Entwurf und Realisierung von Sicherheitsmechanismen für eine Infrastruktur für digitale Bibliotheken

Diplomarbeit

von
Razi Lotfi-Tabrizi

eingereicht bei

Prof. Dr. Oswald Drobnik
Professur für Architektur und Betrieb verteilter Systeme
Fachbereich Biologie und Informatik
Johann Wolfgang Goethe-Universität
Frankfurt am Main

Mai 2002

Erklärung

Hiermit versichere ich, Razi Lotfi-Tabrizi, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kronberg, den 20. Mai 2002 .....

Mein besonderer Dank

gilt Prof. Dr. Oswald Drobnik, der die vorliegende Diplomarbeit mit Umsicht und Hilfsbereitschaft betreut hat. Nicht minder bedeutsam ist sein Engagement als Hochschullehrer, der mir im Rahmen seiner Vorlesungen und Praktika in vielfältiger Weise das „Rüstzeug“ für meine Arbeit vermittelte.

Zu großem Dank bin ich auch Dr. Christian Mönch verpflichtet, dessen profunde fachliche Anregungen und Diskussionsbeiträge für mich sehr wichtig waren.

Danken möchte ich nicht zuletzt allen Professoren und wissenschaftlichen Mitarbeitern, in deren Vorlesungen und Praktika/Seminaren ich ein breites Spektrum an Wissen erworben habe, das mir bei dieser Arbeit von großem Nutzen war.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

> *„Von dem menschlichen Wissen überhaupt, in jeder Art, existiert der allergrößte Teil stets nur auf dem Papier, in den Büchern, diesem papierenen Gedächtnis der Menschheit. Nur ein kleiner Teil desselben ist, in jedem gegebenen Zeitpunkt, in irgendwelchen Köpfen wirklich lebendig. [...] Daher sind die Bibliotheken allein das sichere und bleibende Gedächtnis des menschlichen Geschlechts, dessen einzelne Mitglieder alle nur ein sehr beschränktes und unvollkommenes haben.“*
>
> Arthur Schopenhauer (1788-1860)

1.1 Motivation

Als der deutsche Philosoph Arthur Schopenhauer von den Bibliotheken als dem sicheren und bleibenden Gedächtnis des menschlichen Geschlechts sprach, waren die damaligen Büchereien noch nicht mit den aktuellen Problemen der modernen Bibliotheken konfrontiert. Die Informationsflut der modernen wißbegierigen Gesellschaft hat die traditionellen Bibliotheken an ihre Grenzen gebracht, denn die Realität hat gezeigt, daß jede Bibliothek – genauso wie ihr Schöpfer – nur ein „beschränktes Gedächtnis“ besitzt.

Die Menge der täglich produzierten Informationen hat ein solches Ausmaß angenommen, daß man sie nicht mehr mit den traditionellen Bibliotheken beherrschen kann. Man geht davon aus, daß allein die Zahl der wissenschaftlichen Publikationen sich innerhalb von zehn Jahren verdoppelt[1] [Groe96]. Die Deutsche Bibliothek in Frankfurt am Main hatte Ende 2000 einen Hauptbestand von insgesamt 4.726.037 Titeln. Das ist gegenüber dem Vorjahr ein Zuwachs von 170.670 Titeln[2]. Die vielfältigen Schwierigkeiten hinsichtlich der zu verwaltenden Dokumente beziehen sich nicht nur auf das Problem, das sich bei der Suche nach einer bestimmten Information in einer solchen Menge an Dokumenten ergibt; es geht auch um rein organisatorische Probleme, beispielsweise die Lagerung der Dokumente und das Fassungsvermögen der Bibliotheken. So mußte die Deutsche Bibliothek in Frankfurt am Main 1997 aus Platzmangel in ein größeres Gebäude umziehen.

[1] Diese Verdopplung stellt keine reine Verdopplung des Wissens dar; hier sind natürlich auch häufig Wiederholungen zu beobachten.

[2] Siehe Jahres-Statistik der Satzarten für den Hauptbestand der DDB für das Jahr 2000: http://support.ddb.de/iltis/statistik/ddb/2000-seiten/jahr.htm

Lagerung, Archivierung, Transport und Produktion der Bücher und anderer konventioneller Dokumente sind sehr aufwendig und teuer. Diese Faktoren stellen ein großes Problem dar, weil den Bibliotheken trotz der steigenden Zahl der zu verwaltenden Objekte nicht unbedingt mehr finanzielle Mittel zur Verfügung stehen (sie sind eher mit Mittelkürzungen konfrontiert). Um solche Mengen an Dokumenten trotzdem sinnvoll zu organisieren, sind andere Einrichtungen erforderlich, die die traditionellen Bibliotheken ergänzen. Ein Konzept, das in der Fachliteratur als Lösung (oder zumindest als Teillösung) für diese Probleme erwähnt wird, trägt den Namen „digitale Bibliotheken“; in [Groe96] werden sie wie folgt definiert:

> „Die digitale Bibliothek ist eine digital strukturierte Sammlung digitaler Information, die über digitale Netze bereitgestellt und abgerufen wird, heute in – weltweit – verteilten offenen Informationssystemen, wobei die Kommunikationsschnittstellen des Internets bzw. des World Wide Web (globale digitale Bibliothek) benutzt werden.“

Bei den digitalen Bibliotheken[3] handelt es sich somit um Einrichtungen, die für den Umgang mit den *digitalen Dokumenten* konstruiert sind. Digitale Dokumente sind beliebige digital-codierte Informationen, wie beispielsweise Tonaufnahmen, Bilder, Grafiken, Animationen, Videos oder Texte, die in einer elektronischen Form vorliegen.

Die digitalen Bibliotheken werden in der Literatur häufig auch als *elektronische Bibliotheken* oder auch als *virtuelle Bibliotheken* bezeichnet. Sie heißen „virtuell“[4], weil sie im Gegensatz zu einer realen Bibliothek, die ein Dokument tatsächlich in einer physikalischen Art – beispielsweise als Buch – besitzen muß, nur einen Verweis auf ein Dokument haben können, das an einem anderen Ort existiert. Damit ist eine digitale Bibliothek nicht mehr ein umgrenztes Gebäude, das Dokumente aufbewahrt, sondern eine verteilte Infrastruktur, deren Hauptaufgabe darin besteht, Informationen zu verwalten und zu verteilen. Aus diesem Grund spricht man häufig auch von „library without walls“, also einer Bibliothek ohne Mauern und Wände.

Die Entwicklung der digitalen Bibliotheken geht zurück auf die Mitte des 20. Jahrhunderts. 1945 entwarf Vannevar Bush das *Memex-System*, das für „Memory Extender“ steht. Bei Memex handelt es sich um ein entworfenes System, das aber nicht realisiert wurde. Es hätte auf Mikrofilmen basiert und war als interaktive Erweiterung für eine Enzyklopädie (Encyclopædia Britannica) gedacht. Es hätte außer dem schnellen Zugriff auf die Informationen auch die Möglichkeit zur Verlinkung von Informationen geboten. Ein anderer Vordenker der digitalen Bibliotheken ist Ted Nelson. Auf ihn geht auch der Begriff *Hypertext* zurück [Niel95]. Er versuchte seit dem Ende der sechziger Jahre, dieses Konzept von Hypertexten bei der Entwicklung des *Xanadu-Systems* technisch zu realisieren.

INDIGO, eine Infrastruktur für die digitalen Bibliotheken, wurde an der Universität Frankfurt am Main im Rahmen einer Dissertation [Moe01] entwickelt. Dabei handelt es sich um verteilte Anwendungen, die über ein Netzwerk miteinander verbunden sind. Sobald eine solche Infrastruktur über ein offenes Kommunikationsnetz[5] wie das Internet geführt wird,

[3] Digitale Bibliothek := digital library [engl.]
[4] virtuell := scheinbar vorhanden [Brockhaus]
[5] Ein offenes Kommunikationsnetz ist ein Netzwerk, über das eine *offene Kommunikation* geführt wird. Unter einer offenen Kommunikation versteht man nach [DU93] die Übertragung zwischen Rechnern verschiedener Hersteller über unterschiedliche Übertragungswege. Wichtig ist dabei, daß jeder, der eine entsprechende Zugangsberechtigung besitzt, an der Kommunikation teilnehmen kann.

ist sie automatisch Sicherheitsrisiken ausgesetzt, von denen auch andere Infrastrukturen in diesem Netz betroffen sind.

Als Beispiel für solche Risiken seien die Sicherheitsverletzungen hinsichtlich der Authentizität und Integrität[6] der zu verwaltenden Dokumente erwähnt: Der Nutzer einer digitalen Bibliothek muß bei der INDIGO-Infrastruktur sicher sein können, daß ein Dokument, das er von dieser Bibliothek bezieht, tatsächlich auch von dem angegebenen Autor stammt. Die INDIGO-Infrastruktur muß sich daher mit den Gefahren, die von offenen Netzen ausgehen, auseinandersetzen und versuchen, den Beteiligten ein ausreichendes Maß an Sicherheit zu bieten.

1.2 Ziele und Vorgehensweise

Die vorliegende Diplomarbeit beschäftigt sich mit dem Gebiet der vernetzten verteilten digitalen Bibliotheken in den offenen Kommunikationsnetzen und den daraus resultierenden spezifischen Sicherheitsproblemen. Hierbei wird speziell auf die Sicherheitsaspekte bei der INDIGO-Infrastruktur eingegangen.

