

# Ein Verteilter Zeitstempeldienst basierend auf RSA in Java

Diplomarbeit am Lehrstuhl Prof. Dr. J. Buchmann

Technische Universität Darmstadt

November 2001



Christian Valentin

Betreuer: Markus Ruppert

# Inhaltsverzeichnis

<b>1</b>	<b>Motivation</b>	<b>5</b>
1.1	Signaturen und Zeitstempel . . . . .	5
1.2	Secret Key und Public Key Kryptografie . . . . .	5
1.3	RSA als Public Key Krypto-Verfahren . . . . .	7
1.4	RSA Signaturen . . . . .	8
1.5	Die Time Stamp Authority (TSA) . . . . .	9
1.6	Einwegfunktionen, kryptografische Hashfunktionen . . . . .	9
1.7	Warum Java und nicht C . . . . .	10
1.8	Ziele des Entwurfs . . . . .	11
<b>2</b>	<b>Die verteilte TSA im Überblick</b>	<b>11</b>
2.1	Die TSA in ihrer nichtverteilten Variante . . . . .	11
2.2	Die verteilte TSA . . . . .	11
2.3	Eine TSA mit eigener Schlüsselgenerierung . . . . .	13
<b>3</b>	<b>Das Protokoll des verteilten RSA</b>	<b>13</b>
3.1	Motivation der verteilten Variante . . . . .	14
3.2	Eine Übersicht der beteiligten Komponenten . . . . .	14
3.3	Das Protokoll der verteilten Schlüsselerzeugung . . . . .	15
3.4	Das Verteilte Sieben (Distributed Sieving) . . . . .	16
3.5	Die Verteilte Berechnung des Moduls . . . . .	18
3.6	Der Verteilte Primalitätstest . . . . .	19
3.7	Verteilte Berechnung der Exponenten . . . . .	20
3.8	t-out of-k Secret Sharing . . . . .	21
3.9	Secret Sharing nach einer Shamir Variante . . . . .	21
3.10	Eine andere Variante des Secret Sharings (nach Sakurai) . . . . .	22
3.11	Die Verteilte Signatur . . . . .	22
3.11.1	Signatur nach dem Shamir ähnlichen Sharing Schema . . . . .	22
3.11.2	Signatur nach dem Sakurai Sharing Schema . . . . .	23
<b>4</b>	<b>Die Implentierung des Projekts Verteilte TSA</b>	<b>25</b>
4.1	Die Kommunikation zwischen den Komponenten . . . . .	25
4.1.1	Ein erster Versuch mit JAVA RMI . . . . .	25
4.1.2	Die Lösung mit TCP/IP Sockets . . . . .	26
4.2	Der SharedKeyGenerator . . . . .	26
4.2.1	Kommunikationsstrukturen des SKG . . . . .	27
4.3	Das Protokoll und seine Klassen . . . . .	30
4.4	Optimierung durch mehrere SKG Instanzen . . . . .	31
4.4.1	Die Klasse SkgInstance . . . . .	35
4.4.2	Die Klassen Distributed Sieving, ComputeN und das BGW . . . . .	37
4.4.3	Die Klassen TrialDivision und PrimeTable . . . . .	40
4.4.4	Die Klasse DistPrimeTest . . . . .	42
4.4.5	Die Klassen ExponentShareGeneration und SecretSharing . . . . .	46
4.5	Der ReceptionServer . . . . .	51
4.6	Die PartialSigningServer . . . . .	57

<b>5</b>	<b>Einige abschliessende Betrachtungen</b>	<b>58</b>
5.1	Implementierungen von GUIs zu Test- und Demonstrationszwecken	58
5.2	Gegenüberstellung des Shamir- mit dem Sakurai Schema . . . . .	59
5.3	Offene Fragen, Ausblicke . . . . .	59
5.3.1	Tests . . . . .	60
5.3.2	Die Zeitsynchronisation . . . . .	60
5.3.3	Weitere mögliche Optimierungen . . . . .	61
<b>A</b>	<b>Glossar</b>	<b>63</b>
<b>B</b>	<b>Klassenübersicht</b>	<b>64</b>
B.1	Package SharedKeyGeneration . . . . .	64
B.2	Package ReceptionServer . . . . .	64
B.3	Package ManagerServer . . . . .	65
B.4	Package PartialSigningServer . . . . .	65
B.5	Package JCA . . . . .	65
B.6	Package IETF . . . . .	65
B.7	Package GUI . . . . .	65
<b>C</b>	<b>Zeitmessungen zur Schlüsselgenerierung</b>	<b>67</b>
C.1	768 bit . . . . .	67
C.2	1024 bit . . . . .	68
C.3	1536 bit . . . . .	69
C.4	2048 bit . . . . .	70

## **Ehrenwörtliche Erklärung**

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 10. April 2002

## Vorwort

In dieser Arbeit wird ein verteilter Zeitstempeldienst beschrieben, der als zugrunde liegenden Kryptoalgorithmus eine verteilte Variante des RSA Algorithmus verwendet. Das entwickelte Programm wurde in JAVA implementiert und beinhaltet alle Schritte von der Schlüsselerzeugung bis hin zur pkcs kompatiblen Signatur mit Zeitstempel. Es wurde während der Entwicklung grosser Wert auf die Kompatibilität zur Java JCA gelegt, um eine leichte Anpassung des Programms in bestehende Systeme zu garantieren. Das Projekt wurde innerhalb eines Joint-Venture Projektes zwischen NTT Japan und dem Lehrstuhl für Theoretische Informatik, Kryptografie und Verteilte Systeme der TU Darmstadt entwickelt.

Mein Dank gilt dem TUD und NTT Team Dr. Ingrid Biehl, Prof. Johannes Buchmann, Arnulph Fuhrmann, Thilo Planz, Markus Ruppert, Prof. Sakurai, Gregor Schnee und Dr. Tzuyoshi Takagi.

Besonderer Dank gilt meiner Mutter, B.Valentin für ihre grosse Unterstützung während meines Studiums, Manuela Kernke für ihre Geduld und Hilfe und Cornelius (Neil) Morrissey für seinen einzigartigen Englischunterricht.

In Gedenken an meinen Vater : Dipl. med. A.M. Valentin † 26.03.1995

# 1 Motivation

In diesem Kapitel soll die Notwendigkeit der Verwendung einer Signier - und Zeitstempelanwendung beschrieben werden. Auf die zu grunde liegenden Verfahren soll kurz eingegangen werden, gerade im Hinblick auf die Besonderheiten der verteilten Implementierung im Gegensatz zu konventionellen, nicht verteilten Methoden. Desweiteren sollen kurz die Unterschiede zwischen Public Key und Secret Key Kryptografie erläutert werden, um die Verwendung der in diesem Projekt verwendeten Public Key Algorithmen zu erklären.

## 1.1 Signaturen und Zeitstempel

In einer Zeit der wachsenden Bedeutung des elektronischen Datenverkehrs entstehen neue Anforderungen an die zu übermittelnden Dokumente. Wie im normalen Schriftverkehr möchte man sich sicher sein, dass ein unterschriebenes (signiertes) Dokument gültig ist, der Unterzeichnende wirklich diese Willenserklärung zu dem vermerkten Zeitpunkt zu genau diesem, nachträglich nicht mehr abänderbaren Sachverhalt abgegeben hat. Der Zeitstempel muss in der Signatur enthalten sein, um den Zeitpunkt der Signierung zweifelsfrei und unverfälschbar zu dokumentieren. Sollte nachträglich das Originaldokument geändert werden, kann dieses Dokument nicht die gleiche Signatur ergeben. Ebenso ist es erforderlich, dass eine spätere Änderung der Signatur einfach zu erkennen ist, um Fälschungen zu vermeiden. Diese Überprüfung einer Signatur wird Verifikation genannt. Wird eine digitale Signatur konform zu dem deutschen Signaturgesetz (SigG) erstellt, kann sie vor Gericht als Beweismittel herangezogen werden. Bei Signaturen wird grösstenteils, wie auch in dem hier beschriebenen Projekt, die Public Key Kryptografie benutzt. Ein einfaches Beispiel soll die Anwendung von Zeitstempeln und ihre Wichtigkeit verdeutlichen.

### Beispiel 1.1.1

*Eine Person A sendet an ihre Bank B einen Überweisungsauftrag über 100 DM an Person C. Person C hört die Leitung ab, und kommt so in den Besitz des Überweisungsauftrages. Nun kann er beliebig oft diesen Überweisungsauftrag an die Bank übermitteln und sich immer wieder 100 DM von A's Konto gutschreiben lassen. Das würde sogar funktionieren, wenn die Überweisung signiert ist, da die Signatur von A erstellt wurde. Fügt man nun aber in den Auftrag einen Zeitstempel ein und signiert danach, kann die Bank beim Verifizieren die Duplikate erkennen und der Betrugsversuch scheitert.*

## 1.2 Secret Key und Public Key Kryptografie

In der Kryptografie gibt es zwei wesentliche Verfahren, welche sich vor allem in der Verwendung der Codier - und Decodierschlüssel, dem Austausch dieser Schlüssel zum Zwecke der verschlüsselten Kommunikation und den daraus resultierenden Attacken unterscheiden.

Zum einen gibt es die symmetrische (secret key) Kryptografie, bei der alle Schlüssel von den teilnehmenden Parteien geheimgehalten werden müssen. Wird ein Schlüssel bekannt, ist der jeweils andere Schlüssel entweder identisch oder kann ohne grossen Aufwand berechnet werden. Somit ist das Geheimnis gebrochen, eine weitere Verwendung dieser Schlüssel ist unmöglich. Oft sind auch die Algorithmen zur Ver- und Entschlüsselung geheim, was einen weiteren Schwachpunkt bedeutet, da eventuelle erfolgreiche Attacken eines Angreifers ebenfalls geheimgehalten werden, es somit sehr lange dauern kann, bis überhaupt bekannt wird, dass eine erfolgreiche Attacke möglich ist. Das nächste sehr schwierige Problem bei der symmetrischen Kryptografie ist der Austausch der geheimen Schlüssel. Hierzu müssen sichere Kanäle oder vertrauenswürdige Kurierere eingesetzt werden, da ein Abhören des Schlüsseltauschs oder eine Manipulation der Schlüssel während des Transports fatale Folgen hätte. Ebenso muss man sich sicher sein, dass die Person, der man den gemeinsamen geheimen Schlüssel schickt auch wirklich der gewünschte Kommunikationspartner ist.

### **Beispiel 1.2.1**

*Person A und Person B tauschen über ein (sehr einfaches) Verschlüsselungsverfahren ihre Nachrichten aus. Sie nummerieren die Buchstaben des Alphabets mit den Nummern 1 bis 26 durch und addieren auf die Nummer des Klartextbuchstabens eine Zahl, die sie miteinander vereinbart haben, diese Zahl ist der geheime Schlüssel. So wird aus dem Wort GEHEIM mit dem Schlüssel 5 das Wort 12 10 13 10 14 19. Beide Personen müssen diesen Schlüssel nun geheim halten, findet ein Angreifer bei einem der beiden den Schlüssel, ist der Schlüssel unbrauchbar. Wäre dieser geheime Schlüssel unkodiert über eine unsichere Leitung geschickt worden und ein Angreifer hat ihn mitgehört, wäre alle Geheimhaltung dahin. Nun könnte der Angreifer sogar Nachrichten verschlüsseln und Person A vortäuschen, Person B zu sein oder andersherum.*

Im Gegensatz dazu gibt es die asymmetrische Kryptografie, die public key Kryptografie. Hier ist nicht möglich, aus dem Chiffrier Schlüssel den Dechiffrier Schlüssel zu berechnen (daher auch die Bezeichnung asymmetrische Kryptografie). Daraus resultiert, dass man die Chiffrier Schlüssel öffentlich machen kann. Jede Person X kann nun mit diesem öffentlichen Schlüssel (PublicKey) eine Information verschlüsseln, die nur von einer Person Y mit passendem Dechiffrier Schlüssel entschlüsselt und gelesen werden kann. Es muss nur der Dechiffrier Schlüssel geheim gehalten werden. Als Grundlage für Public Key Kryptoverfahren werden keine geheimen Algorithmen gewählt, sondern Verfahren, die mathematisch relevant sind und von vielen Forschern ständig untersucht werden. Sollte eine Attacke gefunden werden, wird sie sehr schnell bekannt werden und Benutzer des nun nicht mehr sicheren Kryptoverfahrens können entsprechend reagieren.

### 1.3 RSA als Public Key Krypto-Verfahren

Für die hier beschriebene Anwendung wird das RSA Verfahren angewendet (RSA steht für die Entwickler des Verfahrens : Rivest, Shamir, Adleman). RSA gewinnt seine Sicherheit aus der Schwierigkeit der Faktorisierung grosser Zahlen. Wird ein Verfahren zur Faktorisierung grosser Zahlen entdeckt, was mathematisch von grosser Bedeutung ist, wäre RSA nicht mehr sicher. Ob es noch andere Methoden gibt, RSA zu brechen, ohne faktorisieren zu können, ist nicht bekannt. RSA ist somit nicht äquivalent zum Faktorisierungsproblem, sondern die Möglichkeit des Faktorisierens impliziert die Unsicherheit von RSA, jedoch ein Brechen von RSA mit anderen Methoden befähigt nicht zum Faktorisieren. Um RSA anzuwenden, benötigt man 2 Primzahlen  $p$  und  $q$ , mindestens in der Grösse 384 bit. Diese beiden Zahlen sind geheim zu halten. Nun berechnet man die Zahl  $N$ :

**Gleichung 1.3.1**  $N = p * q$

Diese Zahl  $N$  wird als der öffentlichen Modul bezeichnet.  $N$  sollte mindestens eine Bitlänge von 768 bit, besser jedoch von 1024 oder 2048 bit haben, dementsprechend muss die Grösse von  $p$  und  $q$  gewählt werden. Desweiteren bestimmt man die Zahl  $\varphi(N)$  mit

**Gleichung 1.3.2**  $\varphi(N) = (p-1)*(q-1)$

$\varphi(N)$  wird dabei als Gruppenordnung von  $N$  bezeichnet. Hat man  $\varphi(N)$  berechnet, kann man  $p$  und  $q$  löschen, eine weitere Speicherung ist nicht notwendig. Man benötigt nun noch 2 Zahlen  $e$  und  $d$ , welche

**Gleichung 1.3.3**  $d * e \equiv 1 \text{ mod } \varphi(N)$

erfüllen. Dazu wählt man eine Zahl  $e$ , für die gilt:

**Gleichung 1.3.4**  $1 < e < \varphi(N)$

und

**Gleichung 1.3.5**  $\text{gcd}(e, \varphi(N)) = 1$ .

gilt. Das Kürzel  $\text{gcd}$  steht hierbei für den grössten gemeinsamen Teiler (engl. *greatest common divisor*). Es sei noch gesagt, dass, um spezielle Attackeformen zu vermeiden (z.B. die low-exponent-attack), die Zahl  $e$  eine Grösse von ca. 16 bit haben sollte, oft findet die Zahl  $2^{16} + 1$  (die Primzahl 65537) als  $e$  Verwendung. Nun kann leicht ein  $d$  berechnet werden (z.B. mit dem erweiterten Euklidischen Algorithmus), das der in **Gleichung 1.3.3** geforderten Eigenschaft genügt. Diese Zahl  $d$  existiert immer, da **Gleichung 1.3.5** gilt. Nun ist  $d$  der geheime Schlüssel zum dechiffrieren,  $e$  ist der öffentliche Chiffrierschlüssel. Eine

Verschlüsselung sieht nun folgendermassen aus. Man besorgt sich den öffentlich erhältlichen Schlüssel  $e$  und berechnet :

**Gleichung 1.3.6**  $c = m^e \text{ mod } N.$

$c$  ist dann Ciphertext,  $m$  ist der Klartext. Nun schickt man  $c$  an die Person, welche den zu  $e$  korrespondierenden geheimen Schlüssel  $d$  besitzt. Diese berechnet dann :

**Gleichung 1.3.7**  $c^d = (m^e)^d = m^{ed} = m^1 = m \text{ mod } N.$

Somit kann der Besitzer des geheimen Schlüssels  $d$  den Klartext  $m$  aus dem Ciphertext  $c$  wiederherstellen. Ein potentieller Angreifer, der  $d$  nicht kennt, muss zum Entschlüsseln des Ciphertextes den geheimen Schlüssel  $d$  berechnen. Das kann er aber nur, wenn er in der Lage ist,  $N$  zu faktorisieren, da er sonst nicht das eindeutige Inverse von  $e \text{ mod } \varphi(N)$  berechnen kann, welches dann das gesuchte  $d$  ist. Solange also Faktorisieren schwer ist, kann man RSA mit diesem Ansatz nicht brechen. Wollte man etwa versuchen, einen 2048 bit Schlüssel durch ausprobieren zu brechen, müsste man ca  $2^{1024}$  Versuche machen, angenommen, jeder Versuch dauert 1 ms, bräuchte man mehr als  $5 \cdot 10^{297}$  Jahre, das Universum im Vergleich dazu besteht seit etwa  $4 \cdot 10^9$  Jahre, also auch bei viel schnelleren Rechnern und der Verwendung grosser Rechnercluster besteht kaum Aussicht auf Erfolg. Es gibt jedoch geschicktere Verfahren, mit denen man in wesentlich geringerer Zeit RSA Schlüssel brechen kann. Man bezeichnet zur Zeit Schlüssel ab 768 bit Länge als sicher, da (unter hohem Aufwand) 512 bit Schlüssel erfolgreich gebrochen wurden. Weitere interessante Ausführungen zu dem Thema RSA als Public Key Kryptoverfahren findet man in [Buch99] und [Schneier96].

## 1.4 RSA Signaturen

Es wurde im vorangegangenen Abschnitt gezeigt, dass sich RSA sehr gut zum Verschlüsseln von Nachrichten eignet. Doch man kann RSA auch sehr leicht zum Signieren benutzen. Dies soll hier erklärt werden. Ziel einer Signatur ist es, zweifelsfrei zu beweisen, das ein Dokument von einem bestimmten Besitzer stammt, bzw, dass dieses Dokument in genau dieser Form von einer bestimmten Person unterschrieben (signiert) wurde. Der Signierer benötigt dazu einen geheimen RSA Schlüssel. Dieser wird, wie im vorherigen Abschnitt beschrieben, erzeugt. Will man nun ein Dokument  $m$  signieren, berechnet man

**Gleichung 1.4.1**  $s = m^d \text{ mod } N.$

$s$  ist die Signatur. (In dieser Form ist das Verfahren noch anfällig für einige Attacken, auf Verbesserungen wird später noch eingegangen). Will nun eine Person B die Signatur einer Person A überprüfen, benötigt sie nur den öffentlichen Schlüssel  $(N, e)$ . Person B berechnet nun

**Gleichung 1.4.2**  $m = s^e \bmod N$ .

Wurde  $s$  korrekt gebildet, stimmt diese Gleichung nach Gleichung 1.3.7. Person B kann also sicher sein, solange RSA nicht gebrochen wurde, dass der signierte Text von Person A stammt und von ihr signiert wurde, da niemand sonst den geheimen Schlüssel  $d$  kennt oder berechnen kann.

## 1.5 Die Time Stamp Authority (TSA)

Um einen glaubwürdigen Zeitstempel zu vergeben, müssen mehrere Dinge garantiert werden. Die gestempelte Zeit muss unverfälschbar sein, das heisst, es müssen präzise Zeitnehmer vorhanden sein, die nicht manipulierbar sind. Wird das gestempelte Dokument signiert, muss man dieser Signatur trauen können. Es muss möglich sein, jederzeit einen Zeitstempel zu erstellen, ohne grosse Ausfallzeiten. All das lässt sich von privaten Nutzern und von vielen Firmen nicht oder nur sehr teuer und aufwändig realisieren. Deshalb gibt es Stellen, die diese Arbeit übernehmen. Eine solche Zeitstempelstelle nennt man auch eine Time Stamp Authority (TSA). Der Gesetzgeber erhebt strenge Auflagen (siehe [SigG]) an die TSAs, um die oben genannten Kriterien zu erfüllen. Möchte ein Kunde sein Dokument mit einem Zeitstempel versehen, schickt er das Dokument (oder dessen Hashwert) an die TSA. Die TSA erzeugt den Zeitstempel, verknüpft ihn mit dem Dokument und signiert das Resultat mit geheimen ihrem Schlüssel. Danach wird das gestempelte und signierte Dokument an den Kunden (requester, client) zurückgeschickt, der dieses Dokument für eventuelle Verifikationsanfragen aufbewahrt. Genauereres über TSAs kann man in [Val99] erfahren.

## 1.6 Einwegfunktionen, kryptografische Hashfunktionen

Ein weiterer wichtiger Baustein der Public Key Kryptografie sind die sogenannten Hash- oder Einwegfunktionen. Zuerst sollen die Begriffe Hashfunktion und Einwegfunktion erklärt werden. Danach werden ihre Anwendungsgebiete im Zusammenhang mit PK Kryptografie und somit mit hier beschriebenen Projekt beschrieben. Eine Hashfunktion  $H(M)$  verarbeitet eine beliebig lange Nachricht  $M$  und erzeugt als Ergebnis einen Hashwert  $h$  fester Länge.

**Gleichung 1.6.1**  $h = H(M), |h| = m$

Der Wert  $m$  ist die sogenannte Blocklänge. Die Blocklänge sollte aus Sicherheitsgründen eine Länge von mindestens 140 bit haben, um die sogenannte Geburtstagsattacke zu verhindern. (Näheres zu diesem Thema siehe [Buch99]) Wenn eine Hashfunktion eine Einwegfunktion sein soll, muss sie jedoch noch zusätzliche Kriterien erfüllen.

- Hat man ein  $M$  gegeben, muss sich daraus leicht  $h$  berechnen lassen.

