Ausfall eines Knotens zurechtkommt.

Im Falle unseres Projektes kann man diesen Test durchführen, indem die Verbindung zum Client trennt, während die Berechnung läuft, oder die Verbindung eines Slaves trennt, während er eine Berechnung durchführt. Den Client zu trennen testet vor allem die Stärke des umgesetzten Kommunikationsprotokolls, während die Trennung eines Slaves die Ausfallsicherheit testet

### Durchschnittswert-Test:

Mit diesem Test lässt sich überprüfen, ob das System in seiner Laufzeit dem Durchschnitt entspricht oder ob es davon abweicht.

### Lasttests:

Hier kann überprüft werden, mit wie vielen Knoten oder mit wie vielen Primzahlen das System lauffähig bleibt.

### Überlasttest:

Mit dem Überlasttest kann geprüft werden, ob das Cluster im Falle einer so großen Last, dass die Berechnung nicht mehr möglich wird, abstürzt Gegenmaßnahmen einleitet.

## 5. Nachbetrachtung

### Zusammenfassung der Ergebnisse

Zur Entschlüsselung des RSA-Verfahrens wurde in diesem Projekt ein verteiltes System entwickelt, dessen Ziel es ist, mithilfe der Verteilung von Arbeitslast diese rechenaufwändige Aufgabe zu beschleunigen. Das verteilte System basiert auf der Kommunikation von Servern innerhalb eines Netzwerks mit Hilfe von Socketkanälen. Um eine geordnete Delegation der Rechenaufgaben zu gewährleisten, wurde das Master-Slave Prinzip integriert. In dem verteilten System gibt es also einen Master, welcher die Anfragen eines Clients verarbeitet und mehrere Slaves, welche als Proxy Verbindung zwischen Server und Client und außerdem als bereitgestellte Rechenleistung dienen. Verbindet sich ein Client, tut er dies mit einem Slave, welcher einen ClientHandler aufruft, um die Verbindung mit dem Client zu verwalten, und die Nachrichten des Slaves an den Master weiterleitet. Der Master erstellt bei Ankunft einer neuen Nachricht einen RequestHandler, welcher den Typ der Nachricht überprüft und sie dementsprechend verarbeitet. So kann der Client beispielsweise Nachrichten schreiben, welche der RequestHandler in eine Datei einschreibt, ein Slave kann sich vom Master trennen. Der Master besitzt sowohl eine Liste mit allen aktiven RequestHandlern, als auch eine HashMap, in welcher aktive Verbindungen gespeichert werden. Entsteht eine neue Verbindung oder trennt sich eine bestehende Verbindung, so wird die HashMap von dem entsprechenden Requesthandler aktualisiert und der Master broadcastet die aktualisierte Map an alle verbundenen Slaves. Schickt ein Client eine Message mit einer RSA-Payload, welche berechnet werden muss, so delegiert der Requesthandler den Arbeitsaufwand an alle verbundenen Slaves. Erhält ein Slave eine RSA Payload, startet er einen DecryptionHandler. Der DecryptionHandler verarbeitet die RSA-PAYload. Er wendet die gegebenen Entschlüsselungsfunktionen auf den gegebenen Bereich der Primzahlen an und gibt dem verbundenen RequestHandler eine "success" Nachricht zurück, sollte er das eine passende Lösung gefunden haben. In dem Fall wird an alle verbundenen Slaves eine RSA-Stop Nachricht gebroadcastet, woraufhin diese ihre Berechnungen einstellen.

Das entworfene Programm funktioniert und die Berechnung kann auf verschiedenen Slaves parallel stattfinden.

### Zur Verbesserung

Es gibt Aspekte, in welchen das Programm sich noch verbessern kann.

Dazu gehört die Rückmeldung eines Slaves, sollte er es nicht geschafft haben, die Nachricht zu entschlüsseln und damit verbunden eine Nachricht an den Client im Falle, dass keiner der Slaves eine Lösung gefunden hat.

Ein weiterer Aspekt ist die Ausfallsicherheit. Zum jetzigen Zeitpunkt gibt es keine Maßnahmen im Falle eines unerwarteten Ausfalls eines Slaves. Diese können in Zukunft mit einem Heartbeat Prinzip realisiert werden. Ein Slave schickt in einem regelmäßigen Abstand eine Nachricht, einen sogenannten Heartbeat an den Master. Dadurch weiß der Master, dass der Slave noch am Leben ist. Sollte der Slave seit einigen Heartbeat Intervallen keine

Nachricht geschickt haben, so weiß der Master, dass dieser Slave ausgefallen ist und kann den entsprechenden RequestHandler schließen.