Die Arbeit verfolgt zwei *Ziele*:

1. Entwurf von Sicherheitsmechanismen für die INDIGO-Infrastruktur und
2. Realisierung dieser Sicherheitsmechanismen.

Diese beiden Ziele werden in den Entwicklungsphasen *Entwurfsphase* und *Realisierungsphase* erreicht. Abbildung 1.1 bietet eine Übersicht der Vorgehensweise beim Entwurf und der Realisierung der Sicherheitsmechanismen. Dabei erfolgt der Entwurf der Sicherheitsmechanismen in Anlehnung an das empfohlene Vorgehensmodel in [Gri94].

In der Entwurfsphase wird zuerst eine *Sicherheitsanalyse* der INDIGO-Infrastruktur durchgeführt. Anschließend werden – basierend auf der Sicherheitsanalyse – *Sicherheitskonzepte* für diese Infrastruktur entwickelt.

Bei der Sicherheitsanalyse der INDIGO-Infrastruktur werden zunächst durch eine abstrakte Betrachtungsweise der Infrastruktur die einzelnen *Akteure* und die *schützenswerten Güter* extrahiert (*Abstraktion der Sicherheitsziele*)[7]. Anschließend werden aus der Sicht der Akteure bestimmte *Sicherheitsanforderungen* an diese schützenswerten Güter gestellt.

Nach der Sicherheitsanalyse der INDIGO-Infrastruktur folgen die Sicherheitskonzepte. In dieser Phase werden zur Erfüllung der Sicherheitsanforderungen *Sicherheitsmaßnahmen* erörtert. Das Hauptaugenmerk richtet sich dabei auf den Schutz der Metadokumente, der Dokumentmethoden und des INDIGO-Servers. Die Entwurfsphase von Sicherheitsmechanismen für die INDIGO-Infrastruktur ist abgeschlossen, nachdem aus der Menge der Sicherheitsmaßnahmen die *Sicherheitsrichtlinien* der INDIGO-Infrastruktur ausgewählt worden sind.

Nach der Entwurfsphase werden die Sicherheitsmechanismen anhand des dafür erarbeiteten Entwurfs realisiert, das heißt, es werden die beim Entwurf ausgewählten Sicherheitsrichtlinien der INDIGO-Infrastruktur implementiert. Ausgangspunkt der Implementierung ist der in der Dissertation [Moe01] realisierte Prototyp der INDIGO-Infrastruktur.

[6] Die Beschreibung dieser Sicherheitsanforderungen findet sich im Abschnitt 3.1.1.
[7] Die genaue Beschreibung der Begriffe „Akteure“ und „schützenswerte Güter“ erfolgt in Kapitel 4.

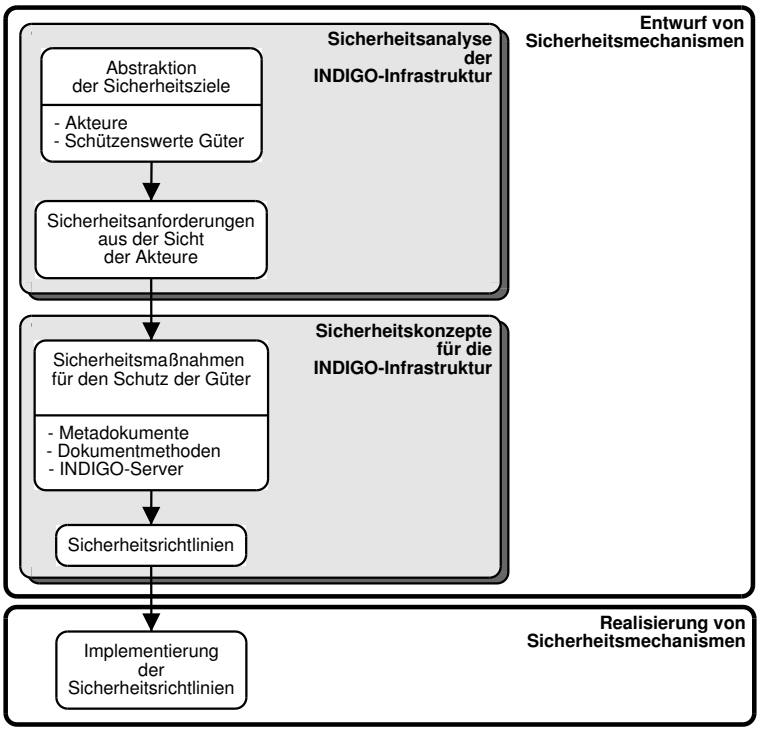

Abbildung 1.1 Vorgehensweise beim Entwurf und bei der Realisierung von Sicherheitsmechanismen für die INDIGO-Infrastruktur

1.3 Gliederung der Diplomarbeit

Die Beschreibung der digitalen Bibliotheken und der INDIGO-Infrastruktur erfolgt im zweiten Kapitel. Hier werden die Komponenten dieser Infrastruktur beschrieben. Außerdem wird deren Funktionalität an Hand einiger Beispiele näher erläutert.

In Kapitel 3 werden die technischen Grundlagen der Sicherheit vorgestellt, die zum Verständnis der Sicherheitsrisiken bei der INDIGO-Infrastruktur notwendig sind. Hier werden neben den wichtigsten Verschlüsselungsverfahren auch kurz die Sicherheitsprotokolle erläutert, die später in Kapitel 5 verwendet werden.

Das vierte und fünfte Kapitel umfassen die Entwurfsphase von Sicherheitsmechanismen. Im vierten Kapitel erfolgt im Rahmen einer sicherheitskritischen Betrachtung die Sicherheitsanalyse der INDIGO-Infrastruktur. In Kapitel 5 werden die Sicherheitskonzepte für die INDIGO-Infrastruktur beschrieben.

In Kapitel 6 werden die im Rahmen der Diplomarbeit angefertigten Implementierungen zur Realisierung der Sicherheitsmechanismen vorgestellt.

Das letzte Kapitel beinhaltet eine Zusammenfassung und einen Ausblick auf die Entwicklung und den Einsatz der Sicherheitsmechanismen bei digitalen Bibliotheken.

Kapitel 2

INDIGO – Infrastruktur für digitale Bibliotheken

Die digitalen Bibliotheken sind als langlebige Einrichtungen zu sehen. Damit sie auch den Bedürfnissen der zukünftigen Anwender gerecht werden, sollten sie gewisse Anforderungen erfüllen. Nach [Moe98] sind die wichtigsten dieser Anforderungen:

- **Skalierbarkeit** Bei der Skalierbarkeit geht es um die Menge und die Größe der zu verwaltenden Datenbestände und Anwender. Eine Bibliothek muß in der Lage sein, eine fast unbegrenzte Menge an Dokumenten und Anwendern zu verwalten. Jede zentral geführte Bibliothek hat aber bezüglich der Skalierbarkeit stets ihre Grenzen. Um diese Grenzen zu erweitern, setzt man bei den digitalen Bibliotheken wie bereits erwähnt auf die verteilte Verwaltung der Dokumentbestände.

- **Erweiterbarkeit** Eine Bibliothek sollte in der Lage sein, neue Dokumenttypen und Dokumentformate zu unterstützen. Allein in den letzten Jahren wurden viele neue Formate entwickelt, die sich bei den Anwendern zu quasi Standards entwickelten. Hier sind als Beispiele das *Extensible Markup Language (XML)* [XML], das Musikformat *MPEG-1 Audio Layer III (MP3)* [MP3] oder das Videoformat *DivX* [DivX] zu erwähnen. Eine digitale Bibliothek sollte so modular aufgebaut sein, daß sie dynamisch auch um eventuell in Zukunft entwickelte Dokumenttypen und Formate ergänzt werden kann.

- **Orthogonalität** Alle verwendeten Datenformate sollten in der Lage sein, einige elementare Operationen zu unterstützen. Orthogonalität stellt sicher, daß gewisse Operationen bei jedem Dokument – unabhängig von dessen Dokumenttyp – verwendet werden dürfen. Beispiele für solche orthogonale Operationen sind Operationen, die zum Präsentieren eines Dokuments verwendet werden. Eine andere wichtige orthogonale Operation kann die Such-Operation sein, mit der man in einem Dokument nach bestimmten Informationen suchen kann.

- **Plattformunabhängigkeit** Der Zugriff auf die Dokumente der digitalen Bibliothek sollte von verschiedenen Plattformen aus möglich sein.

Diese Anforderungen können aber nicht alle gleichzeitig in gleichen Maßen erfüllt werden. Allein, wenn man sich die orthogonale Operation „Suchen“ betrachtet, stellt man

fest, daß die Umsetzung dieser Operation bei den unterschiedlichen Dokumenttypen sehr schwierig zu realisieren ist. Während das Suchen nach einem bestimmten Wort in den Text-Dokumenttypen (sogar bei unterschiedlichen Dokumentformaten) leicht realisiert werden kann, ist das Suchen nach bestimmten Mustern in Bildern, Tonaufnahmen oder gar in Videos sehr umständlich.

An der Universität Frankfurt am Main wurde die *INDIGO-Infrastruktur* für die digitalen Bibliotheken entwickelt, mit dem Ziel die oben beschriebenen Anforderungen zu erfüllen [Moe01]. Bei dieser Infrastruktur besteht eine Bibliothek aus einer Menge von verteilten Servern, die die Dokumente dieser Bibliothek verwalten. Dieser Server-Verbund stellt den Speicher dieser Bibliothek dar. Wenn ein Anwender auf ein bestimmtes Dokument zugreifen möchte, muß er mit einem dieser Server in Verbindung treten. Die wichtigsten Komponenten, die bei dieser Infrastruktur zum Einsatz kommen, sind:

- Dokumente,
- Dokumentmethoden,
- INDIGO-Ausführungs-Server.