- Hat man ein  $h$  vorliegen, muss es sehr schwer sein, ein  $M$  zu berechnen, für das gilt :  $H(M) = h$ .
- Hat man ein  $M$  gegeben, muss es sehr schwer sein, ein  $M'$  so zu berechnen, dass gilt  $H(M) = H(M')$ .

Die ersten zwei Forderungen definieren die Einwegeigenschaften der Funktion, den letzten Punkt nennt man die Forderung nach Kollisionsresistenz, möglichst keine zwei unterschiedlichen Nachrichten sollten denselben Hashwert haben. Wozu werden nun diese Einwegfunktionen benutzt. Als Beispiel kann man sich vorstellen, ein Wissenschaftler möchte seine neueste Erfindung patentieren lassen. Dies macht er, in dem er das Dokument über das Internet an das Patentamt verschickt, wo es mit einem Zeitstempel versehen, signiert und registriert wird. Nun hat aber ein eifersüchtiger Kollege die Möglichkeit, die Leitung des Wissenschaftlers abzuhören. Sendet der Wissenschaftler seine Erfindung im Klartext, kann der Kollege sie abfangen, lesen und unerlaubterweise schon für sich verwenden. Liest er jedoch nur den Hashwert des Dokuments kann er daraus nichts erkennen und wegen der Eigenschaften der Einwegfunktionen kann er den Originaltext auch nicht zurückgewinnen. Der Wissenschaftler kann aber auch nicht nachträglich sein Dokument verändern und behaupten, es so beim Patentamt eingereicht zu haben, da das veränderte Dokument auf Grund der dritten Eigenschaft der Einwegfunktionen nicht mehr den gleichen Hashwert wie das Original liefert. Jede auch noch so kleine Veränderung eines Dokuments, zu welchem bereits ein Hashwert berechnet wurde, liefert einen völlig neuen Hashwert. (Faustregel dabei ist, das die Änderung eines bits im Originaldokument eine mindestens 50 prozentige Änderung der Bitstruktur des Hashwertes ergibt). Somit kann also die Integrität von Dokumenten mit Hilfe von Einwegfunktionen geprüft werden. Ein weiterer Effekt bei der Verwendung von Hashfunktionen ist, dass der Hashwert viel kürzer ist als das Originaldokument, nämlich Blocklänge  $m$ -bit lang, was vor allem bei Netzübertragungen oder Speicherungen auf SmartCards grosse Vorteile bringt.

## 1.7 Warum Java und nicht C

Normalerweise wird für rechenaufwendige Programme meist die Programmiersprache C/C++ verwendet, da sie durch maschinennähere Programmierung einen schnellen und schlanken Code liefert, Java als Hochsprache, deren Bytecode vom Rechner interpretiert wird ist um einiges langsamer. Die Vorteile von Java liegen in der Plattformunabhängigkeit, den standardisierten Bibliotheken und der leichteren Wartbarkeit des Codes. So war dieses Projekt auch ein Test, um zu sehen, ob auch so rechenaufwendige Programme wie das Vorliegende, in Java mit vertretbaren Ergebnissen bezüglich der Effizienz realisierbar sind.

## 1.8 Ziele des Entwurfs

Das Projekt hatte mehrer Ziele, denen bestimmte Entwurfskriterien untergeordnet waren. Es sollte zum einen leicht pfleg- und erweiterbaren Code darstellen, zum anderen aber schlanke und schnelle Protokolle (sowohl mathematische als auch kommunikatorische) realisieren. Um eine möglichst breite Anwendungsfläche zu finden, sollte das Projekt nach aussen hin standardisierte Schnittstellen zur Verfügung stellen, welche die innere verteilte Struktur transparent erscheinen lassen. Somit wurden die Schnittstellen dem Java Cryptography Architecture (JCA) Standard angepasst, mehr über die JCA findet man unter [JCA]. Die einzelnen Teile, vor allem die des mathematischen Protokolls, sind so angelegt, dass man sie leicht austauschen oder erweitern kann. Die Codedokumentationen erfolgten parallel zur Entwicklung mit Javadoc. Die Struktur des gesamten Projekts ist modular, um schwer pflegbare Riesenklassen zu vermeiden. Während der Entwicklung wurden immer wieder Zeit- und Netzlastmessungen durchgeführt, um die Auswirkungen neuer Programmteile und optimale Parameter für den Programmablauf zu ermitteln.

## 2 Die verteilte TSA im Überblick

Im folgenden Kapitel wird die TSA mit ihren Komponenten und ihrer Wirkungsweise dargestellt. Dies soll dem besseren Verständnis der später aufgezeigten und detailliert beschriebenen Einzelkomponenten dienen.

### 2.1 Die TSA in ihrer nichtverteilten Variante

Als Basis soll hier die nichtverteilte TSA vorgestellt werden, um die Unterschiede zur verteilten TSA zu verdeutlichen. Die TSA besteht aus 2 wesentlichen Komponenten, dem Receptionserver und dem Signierserver. Der Receptionserver hat die Aufgabe, die Anträge der Kunden, welche via Internet gestellt werden, entgegenzunehmen und auf Ihre Richtigkeit zu prüfen. Sind alle Angaben verifiziert, versieht er das zu signierende Dokument mit einem Zeitspempel. Dann wird der Antrag an den Signaturserver weitergegeben. Dort wird die Signatur des Dokuments mit dem Zeitstempel erstellt und zum Receptionserver zurück geleitet. Dieser schickt das gestempelte und signierte Dokument zum Kunden zur Aufbewahrung zurück, der Vorgang ist erfolgreich beendet.

### 2.2 Die verteilte TSA

Bei der nichtverteilten Variante fällt sofort ein entscheidender Nachteil auf. Was geschieht mit dem Service, wenn der Signaturserver ausfällt oder korrupt ist. Der Service kann nicht aufrecht erhalten werden. Oftmals werden auf solche einzelnen Server mit sogenannte Denial of Service Attacken lahm gelegt, bei denen

kein Wissen durch die Attacke gewonnen wird, sondern nur die Verfügbarkeit des Servers gestört werden soll. Da aber eine zertifizierte CA nach SigG/SigV [SigG] nur geringe Ausfallzeiten haben darf, will man für hochverfügbare Services ein höheres Mass an Sicherheit erreichen. Dieses liefert die nun beschriebene Variante. Auch hier gibt es den Receptionserver, aber er bekommt nun ein erweitertes Aufgabengebiet. Im Gegensatz zur nichtverteilten Variante gibt es nun mehrere (in unserem Fall 3 oder 5) gleichberechtigte Signaturserver. Diese erhalten nun jeder vom Receptionserver das zu signierende und mit einem Eingangszeitstempel versehene Dokument. Die Signaturserver erstellen eine Teilsignatur (partial signature), da sie nur über einen Teilschlüssel des RSA Signaturschlüssels verfügen. Keiner der anderen (Teil) Signaturserver (PartialSigningServer, PSS) kennt den geheimen Teilschlüssel irgend eines anderen beteiligten PSS. Diese Teilsignaturen werden an den Receptionserver zurückgesendet. Dieser kann nun, ohne Kenntnis des des RSA Geheimschlüssels (RSA Secret Key) die Gesamtsignatur erstellen. (Wie genau das geschieht, wird später detailliert beschrieben). Der Receptionserver vergleicht aber nun auch die Zeitpunkte des Erhalts der einzelnen Teilsignaturen, um auszuschliessen, das sich einzelne Server zu lange im Besitz des Dokuments (für eventuelle Ausspääh oder Manipulationsversuche) befinden oder um zu erkennen, dass Leitungen zu einzelnen Servern gestört sind. Liegen Abweichungen in den Zeiten vor, welche Störungen vermuten lassen, kann der Receptionserver trotzdem eine gültige Signatur erstellen, da eine Mehrheit der teilnehmenden PSS ausreicht, eine gültige Signatur zu erstellen. Im Falle von 3

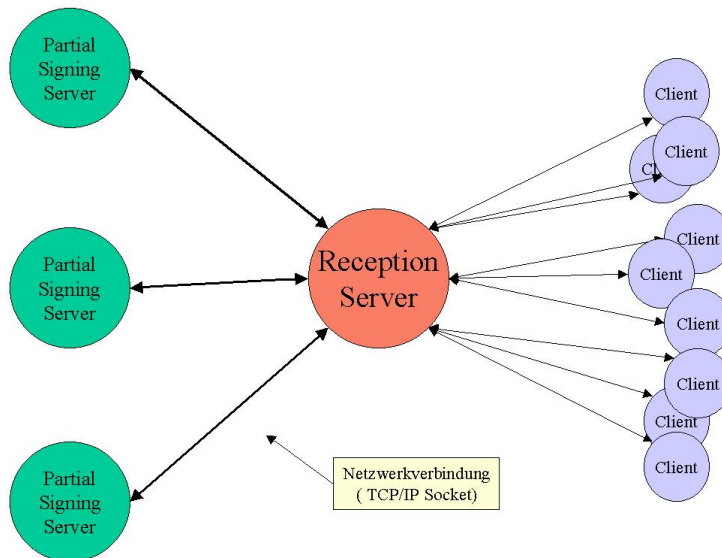


Abbildung 1: die verteilte TSA

verwendeten PSS darf also eine Station ausfallen, im Falle von 5 PSS genügen 3 funktionierende PSS, um den Service aufrecht zu erhalten. Dies erhöht die Aus-

fallsicherheit und Verfügbarkeit des Gesamtsystems und macht Attacken auf den Signaturserver schwieriger, da man nun die Majorität der der teilnehmenden PSS korrumpieren muss (im Gegensatz zur Einzelplatzlösung, die einen sogenannten single point of attack/compromise darstellt). Schafft man es, in einen der PSS einzubrechen, erlangt man kein genügendes Wissen, um den gesamten RSA Secret Key zu brechen.

### 2.3 Eine TSA mit eigener Schlüsselgenerierung

Um einen RSA Secret Key zu erhalten, muss man sich an eine Schlüsselgerierungsinstanz wenden, welcher man vertraut, eine sogenannte Trusted Authority (TA). Dieser Service ist im allgemeinen nicht kostenfrei und der geheime Schlüssel muss auch von dort geholt bzw gebracht werden, da ein Versand über das Internet zu unsicher ist. Wird der Schlüssel durch Attacken oder beschränkte Haltbarkeit unbrauchbar, muss er neu beantragt werden, der Signatur- und Zeitstempelservice ruht solange. Wird die TA erfolgreich attackiert, können in einigen Fällen alle ausgestellten Schlüssel ihre Gültigkeit verlieren, somit auch alle Zertifikate und Zeitstempel einer TSA, die diese Schlüssel benutzt. Um dieses zu verhindern, kann man eine eigene Schlüsselgenerierung benutzen. Dieses wurde in der beschriebenen Anwendung implementiert. Die Schlüsselgenerierung (Keygeneration) ist vom eigentlichen Zeitstempelprozess unabhängig und kann jederzeit parallel zum noch laufenden Service durchgeführt werden. Die Schlüsselgenerierung wird von einem Managerserver (MS) angestoßen, der jedoch nur die erforderlichen Parameter an die teilnehmenden Shared Key Generatoren (Skgs) übergibt. Die Skgs berechnen nun verteilt den RSA Key. Die fertigen Schlüssel werden auf einem Datenträger gespeichert und können dann von den PSS verwendet werden. Da die verteilte Schlüsselerzeugung (shared key generation = skg) wesentlich schwieriger und aufwendiger ist als für den nichtverteilten Fall und die Ausgangsbasis für den verteilten Signaturservice bildet, wird sie in einem eigenen Kapitel ausführlich beschrieben.

## 3 Das Protokoll des verteilten RSA

In diesem Abschnitt wird die verteilte Variante des in Kapitel 1 beschriebenen RSA Algorithmus vorgestellt. Die Ausarbeitungen stützen sich hierbei vor allem auf die Veröffentlichungen [MaWuBo99] und [BoFr97]. Danach steht vor allem die Implementierung dieses verteilten Verfahrens mit seinen Besonderheiten im Mittelpunkt. Einige der in diesem Kapitel enthaltenen Grafiken sind aus [Führ00] und [CDC] entnommen, dort kann man auch einführende und ergänzende Bemerkungen zu diesem Kapitel finden. Den Abschluss des Kapitels bilden einige Zeitmessungen, die während der Implementierung gemacht wurden und Vergleiche zu anderen Varianten bieten sollen.

### 3.1 Motivation der verteilten Variante

Der erste Gedanke wäre, man erstellt ein Schlüsselpaar auf einem Rechner und verteilt den Schlüssel dann in Teilstücken auf die teilnehmenden PSS. Aber dann hätte man das Problem des single point of attack/compromise nicht völlig beseitigt und es gäbe eine Instanz, die den kompletten RSA Schlüssel kennt, was eine Schwäche darstellt. Es muss also ein Protokoll benutzt werden, welches verteilt ein RSA Schlüsselpaar erstellt, so dass kein Rechner das ganze Geheimnis kennt. Die verteilte Variante soll also höhere Sicherheit und Verfügbarkeit zum Preis noch zu vertretender Effizienz und angemessen höherem Implementierungsaufwand liefern.

### 3.2 Eine Übersicht der beteiligten Komponenten

Um die Verständlichkeit des im Folgenden beschriebenen Protokolls zu verbessern, werden die an der verteilten Schlüsselgenerierung (Shared Key Generation) teilnehmenden Komponenten kurz vorgestellt und in ihrer Wirkungsweise beschrieben. Eine detailliertere Erklärung zu den Einzelkomponenten erfolgt in der Implementierungsbeschreibung in einem späteren Kapitel. Im Unterschied zu

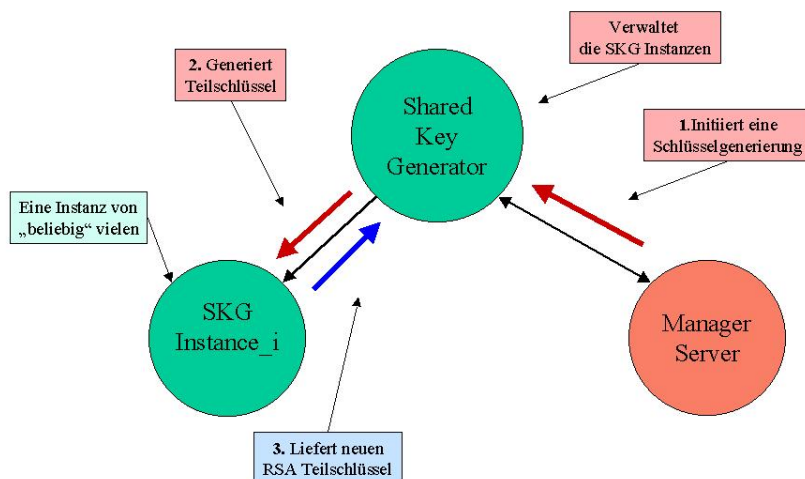


Abbildung 2: Eine erste Übersicht

[MaWuBo99] und [Führ00] wurden im Laufe des Projekts die PSS vom Prozess der Schlüsselgenerierung abgespalten. Sie dienen nur noch, wie der Name auch sagt, zum Erstellen partieller Signaturen. An der Schlüsselgenerierung neh-

men nun  $k$  (i.a.  $k=5$ ) Shared Key Generatoren (SKGs) und ein ManagerServer (MS) teil. Wenn im folgenden Kapitel von einem Server mit Index  $i$  gesprochen wird, so ist damit der  $SKG_i$  gemeint. Der ManagerServer dient lediglich zum Anstossen (Triggern) einer verteilten Schlüsselgenerierung. Der Administrator kann hier die Parameter wie Bitlänge des Schlüssels oder Anzahl der teilnehmenden Server (3 oder 5) angeben. Ausserdem kennt der MS die IP Adressen und Zielports der SGKs, welche sich untereinander vor dem Start der Schlüsselgenerierung nicht kennen. Sobald der MS die Schlüsselgenerierung ausgelöst hat, erhält er keinerlei Informationen über deren Fortgang noch über das Ergebnis. Die SKGs tauschen die während des Protokolls anfallenden Daten miteinander aus. Aus Effizienzgründen enthält jeder SKG mehrere Instanzen, welche er verwaltet und in denen die eigentliche Berechnung des Protokolls stattfindet. Darauf wird später noch genauer eingegangen. Während der Schlüsselgenerierung sieht das Kommunikationsmodell im Groben wie folgt aus. Der genaue Datenfluss wird im Kapitel Kommunikationsstrukturen und Datenflüsse erläutert.

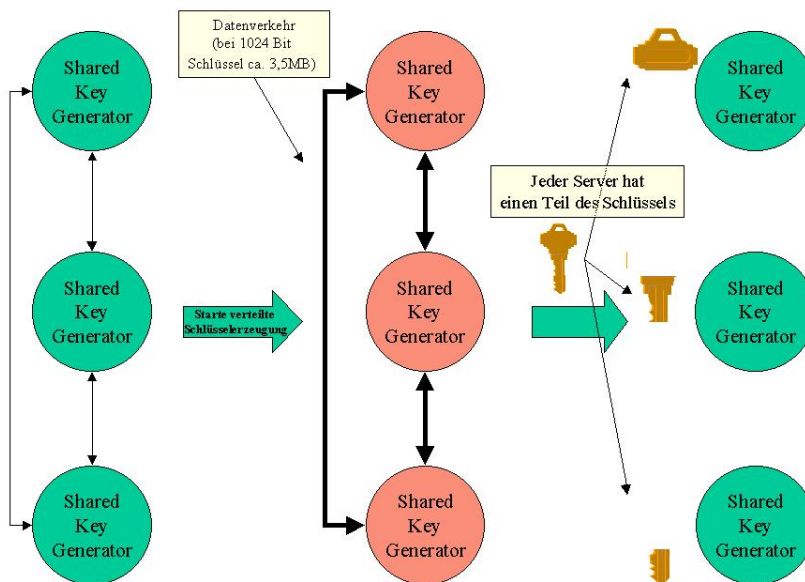


Abbildung 3: Der Datenfluss (grob)

### 3.3 Das Protokoll der verteilten Schlüsselerzeugung

Wie schon in 1.3 beschrieben, ist es das Ziel der RSA Schlüsselgenerierung, zu einem gewissen Modul  $N$  einen geheimen Schlüssel  $d$  und den dazu korrespondierenden öffentlichen Schlüssel  $e$  zu finden. Der Modul  $N$  ist das Produkt zweier Primzahlen  $p$  und  $q$ . Unter den Voraussetzungen der verteilten Variante sollen aber  $p$ ,  $q$  und  $d$  keinem der an der Schlüsselgenerierung teilnehmenden Server

gänzlich bekannt sein. Ein Protokoll hierfür liefern [MaWuBo99] und [BoFr97]. Es soll nun das Protokoll im Überblick dargestellt werden, bevor auf die Implementierung der einzelnen Protokollteile eingegangen wird. Dieses Bild zeigt

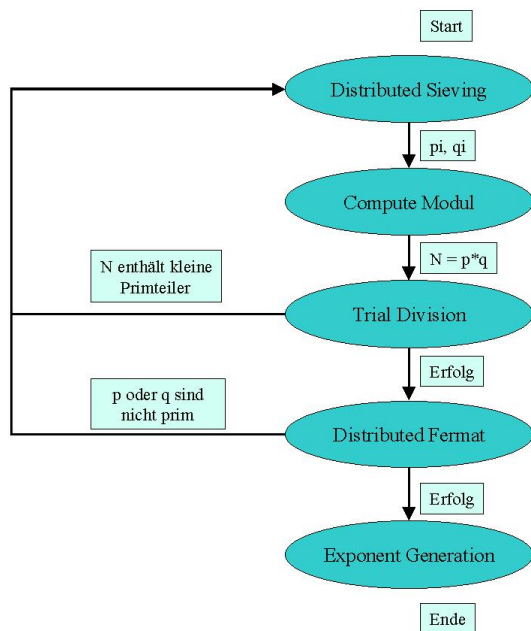


Abbildung 4: das verteilte Protokoll

die wesentlichen Schritte, die durchlaufen werden müssen, um das gewünschte Schlüsselpaar zu den Bedingungen der verteilten Variante zu erhalten. Es müssen zuerst geeignete Kandidaten  $p$  und  $q$  gefunden werden. Danach wird der Modul  $N$  berechnet. Nun wird getestet, ob der Modul  $N$  kleine Primteiler enthält. (kleine Primteiler heisst hier Primteiler in der Grössenordnung von  $ca\ 10^5$ ). Findet man Primteiler dieser Grössenordnung, muss man die Berechnung erneut mit anderen Kandidaten für  $p$  und  $q$  beginnen, da kleine Primteiler im Modul eine Schwäche darstellen und mit relativ geringem Aufwand auszurechnen wären. Findet man keine kleinen Primteiler, wird nun der Test auf Primalität von  $p$  und  $q$  vorgenommen. Fällt der Test negativ aus, muss wieder mit neuen Kandidaten die Berechnung begonnen werden, da sonst die RSA Eingangskriterien verletzt wären. Sind  $p$  und  $q$  prim, ist das Ziel erreicht, und es kann nun das Schlüsselpaar bezüglich des Moduls berechnet werden. Im folgendem sollen nun die einzelnen Schritte des Protokolls einzeln durchleuchtet und auf Besonderheiten in der Implementierung eingegangen werden.