Ein sehr vereinfachtes Beispiel für die Zusammenarbeit dieser Komponenten stellt die Abbildung 2.1 dar. Es ist zu beachten, daß bei der INDIGO-Infrastruktur ein INDIGO-Ausführungs-Server sowohl bei jedem Rechnerknoten der Bibliothek als auch bei jedem Anwender verwendet wird. Bei dem vorliegenden Beispiel initiiert der Anwender die Präsentationsoperation an einem Dokument, das sich auf einem Ausführungs-Server aus dem Server-Verbund der entfernten Bibliothek befindet. Dieser Ausführungs-Server sucht in seiner Datenbank nach dem geforderten Dokument. Anschließend führt er an diesem Dokument – auf der Serverseite – die entsprechende Methode aus. Als Resultat der Methodenausführung wird das Dokument dem Ausführungs-Server des Anwenders geschickt. Dieses Dokument wird von dem lokalen Ausführungs-Server des Anwenders gespeichert. Anschließend wird in der Laufzeitumgebung die entsprechende Methode zum lokalen Präsentieren dieses Dokuments initialisiert.

Diese Komponenten der Infrastruktur werden in diesem Kapitel näher beschrieben. Im Abschnitt 2.4 dieses Kapitels werden anschließend einige Beispiele für die Zusammenarbeit und die Funktionsweise dieser Komponenten erläutert. Die Definitionen in diesem Abschnitt halten sich an die in [Moe01], [Moe98] und [Moe00] beschriebene Infrastruktur.

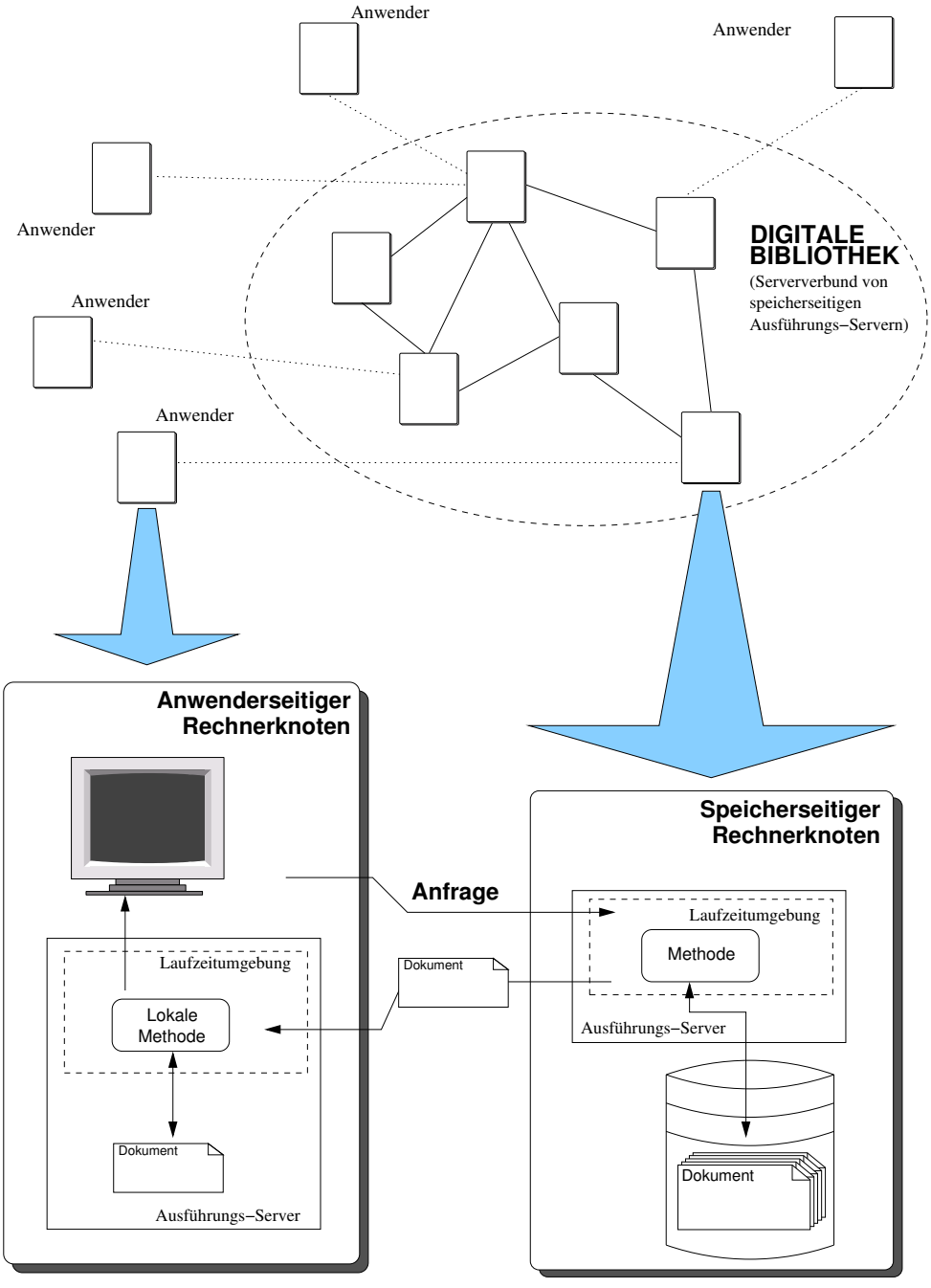

Abbildung 2.1 Komponente der INDIGO-Infrastruktur

2.1 Dokumente

Bei den in dieser Infrastruktur verwendeten *Dokumenten* handelt es sich eigentlich um *Metadokumente*. Die Metadokumente beinhalten außer dem eigentlichen Dokument auch *Metadaten* über das jeweilige Dokument. Unter Metadaten versteht man zusätzliche Daten, die dem Dokument zugrunde liegen, wie beispielsweise der Name des Autors, der Dokumenttyp und die Sprache, in der das Dokument erstellt wurde.

Ein Metadokument kann in der INDIGO-Infrastruktur beispielsweise vom Autor des Dokuments erstellt werden. Ein Beispiel für ein solches Metadokument kann wie folgt aussehen:

```
Content-Type: application/x-metadoc;
 boundary="ekidpUnhoJ"

--ekidpUnhoJ
Content-Type: application/x-metadoc-attributes

author: Razi Lotfi-Tabrizi
uses: base self global io
type: text/plain
title: Testdokument fuer die INDIGO
--ekidpUnhoJ
Content-Type: application/x-metadoc-methods

Present http://62.104.191.241/Text/present.zip application/java
Describe http://62.104.191.241/Text/describe.zip application/java
private.localPresent http://62.104.191.241/Text/lpresent.zip application/java
--ekidpUnhoJ
Content-Type: application/x-metadoc-content

Dies ist ein Testdokument fuer die INDIGO Infrastruktur.
Es enthaelt als Inhalt nur diese beiden Zeilen.

      ⋮

--ekidpUnhoJ--
```

Die Metadokumente halten sich an das in Multipurpose Internet Mail Extensions definierte *MIME-Format* [RFC2045] und sind vom Typ „`application/x-metadoc-attributes`“, der in dieser Infrastruktur definiert worden ist. Die in „`boundary`“ definierte Zeichenkette wird zur Markierung der Begrenzungen zwischen den unterschiedlichen Abschnitten des Metadokuments verwendet.

In der INDIGO-Infrastruktur bestehen die Metadokumente aus den folgenden drei Abschnitten:

- **Attribute** Dieser Abschnitt beinhaltet Informationen über das ursprüngliche Dokument. Er stellt (fast) das Pendant zu den Katalogen bei den realen Bibliotheken dar. Das Metadokument aus dem vorhergehenden Beispiel beinhaltet beispielsweise Informationen über den Autor, die benötigten Schnittstellen (gekennzeichnet durch

den Schlüsselwort „`uses`“), den Dokumenttyp und den Titel des Dokuments. Die Attribute eines Dokuments können nur vom Autor oder vom Ausführungs-Server modifiziert werden; die Dokumentmethoden des jeweiligen Dokuments haben nur einen lesenden Zugriff auf diesen Abschnitt.

- **Methodenzuordnung** Dieser Bereich eines Metadokuments beinhaltet Informationen über die Operationen und Dokumentmethoden, die auf dieses Dokument angewendet werden können. Jede Zeile dieses Abschnittes bezieht sich auf eine bestimmte Dokumentmethode; sie beinhaltet den Namen der Methode (bzw. der Operation), ihre Adresse und die benötigte Laufzeitumgebung. Die Methodenzuordnung des Beispieldokuments von der Seite 8 beinhaltet somit Informationen über drei Operationen `Present`, `Describe` und `private.localPresent`. Auf diesen Abschnitt des Metadokuments dürfen die Dokumentmethoden weder schreibend noch lesend zugreifen. Eine genauere Beschreibung dieses Abschnittes folgt im Abschnitt 2.2.

- **Inhalt** Dieser Bereich beinhaltet das tatsächliche Dokument. Die Kodierung des Inhalts wird vom Erzeuger des Metadokuments vorgenommen; er kann beispielsweise in Base64[1] kodiert sein. Auf diesen Abschnitt der Metadokumente können die Dokumentmethoden lesend und schreibend zugreifen[2].

Die Metadokumente sind somit ein „standardisiertes Austauschformat für Dokumente“, das bei der Kommunikation zwischen unterschiedlichen Instanzen – wie beispielsweise zwischen zwei Ausführungs-Servern – zum Einsatz kommen [Moe01]. Ein Dokument wird also beim Transport zwischen zwei Instanzen als ein Metadokument verschickt. Nach dessen Empfang weist ihm der Ausführungs-Server eine eindeutige Identifikation zu, zerlegt es in seine Einzelteile und speichert jedes der drei Abschnitte in eine separate Datei; es wird eine Datei für die Attribute (`attributes`-Datei), eine Datei für die Methodenzuordnungen (`methods`) und eine Datei für den Inhalt (die `content`-Datei) erzeugt[3]. Beim Speichern des Dokumentinhalts erfolgt keine Umkodierung des Inhalts. Der Ausführungs-Server speichert den Inhalt so wie er ihn beim Metadokument vorfindet. Informationen über die Kodierung des Dokumentinhalts sind nur den dazugehörigen Dokumentmethoden bekannt.