### 3.4 Das Verteilte Sieben (Distributed Sieving)

Der erste Schritt im Protokoll ist das Finden der Kandidaten  $p$  und  $q$ . Diese setzen sich jedoch nun aus der Summe der Kandidaten der einzelnen Server zusammen,

dass heisst :

**Gleichung 3.4.1**  $p = \sum_1^k p_i$  ,  $q = \sum_1^k k_i$  mit  $k=5$ .

Die Summen  $p$  und  $q$  müssen aber nun, laut RSA Voraussetzung, prim sein. Im Folgendem gehen wir immer von  $k = 5$ , dass heisst von 5 Servern aus, da dies auch die Endkonfiguration des Projekts war. Der Betrieb für 3 Server ist aber ebenfalls problemlos möglich. Somit ergibt sich für den Modul  $N$  die Formel :

**Gleichung 3.4.2**  $N = (\sum_1^k p_i) * (\sum_1^k k_i) = p * q$ .

Der erste Ansatz war, einfach  $k$  Kandidaten  $p_i, k_i$  der entsprechenden Bitlänge zufällig auf den  $k$  Skgs zu wählen. Diese Methode führt nach genügend vielen Versuchen auch zum Erfolg, doch die Anzahl der Versuche ist sehr hoch. In der Protokollübersicht sieht man, dass noch einige Berechnungen folgen, bevor die Kandidaten selbst auf Tauglichkeit getestet werden, so dass bei vielen vergeblichen Versuchen auch ebenso viele nutzlose Berechnungen stattfinden. Um dies zu verhindern, untersucht man die zufällig gewählten Kandidatenbruchstücke sofort (bzw deren Summe), bevor man sie zu weiteren Berechnungen verwendet. Diesen Vorgang nennt man verteiltes Sieben. Die Summen  $p$  und  $q$  werden auf kleine Primteiler geprüft und unpassende, kleine Primteiler enthaltende Kandidaten werden ausgesiebt. Die Effizienz bei Benutzung dieser Methode ist ca. 10 mal höher als bei dem naiven Ansatz, einfach solange zu probieren, bis man passende Kandidaten gefunden hat ohne sie vorher zu testen. In der folgenden Beschreibung sei  $M$  das Produkt der kleinen Primzahlen bis zu einer bestimmten Grenze, der sogenannten Siebgrenze,  $M$  sei so gewählt dass  $M < p$ . (Da das Verfahren analog für  $q$  verläuft, sei hier nur der Fall für  $p$  beschrieben).

Jeder Server  $i$  generiert eine Zufallszahl  $a_i < M$  und  $a_i \neq 0$ , welche relativ prim zu  $M$  ist. Daraus folgt, das auch das Produkt  $a = a_1 \cdot \dots \cdot a_k \text{ mod } M$  relativ prim zu  $M$  ist. Man will nun die Verteilung der Faktoren dieses Produkts (multiplikative Verteilung) in eine Verteilung von Summanden (additive Verteilung) umwandeln, so das  $a = b_1 + \dots + b_k \text{ mod } M$  ist und jeder Server  $i$  ein solches  $b_i$  besitzt. Es sollen dabei keine Informationen über  $a$  an die einzelnen Server vermittelt werden. Initial wird bei Server 1 der Wert  $b_{1,1} = a_1$  gesetzt, die anderen Server setzen ihren Wert  $b_{1,i} = 0$ . Nimmt man für  $c_l = \prod_{i=1}^l a_i$  folgende Verteilung an :  $c_l = b_{1,l} + \dots + b_{k,l} \text{ mod } N$ , kann  $c_{l+1} = c_l \cdot a_{l+1} + 1$  recht einfach berechnet werden. Es wird der für die verteilte Berechnung des Moduls  $N$  beschriebene Algorithmus verwendet, mit der entsprechenden Eingabe :  $(b_{1,l} + \dots + b_{k,l}) \cdot (0 + \dots + a_{l+1} + \dots + 0) \text{ mod } M$ . In der Berechnung der des Moduls wird allerdings der errechnete Wert  $N$  öffentlich gemacht (broadcast). Dies will man hier nicht, die dazu nowtwendige Änderung wird ebenfalls in 3.5 beschrieben. Der Algorithmus liefert also die gewünschte additive Verteilung von  $c_{1,l+1}$ . Nach  $k-1$  Iterationen erhält man die additive Verteilung von  $a = b_1 + \dots + b_k \text{ mod } N$ . Es wird nun von jedem der Server  $i$  ein Zufallswert  $r_i \in [0, \frac{2^n}{M}]$  und setzt  $p_i = r_i M + b_i$ . Nun gilt  $p = \sum p_i = a \text{ mod } M$ .  $a$  hat keine kleinen Primteiler, somit hat auch  $p$  keine kleinen Primteiler. Es müssen noch kleine Anpassungen

erfolgen, da  $M$  nicht prim ist, jedoch alle Berechnungen modulo  $M$  erfolgen. Es muss der kleinste Primteiler von  $M$  grösser sein als die Anzahl  $k$  der beteiligten Server. (für  $k=5$  wäre also  $M$  also  $M = \prod_7^{p_i}$  wobei  $p_i$  die Siebgrenze darstellt) Es muss noch verhindert werden, dass  $p$  und  $q$  nicht die Faktoren 2,3 und 5 enthalten. Dies wird über eine Modifikation des  $r_i$  erreicht. Es wird zuerst ein  $m_i \in [0, \frac{2^n}{M \cdot 30}]$  gewählt. Dann ermittelt man einen Wert  $s_i = b_i * m_i^{-1} \text{ mod } 30$ . Server 1 berechnet nun  $r_1 = m_1 \cdot 30 + s_1 + 1$ . Für die anderen Server  $i$  gilt :  $r_i = m_i \cdot 30 + s_i$ . Nun kann das  $p_i$  wie oben angeführt berechnet werden. Sollten, was vorkommen kann, die Primfaktoren  $p$  und  $q$  nicht die richtige Bitlänge haben, müssen auch hier Massnahmen zur Garantierung der exakten Bitlänge getroffen werden. Dazu sei auf [Fuhr00] verwiesen.

### 3.5 Die Verteilte Berechnung des Moduls

Sobald alle  $k$  Server ihre Kandidaten  $q_i$  und  $p_i$  gefunden haben, soll der Modul  $N$  so berechnet werden, dass keiner der Server das Geheimnis der anderen Server erfährt. Das Protokoll für diese Berechnung beruht auf einer in [BGW88] und [MaWuBo99] beschriebenen Technik. Man wählt eine Zahl  $P \in \mathbb{P}$  mit  $P > N$ . (Alle in diesem Abschnitt angeführten Berechnungen werden, wenn nicht anders angegeben, modulo  $P$  gerechnet.) Das Protokoll verläuft in mehreren Schritten. Zuerst wählt jeder der  $k$  Server 2 Zufallspolynome  $f_i$  und  $g_i \in \mathbb{Z}_P[x]$  vom Grad  $l$ , mit  $l = \lfloor \frac{k-1}{2} \rfloor$ . Die Nullstellen von  $f_i$  und  $g_i$  sind  $f_i(0) = p_i$  und  $g_i(0) = q_i$ , die konstanten Glieder der Polynome sind also die Geheimnisse der Server. Die anderen Koeffizienten werden zufällig gewählt. Zusätzlich wählt jeder der Server ein weiteres Polynom  $h_i \in \mathbb{Z}_P[x]$  mit dem Grad  $2l$  und der Nullstelle  $h_i(0) = 0$ . Nun berechnen die  $k$  Server jeweils  $p_{i,j} = f_i(j)$ ,  $q_{i,j} = g_i(j)$  und  $h_{i,j} = h_i(j)$  mit  $j = [1..k]$ . Sind diese Werte errechnet, sendet Server  $i$  das Triple  $(p_{i,j}, q_{i,j}, h_{i,j})$  an alle Server  $j$  ausser im Fall  $j = i$ . Haben alle Server diese Tripel verschickt und die Werte der anderen Server erhalten, besitzt jeder Server  $i$  die Werte  $(p_{i,j}, q_{i,j}, h_{i,j})$  der  $j$  anderen Server mit  $j = [1..k]$ . Jeder Server berechnet nun eine Zahl  $N_i$  nach der Gleichung :

$$\textbf{Gleichung 3.5.1} \quad N_i = (\sum_j^k p_{i,j}) * (\sum_j^k q_{i,j}) + \sum_j^k h_{i,j}$$

Nach erfolgter Berechnung des  $N_i$  broadcastet der Server  $i$  diesen Wert an die anderen Teilnehmer. Nun hat jeder der teilnehmenden Server  $k$  Werte für  $N_i$ . Wenn man nun das Polynom

$$\textbf{Gleichung 3.5.2} \quad \alpha(x) = (\sum_j f_j(x)) * (\sum_j g_j(x)) + \sum_j h_j(x)$$

betrachtet, sieht man, dass  $\alpha(i) = N_i$  ist. Wegen der Definitionen von  $f_i$ ,  $g_i$  und  $h_i$  ist  $\alpha(0) = N$ .  $\alpha(x)$  ist ein Polynom vom Grad  $2l$  und  $l$  ist definiert als  $k \geq 2l + 1$ . Somit hat jeder Server mindestens  $2l + 1$  Punkte des Polynoms  $\alpha(x)$ . Somit kann das Polynom interpoliert und  $\alpha(0)$  berechnet werden.

Es soll noch angefügt werden, wie die Berechnung verändert werden muss, um

einen Wert *nicht* zu veröffentlichen. Dies wird beim Verteilten Sieben benötigt. Will man die Funktion  $N = \sum p_i \cdot \sum q_i$  so berechnen, das der Wert  $N$  nicht öffentlich ist, benutzt man folgende Modifikation. Alle Server haben am Ende ihrer Berechnungen einen Wert  $M_i$  :

$$\text{Gleichung 3.5.3 } \sum_{i=1}^k M_i = \sum_{i=1}^k p_i \cdot \sum_{i=1}^k q_i \text{ mod } P$$

Man unterbricht nun die Berechnung vor dem Broadcasten der Werte  $N_i$ . Jeder Server besitzt einen Punkt auf dem Polynom. Sei die Lagrangeform des Polynoms :

$$\text{Gleichung 3.5.4 } N = \alpha(0) = \sum_{i=1}^k \lambda_i(0) N_i \text{ mod } P$$

Dann ist  $\lambda_i(x) = \prod_{j \neq i} (x - j)(i - j)$  der entsprechende Lagrange Koeffizient. Jeder Server  $i$  kann also anstatt des Broadcasts den Wert  $M_i = \lambda_i(0) N_i$  setzen. Die daraus resultierenden Werte  $M_i$  sind dann die additive Verteilung von  $N$ . bevor man den verteilten Primalitätstest durchführt, kann zu Steigerung der Effizienz auf jedem Server eine Probedivision durchgeführt werden. Diese wird im Implementierungsteil näher beschrieben.

### 3.6 Der Verteilte Primalitätstest

Nach der Berechnung von  $N$  soll geprüft werden, ob  $N$  das Produkt zweier Primzahlen ist. Auch in diesem Schritt soll keiner der beteiligten Server etwas über die Geheimnisse der anderen Server erfahren. Der hier verwendete Test beruht auf dem Fermattest (siehe z.B. [Schneier96]). Bei dem Fermattest können bestimmte Zahlen fälschlicherweise Weise als Primzahlen identifiziert werden, die sogenannten Carmichael-Zahlen (näheres dazu z.B. in [Buch99]). Allerdings ist die Dichte dieser Carmichael-Zahlen sehr gering. So ist z.B. bei 1024 bit Zahlen die Wahrscheinlichkeit, eine Carmichael-Zahl zufällig zu finden ca.  $1:10^{40}$  und somit in dieser Betrachtung vernachlässigbar klein. Andere Tests, welche auch Carmichael-Zahlen finden, sind in [BoFr97] aufgeführt. Die Situation ist wie bisher, jeder Server  $i$  besitzt seine Geheimnisse  $p_i, q_i$ . Es soll nun festgestellt werden, ob das  $N$  mit

$$\text{Gleichung 3.6.1 } N = p^*q = (\sum p_i)^*(\sum q_i)$$

das Produkt der Primzahlen  $p, q$  ist, ohne das  $p, q$  berechnet werden. Dies verläuft in zwei Schritten. Im ersten Schritt wird eine Zufallszahl  $g \in \mathbb{Z}_N^*$  gewählt, welche allen Servern bekannt gegeben wird.

Im 2.Schritt berechnet Server 1 den Wert  $v_1 = g^x \text{ mod } (N)$  mit  $x = N - p_1 - q_1 + 1$ . Alle anderen Server berechnen den Wert  $v_i = g^y$  mit  $y = p_i + q_i$ . Die Server tauschen nun alle Werte  $v_i$  miteinander aus, so dass jeder Server folgende Gleichung berechnen kann.

$$\text{Gleichung 3.6.2 } v_1 = \prod v_i \text{ mod}(N), 2 \leq i \leq k$$

Stimmt diese Gleichung nicht, ist der Test fehlgeschlagen,  $N$  ist nicht das Produkt zweier Primzahlen. Die Berechnung muss von vorn begonnen werden. Im positiven Fall kann die Berechnung fortgeführt werden. Bei der Berechnung wurden keine Informationen über die Geheimnisse  $p_i, q_i$  preisgegeben.

### 3.7 Verteilte Berechnung der Exponenten

Zu diesem Zeitpunkt hat man nun den Modul  $N$  sowie die Primzahlen  $p = \sum p_i$  und  $q = \sum q_i$ . Für den vollständigen RSA Algorithmus benötigt man noch den öffentlichen Exponenten  $e$  und den geheimen Exponenten  $d$ , welche die Gleichung

**Gleichung 3.7.1**  $d = e^{-1} \text{ mod } \varphi(N)$

erfüllen müssen. Der geheime Exponent  $d$  muss wieder verteilt vorliegen ( $d = \sum d_i$ ) und zwar so, dass keiner der Server Informationen über die Teile (shares) von  $d$  der anderen Server kennt. In diesem Projekt wurde eine Lösung gewählt, welche nur für relative kleine  $e$  mit  $e \leq 2^{20}$  funktioniert, aber dafür sehr effizient ist (siehe auch [MaWuBo99]). Der relativ kleine Exponent stellt aber insofern keine Schwäche dar, als das bei einer RSA Signaturverifikation oftmals kleine öffentliche Schlüssel verwendet werden, um die Verifikation schnell durchführen zu können. Eine Lösung für allgemeingültige Exponenten kann in [BoFr97] gefunden werden. Es müssen vier Schritte vollzogen werden, um eine verteilte Exponentengeneration nach den oben angeführten Kriterien durch zu führen.

Im ersten Schritt berechnet Server 1 den Wert  $\varphi_1 = N - p_1 - q_1 + 1$ . Alle anderen Server berechnen  $\varphi_i = -p_i - q_i$ . Es ist leicht zu sehen, dass  $\varphi(N) = \sum \varphi_i$  ist. Im nächsten Schritt berechnen die Server einen Wert  $l = \varphi(N) \text{ mod}(e)$ . Da  $l = \sum \varphi_i \text{ mod}(e)$  ist, kann man  $l$  berechnen, ohne Informationen über die geheimen Shares preiszugeben. Zur Berechnung wird ein k-1 privates Protokoll benutzt, welches von Benaloh in [Ben86] entwickelt wurde. Jeder Server  $i$  bestimmt eine additive Verteilung seines  $\varphi_i$  dergestalt, dass  $\varphi_i = \sum_j \gamma_{i,j} \text{ mod}(e)$  für zufällige  $\gamma_{i,j}$  mit  $1 \leq j \leq k=5$  ist. Dabei werden  $j-1$  Werte per Zufall bestimmt, der  $j$ .te Wert wird aus der Differenz von  $\varphi_i - \sum_{j=1}^{j-1} \gamma_{i,j} \text{ mod}(e)$  gebildet und bezieht somit seine Zufälligkeit aus der Zufallsgenerierung der anderen  $j-1$  Summanden. Dann sendet jeder Server  $i$  seinen für  $\gamma_{i,j}$  berechneten Wert an Server  $j$ . Server  $j$  hat somit die Werte  $\gamma_{i,j}$  für alle  $i$ . Er berechnet dann eine Summe  $\alpha_j = \sum_i \gamma_{i,j} \text{ mod}(e)$  und sendet sein  $\alpha_j$  an alle anderen Server. Jeder Server berechnet nun lokal  $\sum_j \alpha_j$  welche die Gleichung  $\sum_j \alpha_j = \sum_j \varphi_j = l \text{ mod}(e)$  erfüllen.

Sei  $\zeta = l^{-1} \text{ mod}(e)$ . Dann ist  $d = (-\zeta \cdot \varphi + 1)/e$ . Somit kann jeder Server  $i$  lokal berechnen :  $d_i = \frac{-\zeta \cdot \varphi_i}{e}$ . Daraus ergibt sich  $d = \sum d_i + r \text{ mod } \varphi(N)$  mit  $0 \leq r < k=5$ . Es muss nun noch der Rest  $r$  ermittelt werden. Nutzt man die Gleichung  $c^d \equiv c^r \prod c^{d_i} \text{ mod}(N)$  kann einer der Server, z.B. Server 1 den Wert  $r$  leicht ermitteln, in dem er alle möglichen Werte für  $r$  austestet. Subtrahiert er dann Wert  $r$ , der die obige Kongruenz erfüllt, von seinem Teilstück  $d_1$  ist das endgültige Teilstück gefunden.

### 3.8 t-out of-k Secret Sharing

Es wurde nun beschrieben, wie ein k-out of-k sharing des privaten Exponenten funktioniert. Das bedeutet, man benötigt alle  $k$  Teilstücke des Schlüssels, um eine gültige Signatur zu erstellen. Man möchte aber aus verschiedenen Gründen oftmals eine andere Verteilung erreichen, bei der jede Untermenge der Teile (shares) ausreicht, eine gültige RSA Signatur zu erzeugen, ohne das jedoch Wissen über die Geheimnisse der anderen, nicht beteiligten Server erlangt wird. Ein Beispiel für eine solche Anwendung kennt man aus dem Bankgeschäft.

#### Beispiel 3.8.1

*In einer Bank hat der Tresor 5 Schlösser. 5 Angestellte haben jeweils verschiedene Schlüssel. Bei einem k-out of-k sharing müssten immer alle 5 Angestellten da sein, um den Tresor zu öffnen (5-out of-5). Das ist aber unpraktisch, da ein Angestellter krank sein kann oder anderwertig verhindert. Dafür benötigt man einen Mechanismus, dass zum Beispiel beliebige 3 (immer mehr als die Hälfte, Majoritätsprinzip) der 5 Angestellten ausreichen, um den Tresor zu öffnen (t-out of-k, 3-out of-5).*

In den Bereich einer verteilten Anwendung übertragen, liegt das Problem in der Verfügbarkeit der  $k$  teilnehmenden Signatur Server. Fällt einer der Server wegen Hardware- oder Verbindungsproblemen aus, kann der Service nicht mehr betrieben werden. Man sucht daher nach einer Möglichkeit, mit weniger als  $k$  Servern eine gültige, sichere Signatur zu erzeugen.

### 3.9 Secret Sharing nach einer Shamir Variante

Das hier verwendete Protokoll entstammt im Wesentlichen [MaWuBo99] und ist eine Variante des Shamir Secret Sharing (siehe [Stinson95]). Man kann die Shamir Variante nicht direkt übernehmen, da der geheime Schlüssel an *einem* Ort rekonstruiert werden müsste, und genau das soll ja während der gesamten Schlüsselerzeugung und späteren Nutzung nicht passieren. Es soll eine Variante für ein 2-out of-3 Sharing gezeigt werden, der Fall für 5 Server (3-out of 5) ist analog, jedoch als Beispiel etwas unübersichtlich. Man zerlegt den privaten Exponenten  $d$  in Teile so dass  $d = d_1 + d_2 = d_3 + d_4$  ist, ( $d_1 \dots d_4$  sind zufällig gewählt aus dem Bereich  $[-N, N]$ ). Die Bruchstücke verteilt man nun auf die 3 Server. S1 bekommt  $d_1, d_3$ , S2 erhält  $d_2$  und S3 schliesslich  $d_1, d_4$ . Nun kann keiner der Server den privaten Schlüssel  $d$  alleine bilden, jedoch jede Kombination aus 2 Servern kann eine vollständige Signatur liefern. In der verteilten Anwendung wird genau dieses Schema von jedem Server  $i$  auf seinen Share  $d_i$  angewendet und die entstehenden Sharebruchstücke  $d_{i,j}$  an die entsprechenden Server  $j$  verschickt. Somit können auch im Falle ausfallender Stationen, solange die Majorität noch funktionstüchtig ist, gültige Signaturen erstellt werden.

### 3.10 Eine andere Variante des Secret Sharings (nach Sakurai)

Herr Prof. Dr. Kouichi Sakurai entwickelte Ende 2000 eine andere Variante zum *t-out of-k* secret sharing für einen verteilten RSA. Diese Methode wurde in dem hier beschriebenen Projekt erstmals implementiert. Nach der Beschreibung wird deshalb auch eine kurze Analyse der Sakurai Methode erfolgen und der in [MaWuBo99] verwendeten Shamir Variation entgegen gestellt. Man berechnet zuerst die shares von  $d$  wie in bisher beschrieben. Man erhält dann also die 5 shares  $d_1 \dots d_5$  mit  $d = d_1 + d_2 + d_3 + d_4 + d_5$ . Im nächsten Schritt müssen die SKGs vorbereitet werden. Dies geschieht in 3 Schritten.

Jeder  $SKG_i$  berechnet lokal und privat (geheim) 2 zufällige Ganzzahlen  $a_1^{(i)}$ ,  $a_2^{(i)} \in (-2^{2L}, 2^{2L})$   $L = \text{bitlength}(p, q)$ . Danach (immer noch privat) wird ein Polynom der Gestalt  $f^{(i)}(x) = d_i + a_1^{(i)}x + a_2^{(i)}x^2$  berechnet. Dieses Polynom wird nun lokal an den Stellen 1 bis  $k$  berechnet,  $f^{(i)}(1), \dots, f^{(i)}(5)$ . Nun werden diese Werte  $f^{(i)}(j)$  an jeden  $SKG_j$  (mit  $1 \leq j \leq k=5$  und  $i \neq j$ ) geschickt. Jeder  $SKG_i$  ist dann im Besitz der Werte  $f^{(1)}(i), f^{(2)}(i), f^{(3)}(i), f^{(4)}(i), f^{(5)}(i)$  und kann das folgende Polynom berechnen :  $f(i) = f^{(1)}(i) + f^{(2)}(i) + f^{(3)}(i) + f^{(4)}(i) + f^{(5)}(i)$ . Das Resultat ist eine Ganzzahl  $f(i)$  und stellt den neuen secret share des  $SKG_i$  (und später des korrespondierenden PSS) dar (mit  $i = 1 \dots k=5$ ).

### 3.11 Die Verteilte Signatur

Es soll nun beschrieben werden, wie die nach dem in Kapitel 3 beschriebenen Protokoll berechneten Teilschlüssel zur Signatur genutzt werden. Auch wird auf einige Besonderheiten eingegangen, welche für eine Anwendung in der Praxis notwendig sind, im rein mathematischen Protokoll wie z.B. in [MaWuBo99] jedoch keine Rolle spielen.

#### 3.11.1 Signatur nach dem Shamir ähnlichen Sharing Schema

Man kann hier nochmal 2 Fälle unterscheiden. Einmal den Fall, dass man kein *t-out of-k* Sharing benutzt, also davon ausgeht, dass alle  $k$  PSS verfügbar sind und korrekt arbeiten. Benutzt man diese Variante, ist der Signiervorgang sehr einfach, der RS schickt die zu signierende und zu stempelnde Nachricht  $M$  an die  $PSS_i$ . Diese berechnen die Teilsignaturen  $S_i = M^{d_i}$ . Jeder  $PSS_i$  sendet sein  $M_i$  zurück an den RS. Dieser berechnet die endgültige Signatur  $S = \prod_i S_i$ . Löst man die Gleichung auf, erkennt man schnell, dass hier die Multiplikativität des RSA ausgenutzt wird, denn :  $S = M^{d_1} \cdot \dots \cdot M^{d_k} = M^{d_1 + \dots + d_k} = M^d$ , da  $d = \sum_i d_i = d_1 + \dots + d_k$  ist.

Benutzt man das modifizierte Shamir Secret Sharing, dann muss der RS eine Kombination teilnehmender  $PPS_i$  bestimmen. Jeder  $PSS_i$  verfügt nun ausser über seinen Share  $d_i$  ausserdem über 2 Sharebruchstücke  $d_{ix}, d_{iy}$ , wobei  $x$  und  $y$  die Indices der *nicht* teilnehmenden Server sind. Es ergibt sich dann :  $S = M^{d_1 + d_{1x} + d_{1y}} \cdot \dots \cdot M^{d_t + d_{tx} + d_{ty}}$ . Da die Sharebruchstücke  $d_{ix}$  und  $d_{iy}$  als Summe

die Shares  $d_x$  und  $d_y$  ergeben, wird die Gleichung  $S = M^d$  wieder gehalten. Allerdings muss in dieser Variante den teilnehmenden  $PSS_i$  die Kombination der Teilnehmer mitgeteilt werden, damit sie die passenden fehlenden Sharebruchstücke der Nichtteilnehmer richtig auswählen, das heisst, der RS muss für jeden Signaturauftrag die momentan aktuelle Kombination der teilnehmenden PSS an die  $t$   $PSS_i$  mitsenden.

### 3.11.2 Signatur nach dem Sakurai Sharing Schema

Es muss zuerst der ReceptionServer (RS) auf das Sakurai Schema vorbereitet werden. Dafür verwendet man folgendes Parameter-Set.

$$\Delta a + eb = 1$$

$$\Delta = \prod_i i = 1 * 2 * 3 * 4 * 5 = 120$$

$$e = 2^{16} + 1 = 65537$$

$$a = 3823$$

$$b = -7$$

Desweiteren benötigt man  $\Lambda(I, i) = \Delta \prod_j \frac{-j}{i-j}$  mit  $j \in I \setminus \{i\}$  und  $I \subset \{1, \dots, k=5\}$  und  $\#I = 3$ . Ein Beispiel für ein solches Parameter-Set sei hier gegeben :

#### Beispiel 3.11.1

Sei  $I = \{1, 3, 5\}$ , dann ergeben die Berechnungen nach obigem Muster :

$$\Lambda(I, 1) = \Delta \frac{-3}{1-3} \frac{-5}{1-5} = 225$$

$$\Lambda(I, 3) = \Delta \frac{-1}{3-1} \frac{-5}{3-5} = -150$$

$$\Lambda(I, 5) = \Delta \frac{-1}{5-1} \frac{-3}{5-3} = 45$$

Sei  $M$  nun eine Nachricht, die signiert werden soll. Der ReceptionServer RS sendet  $M$  an alle  $PSS_i$ . Jeder der  $PSS_i$  berechnet :  $S_i \equiv M^{f(i)} \pmod{N}$  und sendet dieses Ergebnis  $S_i$  zurück zum RS. Der RS wählt nun zufällig eine Kombination  $I \subset \{1, 2, 3, 4, 5\}$  mit  $\#I = 3$ . Dann berechnet der RS  $S(I) \equiv \prod_i S_i^{\vartheta} \pmod{N}$  mit  $i \in I$  und  $\vartheta = \Lambda(I, i)$ . Nun kann die vollständige Signatur  $S$  für die Nachricht  $M$  durch den RS berechnet werden. Dies erfolgt über die Gleichung  $S \equiv S(I)^a M^b \pmod{N}$ . An dieser Stelle muss man jedoch zwischen der reinen Zahlentheorie und der Anwendung in der Praxis unterscheiden. Zahlentheoretisch ist diese Gleichung problemlos anwendbar. In der Praxis jedoch wird, um eine pkcs#? gemässe Signatur zu erzeugen, die Nachricht  $M$  als Hashwert (z.B. MD5) auf den PSS verarbeitet. Dort wird vor dem Hashen noch ein padding durchgeführt, um bestimmte Attacks (z.B. existentielle Fälschung, siehe [Buch99])

zu vermeiden und um dem pkcs1 Standard zu genügen. Nehmen wir an, die Paddingfunktion heiße  $p$  und die Hashfunktion heiße  $H$ . Dann ist der in den  $PSS_i$  verarbeitete Wert nicht  $M$ , wie vom RS übermittelt, sondern  $H(p(M))$ . Dem muss Rechnung getragen werden, da sonst die Kongruenz der Verifikation  $S^e \equiv M \pmod{N}$  nicht gilt. Es muss nun heißen  $S \equiv S(I)^a H(p(M))^b \pmod{N}$ . Auch die Verifikationskongruenz muss angepasst werden, sie lautet nun  $S^e \equiv H(p(M))$ . Zu diesem Zweck muss mindestens einer der  $PSS_i$  (mit  $i \in I$ ) sein  $H(p(M))$  zusätzlich zu dem von ihm errechneten Wert  $H(p(M))^{d_i}$  an den RS zurückschicken, um diesen Wert dem RS für die Berechnung der endgültigen Signatur und deren Verifikation zur Verfügung zu stellen. Dafür entfällt aber hier das Versenden der momentanen Teilnehmerkombination an die  $PSS_i$ , da die Sharebruchstücke der nicht teilnehmenden PSS bereits in den Shares  $f(i)$  eines jeden PSS enthalten sind.

## 4 Die Implementierung des Projekts Verteilte TSA

In diesem Kapitel soll auf Details der Implementierung eingegangen werden. Zuerst wird ein grober Überblick auf die Komponenten gegeben und die Grundlagen der Kommunikation erläutert, dann werden die einzelnen Komponenten spezieller analysiert. Wie schon in Kapitel 3 erwähnt, wurde eine Aufteilung der Komponenten nach Signaturteil und Schlüsselgenerierungsteil vorgenommen. Ferner wurde auch ein Testclient entwickelt, der den Kunden für eine Zeitstempelung simulieren soll. Diese Teile werden nun einzeln vorgestellt sowie ihre Schnittstelle, nämlich die Schlüsselübergabe, beschrieben.

### 4.1 Die Kommunikation zwischen den Komponenten

Da bei allen Prozessen, sei es die Schlüsselgenerierung oder die Signaturerstellung mit Zeitstempel, Kommunikation zwischen den beteiligten Stationen notwendig ist, war es wichtig, die Kommunikationswege schnell, aber auch flexibel zu halten. Ausserdem soll die Kommunikation für die benutzenden Klassen so transparent wie möglich sein. Das heisst, das eigentliche Einrichten von Verbindungen, Senden und Empfangen von Daten soll in gekapselten Kommunikationsklassen verlaufen, die für die Berechnungen zuständigen Klassen sollen lediglich eine abstrakte Sicht (etwa : sende diese Daten an  $Server_i$ ) haben. Das macht die einzelnen Klassen leichter pflegbar und erlaubt den einfacheren Austausch oder Änderungen der Kommunikationsklasse möglich, da sich für die nutzenden Klassen aus ihrer Sicht nichts ändert. Diesem Aspekt wurde viel Aufwand zudedacht, da er ein Kernstück der Anwendung darstellt. Da das verwendete mathematische Protokoll, wenn alle Kommunikationskanäle gleichzeitig ausgespäht werden, nicht mehr sicher ist, wurde die gesamte Kommunikation mit SSL zur Installierung sicherer Kanäle betrieben. Als SSL Suite wurde die Implementierung von IAIK ([IAIK] verwendet.

#### 4.1.1 Ein erster Versuch mit JAVA RMI

In der Frühphase des Projekts wurden Versuche unternommen, die Kommunikation über die JAVA RMI Schnittstelle zu realisieren. JAVA RMI ist ähnlich den aus der C-Welt bekannten RPCs. Man kann mit Hilfe dieses Mechanismus eine Funktion auf einem entfernten (remote) Rechner aufrufen, mit Übergabeparametern und Rückgabewerten, so als wäre diese Funktion Bestandteil im lokal ausgeführten Programm. Die Implementierung solcher RMI Prozeduren ist sehr einfach und benötigt kein eigenes Kommunikationsprotokoll. Somit wurden erste Schritte der Schlüsselgenerierung mit RMI Prozeduren entworfen. Dies funktionierte auch wunschgemäss mit wenig Aufwand, bis der Punkt erreicht wurde, an dem echt verteilt gerechnet wurde, und die Ergebnisse einer Berechnung eines  $SKG_i$  Grundlage für weitere Berechnungen der anderen  $SKG_j$  waren,  $SKG_i$  aber ebenfalls auf Ergebnisse der Berechnungen anderer  $SKG_j$  zum Weiterrechnen

benötigte. Hier ergibt sich das Problem, dass eine Funktion eines  $SKG_i$  Ergebnisse der anderen  $SKG_j$  benötigt, um weiterzurechnen. Nun muss sie einen SKG nach dem anderen nach dem Ergebnis fragen. Geschieht dies über einen remote function call, bleibt die aufrufende Funktion solange blockiert, bis das Ergebnis der Remote Funktion vorliegt. Dies kann manchmal jedoch dauern, die anderen SKG haben ihr Teilergebnis vielleicht schon fertig, aber es wird nicht abgerufen, da die Funktion noch auf das vorher abgefragte Argument wartet. Nun können aber die bereits fertigen Funktionen der anderen SKG nicht fortfahren, da  $SKG_i$  noch kein Ergebnis geliefert hat, er wartet ja noch auf das erste Teilergebnis. Dies ist ineffizient und blockierend, kann im Extremfall sogar zu Verklemmungen und Deadlocks führen. Diese Art der Kommunikation, bei der eine Station blockiert bleibt bis zum Empfang der Antwort auf eine ausgesendete Nachricht, auch synchrone Kommunikation genannt, ist für unser Projekt nicht praktikabel. Ausserdem erwies sich die Geschwindigkeit unter RMI als wesentlich geringer als z.B. bei TCP/IP Socket Verbindungen. Mehr zum Thema RMI/RPC und synchroner Kommunikation kann man in [Mattern97] nachlesen.

#### 4.1.2 Die Lösung mit TCP/IP Sockets

Im Laufe der Implementierung wurde klar, dass die anfallenden Daten sofort abgesendet und empfangen werden müssen, ohne dabei Blockaden zu erzeugen oder den Berechnungsablauf zu stören. Ob die Gegenseite bereits die Daten benötigt oder sie diese noch zurückstellen kann, soll für den Sender dabei keine Rolle spielen. Diese Art der Kommunikation (fire and forget, asynchron) lässt sich am besten über TCP/IP Socket realisieren. Hierbei werden die Daten an die Kommunikationsenden, die Sockets, weitergereicht. Diese übernehmen Versand und Empfang, so dass die laufenden Berechnungen nicht warten müssen, bis etwas versendet oder empfangen wurde. Nach der Übergabe der Daten an die Sockets kann mit der Berechnung fortgefahren werden. Man sollte hier unbedingt gepufftere (buffered) Übertragungsmodi benutzen, da es sonst zu erheblichen Geschwindigkeitseinbußen aufgrund eines erhöhten Kommunikations- und Verarbeitungsoverheads kommt. Bei der Verwendung von Sockets muss zwar ein anfänglich aufwendigeres Kommunikationsprotokoll erstellt werden, doch das Vermeiden von Blockaden und die wesentlich höhere Geschwindigkeit rechtfertigen diesen Aufwand.

## 4.2 Der SharedKeyGenerator

In Kapitel 3 wurde bereits eine Bild gezeigt, welches den SharedKeyGenerator und seine Stellung innerhalb des Schlüsselgenerierungsprozesses zeigt. Die Klasse `SharedKeyGenerator` stellt den Ausgangspunkt für das Package der Schlüsselgenerierung dar. Man kann bereits erkennen, welche Komponenten der SharedKeyGenerator benötigt. Es ist zum einen eine Kommunikationsschnittstelle notwendig, um mit den anderen SKGs Daten auszutauschen und die notwendigen Informationen über die Parameter der Schlüsselgenerierung und deren



jekts benutzt wurde. Um ein leichtes Routing der Datenpakete zu ermöglichen, werden alle Daten (Kommandos) durch eine eigene Klasse `DTSSCommand` dargestellt, in welchem neben Steuerinformationen wie Sender, Empfänger (physisch als Rechner eines Netzes) oder Broadcast die Daten und der Empfänger (logisch als Teil des Protokolls) des Kommandos innerhalb der Protokollhierarchie enthalten sind. Somit kontrolliert jede Stufe der Hierarchie ein ankommendes Datenpaket nur auf Befehle, die für diese Stufe relevant sind. Enthält das Paket keinen Befehl für diese Stufe, wird es in die nächste Stufe der Hierarchie weitergeleitet. Wird ein zu der Stufe passendes Kommando gefunden, wird es ausgeführt und die dazugehörigen Daten verarbeitet. Dies ermöglicht ein schnelles Routing

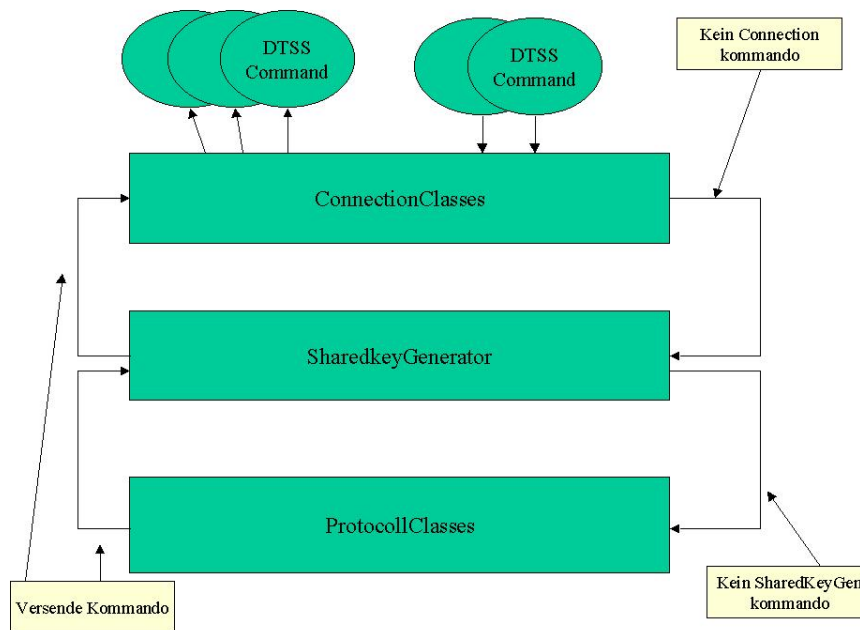


Abbildung 6: Routing der Kommandos

der Daten durch alle Instanzen der Hierarchie und keine Klasse ist angehalten, für sie unwichtige Daten zu untersuchen. Zwischen den SKGs muss eine vollständige Vermaschung der Leitungen bestehen, jeder kann mit jedem Daten austauschen. Um  $n$  Server vollständig zu vermaschen, benötigt man  $\frac{n*(n-1)}{2}$  Verbindungen. Jede Klasse, die Kommandos empfängt und ausführt, enthält eine Funktion vom Typ `public void evalCommand(DTSSCommand cmd)`. Diese Funktion wertet die eingehenden Kommandos aus. Das Muster ist immer das selbe. Ist der Kommandocode für *diese* Klasse bestimmt, dann wird die lokale Codeauswertungstabelle angesprochen. Wenn nicht, wird der Code an die nächst höhere Instanz (*parent*) weitergeleitet. Als Beispiel sei hier `evalCommand` Funktion der Klasse `SkgConnection` aufgezeigt, welche das Prinzip verdeutlichen soll. Die `evalCommand()` Funktionen der anderen Klassen sehen strukturell genauso aus, nur die Namen der Sprungziele der Auswertungstabelle und der Vergleichsoperator ändern sich, wobei `<SKG_COMMAND` Kommunikationskom-

mandos darstellen, welche sich auf Verbindungsaufbau und -abbau beziehen, also die Transportschicht betreffen, =SKG\_COMMAND Kommandos für die Klasse `SharedKeyGenerator` sind, >SKG\_COMMAND sind Kommandos für die Klassen des mathematischen Protokolls. Die beiden letzten genannten Kommandotypen beziehen sich auf das mathematische Protokoll der Schlüsselerzeugung, die Protokollschicht.

```

/**
 * Receives incoming commands and if they belong to this class
 * they are evaluated here.
 * If this is not the case the command is given to the parent
 * class for further evaluation.
 */
public void evalCommand(DTSSCommand cmd)
{
    if(cmd.connectionCommand < SKG_COMMAND)
    {
        switch(cmd.connectionCommand)
        {
            case ALL_CONNECTIONS_ESTABLISHED:
                Functionbody();
                break;
            case READY_FOR_TERMINATION:
                Functionbody();
                break;
            default:
                GuiOutput.println(...);
                break;
        }
    }
    else
    {
        parent.evalCommand(cmd);
    }
} // end of evalCommand

```

Die Klassen, welche für die Kommunikation zwischen den SKGs zuständig sind, sind im Folgenden aufgelistet.

```

SkgConnection.java
SkgConnectionReceiveThread.java
SkgConnectionSendThread.java

```

Der Code der Server ist symetrisch, das heisst es gibt keinen dedizierten Server wie z.B. `Server1`. Jeder Server kann jede Nummer innerhalb des Protokolls zugeteilt bekommen. Die Nummernvergabe für einen Protokollverlauf erfolgt zu

Beginn durch den MS nach der Reihenfolge der eintreffenden Bereitmeldungen der SKGs an den MS. Die Sendevorgänge innerhalb der SKGs laufen ebenfalls nach dem hierarchischen Prinzip ab, eine Klasse, die etwas versenden will erstellt das Kommando, packt die Daten dazu und ruft seine *parent* Klasse auf, welche die Daten weiterleitet bis hin zum `SkgConnectionSendThread`. Beispielfhaft hier der Versendevorgang von Daten der Protokollklasse `DistPrimeTest.java`.

```
public void sendCommand(DTSSCommand cmd)
{
    cmd.skgCommand = SkgInstance.DIST_PRIME_TEST;
    this.parent.sendCommand(cmd);
}
```

Das eigentliche Versenden geschieht in der Klasse `DTSSCommand`, welche sich quasi (getriggert durch den Sendebefehl aus der Klasse `SkgConnectionSendThread`) selbst verschickt. Auch diese Massnahme bietet eine leichtere Austauschbarkeit der Kommunikationskomponenten. Würde man eine direkte Verbindung jeder Klasse zu der Kommunikationssuite via Methoden/Funktionsaufruf implementieren, müsste im Falle einer Änderung dieser Methoden jede Klasse mitgeändert werden, welche eine dieser Methoden benutzt. In unserem Fall jedoch wird nur die Klasse geändert, welche direkt auf der Kommunikationssuite aufsetzt, in der Regel die Klassen `SkgConnectionSendThread` und `DTSSCommand`. Das Selbstversenden des `DTSSCommand`s hat den Vorteil, das bei etwaigen Umstellungen der Kommando- oder Datenstruktur nur diese eine Klasse von der Änderung betroffen ist und somit der Sendevorgang in der Kommunikationssuite nicht angepasst werden muss. Man kann also sehen, eine Änderung in der Kommunikation betrifft nicht den mathematischen Teil, und eine Änderung im mathematischen Teil lässt die Kommunikationsebene unberührt.