In der INDIGO-Infrastruktur hat jedes Dokument eine systemweit eindeutige Identifikation. Diese setzt sich aus der systemweit eindeutigen Identifikation des Ausführungs-Servers und der eindeutigen Identifikation des Dokuments bei diesem Ausführungs-Server zusammen.

[1] Eine kurze Bescheibung von Base64-Kodierung erfolgt auch im Abschnitt 6.1.3 auf Seite 6.1.3.
[2] Dieses Verhalten unterliegt der Sicherheitspolitik des Server-Betreibers; siehe Kapitel 4.
[3] Dies gilt besonders beim serverbasierten Transport von Dokumenten; vgl. Abschnitt 2.3.3.1 und 2.4.1.

2.2 Dokumentmethoden

Unter einer *Dokumentmethode*[4] versteht man ein dokumentspezifisches mobiles Programm, das besondere Operationen an dem dazugehörigen Dokument ausführt, wobei die Operationen Manipulationen oder Transformationen an einem bestimmten digitalen Dokument darstellen (vgl. [Moe01, S. 8]).

In der INDIGO-Infrastruktur bringt jedes Metadokument seine eigenen Methoden mit. Diese Methoden werden aber beim Transport der Metadokumente nicht mitverschickt. Die Metadokumente enthalten lediglich Verweise auf diese Methoden. Diese Verweise befinden sich im Methodenzuordnungs-Abschnitt des Metadokuments. Der Methodenzuordnungs-Abschnitt eines Metadokuments kann beispielsweise aus den folgenden Zeilen bestehen:

```
--ekidpUnhoJ
Content-Type: application/x-metadoc-methods

Present http://62.104.191.241/Text/present.zip application/java
Describe http://62.104.191.241/Text/describe.zip application/java
private.localPresent http://62.104.191.241/Text/lpresent.zip application/java
--ekidpUnhoJ
```

In jeder Zeile stehen der Name der unterstützten Operation, ein Verweis auf die Dokumentmethode, die diese Operation durchführen kann, und anschließend der Name der von der Methode benötigten Laufzeitumgebung[5], wobei der Verweis stets als eine URL-Adresse (Uniform Resource Locator [URL]) angegeben ist. Der Ausführungs-Server holt sich vor dem Ausführen einer Operation die benötigten Methoden bei Bedarf von der jeweiligen Adresse ab. Es ist zu beachten, daß diese Adresse nicht immer direkt auf eine Methode zeigt; die Adresse kann wie bei jeder Methode aus dem vorherigen Beispiel auch auf eine Archivdatei hinweisen, die wiederum diese Methode beinhaltet.

Es gibt zwei Arten von Dokumentmethoden: Methoden, die *orthogonale Operationen* an Dokumenten durchführen, und Methoden, die *private Operationen* an Dokumenten vornehmen.

2.2.1 Orthogonale Operationen

Um ein minimales Maß an Orthogonalität zu bieten, wurden bei der INDIGO-Infrastruktur vier *orthogonale Operationen* definiert. Diese Operationen müssen von jedem Dokumenttyp und Dokumentformat unterstützt werden. Diese vier orthogonale Operationen sind:

- **Present** Diese Operation wird zum Präsentieren des Dokuments verwendet. Sie erhält als Argument die Adresse des Ausführungs-Servers, auf dem das Dokument präsentiert werden soll. Als Ergebnis erhält man einen booleschen Wert über den Erfolg der Ausführung.

[4] In dieser Arbeit wird anstatt des Begriffes „Dokumentmethode“ der Einfachheit halber häufig auch der Begriff „Methode“ verwendet.

[5] Es gibt zwei Implementierungen des Ausführungs-Servers: einer wurde in der Programmiersprache Java und der andere in der Programmiersprache Python entwickelt. Es existieren auch entsprechend zwei Laufzeitumgebungen, die mit dem Namen `application/java` und `application/python` gekennzeichnet sind. Diese Arbeit setzt sich aber fast ausschließlich mit der in Java programmierten Implementation auseinander.

- **Describe** Diese Operation wird ohne Argumente aufgerufen. Sie liefert als Ergebnis eine Inhaltsbeschreibung des dazugehörigen Dokuments, wobei die Syntax dieser Inhaltsbeschreibung von der INDIGO-Infrastruktur vorgeschrieben ist. Mit Hilfe dieser Operation kann man in einem Dokument inhaltsbasiert nach bestimmten Begriffen suchen. Ein Beispiel für einen Such-Algorithmus, der diese Describe Operation verwendet, findet man in [Moe01, Seite 93].

- **Copy** Mit Hilfe dieser Operation kann ein Dokument, das sich auf einem Ausführungs-Server befindet, auf einen anderen Server kopiert werden. Zu diesem Zweck erhält die Operation als Argument die Adresse des Ziel-Servers. Die Operation liefert als Ergebnis einen booleschen Statuswert.

- **Move** Diese Operation wird genauso wie die Copy Operation aufgerufen und liefert ebenfalls einen booleschen Wert. Sie führt auch das gleiche wie die Copy Operation aus, mit dem Unterschied, daß nach dem Kopieren das Dokument vom Ursprungs-Server entfernt wird.

Die Operationen *Present* und *Describe* sind *obligatorische Operationen*. Dies bedeutet, daß jedes Dokument die entsprechenden dokumentspezifischen Methoden bereitstellen muß, die diese beiden Operationen durchführen.

Die Operationen *Copy* und *Move* sind dagegen *optional*. Jeder Ausführungs-Server bietet intern Funktionen, die diese Operationen auf einer Ebene von Bit-Folgen beherrschen. Falls ein Dokument keine Methoden für diese Operationen bereitstellt, werden diese Operationen automatisch von dem Ausführungs-Server intern durchgeführt. Diese Operationen wurden bei der Entwicklung der Infrastruktur als orthogonale Dokumentmethoden definiert, um einem Dokument zu ermöglichen, seine eigene Kopier- und Verschiebe-Operationen mit erweiterter Funktionalität zu definieren. Es ist beispielsweise möglich, eine besondere Kopier-Dokumentmethode zu definieren; diese Methode kopiert bei einem Dokument mit einem Inhalt, der außer dem statischen Bereich auch dynamische Bereiche besitzt, nur den statischen Abschnitt (der von dem Original nicht abweicht) auf einen Ziel-Server.

2.2.2 Private Operationen

Jedes Dokument darf neben den orthogonalen Dokumentmethoden, die orthogonale Operationen durchführen, auch weitere Methoden anbieten, die andere Operationen unterstützen. Diese Operationen nennt man auch *private Operationen* und die dazugehörigen Methoden *private Dokumentmethoden*. Dieses Verhalten macht es möglich, Operationen zu definieren, die das spezielle Verhalten der jeweiligen Dokumenttypen ausreizen können. Man kann beispielsweise für Bild-Formate spezielle Filter-Methoden definieren, die zum Konvertieren der farbigen Bilder in schwarz-weiße Bilder geeignet wären; eine solche spezielle Operation würde aber beispielsweise bei den Musik-Formaten keinen Sinn ergeben.

Für die Kennzeichnung der privaten Methoden sind ihre dazugehörigen Operationen bei der Methodenzuordnung mit dem Präfix „`private`“ zu markieren (beispielsweise `private.localPresent`; siehe Methodenzuordnungs-Abschnitt aus dem Beispiel auf Seite 10). In Abschnitt 2.4 werden einige Beispiele angesprochen, in denen auch private Methoden zum Einsatz kommen.

2.3 Ausführungs-Server

Der Ausführungs-Server kommt in der INDIGO-Infrastruktur sowohl auf der Anwenderseite als auch auf der Bibliothekseite zum Einsatz. Die Hauptaufgabe der Ausführungs-Server auf der Bibliothekseite ist die Speicherung der Dokumente. Diese bilden auch gemeinsam den Speicher der Bibliothek[6]; daher heißen diese Server auch *speicherseitige Ausführungs-Server*. Wie auch der Abbildung 2.1 zu entnehmen ist, bildet in der INDIGO-Infrastruktur der Verbund dieser speicherseitigen Ausführungs-Server, die in einer Nachbarschaftsbeziehung zueinander stehen, die digitale Bibliothek.

Eine wichtige Forderung an die speicherseitigen Server ist ihre ständige Verfügbarkeit. Im Gegensatz zu diesen Servern müssen die *anwenderseitigen Ausführungs-Server* nicht stetig erreichbar sein. Diese Server laufen bei den Anwendern, und die Verfügbarkeit dieser Server richtet sich nur nach dem jeweiligen Anwender.

Die anwenderseitigen Server dürfen genauso wie die speicherseitigen Server Dokumente speichern. Die beiden Ausführungs-Server bieten ebenfalls die Möglichkeit zur Ausführung von Dokumentmethoden und stellen eine Schnittstelle zum Ausführen der Server-Befehle bereit.

2.3.1 Server-Befehle

Der Zugriff auf die Dienste eines INDIGO-Ausführungs-Servers geschieht über eine Schnittstelle. An diese Schnittstelle kann man Befehle mit einer wohldefinierten Syntax – die sogenannten *Server-Befehle* (server commands) – schicken, um den Ausführungs-Server zum Erbringen seiner Dienste zu bewegen. Die Server-Befehle stellen somit Protokolle zur Kommunikation mit den INDIGO-Ausführungs-Servern dar. Die Server-Befehle beim INDIGO-Ausführungs-Server sind:

- **Documents** Dieser Befehl hat keine Parameter; der Ausführungs-Server antwortet auf diesen Befehl mit der Liste der Dokumente, die er speichert.

- **Runtimes** Dieser Befehl wird ebenfalls ohne Argumente aufgerufen; der Server liefert daraufhin den Namen der Laufzeitumgebungen, die ihm zur Verfügung stehen.

- **Attributes** Dieser Befehl hat – genauso wie der nächste Befehl „Operations“ – den Namen (genauer gesagt die Identifikation) eines Dokuments als Parameter. Falls dieses Dokument auf dem Server existiert, schickt er als Antwort die Attribute des Dokuments zurück.

- **Operations** Der Server liefert auf die Anfrage die Liste der Operationen, die das angegebene Dokument unterstützt.

[6] Diese verteilte Speicherung der Dokumente birgt auch einige Probleme. Ein Problem könnte beispielsweise dann entstehen, wenn man in einem solchen Datenbestand nach einem bestimmten Dokument suchen möchte, wobei dieser Datenbestand über eine Menge von Ausführungs-Servern, die in einer Nachbarschaftsbeziehung zueinander stehen, verteilt ist. Zu diesem und vielen anderen Problemen, die in diesem Kontext entstehen könnten, wird sehr detailiert in [Moe01] eingegangen.

- **Invoke** Mit diesem Befehl bewegt man einen Server dazu, an einem bestimmten Dokument, das er besitzt, eine Operation (bzw. Methode) auszuführen. Er hat als Parameter den Namen dieses Dokuments, eine Operation sowie eine Reihe von Argumenten für diese Operation.

- **Deliver** Mit Hilfe dieses Server-Befehls kann man ein Metadokument zu einem Ausführungs-Server schicken. Deliver erhält dazu ein Metadokument als Argument. Falls man nach dem Transport eine Operation auf dieses Metadokument ausführen möchte, kann man optional auch den Namen dieser Operation – gefolgt von einer Reihe von Argumenten – direkt mit dem Deliver-Befehl an den Server schicken.

  Der Ausführungs-Server weist jedem Metadokument bei dessen Speicherung eine serverbezogene eindeutige Identifikation zu (vgl. 2.1). Diese Identifikation schickt er als Ergebnis an den Aufrufer des Deliver-Befehls zurück.

2.3.2 HTTP als Basis-Protokoll

Das *Hypertext Transfer Protocol* (HTTP) [HTTP] stellt bei der INDIGO-Infrastruktur das Basis-Protokoll für die Kommunikation zwischen den unterschiedlichen Instanzen dar. Dies bedeutet auch, daß beispielsweise beim Versenden der Server-Befehle zu einem Ausführungs-Server diese in einer HTTP-konformen Syntax verschickt werden. Zu diesem Zweck werden ein Server-Befehl und seine Argumente zusammen als URL an einen HTTP-Request (`GET` oder `POST`) übergeben. Wenn man beispielsweise mit Hilfe des Server-Befehls „Documents“ die Liste der Dokumente, die ein Server enthält, erhalten will, schickt man diesem Server die folgende Zeile:

```
GET /diglib/documents HTTP/1.0
```

Daraufhin schickt der Server diesem Aufrufer eine HTTP-konforme Antwort. Die Antwort auf den oben erwähnten Server-Befehl könnte wie folgt aussehen:

```
HTTP/1.0 200 Ok
Server: Java-Dali Server V0.1
Date: Thu Jul 26 22:31:04 GMT+02:00 2001
Content-type: text/plain

documents:
127.0.0.1-135071748-935969511155
127.0.0.1-135072748-935969511155
127.0.0.1-135073748-935969511155
```

Die eigentliche Antwort, also die Namen der vorhandenen Dokumente, steht im Rumpf dieser HTTP-Response. Dieser Server beinhaltet somit drei Dokumente. Die Tabelle 2.1 faßt noch einmal die Liste der möglichen Server-Befehle mit deren Pendant als HTTP-Request zusammen.

Die Verwendung des HTTP als Basis-Protokoll ermöglicht den Anwendern, zur Kommunikation mit einem INDIGO-Ausführungs-Server jeden beliebigen HTTP-fähigen Client – wie beispielsweise einen Telnet-Client oder einen Web-Browser – einzusetzen. Falls

Server-Befehl	**HTTP-Request**
Documents	`GET /diglib/documents`
Runtimes	`GET /diglib/runtimes`
Attributes	`GET /diglib/attributes/`*Dokumentname*
Operations	`GET /diglib/operations/`*Dokumentname*
Invoke	`GET /diglib/invoke/`*Dokumentname?Methode&Argumente*
Deliver	`POST /diglib/deliver`*?Methode&Argumente*

Tabelle 2.1 Umsetzung der Server-Befehle auf HTTP-Requests

man beispielsweise bei einem auf der Socket-Adresse „`127.0.0.1:7333`“ zu erreichenden INDIGO-Server den Server-Befehl „Documents“ ausführen möchte, kann man bei einem Web-Browser die folgende URL-Adresse eingeben:

```
http://127.0.0.1:7333/diglib/documents
```

2.3.3 Laufzeitumgebung

Die eigentliche Ausführung der Dokumentmethoden geschieht in einer Umgebung, in der sogenannten *Laufzeitumgebung*. Jeder Ausführungs-Server kann zur Ausführung der Methoden unterschiedliche Laufzeitumgebungen bieten. Jede Laufzeitumgebung zeichnet sich dabei durch zwei wichtige Eigenschaften aus: durch den Programmcode, den sie ausführen kann, und durch die Schnittstellen, die sie den Dokumentmethoden zum Zugriff auf Ressourcen des Ausführungs-Servers zur Verfügung stellt.

Die Liste der unterschiedlichen Laufzeitumgebungen, die ein Ausführungs-Server bereitstellt – also somit auch die Liste der unterschiedlichen Programmcodes, die dieser Server ausführen kann – erhält man durch den Server-Befehl „Runtimes“.

Da es bei der Ausführung der Dokumentmethoden zu Sicherheitsverletzungen kommen kann, darf bei der Wahl der Laufzeitumgebung nur diejenige in Betracht gezogen werden, die eine geschützte und abgekapselte Ausführung der Methoden bietet. Dieses sicherheitsbezogene Konzept, das unter dem Namen *Sandbox-Konzept* (Sandkasten-Konzept) bekannt ist, wird von einigen Programmiersprachen wie beispielsweise Java unterstützt. Durch dieses Sandbox-Konzept wird ein kontrollierter Zugriff auf die Ressourcen garantiert, indem der Zugriff indirekt über die Schnittstellen der Laufzeitumgebung geleitet wird.

Die Ressourcen, die von den Dokumentmethoden benötigt werden, kann man in zwei Klassen unterteilen: in die *elementaren Ressourcen* und die *INDIGO-spezifischen Ressourcen*. Zu den elementaren Ressourcen gehören Ressourcen wie beispielsweise die Speichermedien, die Kommunikationsmedien und die Präsentationsmedien. Zu den INDIGO-spezifischen Ressourcen zählen hingegen Ressourcen (wie der Zugriff auf die Dokumentinhalte), die speziell von den INDIGO-Ausführungs-Servern angeboten werden. Der Zugriff auf die INDIGO-spezifischen Ressourcen geschieht über drei unterschiedliche Funktionsarten:

- Server-Funktionen,
- Dokument-Funktionen,
- Traversierungs-Funktionen.

2.3.3.1 Server-Funktionen

Diese Funktionen bieten den Dokumentmethoden den Zugriff auf die Server-Befehle eines lokalen bzw. eines entfernten Ausführungs-Servers. Über diese Funktionen kann beispielsweise eine Dokumentmethode, die auf einem lokalen Ausführungs-Server läuft, ihren Dokumentinhalt zu einem entfernten Server kopieren (mittels Deliver) und anschließend darauf eine neue Methode starten (dies ist beispielsweise beim entfernten Präsentieren eines Dokuments der Fall; vgl. auch 2.4).

Die Server-Funktionen bieten (im Gegensatz zu ihrem Pendant bei den Server-Befehlen) zwei Arten von Deliver zum Transport der Metadokumente:

- **Serverbasierter Transport** Bei dem *serverbasierten Transport* wird beim Transport eines Metadokuments auch sein Dokumentinhalt mitverschickt, was das Kopieren des gesamten Dokuments bedeutet. Diese Art des Transports heißt serverbasiert, da der gesamte Transport von dem Ausführungs-Server erledigt wird.

- **Methodenbasierter Transport** Im Gegensatz zum serverbasierten Transport wird beim *methodenbasierten Transport* der Transport des Dokumentinhalts von einer Methode kontrolliert[7]. Bei dieser Art des Transports wird zwar das Metadokument ebenfalls zu einem entfernten Ausführungs-Server geschickt, der Inhalts-Abschnitt des Metadokuments bleibt bei diesem Transport aber leer. Dieser Transport ist beispielsweise bei einem streamorientierten Transport sinnvoll, in dem große Mengen an Daten wie im Fall einer Video-Übertragung von einem Server zu einem anderen geschickt werden müssen. In Abschnitt 2.4 wird der Online-Transport vorgestellt, der diese Server-Funktion anwendet.

2.3.3.2 Dokument-Funktionen

Die Dokument-Funktionen bieten den Dokumentmethoden Schnittstellen für den Zugriff auf den Inhalt und die Attribute der gespeicherten Dokumente, auf denen diese Methoden ausgeführt werden. Dazu zählen Funktionen wie beispielsweise „open“, „close“, „read“, „write“ und „delete“.