### 4.3 Das Protokoll und seine Klassen

Nachdem alle notwendigen Verbindungen zwischen den SKGs aufgebaut sind und die Parameter vom MS übergeben wurden, beginnt die verteilte Schlüsselerzeugung. Hierbei entsprechen die Klassen, welche die Schlüsselerzeugung implementieren im Wesentlichen den Schritten des Protokolls von [MaWuBo99]. Wie bereits in Kapitel 3 dargestellt, benötigen die aufeinanderfolgenden Stufen des Protokolls Daten der Vorgängerstufe und an 2 Stellen kann das Protokoll neu gestartet werden, wenn die Tests (Primalitätstest und Trialdivision) fehlschlagen. Es findet ein reger Datenaustausch zwischen den entsprechenden Instanzen der Server statt. Diesen gilt es zu synchronisieren, jedoch Blockaden zu vermeiden und die Daten an die richtigen Instanzen weiterzuleiten. Die Weiterleitung der Datenpakete wurde bereits besprochen, die protkollinterne Datenverarbeitung soll nun analysiert werden. Zuerst sollen die am Protokoll beteiligten Klassen genannt werden, anschliessend wird der Weg der Schlüsselerzeugung durch die Klassen verfolgt. Auf wichtige Details der Protokollklassen wird dann im einzel-

nen eingegangen.

```
BGW.java  
ComputeN.java  
DistPrimeTest.java  
DistributedSieving.java  
ExponentShareGeneration.java  
SecretSharing.java  
TrialDivision.java
```

Das package der SKGs enthält weitere Klassen, welche verwendete Datentypen oder Hilfsklassen definieren (z.B. `FpPolynomial.java`, `PrimeTable.java` usw), die im Bedarfsfall extra angesprochen werden ansonsten aber im Anhang zu finden sind.

#### 4.4 Optimierung durch mehrere SKG Instanzen

Es wurde in der vorgangegangenen Sektion auf die erhebliche Kommunikation eingegangen, welche im Verlaufe der Schlüsselgenerierung anfällt. Daraus entstehen immer wieder Pausen, in welchen ein Protokollteil auf Daten anderer SKGs bzw deren entsprechender Protokollteile warten muss. Dies führt zu einer sehr schlechten Ausnutzung der Prozessorzeit. Ausserdem werden sehr viele

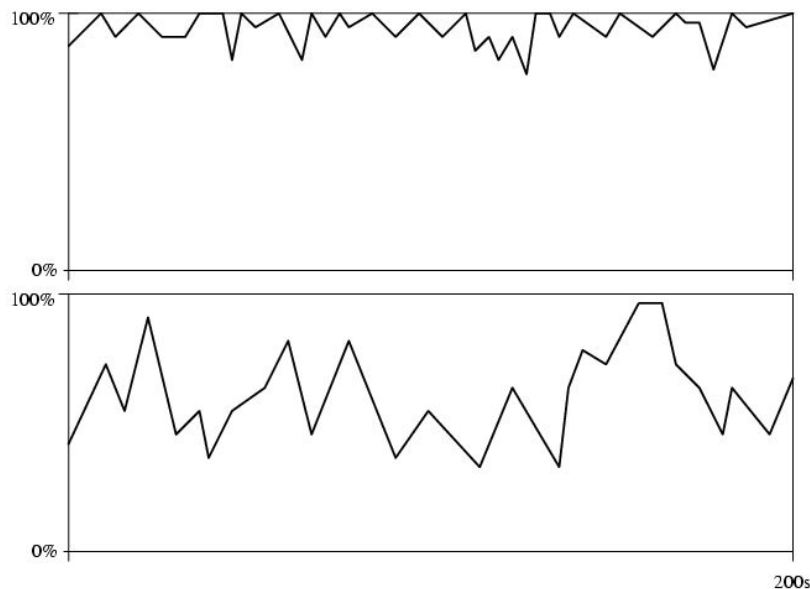


Abbildung 7: CPU Auslastung mit und ohne mehreren Instanzen

der Kandidaten bei der TrialDivision oder dem verteilten Fermattest verworfen, so dass viele Iterationen der Protokollpipeline notwendig sind. Diese zwei effizienz- und zeitraubenden Probleme können mit einer Optimierung verringert werden. Dabei kommt einem auch der schon angesprochene symmetrische Code der SKGs entgegen. Die Idee ist, dass ein `SharedKeyGenerator` mehrere Protokollpipelines verwaltet, welche unterschiedliche Rollen innehaben. Beim Blick auf das in Kapitel 3 beschriebene Protokoll sieht man, dass z.B. Server 1 beim DistPrimeTest erst eine Berechnung anstellen muss, auf deren Ergebnis alle anderen Server warten müssen. Lässt man nun auf jedem der Server eine Instanz eines `SKG1` laufen, ist immer eine Instanz eines Servers aktiv, während die anderen Instanzen des Servers auf Ergebnisse entsprechender Instanzen anderer Rollen warten. Dieses Vorgehen ermöglicht, in kommunikationsbedingten Wartezeiten einer Rolle  $x$  Berechnungen einer oder mehrerer weiteren Rollen durchzuführen. Ebenso wird die Wahrscheinlichkeit erhöht, dass wenigstens *eine* der Instanzen schnell passende Kandidaten findet. Nun sieht also das Bild der verteilten Schlüsselgenerierung aus wie in der folgenden Abbildung dargestellt. Es ergeben sich hiermit neue Aufgaben für die Klassen `SharedKeyGenerator`,

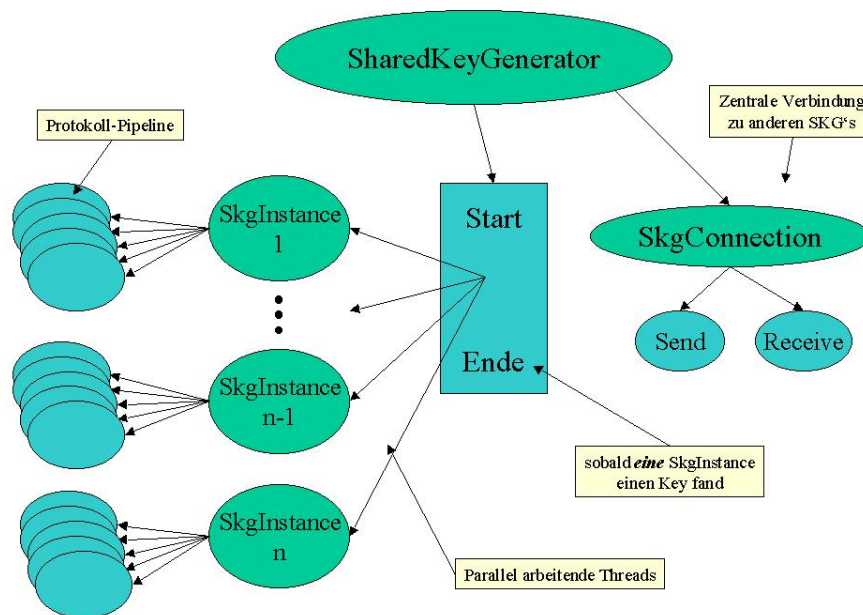


Abbildung 8: Verteilte Schlüsselerzeugung mit mehreren Instanzen

da die Klasse `DTTSCOMMAND` von dieser Optimierung unberührt bleibt, nichts von verschiedenen Instanzen weiss. Dadurch wird verhindert, dass die Kommandostruktur verändert werden muss, und somit alle sie benutzenden Klassen. Einzig der `SharedKeyGenerator` muss um die neue Funktionalität erweitert werden. Er muss nun die ankommenden Pakete an die richtige Instanz weiterreichen und auf Abbruchkommandos einzelner Instanzen im Falle fehlgeschlagener und erfolgreicher Kandidatentests reagieren. Im Fall eines fehlgeschlagenen Tests ei-

ner Instanz muss diese Instanz auf allen Servern neu gestartet werden, ganz wie es das Protokoll vorschreibt. Findet jedoch *eine* Instanz ein passendes Kandidatenpaar, müssen *alle* anderen berechnenden Instanzen gestoppt und terminiert werden um zu verhindern, das etwa eine andere Instanz ebenfalls ein passendes Kandidatenpaar findet und man mehr als einen verteilten RSA Schlüssel erhält. Wird also eine neue Schlüsselerzeugung angestossen, legt jeder SharedKeyGenerator eine vom ManagerServer übermittelte Anzahl von SkgInstances an.

```
for(int i = 0; i<this.threadCount; i++)
{
    int role;
    role=(this.ownNumber+i)%this.threshold;
    if(role==0)
        role=this.threshold;

    skgInstance[i] = new SkgInstance(role, threshold, modulBitLength, i,
                                     this, trialDivisionBound);
}
```

Die Nummer der Rolle, welche eine Instanz ausführt, wird nach dieser Gleichung ermittelt :  $role = (this.ownNumber + i) \bmod (this.threshold)$ . *role* ist die resultierende Rollenummer, welche dann die Instanz ausführt, die diese Nummer zugeteilt bekommt. *ownNumber* ist die vom ManagerServer zugewiesene Servernummer. *threshold* ist die Gesamtzahl der teilnehmenden Server, hier also meist 5. Man braucht diese virtuelle Rollenummer, um die Nummer der Rolle der Instanz von der echten Servernummer unterscheiden zu können. Beim Versenden der Pakete für eine bestimmte Instanz wird diese Rollenummer auf das SKG\_COMMAND addiert, somit weiss der empfangende SharedKeyGenerator, an welche Instanz das Paket weitergeleitet wird und es muss kein neues Datenfeld im DTSSCommand angelegt werden, die Benutzung des Rollenkonzepts bleibt transparent für die Transportschicht und die Protokollteile. Nur der verwaltende SharedKeyGenerator muss sich mit dem Rollenkonzept auseinandersetzen. Die Protokollinstanzen rechnen mit der virtuellen Servernummer, als wäre es die vom ManagerServer vergebene Nummer. Die Routine sendCommand Routine im SharedKeyGenerator sieht dann folgendermassen aus :

```
public void sendCommand(DTSSCommand cmd, int skgInstanceIndex)
{
    .....

    // the number of the thread is coded into connectionCommand
    cmd.connectionCommand = (short)(SkgConnection.SK_G_COMMAND
                                     +skgInstanceIndex+1);
```

```

        // executed, if connectionCommand is not a broadcast, then one must
        // figure out, which Pss is the real receiver, that is done here.
        if(cmd.receiver != 0)
        {
            cmd.receiver = (cmd.receiver - skgInstanceIndex) % threshold;
            if(cmd.receiver <= 0)
            {
                cmd.receiver = cmd.receiver + this.threshold;
            }
        }
        this.skgConnection.sendCommand(cmd);
    }
}

```

Die entsprechende Rückberechnung übernimmt ebenfalls der SharedKeyGenerator in der schon aus anderen Klassen bekannten Funktion `evalCommand(DTSSCommand)`. Ist die passende Instanz ermittelt, wird ihr das Datenpaket übergeben.

```

/**
 * EvalCommand takes commands from SkgConnection and hands them to destination
 * SkgInstances. This function distinguishes
 * between broadcast to all threads and directed
 * messages for a single thread.
 */
public void evalCommand(DTSSCommand cmd)
{
    int targetThread;

    // translate from connectionCommand to number of thread
    targetThread = cmd.connectionCommand - this.skgConnection.SKG_COMMAND - 1;

    ....

    else
    {
        if(targetThread >= 0)
        {
            // hand it to a specific skgInstance thread
            cmd.sender = (cmd.sender+targetThread)%threshold;
            if(cmd.sender==0)
                cmd.sender = threshold;
            skgInstance[targetThread].evalCommand(cmd);
        }
        else if(targetThread == -1)
        {
            // broadcast to all threads

```

```

        GuiOutput.println("Broadcast to all threads");
        for(int i = 0; i < this.threadCount; i++)
        {
            this.skgInstance[i].evalCommand(cmd);
        }
    }
    else
    {
        // error
        GuiOutput.println(...);
    }
}
}
}

```

#### 4.4.1 Die Klasse `SkInstance`

Die Klasse `SkInstance` übernimmt die Verwaltung der Klassen, welche eine Protokollpipeline ausführen. Hier werden die ankommenden Daten anderer Instanzen empfangen und an die entsprechenden Protokollklassen weitergereicht, dies geschieht natürlich wieder in der bekannten `evalCommand(...)` Funktion.

```

/**
 * Evaluates commands and if they belong to one of the protocol parts they
 * are given to them.
 */
public synchronized void evalCommand(DTSSCommand cmd)
{
    .....

    switch(cmd.skgCommand)
    {
        case DISTRIBUTED_SIEVING:
            distributedSieving.evalCommand(cmd);
            break;
        case COMPUTE_N:
            computeN.evalCommand(cmd);
            break;
        case TRIAL_DIVISION:
            trialDivision.evalCommand(cmd);
            break;
        case DIST_PRIME_TEST:
            distPrimeTest.evalCommand(cmd);
            break;
        case EXPONENT_GENERATION:
            expGen.evalCommand(cmd);
    }
}

```

```

        break;
    default:
        GuiOutput.println("SharedKeyGenerator.evalCommand(): unkown Command");
        System.exit(0);
        break;
    }
    // Wake up the actual SkgInstance to check
    // if it was the data he was waiting for.
    receivedData = true;
    this.notify();
}

```

Vorher aber müssen die einzelnen Klassen, welche die Protokollteile ausführen, in der richtigen Reihenfolge angestartet und gegebenenfalls mit Startparametern für ihre Berechnung initialisiert werden. Das Protokoll wird ausgeführt, solange kein Schlüssel in der aktuellen Instanz gefunden wurden, oder bis ein Terminationskommando ankommt (also in einer anderen Instanz ein Schlüssel ermittelt wurde). Da es bei Fehlschlagen der Trialdivision oder des Primalitätstests zum mehrmaligen Ausführen des Protokolls kommt, werden erst alle Werte der Protokollklassen auf definierte Defaultwerte zurückgesetzt (reset).

```

this.distributedSieving.reset();
this.computeN.reset();
this.trialDivision.reset();
this.distPrimeTest.reset();

```

Hat man einen definierten Startzustand hergestellt, werden die Protokollteile gestartet. Dabei haben alle Protokollklassen eine Funktion `void startComputation()`, welche nach dem eventuellen Setzen nötiger Startwerte aufgerufen wird. Ist dieser Protokollteil fertig, werden, wenn nötig, seine Resultate der im Protokoll nachfolgenden Klasse übergeben, dann wird der nächste Protokollteil gestartet.

```

distributedSieving.startComputation();
this.pi=distributedSieving.getPi();
this.qi=distributedSieving.getQi();

computeN.setValues(pi,qi);
computeN.startComputation();
this.N=computeN.getN();

```

Es gibt zwei Sonderfälle, nämlich in den Klassen `TrialDivision` und `DistPrimeTest`. Dort hängt das weitere Verfahren innerhalb des Protokolls davon ab, ob die dort durchgeführten Tests positiv oder negativ verlaufen sind. Deshalb ist hier der Rückgabewert der Funktion `startComputation` nicht `void`, sondern ein boolescher Wert. Gelingt ein Test, wird ein `true` zurückgeliefert, ansonsten ein `false`.

Das genügt, da das Ergebnis der Berechnungen nicht von Bedeutung ist, sondern die Aussage, ob der Test erfolgreich war oder nicht. Nur im Falle einer erfolgreichen Berechnung wird der nächste Protokollteil angestartet, ansonsten muss das gesamte Protokoll von vorn, mit neuen Kandidaten begonnen werden.

```

if(trialDivision.startComputation()==true)
    {
        distPrimeTest.setValues(N, pi, qi);
        if(distPrimeTest.startComputation() == true)
            {
                this.expGen.setValues(N, pi, qi);
                this.expGen.startComputation();
                this.keyFound = true;
                this.e = expGen.getPublicExponent();
                this.d = expGen.getSecretExponent();
                parent.setKey(N, this.e, this.d,numIterations*numThreads);
            }
    }

```

Wird die Variable `keyFound` auf den Wert `true` gesetzt, wird keine weitere Protokolliteration gestartet, es wurde ein passender RSA Schlüssel gefunden, alle anderen Instanzen müssen terminiert werden. Andernfalls beginnt das Protokoll von vorn.

#### 4.4.2 Die Klassen `Distributed Sieving`, `ComputeN` und das `BGW`

Hier werden, wie in Kapitel 3 bereits erwähnt, Kandidaten für  $p_i$ ,  $q_i$  ausgewählt und unbrauchbare Kandidaten ausgesiebt. Da die Klasse `BGW` von den Klassen `DistributedSieving` und `ComputeN` benutzt wird, wurde sie als `SubProtocolPart` implementiert. Dadurch kann man Protokollhilfsklassen, welche von mehr als einer Klasse verwendet werden, zur Verfügung stellen und verhindert Codeduplizität in den eigentlichen Protokollklassen. In der Klasse `BGW` wird der Lagrangefaktor berechnet, der im Protokoll benötigt wird. Es ist hier darauf zu achten, dass dabei keine echte Division stattfindet, sondern, da alle Berechnungen bezüglich eines Moduls verlaufen, eine Inversenbildung stattfindet.

```

/**
 * Calculates the Lagrange Factor lambda (only for x=0).
 */
private DTSSBigInt lagrangeFactor(int index)
{
    DTSSBigInt tempLambda;
    DTSSBigInt tempNenner;

    int nenner=1;
    int zaehler=1;

```

```

    for(int i=1; i<=this.threshold; i++)
    {
        if(i!=index)
        {
            zaehler=zaehler*(0-i);
            nenner=nenner*(index-i);
        }
    }
    tempNenner=(new DTSSBigInt(new Integer(nenner).toString())).
                modInverse(this.globalModul);
    tempLambda=new DTSSBigInt(new Integer(zaehler).toString());

    return tempLambda.multiply(tempNenner).mod(this.globalModul);
}

```

Ansonsten folgt die Berechnung der Gleichungen wie in Kapitel 3 beschrieben. Es wird eine Klasse `FpPolynomial` verwendet, welche durch die Verwendung des Horner Schemas zur Polynomevaluation und der daraus resultierenden Reduktion der Anzahl Multiplikationen einen Effizienzgewinn für Programmabläufe mit vielen Polynomevaluationen (wie das BGW) von ca 20 Prozent erreicht.

```

/**
 * Evaluating the polynomial at point. This function uses the
 * Horner's Rule for reducing number of multiplications.
 */
public DTSSBigInt evaluate(DTSSBigInt point)
{
    DTSSBigInt p;

    p = this.coefficients[this.degree];
    for(int i=this.degree-1; i>=0; i--)
    {
        p = p.multiply(point);
        p = p.add(this.coefficients[i]);
        p = p.mod(this.modulus);
    }
    return p;
}

```

In der Klasse `DistributedSieving` werden die Berechnungen zum Aussieben ungeeigneter vorgenommen. Der Aufruf an das BGW Protokoll erfolgt, nachdem die Testkandidaten  $ap$  und  $aq$  gewählt wurden.

```

/**
 * After this method is executed bp[ownNumber-1] and bq[ownNumber-1]
 * contain the final shares to create pi and qi.

```

```

*/
private void convertToAdditiveSharing(DTSSBigInt ap, DTSSBigInt aq)
throws NoMoreDataException
{
    int offSet;

    this.setBpBq(ap, aq);

    // ap : do the conversions for ap2 - apk
    for(int i=0; i<threshold-1; i++)
    {
        if(i == this.ownNumber - 2 )
            this.bgw[i].setValues(bp[ownNumber-1], ap);
        else
            this.bgw[i].setValues(bp[ownNumber-1], DTSSBigInt.zero);
        this.bgw[i].startComputation();
        bp[ownNumber-1] = this.bgw[i].getMi();
    }

    // aq : do the conversions for aq2 - aqk
    // offSet assures new bgw's instances (to avoid communication conflicts)
    offSet = threshold-1;
    for(int i=0; i<threshold-1; i++)
    {
        if(i == this.ownNumber - 2 )
            this.bgw[i+offSet].setValues(bq[ownNumber-1], aq);
        else
            this.bgw[i+offSet].setValues(bq[ownNumber-1], DTSSBigInt.zero);
        this.bgw[i+offSet].startComputation();
        bq[ownNumber-1] = this.bgw[i+offSet].getMi();
    }
}

```

In der Klasse ComputeN wird wieder das BGW Protokoll ausgeführt. Sind alle Werte  $N_i$  berechnet wird daraus der Modul  $N$  gebildet.

```

public boolean startComputation() throws NoMoreDataException
{
    bgw.setValues(pi, qi);
    bgw.startComputation();

    Ni[ownNumber-1] = bgw.getMi();
    this.broadcastNi();
    N = this.combineNi();
    return true;
}

```

#### 4.4.3 Die Klassen `TrialDivision` und `PrimeTable`

Die Testdivision (`TrialDivision`) dient dazu, festzustellen, ob der Modul  $N$  kleine Primzahlen enthält. Man mag sich fragen, warum nicht gleich der Verteilte Fermattest gemacht wird, der ja relativ sicher feststellt, ob  $p$  und  $q$  prim sind und man prüft dann bei diesen vielversprechenden Kandidaten, ob kleine Primzahlen enthalten sind. Schaut man sich den Verteilen Fermattest an, sieht man, dass jeder Server eine modulare Exponentiation durchführen muss, mit Zahlen der Größenordnung der halben Bitlänge des Moduls. Diese Exponentiation ist sehr zeitaufwendig. Geht man von einem 2048 bit Modul aus, und nimmt ca 3000 Iterationen der Protokollpipeline an (ein Durchschnittswert, welcher während der Tests ermittelt wurde), kommt man auf 3000 Exponentiationen mit einer ca 1024 Bitzahl im Exponenten pro Server nur beim Fermattest. Unter Java auf in der im Ahnhang beschriebenen Testumgebung braucht man dazu ca 18 - 20 min. Zusätzlich sind mehrere Versendevorgänge nötig, die ebenfalls Zeit kosten, was sich bei der hohen Anzahl von Iterationen bei grossen Modullängen durchaus bemerkbar macht. Benutzt man eine Testdivision, werden ca 60 Prozent aller Kandidaten als nicht tauglich erkannt. Die Division ist wesentlich schneller auszuführen als die modulare Exponentiation. Man kann den Vorgang noch beschleunigen, indem man  $N$  nicht durch jede der kleinen Primzahlen einzeln dividiert, sondern Primzahlprodukte bildet. Diese kann man im voraus berechnen und abspeichern, so das die Berechnungen der Produkte während der Schlüsselgenerierung wegfallen. Da die Primzahlprodukte anfänglich klein sind, jedoch bis 31 bit Länge ansteigen (grössere Zahlen als 31 bit erwiesen sich als uneffektiv bezüglich der der Rechenzeit), kann man je nach Servernummer viele kleine Produkte oder wenige grosse Produkte in der Testdivision verwenden, so das ein Lastausgleich sehr gut möglich ist. Die Anzahl der zu testenden Primzahlen ist ein Parameter, der je nach Schlüssellänge und teilnehmender Serveranzahl unterschiedlichen Effizienzgewinn bringt. Im Ahnhang werden dazu Tabellen aufgezeigt, welche Ergebnisse aus durchgeführten Testreihen dokumentieren. Die Primzahlen, soweit nicht vorberechnet, werden in der Klasse `PrimeTable.java` mit dem Sieb des Erathostenes ermittelt.

```
/* Sieve of Erathostenes */
    numberArray[0] = false;

    for(int i = 2; i < numberArray.length; i++)
    {
        numberArray[i] = true;
    }
    int loopCount = (int)(Math.sqrt(upperBound)+1);
    // +1, cause sqrt is rounded down

    for(int i = 2; i <= loopCount; i++)
    {
        for(int j = i*i; j <= upperBound;)
```

```

    {
        if(numberArray[j] == true)
            primeCount--;
        numberArray[j] = false;
        j = j+i;
    }
}