Weiterhin stellen die Dokument-Funktionen den Methoden Schnittstellen für den Zugriff auf die dokumentspezifischen Semaphoren zur Verfügung. Mittels dieser Semaphoren können mehrere Methoden gleichzeitig kontrolliert auf die kritischen Abschnitte eines Dokuments zugreifen.

2.3.3.3 Traversierungs-Funktionen

Wie bereits am Anfang des Abschnitts 2.3 erwähnt, besteht eine INDIGO-Bibliothek aus einem Verbund von Ausführungs-Servern, die in einer Nachbarschaftsbeziehung zueinander stehen. Über die Traversierungs-Funktionen können die Dokumentmethoden die Menge der benachbarten Ausführungs-Server ermitteln. Sie bieten außerdem andere Traversierungsdienste, wie beispielsweise Funktionen zur Markierung der Server-Knoten.

[7] Da für diese Art des Transports indirekt das Dokument verantwortlich ist, heißt diese Art von Transport auch *dokumentbasierter Transport*.

2.4 Beispiele für Interaktion der INDIGO-Komponenten

Im vorliegenden Abschnitt werden zur Verdeutlichung der Funktionsweise der in den vorherigen Abschnitten erwähnten Komponenten – besonders zur Erklärung der Unterschiede zwischen dem serverbasierten und dem methodenbasierten Transport – zwei Beispiele für die Präsentation der Dokumente vorgestellt:

2.4.1 Offline-Präsentation

Unter einer *Offline-Präsentation* versteht man in der INDIGO-Infrastruktur die Präsentation von Dokumenten auf einem Ausführungs-Server, zu deren Präsentation (genauer gesagt, während des tatsächlichen Präsentationsvorgangs) kein Kontakt zu den weiteren Ausführungs-Servern notwendig ist. Um auf einem anwenderseitigen Ausführungs-Server eine solche Präsentation vorführen zu können, muß zuerst dieses Dokument vollständig auf diesen Server transportiert werden. Die Offline-Präsentation wird bei Dokumenten verwendet, bei denen man einen wahlfreien Zugriff auf den Dokumentinhalt haben soll. Außerdem wird diese Art von Transport bei nicht datenintensiven Dokumenten, wie beispielsweise Bildern oder Texten, verwendet. Eine Offline-Präsentation könnte wie folgt ablaufen:

Angenommen, ein Anwender möchte ein Dokument präsentieren. Dieses Dokument befindet sich aber auf einem speicherseitigen Ausführungs-Server. Zu diesem Zweck baut er mittels eines Clients eine Verbindung zu diesem Server auf und führt auf diesem Dokument die orthogonale Operation „Present“ aus. Dieser Aufruf könnte wie folgt aussehen:

```
GET /diglib/invoke/935969511155?Present&213.7.27.211&7334 HTTP/1.0
```

„`935969511155`“ ist der Name des Dokuments. Dieser Server-Befehl erhält als Argument ebenfalls die lokale Socket-Adresse – also die lokale Adresse „`213.7.27.211`“ und die Port-Adresse „`7334`“ – des anwenderseitigen Ausführungs-Servers.

Der speicherseitige Ausführungs-Server sucht nach dem Empfang dieses Befehls nach dem geforderten Dokument. Anschließend sucht er im Dokumentmethoden-Abschnitt dieses Dokuments nach der Adresse der zur Present Operation gehörenden Dokumentmethode. Nach dem Laden der benötigten Dokumentmethode initiiert er sie in der Laufzeitumgebung; die Methode erhält bei der Initiierung ebenfalls die Socket-Adresse des anwenderseitigen Servers als Argument.

Die Present-Methode nutzt die Server-Funktionen der Laufzeitumgebung und schickt dieses Dokument in Form eines Metadokuments mittels serverbasiertem Transport an den anwenderseitigen Server, dessen Adresse sie bei seinem Aufruf erhielt. Nach dem vollständigen Kopieren des Metadokuments fordert die Methode den anwenderseitigen Server auf, die lokale Präsentationsoperation „private.localPresent“ an diesem Dokument auszuführen (der Name dieser Operation wird beim Verschicken mittels serverbasiertem Deliver als Argument an diese Server-Funktion übergeben). Daraufhin besorgt der anwenderseitige Server die erforderliche Methode zum lokalen Präsentieren und initiiert diese in der Laufzeitumgebung. Nach der erfolgreichen Initiierung der private.localPresent-Methode terminiert die Present-Methode. Somit besteht keine Verbindung mehr zwischen diesen beiden Ausführungs-Servern.

Von diesem Zeitpunkt an beginnt die eigentliche Präsentation des Dokuments. Zu diesem Zweck verwendet die für private.localPresent zuständige Methode die elementaren und die

INDIGO-spezifischen Ressourcen des anwenderseitigen Servers. Nachdem die Präsentation des Dokuments beendet ist, entfernt die lokale Präsentationsmethode dieses Dokument von dem anwenderseitigen Ausführungs-Server.

2.4.2 Online-Präsentation

Die Offline-Präsentation kann nicht in jedem Umfeld verwendet werden. Beim Umgang mit datenintensiven oder zeitkontinuierlichen Medien – wie beispielsweise Video-Dokumenten – entsteht der Bedarf nach einer anderen Art der Präsentation. Wenn man beispielsweise eine große Video-Datei mittels Offline-Präsentation präsentieren möchte, entstehen einige Probleme: Bei einer Offline-Präsentation muß eine große Datei vor der tatsächlichen Präsentation zum Anwender transportiert werden; angesichts der Datendurchsätze, die den Anwendern derzeit normalerweise zur Verfügung stehen, würde dies zu langen Ladezeiten führen. Außerdem würde ein solcher Transport die Verfügbarkeit großer Speicherkapazitäten bei den Anwendern voraussetzen. Um solche Probleme zu vermeiden, wird in Einsatzgebieten dieser Art die *Online-Präsentation* verwendet. Bei dieser Art von Präsentation wird der Transport des Dokumentes von einem Protokoll zwischen den beiden Dokumentmethoden, die gleichzeitig auf dem speicherseitigen und auf dem anwenderseitigen Ausführungs-Server agieren, bestimmt. Eine solche Online-Präsentation kann in der INDIGO-Infrastruktur wie folgt konstruiert sein:

Der Anwender baut zu dem speicherseitigen Ausführungs-Server eine Verbindung auf und schickt ihm den Server-Befehl mit der Aufforderung, ein bestimmtes Dokument zu präsentieren. Dieser Server führt daraufhin an diesem Dokument die Present-Methode aus. Bis dahin geschieht alles wie bei einer Offline-Präsentation. Der eigentliche Unterschied liegt in der Konstruktion der verwendeten Dokumentmethoden zur Präsentation des Dokuments, denn die Present-Methode schickt bei der Online-Präsentation das Dokument mittels methodenbasierten Transports. Somit erhält der anwenderseitige Ausführungs-Server ein Metadokument, dessen Inhalt leer ist.

Auf dieses Metadokument wird auf dem anwenderseitigen Server die lokale Präsentationsmethode, die die Operation private.localPresent implementiert, gestartet. Bei einer Offline-Präsentation würde die Present-Methode nach der erfolgreichen Initiierung der lokalen Präsentationsmethode terminieren. Bei einer Online-Präsentation besteht hingegen zwischen diesen beiden Methoden ein gemeinsamer und von der Present-Methode erzeugter *Kommunikationsendpunkt*, über den die beiden Methoden miteinander kommunizieren. Der Kommunikationsendpunkt stellt einen Kommunikationskanal dar, über den Daten verschickt und empfangen werden.

Im Falle eines Video-Dokuments kann der Anwender beispielsweise mittels der lokalen Präsentationsmethode über diesen Kanal Steuerungsbefehle an die Present-Methode schicken, die das Versenden bestimmter Datenabschnitte des Dokuments, Vorspulen, Zurückspulen oder das Stoppen der Datenübertragung zur Folge hätte. Durch die Online-Präsentation werden somit nicht das gesamte Dokument an den anwenderseitigen Ausführungs-Server transportiert, sondern nur jene Abschnitte des Dokuments, die zur Präsentationszeit von der entsprechenden Methode benötigt werden.

Die Present-Methode ist auf dem speicherseitigen Ausführungs-Server so lange aktiv, bis die lokale Präsentationsmethode auf dem anwenderseitigen Server beendet wird. Bei der

Online-Präsentation wird genauso wie bei der Offline-Präsentation nach dem Beenden der Methode das gesamte Dokument von dem anwenderseitigen Ausführungs-Server entfernt.

Die Art, wie ein Dokument präsentiert wird, wird einzig und allein von dem Dokument bzw. von seinem Autor bestimmt: Die Present-Dokumentmethode kann beispielsweise so konzipiert sein, daß sie dem Anwender die beiden Arten der Präsention – also Online- und auch Offline-Präsentation – bietet; das Entfernen des Dokumentinhalts nach dem Beenden der Präsentation kann ebenfalls entfallen. Diese Möglichkeiten werden vom Autor durch die Festlegung der verwendeten Dokumentmethoden bestimmt.

Einige Beispiele für die Offline- sowie für die Online-Präsentation findet man in [Moe01] (in Kapitel 6). Dort wird auch erläutert, wie bei der INDIGO-Infrastruktur eine kooperative Benutzung eines Dokuments, bei der mehrere Anwender gleichzeitig dasselbe Dokument bearbeiten können, gestaltet werden kann.