```

Ist das Sieb durchgelaufen, bezeichnet der Index des Arrays, an dessen Stelle der boolesche Wert *true* steht, eine gefundene Primzahl. Diese werden dann in einem Array gespeichert und können dann als Faktoren für Primzahlprodukte oder als Einzelteiler verwendet werden. Die Testdivisionen mit den gefundenen Primzahlen oder ihren Produkten werden in der Klasse `TrialDivision` mit Hilfe des Euklidischen Algorithmus, des ggT (englisch Greatest Common Divisor, gcd) durchgeführt.

```

// trial division using PrimeProducts
while(i < primeProducts.length && trialDivisionResult)
{
    if(RSASModul.gcd(primeProducts[i]).compareTo(DTSSBigInt.one) != 0)
    {
        trialDivisionResult = false; // N is a product of small primes
        break;
    }
    i = i + threshold;
}

```

Wird nun bei *einem* Server entdeckt, dass  $N$  das Produkt der getesteten Primzahlen ist, wird nur ein Broadcast an alle anderen Server geschickt, das diese Iteration abgebrochen werden kann. Wird kein Teiler ermittelt, wird dieses Ergebnis ebenfalls den anderen SKGs per Broadcast mitgeteilt.

```

if(trialDivisionResult)
{
    sendCommand(new DTSSCommand (N_IS_QUITE_PRIME, null, DTSSCommand.BROADCAST));
}
else
{
    sendCommand(new DTSSCommand (N_IS_NOT_PRIME, null, DTSSCommand.BROADCAST));
}

```

Es ist kein weiteres Versenden von Zahlen und Warten auf Zwischenergebnisse wie im Fall des Verteilten Fermattests nötig. Die Verwendung der Testdivision ist also für die Effizienz der verteilten Schlüsselgenerierung unerlässlich, das Finden

der richtigen Parameter in Abhängigkeit von Serveranzahl und Schlüsselgröße ein interessantes Testfeld. Um zusätzliche Zeiteinsparungen zu erreichen, wurde versucht, diese Berechnungen über das JNI Interface in einer C/C++ Bibliothek der TU Darmstadt auszuführen. Diese Bibliothek nennt sich LiDIA und ist besonders für die Berechnung grosser Ganzzahlen geeignet. Der reine Geschwindigkeitsgewinn einer Berechnung lag bei einem Faktor von ca 2. Doch es ergaben sich zwei Probleme, die dazu führten, LiDIA nicht zur Verwendung für diesen Teil des Projekts heranzuziehen. Das erste Problem ist, dass ein JNI-call einen gewissen Overhead mit sich bringt, der bei mehreren 1000 JNI-calls durchaus ins Gewicht fällt. Man sollte also bei der Benutzung von JNI-Funktionen darauf achten, sie nicht innerhalb einer grossen Iteration zu benutzen. Dies hätte man noch umgehen können, wenn man die gesamten Primzahlarrays als Argument an die berechnende JNI-Funktion übergibt. Das wesentlich grössere Problem war, dass LiDIA nicht für den Multithreadingbetrieb programmiert wurde. Benutzt man also das schon beschriebene Rollenkonzept mit mehreren berechnenden Threads (Instanzen) je Server, entstanden sogenannte *race conditions*. Hatte eine Instanz gerade eine Berechnung in der LiDIA Bibliothek begonnen, war es möglich, dass dieser Thread vom Threadscheduler des Betriebssystems zugunsten eines anderen deaktiviert wurde. Der nun aktive Thread führt seine Berechnungen durch und verlässt die LiDIA Bibliothek. Kommt nun der vorher suspendierte Thread wieder an die Reihe, sind seine Argumente, die er möglicherweise vorher schon an die LiDIA Funktionen übergeben hat und dortige Zwischenergebnisse nicht mehr vorhanden, die Berechnung kann nicht mehr erfolgreich zu Ende gebracht werden. LiDIA ist also nicht *threadsave* programmiert und konnte somit nicht in diesem Projekt verwendet werden. Dies ist sehr bedauerlich, da bei allen Berechnungen, vor allem bei den Exponentiationen die Rechenzeit wesentlich geringer ausgefallen wäre.

#### 4.4.4 Die Klasse DistPrimeTest

Ist die Testdivision positiv verlaufen und es wurden keine kleinen Primteiler gefunden, wird der finale Test auf die Primalität der Kandidaten  $p$  und  $q$  durchgeführt.

```

/**
 * Server number one picks g, the other receive g from him.
 */
public void pickG() throws NoMoreDataException
{
    if(this.ownNumber==1)
    {
        this.g = new DTSSBigInt(this.modulBitLength,0);
        sendCommand(new DTSSCommand(BROADCAST_G, this.g.toByteArray(), 0));
        this.receivedG = true;
    }
    else

```

```

    {
        while(!this.receivedG)
        {
            waitUntilReceived();
        }
    }
}

```

Hierbei muss der Server, welcher die Rolle der Servers<sub>1</sub> spielt, Berechnungen vornehmen, während die anderen Server auf diese Ergebnisse warten. Hier bietet sich das verwendete Rollenkonzept im Besonderen an, da in den freien Rechenzeiten die Server nun ihrerseits Instanzen mit der Rolle eines Servers<sub>1</sub> ausführen können.

```

/**
 * Server number one picks g, the other receive g from him.
 */
public void pickG() throws NoMoreDataException
{
    if(this.ownNumber==1)
    {
        this.g = new DTSSBigInt(this.modulBitLength,0);
        sendCommand(new DTSSCommand(BROADCAST_G, this.g.toByteArray(), 0));
        this.receivedG = true;
    }
    else
    {
        while(!this.receivedG)
        {
            waitUntilReceived();
        }
        if(DEBUG)
            GuiOutput.println("I received g "+this.g);
    }
}
}

```

Man kann sehen, dass nur Server<sub>1</sub> beschäftigt ist, alle anderen warten. Sind nun alle Berechnungen der Server beendet, senden sie ihre Werte  $v_i$  zurück an Server<sub>1</sub>, welcher die finale Berechnung durchführt und das Ergebnis an alle anderen Server (besser den entsprechenden Instanzen auf den anderen Servern) mitteilt.

```

/**
 * Finally server number one checks the primality of
 * P and Q and broadcasts the result.
 * The Result is stored global to be able

```

```

    * to receive it from other servers.
    */
private void testN() throws NoMoreDataException
{
    if(this.ownNumber == 1)
    {
        DTSSBigInt temp = DTSSBigInt.one;

        for(int i=1; i<this.threshold; i++)
        {
            temp = temp.multiply(this.V[i]).mod(this.N);
        }
        if(this.V[0].compareTo(temp) == 0)
        {
            this.distPrimeTestResult = true;
            // broadcast result
            sendCommand(new DTSSCommand (N_IS_QUITE_PRIME, null,
                                         DTSSCommand.BROADCAST));
        }
        else
        {
            this.distPrimeTestResult = false;
            // broadcast result
            sendCommand(new DTSSCommand (N_IS_NOT_PRIME, null,
                                         DTSSCommand.BROADCAST));
        }
    }
    else
    {
        while(this.distPrimeTestFinished == false)
        {
            waitUntilReceived();
        }
    }
}

```

Man sieht, dass inaktive, wartende Protokollteile eine Funktion `waitUntilReceived()`; aufrufen. Diese Funktion ist in der Klasse `SharedKeyGenerator` implementiert und sorgt dafür, dass Instanzen, welche keine Berechnungen durchführen, da sie auf eingehende Daten oder Zwischenergebnisse der Berechnungen anderer Server warten, Rechenzeit freigeben, um sie anderen Instanzen, deren Daten mittlerweile eingetroffen sind, zur Verfügung zu stellen. Diese Funktion wird von jedem Protokollteil aufgerufen, sobald er sich in einer Wartestellung befindet.

```

/**
 * Wait until new data arrives from network or a stopKeyGen command is send

```

```

    * (over network or from this process directly to do a reset)
    */
public void waitUntilReceived()
{
    try
    {
        // get a monitor for this object and then fall asleep
        synchronized(this)
        {
            // do a last check if someone did send some data
            // while on the way to this point of the programm
            if(receivedData == true)
            {
                receivedData = false;
                return;
            }
            wait();
            receivedData = false;
        }
    }
    catch(InterruptedException ie)
    {
        GuiOutput.println(...);
    }
}

```

Treffen neue Daten ein, übergibt gibt der SharedKeyGenerator diese Daten der Klasse `SkInstance` der auf die Daten wartenden Instanz. Die Klasse `SkInstance` übergibt die Daten an den entsprechenden Protokollteil und teilt diesem das Eintreffen der neuen Daten mit. Dies geschieht mittels eines Aufrufs der `notify` Methode. Somit aktiviert sich die wartende Instanze selbst.

```

/**
 * Evaluates commands and if they belong to one of the protocol parts they
 * are given to them.
 */
public synchronized void evalCommand(DTSSCommand cmd)
{
    ...
    switch(cmd.skgCommand)
    {
        case DISTRIBUTED_SIEVING:
            distributedSieving.evalCommand(cmd);
            break;
        case COMPUTE_N:
            computeN.evalCommand(cmd);
    }
}

```

```

        break;
    case TRIAL_DIVISION:
        trialDivision.evalCommand(cmd);
        break;
    case DIST_PRIME_TEST:
        distPrimeTest.evalCommand(cmd);
        break;
    case EXPONENT_GENERATION:
        expGen.evalCommand(cmd);
        break;
    default:
        GuiOutput.println(...);
        System.exit(0);
        break;
}
// Wake up the actual SkgInstance to check
// if it was the data he was waiting for.
receivedData = true;
this.notify();
}

```

Der Protokollteil, welcher auf eintreffende Daten wartet und die wait Funktion aufgerufen hat, prüft nun die angefallenen Daten. Sind noch nicht alle Daten eingetroffen, die er benötigt, gibt er wieder durch den Aufruf der Funktion `waitUntilReceived()` die Rechenzeit zugunsten anderer Instanzen ab. Ansonsten führt er nun seine Berechnungen mit den nun vollständig eingetroffenen Daten fort.

Doch nun zurück zum verteilten Fermattest. Schlägt der Test fehl, kann man in den oberen Codefragmenten sehen, dass ein Broadcast erfolgt, der alle anderen Server veranlasst, *diese* Iteration abubrechen und das Protokoll neu zu starten. Ist der Test erfolgreich, werden alle Server benachrichtigt und somit veranlasst, die *anderen* Instanzen zu schliessen und nur noch die Gewinnerinstanz weiterzuführen. Diese muss nun den verteilten Exponenten  $d$  berechnen.

#### 4.4.5 Die Klassen `ExponentShareGeneration` und `SecretSharing`

Der letzte Teil des Protokolls, welcher ausschliesslich bei Erfolg *einer* Instanz durchgeführt wird, ist das Berechnen des geheimen Exponenten  $d$  derart, dass keiner der Server ihn in seiner Gesamtheit kennt oder ihn rekonstruieren kann. Danach muss der Schlüssel sicher in einem `JAVAKeystore` gespeichert werden. Soll noch ein t-out of-k `SecretSharing` (nach Shamir oder Sakurai erfolgen, geschieht dies im Anschluss an die Berechnung des Exponenten). Dabei werden die in Kapitel 3 beschriebenen Schritte durchgeführt.

```
public boolean startComputation() throws NoMoreDataException
```

```

{
    computePhi_i();
    computeAdditivePhi_iShares();
    computeReceivedAddSharesSum();
    computeL();
    computeShareOfSecretExponent();
    trialSigning();
    computeR();
    if(this.shareOfSecretExpFound)
    {
        return true;
    }

    // no Share of d found
    else
        return false;
}

```

Der Server mit der Rolle der Servers  $SKG_1$  muss den Rest  $r$  der Gleichung aus Kapitel 3 ermitteln. Dies erfolgt über Probever- und -entschlüsselungen.

```

/**
 * computeR is only executed by PartialSigningServer No.1,
 * to determine the exact value
 * of share d1, subtracting the remainder that
 * is caused by the integer division in function
 * computeShareOfSecretExponent().
 */
public void computeR() throws NoMoreDataException
{
    boolean found = false;
    if(ownNumber > 1)
        return;

    ...

    while(receivedCipherTextCount < threshold-1)
    {
        waitUntilReceived();
    }

    for(int j = 0;j < this.threshold;j++)
    {
        tempProduct = tempProduct.multiply(this.ciphers[j]);
    }
    tempProduct = tempProduct.mod(this.N);
}

```

```

for(int i = 0;i <= this.threshold;i++)
{
    this.R = new DTSSBigInt(Integer.toString(i));
    cipherModPowR = new DTSSBigInt(testPlainText).modPow(this.R, this.N);
    cipherModPowD = tempProduct.multiply(cipherModPowR).mod(this.N);
    cipherModPowD = cipherModPowD.modPow(this.publicExponent, this.N);
    if(testPlainText.compareTo(cipherModPowD) == 0)
    {
        if(R.compareTo(DTSSBigInt.zero) < 0)
            this.shareOfSecretExponent=this.shareOfSecretExponent.add(R);
        else
            this.shareOfSecretExponent=this.shareOfSecretExponent.add(R);

        this.shareOfSecretExpFound = true;
        found = true;
        break;
    }
    if(i == this.threshold && !found)
        GuiOutput.println("NO KEY FOUND, N PROBABLY
                            NOT A PRODUCT OF 2 PRIMES.");
}
}

```

Sollte *kein* Rest  $r$  gefunden werden, ist die Wahrscheinlichkeit, dass  $N$  doch kein Produkt zweier Primzahlen ist, sehr hoch und es muss eine neue Schlüsselgenerierung gestartet werden. Bei den vielen Tests während der Entwicklung dieses Projekts, welche oftmals als Dauertests tagelang liefen, ist dieser Fall nie eingetreten.

Das Speichern des Schlüssels übernimmt die Klasse `SharedKeyGenerator`, welche auch im Bedarfsfall ein *t-out of-k* SecretSharing anstösst, um die Sharebruchstücke dem KeyStore hinzuzufügen. Die eigentliche Protokollpipeline ist nach der Exponentengenerierung beendet. Die Klasse `SecretShare` berechnet die Sharebruchstücke, welche für ein *t-out of-k sharing* benötigt werden. In der ersten Entwicklungsphase wurde das SecretSharing Schema implementiert, welches auf einem leicht modifizierten Shamir Secret Sharing beruht. Später wurde dann das Secret Sharing nach Sakurai benutzt, um diese neue Methode erstmals innerhalb einer Anwendung zu testen. Die Implementierung dieser Methode kommt mit sehr wenig Code aus. Ausser der obligatorischen Methode `startComputation()` werden nur 3 weitere klassentypische Methoden zur Implementierung verwendet. Benutzt man kein Rollenkonzept, können hier einige Berechnungen nach Erhalt der Servernummer bereits im Vorhalt durchgeführt werden, wie z.B. das Erstellen der Koeffizienten und die Evaluation des Polynoms an der Stelle  $x$ , welches bereits erhaltenen Servernummer entspricht. Die drei Funktionen, welche jeweils kaum mehr als 10 Zeilen Code entsprechen, sind :

```

private Polynomial createPolynom()
{
    Polynomial result;
    result = new Polynomial();
    result.setCoefficient(ownShare,0);
    result.setCoefficient(new DTSSBigInt(2048,0),1);
    result.setCoefficient(new DTSSBigInt(2048,0),2);
    return result;
}

```

Bis auf den Koeffizienten an der Stelle  $x^0$ , welcher dem  $d_i$  des SKG $_i$  entspricht, können die restlichen Koeffizienten bereits vorher berechnet werden.

```

private void computeSharesToSend()
{
    DTSSBigInt evaluationPoint;
    for(int i = 0;i < threshold;i++)
    {
        evaluationPoint = new DTSSBigInt(java.lang.Integer.toString((i+1)));
        sharesToSend[i] = polynom.evaluate(evaluationPoint);
        if((i+1) != ownNumber)
        {
            sendCommand(new DTSSCommand (this.SECRETSHARE,
                                           sharesToSend[i].toByteArray(), i+1));
        }
    }
}

```

Auch hier kann bereits vor Erhalt des eigenen Shares  $d_i$  ein Teil der Berechnung durchgeführt werden. Das Polynom  $f^{(i)} = a_1^{(i)}x + a_2^{(i)}x^2$  kann bereits berechnet werden. Das fehlende additive Glied  $d_i x^0$  kann dann nachträglich, nach erfolgreicher Beendigung der Protokollpipeline hinzugefügt werden. Man kann also die Funktion `private Polynomial createPolynom()` und einen Teil der Funktion `private void computeSharesToSend()` vorberechnen und im Falle der finalen Keyerzeugung auf diese Zwischenergebnisse zurückgreifen.

```

private void computeOwnSecret() throws NoMoreDataException
{
    this.ownSecret = new DTSSBigInt("0");
    sharesToHold[ownNumber - 1] = sharesToSend[ownNumber - 1];
    while(this.receivedCounter < this.threshold - 1)
    {
        synchronized(this)
        {
            try

```

```

        {
            //GuiOutput.println("waiting for shares.");
            wait();
        }
        catch(InterruptedException e)
        {
            GuiOutput.println(e);
        }
    }
}
for(int i = 0; i<this.threshold; i++)
{
    GuiOutput.println("f("+(i+1)+") = " + sharesToHold[i]);
    this.ownSecret = this.ownSecret.add(sharesToHold[i]);
}
GuiOutput.println("sum of all fi "+this.ownSecret);
}

```

Sind auch diese Sharebruchstücke berechnet, kann der gesamte Schlüsseldatensatz gespeichert werden. Dies erfolgt in einem JAVA KeyStore, um den gängigen Sicherheitsstandards zu genügen. In der Frühphase des Projekts wurde noch eine eigene Klasse für den DTTSKey implementiert (siehe [Fuhr00]). Diese fand in der Endversion jedoch keine Anwendung mehr. Im Unterschied zur Shamir Secret Sharing Variante, in der alle Sharebruchstücke mitgespeichert werden müssen, genügt bei Verwendung des Sakurai Protokolls nur das Ergebnis des Polynoms  $f(i)$  als private key.

```

myKeyShare = new
RSAPrivKey(this.N.getJavaBigInt(),
            this.finalSecretExponent.getJavaBigInt());

keystore.setKeyEntry("My Share-",myKeyShare,geheim,chain);

try
{
    GuiOutput.println("Saving KeyStore to disk");
    keystore.store(out,geheim);
    out.close();
}
catch(IOException ioe)
{
    GuiOutput.println("Could not store KeyStore to disk."+ioe);
}
catch(CertificateException ce){
    GuiOutput.println("Cert Error:"+ce);
}

```

```

        catch(NoSuchAlgorithmException nsa){
            GuiOutput.println("Could not find KeyStore algorithm:"+nsa);
        }

```

Ist der Schlüssel gespeichert, können die SKGs beendet werden. Die Schlüssel zur Signaturerstellung liegen nun gebrauchsfertig im KeyStore bereit und können von den jeweiligen PSS eingelesen werden.

## 4.5 Der ReceptionServer

Der ReceptionServer hat die Aufgabe, die anfallenden Timestamprequests (TSR) entgegenzunehmen und an die verarbeitenden Instanzen weiterzuleiten. Die Vorgehensweise ist in etwa folgende.

1. der RS horcht an seinem Port auf eingehende SSL Verbindungen
2. ein Client macht einen erfolgreichen Verbindungsrequest
3. der RS öffnet einen ClientConnection thread zur weiteren Kommunikation
4. der ClientConnection thread liest den eingehenden TimeStampRequest
5. der ClientConnection thread stösst ein MD5withRSA Klasse mit dem Signaturauftrag an
6. MD5withRSA beantragt beim ReceptionServer ein SignatureControlObject (SCO)
7. das SCO übernimmt die Kommunikation mit den PSS und kombiniert die erhaltenen Teilsignaturen
8. MD5withRSA übergibt das SCO an den RS für weitere Berechnungen
9. der ClientConnection thread sendet die Timestampresponse zurück zum Client und wird geschlossen.

Der entstehende TSR, welcher an die PSS gesendet wird, ist pkcs7 kompatibel ( siehe [PKCS]) und kann somit von jedem gängigen Programm erkannt und ausgewertet werden. Es muss, bevor der Antrag an die PSS geleitet wird, die Zeit des Eintreffens des Requests in den Antrag übernommen werden. Die eingebettete Zeit stellt den Zeitstempel dar und muss *vor* der Signatur hinzugefügt werden, um nicht nachträglich verfälscht werden zu können. Dies geschieht in der Klasse `ClientConnection` mit Hilfe der Klassen des packages `IETF`. Die Einbettung erfolgt in einen ASN1 Registered Type.

```

.....
// make TSTInfo structure (the timestamp)

```

```

TSTInfo tst =
new TSTInfo(tsr.messageImprint,serialNumber,time,nonce);

// encode TSTInfo
TSTInfoData encoded = new TSTInfoData(tst);

// put encoded TSTInfo into SignedData
SignedData signedData = new SignedData();
signedData.setContent(encoded);
SignerInfo signInfo = new SignerInfo(cert, "MD5withRSA");
// sign it using the Codec
codec.pkcs7.Signer sign = new codec.pkcs7.Signer(signedData,
                                                    signInfo, ReceptionServer.key);

sign.update();
.....

```

Danach kann der Signiervorgang gestartet werden. Als Kryptoprovder wurde in diesem Projekt der CDCStandardProvider (TU Darmstadt), der CDCSharedProvider (TU Darmstadt) und der JSSE SSL Provider (sun) verwendet. Der CDCSharedProvider setzt auf den CDCStandardProvider auf und erweitert ihn um die *verteilte* MD5withRSA Signatur.

```

/**
 * The CDCStandardProvider registers the distributed MD5withRSA signer.
 * <p>
 */

public class CDCSharedProvider extends java.security.Provider{

    public CDCSharedProvider(){
        super("CDCShared", 0.1, "");

        // Distributed Signature
        put("Signature.MD5withRSA", "cdc.dtss.jca.MD5withRSA");

        // register CDC provider because we need it ourselves
        java.security.Security.addProvider(new cdc.standard.CDCStandardProvider());
    }
}

```

Die Klassen MD5withRSA und DTSStoKey nehmen die notwendigen Änderungen für die verteilte Anwendung vor, so dass nach aussen völlige JCA-Kompatibilität besteht. So müssen z.B. Konfigurationsdaten wie IP Adressen, Anzahl der beteiligten Server usw transparent für die den Service benutzende Anwendung in das Schema, einen JCA kompatiblen PrivateKey eingebettet werden. Dieser kann dann zur Initialisierung (JCA kompatible Funktion `initSign()` der neuen MD5withRSA Klasse verwendet werden. Die benötigten Daten werden aus einem

Konfigurationsfile gelesen. Beispielhaft ein Auszug aus der Klasse KeyToDTSS, in dem exemplarisch das Einlesen einiger der Parameter aufgezeigt wird.

```
public class KeyToDTSS implements PrivateKey{

    .....

    public boolean readConfigFile(String configFileName)
    {
        BufferedReader reader;
        String parameterString;
        String pssIP = "PartialSigningServerIP";
        String pssPortBase = "PartialSigningServerPortBase";
        String ownPortBase = "ReceptionServerPortBase";
        String threshold = "Threshold";
        int linesCounter = 0;
        int ipCounter = 0;

        try
        {
            reader = new BufferedReader(new FileReader(configFileName));
        }
        catch(FileNotFoundException e)
        {
            .....
        }
        try
        {
            parameterString = reader.readLine();
            while(parameterString != null)
            {
                if(parameterString.startsWith(threshold) && linesCounter == 0)
                {
                    this.threshold = Integer.parseInt(parameterString.
                        substring(threshold.length()).trim());
                    this.pssIpAddresses = new String[this.threshold];
                }
                else if(parameterString.startsWith(pssPortBase) && linesCounter > 0)
                {
                    this.pssPortNumber = Integer.parseInt(parameterString.
                        substring(pssPortBase.length()).trim());
                }
            }

            .....

        }

        public String getAlgorithm(){
            return "DTSS Configuration Information";
        }
    }
}
```

```

    }
    .....
}

```

Die Klasse `MD5withRSA` übernimmt nun diese eingebetteten Daten und stellt in ihrer Funktionalität und Struktur eine JCA kompatible Klasse dar, welche die JCA Klasse `SignatureSpi` erweitert. Die für den erweiteren verteilten Einsatz notwendigen Daten aus der `KeyToDTSS` Klasse werden in der Funktion `protected void engineInitSign(PrivateKey privateKey)` entgegengenommen, dort wird auch das SCO angefordert.

```

protected void engineInitSign(PrivateKey privateKey) throws
InvalidKeyException{
    if (!(privateKey instanceof KeyToDTSS)) throw new
InvalidKeyException("We need a KeyToDTSS!");
    confInfo = (KeyToDTSS)privateKey;
    // get SignatureControlObject from ReceptionServer
    sco = ReceptionServer.getSignatureControlObject();
    sco.init(confInfo);
    // clear buffer
    message=null;
}

```

Die Klasse `SignatureControlObject` unterhält die Verbindungen zu den PSS und berechnet aus den erhaltenen Teilsignaturen die Gesamtsignatur. Es werden jedoch auch einige Sicherheitsüberprüfungen vor Inbetriebnahme des Servers ausgeführt. So fordert das SCO von allen teilnehmenden PSS den Modul  $N$  und den public key  $e$  an. Somit wird sichergestellt, dass alle PSS mit dem gleichen Schlüssel arbeiten. Ist dies nicht der Fall, wird das Programm sofort gestoppt und beendet. Sind alle Server ansprechbar und haben den gleichen (öffentlichen) Schlüssel und Modul, kann bei Verwendung des Sakurai Protokolls bereits eine Auswahl der Server getroffen werden, welche an der Signatur teilhaben sollen. In der Endversion liefen alle Berechnungen mit einem Pool aus 5 Servern, somit konnte man 3 aus 5 Server wählen. Als Kriterien zur Auswahl der 3 Server könnte man z.B. schnelle Antwortzeit im Netzwerk nehmen (geringer ping). In dieser Implementierung wurde die Kombination der 3 Server bei jeder Signatur zufällig neu gewählt. Die Parameter der jeweiligen Kombination wurden dann einmal berechnet, dies kann aber auch im Vorhalt geschehen, da das Parameterset der Sakurai Methode sehr gut geeignet ist, im Voraus berechnet zu werden, wie in Kapitel 3 beschrieben.

```

/**
 * precomputes all combinations for a 3 out of 5 secretsharing
 * (Sakurai)
 */
private void prepareSecretShareProtocol()

```

```

    {
        int test [] = {1,3,5};
        DTSSBigInt temp [];
        combinations = new Combinations[10];
        combinations[0] = new Combinations(1,2,3);
        combinations[1] = new Combinations(1,2,4);
        combinations[2] = new Combinations(1,2,5);
        combinations[3] = new Combinations(1,3,4);
        combinations[4] = new Combinations(1,3,5);
        combinations[5] = new Combinations(1,4,5);
        combinations[6] = new Combinations(2,3,4);
        combinations[7] = new Combinations(2,4,5);
        combinations[8] = new Combinations(2,3,5);
        combinations[9] = new Combinations(3,4,5);
    }

/**
 * Chooses one combination out of 10 randomly, that determines,
 * which 3 out of the 5 Pss are used for signing.
 */
private int chooseCombination()
{
    int result;

    result = (int) (java.lang.Math rint(java.lang.Math.random()*10));
    if(randomCombination < 1)
        randomCombination++;
    if(randomCombination >10)
        randomCombination--;
    return result ;
}

```

Die dafür notwendigen Berechnungen geschehen in der Klasse `Combinations`. Ist nun eine Kombination ermittelt, können Signaturen mit Zeitstempel erstellt werden. Das SCO versendet an alle beteiligten PSS einen Aufruf zur Signatur der übermittelten Nachricht und wartet dann passiv, bis alle Teilsignaturen der PSS eingetroffen sind.

```

/**
 * This method is used to sign a byte array.
 * the name is used because we want to be JCA conform.
 */
public byte[] engineSign (byte[] msg) throws SignatureException
{
    message = msg;
}

```

```

DTSSBigInt completeSignedMessage = DTSSBigInt.one, decryptedSignature;

SigningRequest req = new SigningRequest (message, "MD5withRSA", participants);

DTSSCommand cmd = req.getCommand();

// set the boolean signComplete back to false for the next signing order
signComplete = false;

for(int i = 1; i <= configurationInformation.threshold; i++)
{
    toPssConnection.getSendThread(i).sendCommand(cmd);
}

.....
}

```

Sind alle Teilsignaturen eingetroffen, muss die finale Signatur ermittelt werden. Dies geschieht nach dem Sakurai Schema wie in Kapitel 3 beschrieben. Es werden zwei Funktionen dazu benutzt, die den 2 finalen Schritten des Sakurai Schemas entsprechen.

```

/**
 * Computes the Signature S(I) of the chosen combination of 3 PSS.
 */
private DTSSBigInt computeSignatureOfCombination(int combinationNumber)
{
    DTSSBigInt result = DTSSBigInt.one;
    DTSSBigInt [] lambdas = combinations[combinationNumber-1].getLambdas();

    DTSSBigInt temp1, temp2, temp3;

    // gives the numbers of the 3 pss, that shall be used
    int [] usedPss;

    usedPss = combinations[combinationNumber-1].participants;

    for(int i = 0; i < 3; i++)
    {
        result = result.multiply(partialSignedMessages[usedPss[i]-1].modPow(
            lambdas[i], modul));
        result = result.mod(modul);
        GuiOutput.println(i+" .round complete");
    }
    return result;
}

```

```

/**
 * Computes the final signature S.
 */
private DTSSBigInt computeFinalSignature(DTSSBigInt combiSignature, byte []message)
{
    DTSSBigInt result;
    DTSSBigInt temp;
    DTSSBigInt temp2;

    temp2 = this.pkcsCompatibleMessage.modPow(b,modul);
    result = combiSignature.modPow(a,modul);
    result = result.multiply(temp2);
    result = result.mod(modul);
    return result;
}

```

Die Signatur der Nachricht ist nun beendet. Die Signatur wird noch verifiziert. Tritt dabei ein Fehler auf und die Verifikation schlägt fehl, muss man davon ausgehen, dass einer der PSS korrupt ist. Das gleiche Ergebnis tritt auf, wenn das SCO zu lange (die Wartezeit ist dabei wählbar) auf einen der PSS warten muss. Man kann dann von einem Ausfall des PSS oder einer Leitungsstörung ausgehen. In diesem Fall überprüft das SCO, ob noch genug andere PSS verfügbar sind, um eine gültige Signatur zu erzeugen. Ist dies der Fall, wird eine neue Kombination unter Ausschluss des korrupten Servers gewählt und es wird ein neuer Signaturversuch unternommen. War die Verifikation erfolgreich, muss der zeitgestempelte und signierte Request nun noch an den Antragssteller zurückgesendet werden.

## 4.6 Die PartialSigningServer

Das Package PSS beinhaltet die PartialSigningServer und deren Hilfsklassen. Die Klasse `PartialSigningServer` stellt hier die Hauptklasse dar. Sie regelt unter Benutzung der Kommunikationsklassen den Datenaustausch mit dem ReceptionServer, liest aus dem JavaKeyStore die während der Schlüsselgenerierung erstellten Parameter für die Signaturerstellung und signiert die anfallenden TSRs. Ist ein Request bearbeitet, wird er zum SCO zurückgesendet. Die Erstellung der Signatur des TSRs erfolgt in der Funktion `sign(request)`.

```

/**
 * signing requests can be satisfied here.
 */
private void sign(SigningRequest req)
{
    DTSSCommand cmd;

```

```

byte[] tbs = req.tstInfo;

// sign it and send the result back
try{
    signer.update (tbs);
    byte[] signed = signer.sign();
    if(pkcsCompatibleMessage==null)
    {
        GuiOutput.println("PKCS compatible message is NULL, ERROR !");
        System.exit(1);
    }
    cmd = new DTSSCommand(PKCSCOMPATIBLEMESSAGE,
                          pkcsCompatibleMessage.toByteArray());
    sendCommand(cmd);

    cmd = new DTSSCommand(SIGNED,signed);
    sendCommand(cmd);
}
catch (SignatureException e){
    GuiOutput.println("Signature exception! "+e);
    System.exit(1);
}

```

Sobald ein TSR zurückgesendet wurde, ist für den PSS die aktuelle Berechnung beendet und er ist bereit, den nächsten request zu bearbeiten.

## 5 Einige abschliessende Betrachtungen

Zum Abschluss der Beschreibung des Projekts sollen noch einige Betrachtungen angestellt werden sowie abschliessende Bemerkungen. Diese betreffen sowohl Arbeiten, welche die Anwendung im Kern nicht verändern, aber trotzdem der Vollständigkeit halber erwähnt werden sollen. Ausserdem sollen Anregungen gegeben werden, was bei einem weiteren Ausbau der Anwendung dem jetzigen Stand hinzugefügt werden könnte.

### 5.1 Implementierungen von GUIs zu Test- und Demonstrationszwecken

Die hier beschriebene Anwendung ist einerseits wegen der Verteiltheit oft schwer zu testen, Fehler können verschleppt werden und schlagen oft sehr spät und nicht regelmässig durch. Andererseits ist es bei Demonstrationen schwierig, den Fortgang des Protokolls und einzelner Schritte vorzuführen. Aus diesen Gründen wurde während der Entwicklung der Hauptanwendung ein GUI mitentwickelt, welches markante Eckdaten des Ausführungsverlaufs grafisch darstellt. So kann

mann z.B. bei Vorführungen die Anzahl der Iterationen, den aktuell ausgeführten Protokollschritt, den Datenverkehr und natürlich auch Zwischenergebnisse sichtbar machen. Ausserdem erleichtern GUIs die Arbeit der Administratoren bei der Eingabe bestimmter Parameter, im Gegensatz zu reinen Kommandozeilenparametern. Die Anwendung kann jedoch völlig ohne GUI benutzt werden.

## 5.2 Gegenüberstellung des Shamir- mit dem Sakurai Schema

Wie bereits in Kapitel 3 erwähnt, handelte es sich bei der Sakurai Version des *t-out of-k* Secret Sharings um eine Erstimplementierung innerhalb eines praxisnahen Projekts. Es lohnt sich daher, es dem bekannten Shamir Schema gegenüberzustellen. Der Implementierungsaufwand der Sakurai Version ist vor allem bei der Berechnung und Verteilung der Sharebruchstücke wesentlich geringer als in der Shamir Version. Allerdings ist die Erstellung des Parametersets im RS etwas aufwendiger, kann allerdings grösstenteils vorberechnet werden. Die Sharebruchstücke haben im Gegensatz zur Shamir (mit BGW) Methode alle die gleiche Grösse, sind aber insgesamt grösser. Dies spiegelt sich in längeren Signaturzeiten wieder, die mehr doppelt so gross ist als bei der Verwendung von Shares aus der Shamirmethode. Dies liegt auch an der grösseren Anzahl modularer Exponentiationen, die für eine Signatur nötig sind. Dafür ist es wesentlich leichter zu erkennen, welcher PSS eine falsche Signatur liefert und daraufhin eine andere Serverkombination zu wählen. Man kann abschliessend sagen, das in fehleranfälligen Systemen die Sakurai-Methode die effizientere ist aufgrund ihrer besseren Fehlerbehandlungsmöglichkeiten. Geht es jedoch um den hohen Durchsatz von Signaturen in einem bestimmten Zeitraum und kurze Rechenzeiten, ist der Shamir-Methode der Vorzug einzuräumen.

	786 bit	1024 bit	1536 bit	2048 bit
Signatur in Java	102	220	679	1500
modPow in Java	95	195	640	1350
Signatur in LiDIA nach Shamir	74	151	365	716
Signatur in LiDIA nach Sakurai	151	303	739	1500

Abbildung 9: Dauer verteilter Signaturen

## 5.3 Offene Fragen, Ausblicke

Wenn ein Projekt zu ende geht, stellt man sich immer die Frage, was hätte man noch mehr, noch besser tun können. Hier sollen einige Anregungen zu weiterführenden Arbeiten und noch offenen Fragen gegeben werden.

### 5.3.1 Tests

Während der Entwicklung der Anwendung wurden zahlreiche Tests, auch Dauertests durchgeführt. Die Testumgebung bestand aus 6 Intel Pentium III 500, JDK 1.2.2 und JDK 1.3 von SUN, Microsoft Windows NT4.0 und einem 10 mbit Ethernet Netzwerk. Im Bereich der Schlüsselgenerierung ist dies noch verhältnismässig einfach. Während dieser Tests konnten auch die optimalen Testparameter für diese Umgebung ermittelt werden. Benutzt man eine andere Testumgebung, ist es durchaus möglich, dass die Parameter variiert werden müssen. Der Einfluss der Siebgränze und der Anzahl der zu benutzenden Instanzen (Threads) ist in [Fuhr00] ausführlicher beschrieben. Will man jedoch

	786 bit	1024 bit	1536 bit	2048 bit
Dauer einer Iteration	73 ms	104 ms	243 ms	417 ms
Instanzen je Server (Threads)	20	10	10	5
Probefdivisionsgränze	20000	40000	80000	140000
Anzahl der Iterationen	729	1296	2209	3721
mittlere Gesamtdauer	53.2 s	134.8 s	8.95 min	25.86 min

Abbildung 10: Zeitmessungen der Schlüsselerzeugung mit optimalen Parametern

messen, wie sich die RS/PSS Klassen im Wirkbetrieb verhalten, wird es schon schwieriger. Hier fallen verschiedene Faktoren zusammen, welche Einfluss auf den Zeitstempeldienst haben. In diesem Projekt wurden lediglich ein gleichverteiltes Lastverhalten getestet, welches in der Praxis so sicher nicht vorkommt. Die Anzahl der zu bearbeitenden Requests ist tagsüber sicher um ein Vielfaches höher als zu Nachtzeiten. Diesen peak Zeiten muss sicherlich durch Pufferung anfallender Anträge, welche nicht sofort weitergeleitet werden können, da die PSS noch mit einem früheren Antrag beschäftigt sind, Rechnung getragen werden. Auch sind sicher nicht alle möglichen Fehlerfälle, welche während des Betriebes einer TSA auftreten können wie Stromausfall während einer laufenden Berechnung untersucht und abgefangen worden. Hier ergäbe sich sicher noch ein weites und interessantes Arbeitsfeld.

### 5.3.2 Die Zeitsynchronisation

Bei dieser Art der verteilten Anwendung ist es wichtig, dass alle beteiligten Stationen über eine globale, allen zugängliche Zeit verfügen. Dies kann man algorithmisch angehen (siehe [Mattern97]) oder aber über Hardware- und Softwaremechanismen. Hier wurde zur Synchronisierung der Server das *ntp* Protokoll benutzt. (siehe [ntp]) Dieses ist jedoch nur im mehrere 100 ms umfassenden Bereich genau und kann sicherlich auch beeinflusst werden. Eine Idee der NTT Laboratories (siehe [NTTLabs99]) beschreibt eine Lösung über mehrere unabhängige GPS Zeitgeberquellen und eine weitere Lösung mittels einer ISDN Zeitsynchronisation. Diese bieten sicher einen höheren Schutz gegen Aus-

fall und Verfälschung. Leider war es innerhalb dieses Projektes aus Zeit- und Kostengründen nicht mehr möglich, diese interessanten Varianten auszutesten.

### 5.3.3 Weitere mögliche Optimierungen

Benutzt man das *t-out of-k sharing* nach Shamir erhält man im Zusammenhang mit dem verwendeten BGW einen Share, welcher wesentlich grösser (fast doppelt so gross) ist als die Shares der anderen Server. Dies hat den Effekt, dass die Gesamtsignaturzeit gleich der Zeit der Signatur des Servers mit dem grösseren Share ist. Bei einem 2048 bit Modul benötigen die Server mit der normalen Sharegrösse ca. 370 ms für eine Signatur, der Server mit dem grossen Share jedoch benötigt ca. 740 ms. Dies scheint erstmal ein grosser Nachteil zu sein. Man kann diesen Nachteil jedoch reduzieren, indem man eine Pipeline benutzt, in der jeder der PSS (ähnlich dem schon vorgestellten Rollenkonzept) die Rolle eines Server mit kurzem und langem Share übernehmen kann. Dazu müssen allerdings

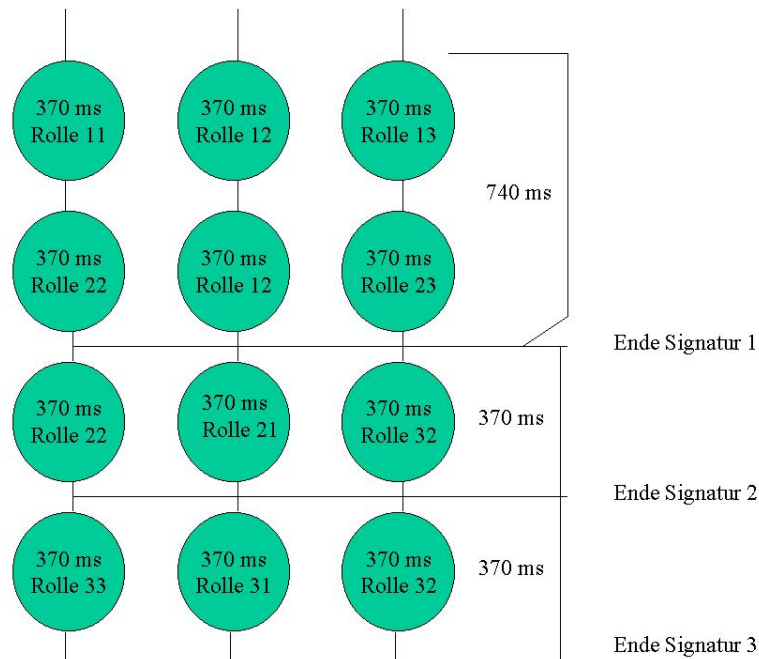


Abbildung 11: Rollenkonzept für eine Signaturpipeline

mehrere Schlüssel erzeugt werden und deren Shares so auf die die PSS verteilt werden, dass jeder im Besitz mindestens eines langen Shares ist. Haben nun PSS mit kurzen Shares ihre Arbeit beendet, bekommen sie keinen neuen Signaturauftrag, da der PSS mit dem langen Share noch am Rechnen ist. Hat man aber das Rollenkonzept implementiert, kann man einen neuen Signaturauftrag vergeben, nur das jetzt ein anderer Schlüssel benutzt wird, bei dem die Aufteilung der Shares so gestaltet ist, dass einer der jetzt untätigen PSS einen langen Share benutzt, der Server mit dem aktuell langen Share wird seine Berechnung

beenden und noch genug Zeit haben, den neuen Request mit nun kurzem Share zu signieren. Dies bringt zwar keine Erhöhung der Geschwindigkeit *einer* Signatur, die Rechenzeit bleibt gleich. Aber der Durchsatz der in einem Zeitabschnitt möglichen Signaturen wird um ca 40 Prozent erhöht. Die Zeit *einer* Signatur berechnet sich nach  $t_{sign_{all}} = \max(t_{sign_1}, \dots, t_{sign_k})$ . Daraus folgt man schafft eine Signatur je ca. 740 ms, somit benötigen 3 Signaturen ca 2220 ms. Benutzt man das Rollenkonzept und eine Pipeline erhöht sich der Durchsatz derart, dass 3 Signaturen nun ca 1480 ms benötigen, was einer Rate (abzüglich des zusätzlichen Verwaltungsaufwandes) von ca 40 - 45 Prozent entspricht. Geht man (theoretisch) von einer konstanten Gleichverteilung der Requests aus, schafft man ohne Pipeling ca 115000 Signaturen am Tag, mit Pipelining werden es ca 160000 Signaturen, die berechnet werden können. Dies sind jedoch theoretische Überlegungen, welche in dem Projekt nicht mehr realisiert wurden.