2.5 Zusammenfassung

Bei der Konstruktion der INDIGO-Infrastruktur wurde das Ziel der Entwicklung einer skalierbaren und erweiterbaren Infrastruktur für die digitalen Bibliotheken verfolgt, die den Zugriff auf ihre Ressourcen von unterschiedlichen Plattformen aus erlaubt. Eine weitere wichtige Anforderung betraf die Orthogonalität der entwickelten Infrastruktur. Diese Anforderungen werden bei der INDIGO-Infrastruktur wie folgt erfüllt:

- In der INDIGO-Infrastruktur besteht eine digitale Bibliothek aus einem Verbund von speicherseitigen Ausführungs-Servern, die in einer Nachbarschaftsbeziehung zueinander stehen. Diese Konstruktion bietet die Möglichkeit, die Bibliothek durch die Aufnahme neuer Ausführungs-Server in den bestehenden Serververbund zu erweitern, was wiederum zur Skalierbarkeit der gesamten digitalen Bibliothek beiträgt.
- Die Erweiterbarkeit einer digitalen Bibliothek wird bei der INDIGO-Infrastruktur erzielt, indem sie jederzeit um neue Dokumentformate und Dokumenttypen erweitert werden kann. Dieses Verhalten wird durch die autonome Definition der privaten Methoden von Seiten des Autors erreicht. Die Interpretation jeglicher Dokumente wird in dieser Infrastruktur von den Dokumentmethoden erledigt.
- Die Orthogonalität wird bei der INDIGO-Infrastruktur durch die Definition der vier orthogonalen Operationen Present, Describe, Copy und Move erreicht (vgl. 2.2.1).
- Der Ausführungs-Server wurde in der Programmsprache Java entwickelt. Da Java-Compiler bzw. Java-Interpreter unter nahezu allen Plattformen verfügbar sind, ist es somit ebenfalls möglich, den Ausführungs-Server auf diesen unterschiedlichen Plattformen zu verwenden.

Durch die Verwendung einiger in diesem Kapitel beschriebenen Konzepte – vor allem der Möglichkeit zur Ausführung mobiler Programme (Dokumentmethoden) – entstehen für die in dieser Infrastruktur beteiligten Akteure große Sicherheitsrisiken. Diese Sicherheitsrisiken werden in Kapitel 4 näher analysiert. In dem folgenden Kapitel 3 werden kurz einige Sicherheitsverfahren, die später bei der Beschreibung der Sicherheitsrichtlinien zum Einsatz kommen, näher beschrieben.

Kapitel 3

Grundlagen der Sicherheit

Ziel dieser Arbeit ist der Entwurf und die Realisierung von Sicherheitsmechanismen für die INDIGO-Infrastruktur. Aus diesem Grund werden in diesem Kapitel die wichtigsten Grundbegriffe der Sicherheit in der Informationstechnologie beschrieben. Außerdem werden hier die am häufigsten eingesetzten Verschlüsselungsverfahren und Sicherheitsprotokolle erläutert.

3.1 Sicherheitstechnische Begriffe

Es gibt unterschiedliche Definitionen des Begriffes *Sicherheit*. Im REMO (Referenzmodell für sichere IT-Systeme) [REMO] lautet die Definition der „Sicherheit von IT-Systemen“:

> „Unter Sicherheit von IT-Systemen versteht man eine Eigenschaft eines IT-Systems, bei der Maßnahmen gegen die im jeweiligen Einsatzumfeld als bedeutsam angesehenen Bedrohungen in dem Maße wirksam sind, daß die verbleibenden Risiken tragbar sind.“

Wichtig ist hier in diesem Zusammenhang die Zuordnung von Maßnahmen zu den vorher konkret identifizierten Bedrohungen. Diese Bedrohungen, denen das System ausgesetzt ist, sind von der Sensitivität der im System verarbeiteten Informationen abhängig. Die Bedrohungen und die Sensitivität der verwendeten Informationen sind andererseits von der *Systemumgebung* (Umwelt) abhängig [ITSK89].

In der oben erwähnten Definition wird ebenfalls das Problem der verbleibenden Risiken angesprochen. Kein System kann eine hundertprozentige Sicherheit bieten. Die Aufgabe ist eher das Erkennen eines verbleibenden Risikos und seiner Begrenzung.

Die Sicherheit in der Informations- und Kommunikationstechnik unterteilt sich in zwei Bereiche. In dem einen Bereich wird die *nicht-kryptographische Sicherheit* behandelt. Lösungsbeispiele für die nicht-kryptographische Sicherheit sind Firewalls. Die *kryptographische Sicherheit* bezieht sich eher auf die Protokolle und Verfahren, die die Sicherheit mit Hilfe der Kryptographie zu sichern versuchen. In dieser Arbeit wird fast ausschließlich auf die kryptographischen Verfahren eingegangen.

3.1.1 Sicherheitsanforderungen

Sicherheitsanforderungen sind Anforderungen an die Sicherheit eines Systems. Im REMO werden die Sicherheitsanforderungen wie folgt definiert:

> „Eine *Sicherheitsanforderung* ist eine Systemanforderung, gegen deren *Erfüllung* Bedrohungen gerichtet sind, die als wichtig erscheinen. *Hinweis zum Kontext:* Die Einstufung einer Anforderung an ein System als sicherheitsrelevant hängt vom konkreten Anwendungsfall, der Einsatzumgebung und der Einschätzung des Anwenders ab."

Der Verlust der Vertraulichkeit, der Verlust der Integrität und der Verlust der Verfügbarkeit sind die drei wichtigsten Bedrohungen, denen ein IT-System ausgesetzt ist. Die Bedrohungen können für unterschiedliche IT-Systeme auch sehr unterschiedlich gewichtet sein. Dieselbe Infrastruktur kann sogar in unterschiedlichen Umgebungen unterschiedlich gewichteten Bedrohungen ausgesetzt sein. Man kann keine allgemein gültigen Sicherheitsanforderungen stellen; die Anforderungen sind stark vom Einsatzgebiet der Infrastruktur abhängig. Ein Server muß beispielsweise im kommerziellen Umfeld andere Anforderungen erfüllen als der gleiche Server, der in einer öffentlichen Bibliothek integriert wurde.

Die am häufigsten an ein System gestellten Sicherheitsanforderungen sind:

- **Vertraulichkeit** Vertraulichkeit[1] verlangt, daß die Information eines Computersystems (also lokal gespeicherte Daten) sowie die übertragenen Informationen nur einer zugelassenen Gruppe zum Lesen zur Verfügung stehen. Vertraulichkeit möchte den unbefugten Informationsgewinn verhindern. Sie wird z.B. bei der Übertragung der Paßwörter oder anderer sensibler Daten gefordert.

  Bei einer vertraulichen Kommunikation sollte ebenfalls der Schutz des Datenflusses vor der Analyse garantiert sein. Dies bedeutet, daß kein Angreifer in der Lage sein darf, Quelle, Ziel, Häufigkeit, Dauer oder andere Eigenschaften des Datentransfers bei einer Kommunikationseinrichtung zu observieren.

  Die Vertraulichkeit kann für einen Nachrichtenfluß, eine einzelne Nachricht oder bestimmte Bereiche innerhalb einer Nachricht gelten. Die Entscheidung über die verwendete Strategie hängt meistens vom Einsatzgebiet des Systems ab.

- **Authentizität** Die Authentifizierung (oder Authentifikation) ist der Nachweis eines Kommunikationspartners, daß er tatsächlich derjenige ist, als der er sich ausgibt. Dabei wird also die *Identität* der Kommunizierenden nachgewiesen. Dieser Vorgang, der zum Nachweis der Authentizität[2] der miteinander kommunizierenden Personen führt, ist die Basis für die meisten anderen Sicherheitsanforderungen. Eine Person erhält beispielsweise bei einem Dienstanbieter die Autorität zur Ausführung gewisser Operationen, nachdem diese Person sich eindeutig bei dem Anbieter identifiziert hat.

  Der Nachweis der Authentizität ist besonders bei der Übertragung des öffentlichen Schlüssels (die Echtheit des öffentlichen Schlüssels muß stets gesichert sein) und bei Zugriffen auf gesicherte Objekte wichtig. Bei den meisten Anwendungen ist es

[1] Vertraulichkeit von Daten := data confidentiality [engl.]
[2] Authentizität := authentication [engl.]

üblich, seine Identität mittels eines Benutzernamens und eines Paßwortes zu authentifizieren. Die Authentifizierung kann aber auch z. B. durch ein Zertifikat einer Zertifizierungsstelle erfolgen[3].

- **Zugriffskontrolle** Diese Anforderung (auch *Autorität* genannt) ist für die Verwaltung der Objekte zuständig. Die Autorität verlangt gewisse Mechanismen zur Steuerung des Zugriffs auf die Anwendungen und Ressourcen (Zugriffssteuerung[4]). Diese Mechanismen entscheiden, ob ein Anwender (genauer gesagt Akteur; vgl. Abschnitt 4.1.1) auf bestimmte Ressourcen zugreifen darf oder nicht. Zu diesem Zweck wird häufig eine Datenbank verwendet, in der die Zugriffs- und Aktionsrechte beschrieben sind.

- **Integrität** Integrität[5] verlangt, daß die lokal gespeicherten Daten und die übertragenen Informationen nur von einem berechtigten Personenkreis geändert werden können. Jede unbefugte Modifikation von Informationen sollte festgestellt werden. Die Mechanismen, die zur Durchsetzung der Integrität verwendet werden, überprüfen die Richtigkeit der empfangenen Daten. Falls die Daten beispielsweise während der Übertragung manipuliert wurden, muß der Empfänger stets in der Lage sein, die Manipulation festzustellen. Zu den Manipulationen gehören in diesem Kontext Schreiben, Änderung, Zustandsänderung, Löschen und Erstellen.