Eine andere, eher praktische Optimierung bietet die Verwendung von Rechner-

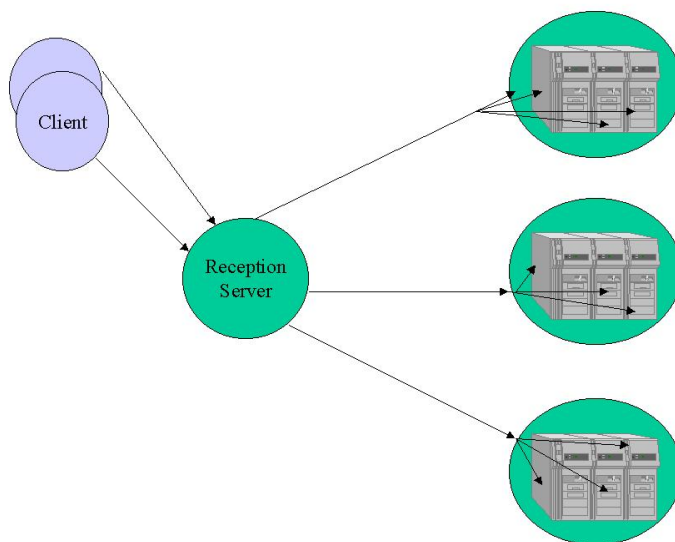


Abbildung 12: Benutzung von Rechner (Signatur) Clustern

clustern. Stellt man statt eines PSS in den gleichen Raum mehrere Stationen, die den gleichen Key benutzen, können viel mehr Signaturen in der gleichen Zeit erstellt werden. Die Lastenkontrolle könnte leicht vom SCO übernommen werden, welches freie PSS mit neuen Anträgen beliefert und Servern, welche gerade eine Signatur erstellen, keine neuen Anträge zustellt. Der Vorteil dieser Methode ist die einfache Realisierung und die leichte Skalierbarkeit, der Nachteil sind sicherlich die anfallenden Kosten für die Hardware.

## A Glossar

CA : Certification Authority  
TTP : Trusted Third Party  
TA : Trusted Authority  
Skg : Shared Key Generator  
PSS : Partial Signing Server  
RPC : Remote Procedure Call  
RS : Reception Server  
SSL : Secure Socket Layer  
MS : Manager Server  
RSA : Verschlüsselungs Verfahren nach Rivest Shamir Adleman  
JCA : Java Cryptographic Architecture  
BGW : Ben Ohr Goldwasser Widgerson  
PKCS : Public Key Cryptographic Standard  
RMI : Remote Method Invocation  
TCP/IP : Transmission Control Protocol/Internet Protocol  
TSA : Time Stamp Authority  
PKI : Public Key Cryptography  
MD5 : Message Digest 5  
gcd : greatest common divisor  
SigG : deutsches Signaturgesetz  
SSL : Secure Socket Layer  
dos : denial of service

## B Klassenübersicht

### B.1 Package SharedKeyGeneration

BGW.java  
ComputeN.java  
DisplayKey.java  
DistPrimeTest.java  
DistributedSieving.java  
DTSSBigInt.java  
DTSSCommand.java  
DTSSFpPolynomial.java  
DTSSKey.java  
ExponentShareGeneration.java  
FpPolynomial.java  
GlobalModul.java  
NoMoreDataException.java  
Polynomial.java  
PrimeTable.java  
ProgressWindow.java  
ProtocolPart.java  
SecretSharing.java  
SequenceNumber.java  
SkgConnection.java  
SkgConnectionReceiveThread.java  
SkgConnectionSendThread.java  
SkgInstance.java  
SkgListener.java  
StartSharedKeyGenerator.java  
StartSharedKeyGeneratorThread.java  
SubProtocolPart.java  
ToManagerServerConnection.java  
ToManagerServerReceiveThread.java  
ToManagerServerSendThread.java  
TrialDivision.java

### B.2 Package ReceptionServer

ClientConnection.java Combinations.java  
FromPartialSigningServerReceiveThread.java ReceptionServer.java  
ReceptionServerThread.java SignatureControlObject.java  
  
ToPartialSigningServerConnection.java  
ToPartialSigningServerSendThread.java

### **B.3 Package ManagerServer**

ManagerServer.java  
ManagerServerListener.java  
ManagerServerReceiveThread.java  
ManagerServerSendThread.java  
ManagerServerThread.java

### **B.4 Package PartialSigningServer**

CertificateWriter.java  
FromReceptionServerReceiveThread.java  
PartialSigningServer.java  
PartialSigningServerThread.java  
SigningRequest.java  
ToReceptionServerSendThread.java

### **B.5 Package JCA**

CDCSharedProvider.java  
KeyToDTSS.java  
MD5withRSA.java

### **B.6 Package IETF**

MessageImprint.java  
PKIStatusInfo.java  
PolicyInformation.java  
SocketTransportProtocol.java  
TimeStampReq.java  
TimeStampResp.java  
TSTInfo.java  
TSTInfoData.java

### **B.7 Package GUI**

ClientFrame.java  
ClientGUI.java  
GuiOutput.java  
ManagerServerFrame.java  
ManagerServerGUI.java  
PartialSigningServerFrame.java

PartialSigningServerGUI.java  
SharedKeyGeneratorFrame.java  
SharedKeyGeneratorGUI.java

## C Zeitmessungen zur Schlüsselgenerierung

Die hier verwendeten Bilder entstanden in Zusammenarbeit mit Dipl.Inf Ar-nulph Fuhrmann während der Vorbereitung der Dokumente [CDC] und [Fuhr00] und dürfen mit seiner freundlichen Genehmigung verwendet werden.

### C.1 768 bit

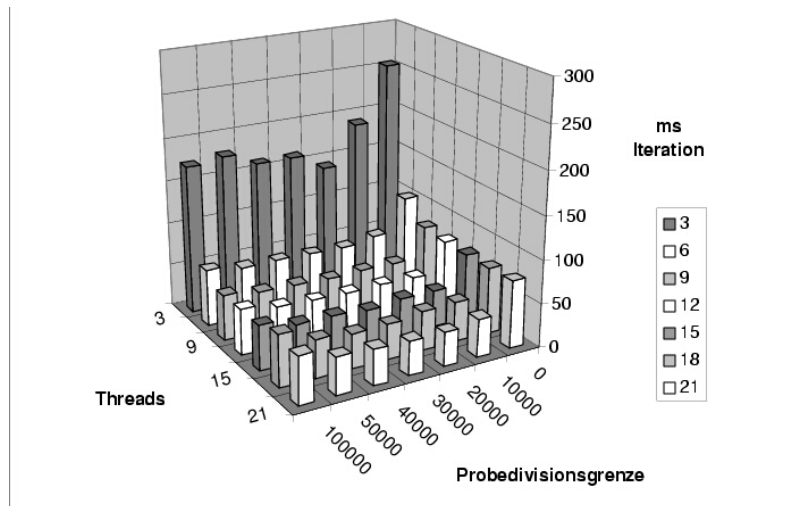


Abbildung 13: Zeit pro Iteration bei 768-Bit und 3 Servern

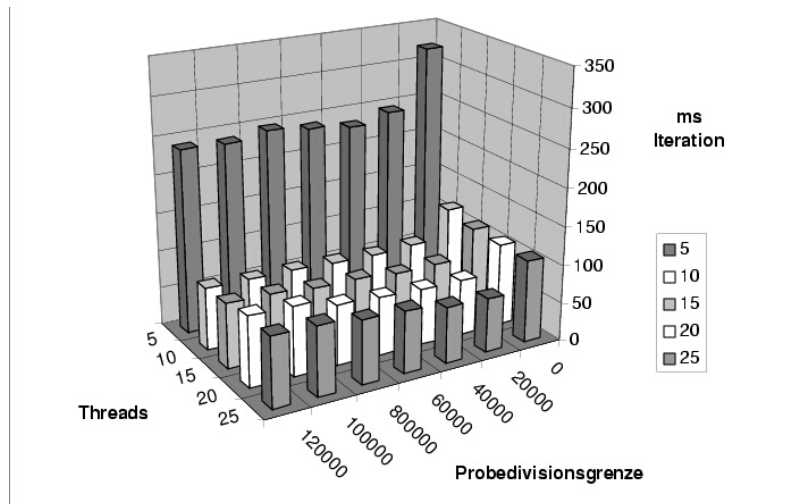


Abbildung 14: Zeit pro Iteration bei 768-Bit und 5 Servern

## C.2 1024 bit

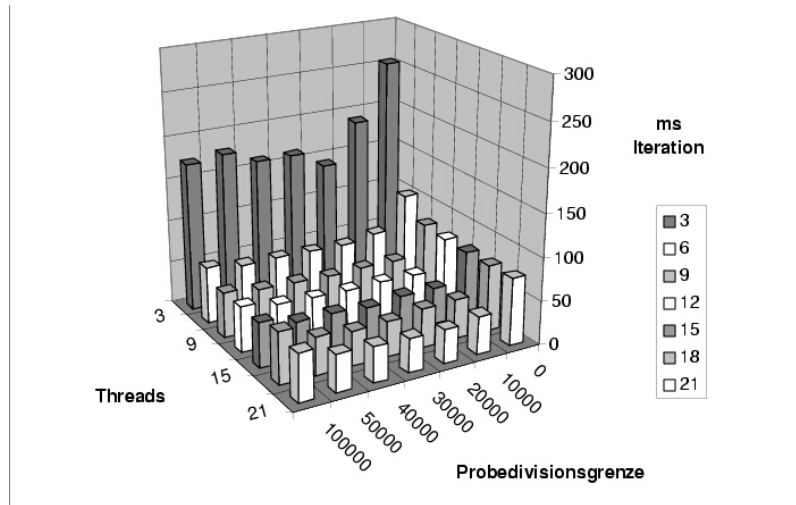


Abbildung 15: Zeit pro Iteration bei 1024-Bit und 3 Servern

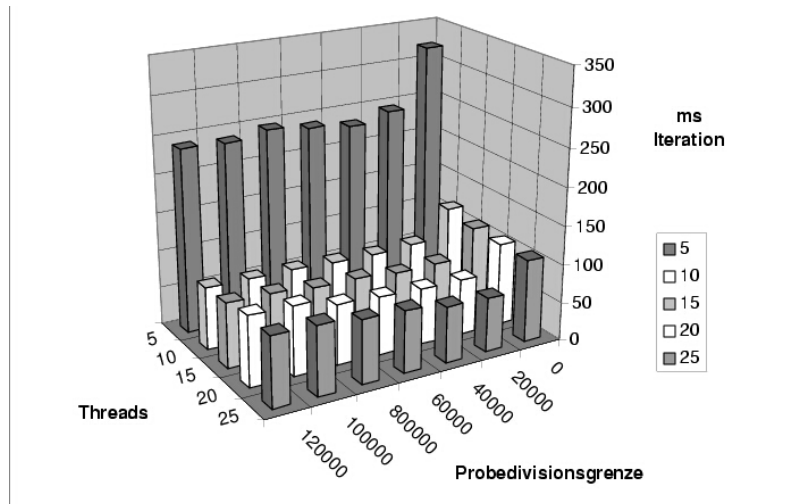


Abbildung 16: Zeit pro Iteration bei 1024-Bit und 5 Servern

### C.3 1536 bit

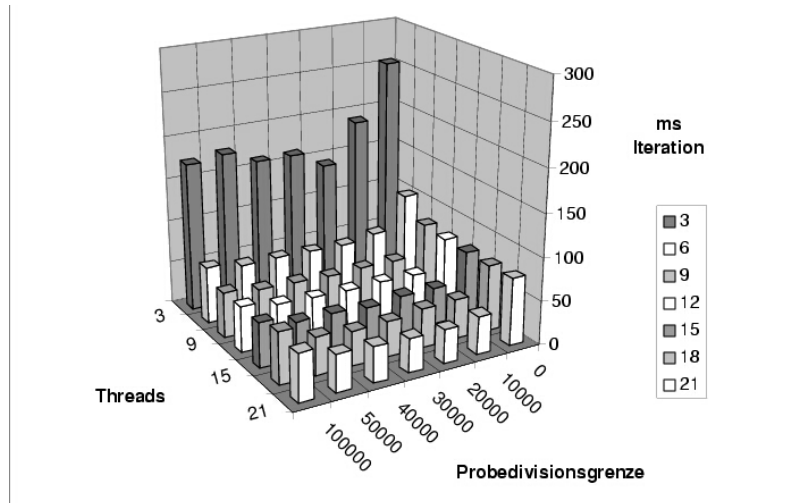


Abbildung 17: Zeit pro Iteration bei 1536-Bit und 3 Servern

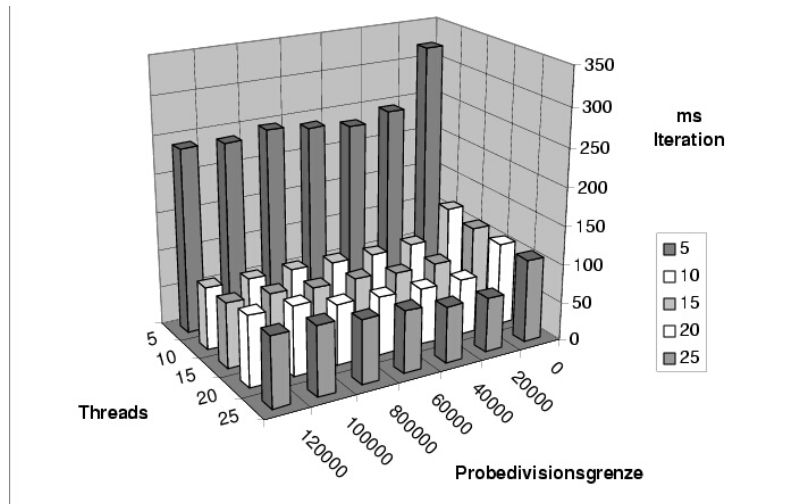


Abbildung 18: Zeit pro Iteration bei 1536-Bit und 5 Servern

## C.4 2048 bit

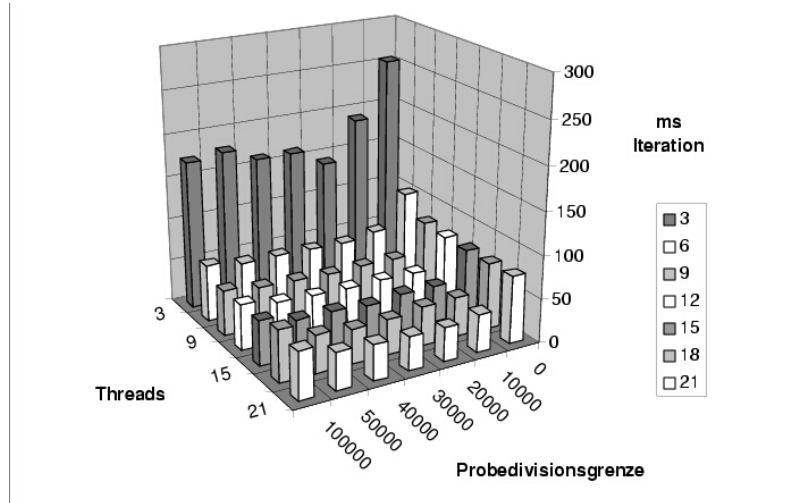


Abbildung 19: Zeit pro Iteration bei 2048-Bit und 3 Servern

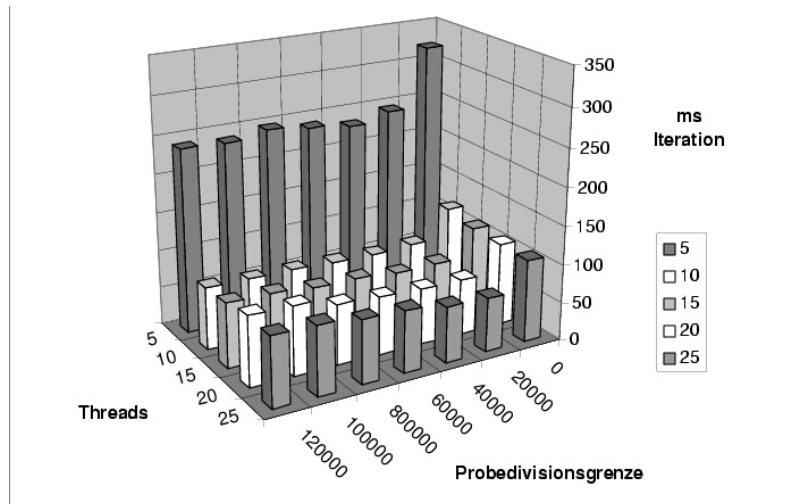


Abbildung 20: Zeit pro Iteration bei 2048-Bit und 5 Servern

## Literatur

- [JNU] David Flanagan, *Java in a Nutshell*, O'Reilly 1998
- [JSC] Scott Oaks, *Java Security – Java 1.2 Edition*, O'Reilly 1998
- [JCA] Jonathan Knudsen, *Java Cryptography – Java 1.2 Edition*, O'Reilly 1998
- [BoFr97] D. Boneh, M. Franklin, *Efficient Generation of Shared RSA keys* Konferenzband Advances in Kryptology - Crypto 97, LNCS 1294 S. 425-439
- [Buch99] J. Buchmann, *Einführung in die Kryptografie* Springer Verlag, 1999
- [CDC] H. Appel, I. Biehl, A. Fuhrmann, M. Ruppert, T. Takagi, A. Takura, Ch. Valentin *Ein sicherer, robuster Zeitstempeldienst auf der Basis verteilter RSA-Signaturen*, Technical Report No. TI-21/99 <http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/TR>, 1999
- [Mattern97] F. Mattern [www.informatik.tu-darmstadt.de/VS/Lehre/SS97](http://www.informatik.tu-darmstadt.de/VS/Lehre/SS97)
- [Ben86] J. Benaloh (Cohen), *Secret sharing homomorphisms: keeping shares of a secret*, Crypto 86, S.251-260
- [Fuhr00] A. Fuhrmann *Studienarbeit : Verteilte effiziente RSA-Schlüsselerzeugung in Java*, TU Darmstadt 2000
- [BGW88] M. Ben-Or, S. Goldwasser, A. Wigderson *Completeness theorems for non-cryptographic fault tolerant distributed computation* STOC 1988, S.1-10
- [Gord98] R. Gordon, *Essential JNI : Java Native Interface (Essential Java)*, Prentice Hall, 1998.
- [Harold] E.R. Harold, *JAVA Network Programming*, O'Reilly 1997
- [LiDIA] LiDIA - A Library for Computational Number Theory, <http://www.informatik.tu-darmstadt.de/TI/LiDIA/>
- [MaWuBo99] M. Malkin, Th. Wu und D. Boneh, *Experimenting with Shared Generation of RSA keys*, Konferenzband Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS), S. 43-56.
- [ntp] TUD HRZ: Netzwerk [www.tu-darmstadt.de/hrz/netz/netzdienste/ntp.html](http://www.tu-darmstadt.de/hrz/netz/netzdienste/ntp.html)
- [Schneier96] B. Schneier *Angewandte Kryptographie* Addison Wesley Verlag, 1996
- [SigG] Deutsches Signaturgesetz [jurcom5.juris.de/bundesrecht/sigg\\_2001/](http://jurcom5.juris.de/bundesrecht/sigg_2001/) Bundesministerium der Justiz 2001
- [NTTLabs99] H. Fukuda und Satoshi Ono *Internet Quality Measurement System*, NTT Information SHaring Platform Laboratories

- [Stinson95] Douglas R. Stinson *Cryptography, Theory and Practice*, CRC Press, 1995
- [Val99] Ch. Valentin, *PKI Seminar, Vortrag über die Verwendung und den Einsatz von Zeitstempeln*
- [PKCS] PKCS-Standards, RSA Laboratories Public Key Cryptographic Standards, *www.rsalabs.com/pkcs*
- [IAIK] IAIK-JCE Java-Paket, Implementierung eines Java Cryptographic Providers, *jcewww.iaik.tu-graz.ac.at*