- **Verbindlichkeit und Unabstreitbarkeit** Bei der Verbindlichkeit steht der Schutz der kommunizierenden (bzw. handelnden) Personen voreinander im Vordergrund. Bei einem Handel kann der Käufer die Bestellung oder den Erhalt der Ware abstreiten. Auf der anderen Seite kann der Verkäufer den Erhalt der Bezahlung abstreiten. Er kann sogar behaupten, daß die Ware bereits geliefert wurde, obwohl das nicht den Tatsachen entspricht.

  Die gleichen Probleme existieren auch bei der elektronischen Kommunikation (bzw. beim elektronischen Handel). Die Unabstreitbarkeit[6] bzw. die Nichtzurückweisung soll den Sender bzw. Empfänger daran hindern, die Übertragung einer Nachricht zu leugnen. Dies bedeutet, daß der Empfänger die Herkunft der Nachricht eindeutig dem angegebenen Absender zuordnen kann. Der Absender kann ebenfalls beweisen, daß die Nachricht von dem Empfänger empfangen wurde.

  Bei der Betrachtung der Verbindlichkeit fällt auf, daß sich in diesem Zusammenhang die Frage der technischen Sicherheit stark mit den juristischen Aspekten und Fragestellungen vermischt. Man spricht auch von der *Rechtsverbindlichkeit* [Gri98].

- **Verfügbarkeit** Die Dienste eines Systems sollten bei Bedarf für die berechtigten Personen erreichbar sein. Die unbefugte Beeinträchtigung der Funktionalität führt bei einem System zum Verlust der Verfügbarkeit[7].

[3] Beispiel für Authentifikation aus [Gri98]: „Das Zertifikat eines Euroschecks ist die Euroscheckkarte, ggf. in Verbindung mit einem Personalausweis. Der Besitzer eines Euroschecks authentifiziert sich, indem er vor den Augen des Empfängers unterschreibt. Der Empfänger des Euroschecks vergleicht die Unterschrift auf dem Scheck mit der auf der Scheckkarte und ggf. mit der im Personalausweis. Er vermerkt die Scheckkartennummer auf dem Scheck als Hinweis dafür, daß das Zertifikat vorgelegen hat.“

[4] Zugriffssteuerung:= access control [engl.]

[5] Integrität von Daten := data integrity [engl.]

[6] Unabstreitbarkeit := non-repudiation of delivery [engl.]

[7] Verfügbarkeit := availability [engl.]

- **Anonymität** Manchmal ist es wichtig, daß die Kommunikationpartner völlig anonym miteinander kommunizieren. Dies bedeutet beispielsweise, daß in einer gewissen Umgebung die Benutzer die Möglichkeit bekommen, auf Dienste ohne Preisgabe ihrer Identität zuzugreifen. Bei öffentlichen Auskünften und Beratungen sollte man beispielsweise die Möglichkeit haben, anonym an die Informationen zu gelangen.

- **Pseudonymität** Die Forderung nach Pseudonymität wird besonders im kommerziellen Bereich immer wichtiger. Die Pseudonymität verlangt, daß der Benutzer imstande sein sollte, Dienste *anonym* zu benutzen. Andererseits sollte der Dienstanbieter die Sicherheit haben, für die angeboten Dienste bezahlt zu werden. Diese Pseudonymität wird im realen Handel durch die „Barzahlung“ garantiert. Für „Wahlen“ gelten ebenfalls Forderungen nach Pseudonymität. Um wählen zu dürfen, muß man sich identifizieren. Nach der Identifikation geschieht das Wählen aber anonym.

- **Rechtzeitigkeit** Eine mitgeschnittene fremde Nachricht (es kann sogar eine verschlüsselte Nachricht sein) kann man aufzeichnen und zu einem späteren Zeitpunkt benutzen, um mit dem entsprechenden Server zu kommunizieren. In diesem Fall kann man als eine Vertrauensperson mit dem Server kommunizieren. Die Forderung nach Rechtzeitigkeit möchte diese sogenannten *Replay-Angriffe* bekämpfen. Durch die Verwendung eines *Zeitstempels* oder eines *Einmalpaßworts* für jede Anfrage wird der Forderung nach der Rechtzeitigkeit genüge getan.

  Diese Form der Störung eines Kanals kann ebenfalls ein *denial-of-service-Angriff* sein. Es geht dabei nicht immer um ein Abfangen oder eine Verfälschung von Daten, sondern um die Störung des Datenverkehrs und Beeinträchtigung oder Fehlfunktion eines Systems, ohne daß der Urheber zu entlarven wäre.

Nach der klassischen Sicht der Sicherheit sind Integrität, Verfügbarkeit und Vertraulichkeit die wichtigsten Anforderungen, die man an ein System stellen soll (*dreidimensionales Sicherheitsmodell*). Nach [Gri98] muß man außerdem mindestens die Authentizität und die Verbindlichkeit bei einem System beachten.

3.1.2 Sicherheitsangriffe

Es gibt unterschiedliche Bedrohungen und Angriffe, die sich gegen die Systemanforderungen richten. Einer dieser Angriffe, der Replay-Angriff, wurde bereits im Zusammenhang mit der Sicherheitsanforderung „Rechtzeitigkeit“ erwähnt. Die Angriffe können nach [Sta95] in vier Kategorien unterteilt werden:

- **Abfangen** Bei dieser Kategorie fängt der Angreifer die Informationen, die nur für einen bestimmten Personenkreis gedacht waren, ab. Dieser Angriff richtet sich gegen die Sicherheitsanforderung *Vertraulichkeit*. Beispiele für diese Kategorie von Angriffen sind die Lauschangriffe auf Telefongespräche oder auf den Email-Verkehr.

- **Unterbrechung** Bei einer Unterbrechung wird ein Teil der Informationskette so unterbrochen, daß beispielsweise der Empfänger die vom Sender gesendeten Informationen nicht erhält. Diese Art von Angriffen richtet sich beispielsweise gegen die Sicherheitsanforderung *Verfügbarkeit*. Ein Beispiel für diesen Angriff ist die physikalische Zerstörung eines Rechners oder das Kappen der Informationskanäle. Angriffe wie

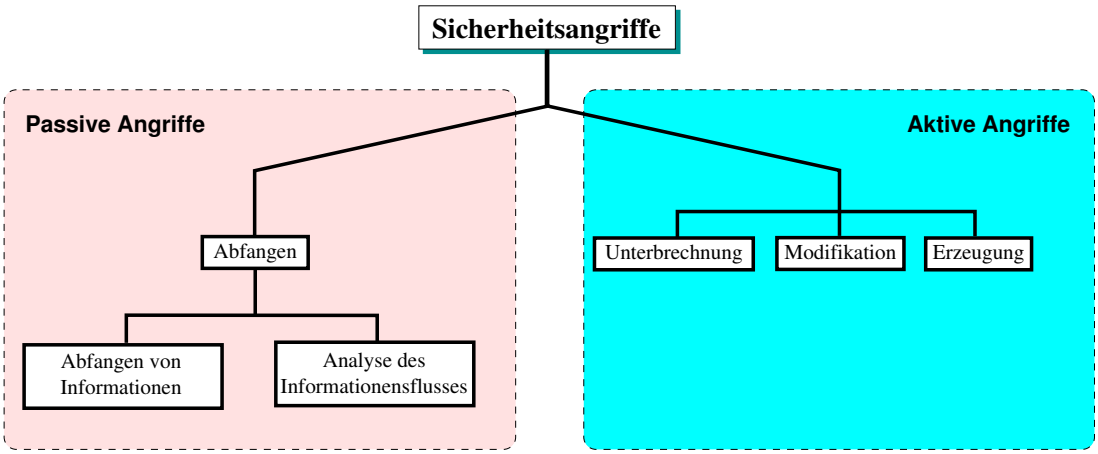

Abbildung 3.1 Sicherheitsangriffe

beispielsweise der denial-of-service-Angriff fallen ebenfalls in diese Kategorie. Man spricht bei dieser Kategorie auch von *Verweigerung der Dienste*.

- **Modifikation** Bei diesem Angriff ändert der Angreifer bestimmte Informationen. Dabei handelt es sich um einen Angriff auf die *Integrität*. Ein Angreifer kann beispielsweise den Inhalt einer Nachricht aktiv verändern und diese veränderte Nachricht anschließend an den ursprünglichen Empfänger weiterleiten. Darunter fällt auch die Modifikation eines Programms oder einer Datei wie der Paßwortdatei.

- **Erzeugung** Dieser Angriff zielt gegen die Sicherheitsanforderung *Authentizität*. Der Angreifer erzeugt eine gefälschte Information und schleust diese in ein System. Dies ist beispielsweise der Fall, wenn ein Angreifer einer anderen Person unter falscher Identität Informationen schickt.

Diese Angriffskategorien lassen sich wiederum in *passive* und *aktive* Angriffe unterteilen. Diese Unterteilung wird in der Abbildung 3.1 veranschaulicht.

3.1.2.1 Passive Angriffe

Die passiven Angriffe zielen entweder direkt auf die (übertragenen) Informationen oder auf die Spur der Informationen.

Bei der ersten Art dieser Angriffe versucht man beispielsweise, den Email-Verkehr oder ein Telefongespräch abzuhören. Bei diesen Angriffen interessiert den Angreifer, welche Information eine Nachricht oder Datei enthält. Diese Art der passiven Angriffe können (meistens) durch die Anwendung der kryptologisch sicheren Werkzeuge abgewehrt werden. Trotzdem bieten viele dieser Werkzeuge keinen Schutz gegen die Analyse des Nachrichtenverkehrs: Eine Person kann beispielsweise mit Hilfe der Kryptologie seine gesamte Kommunikation mit einer Bank schützen. Solange der Nachrichtenfluß nicht gegen eine Analyse geschützt ist, kann das Finanzamt trotzdem herausfinden, ob und wie häufig diese Person mit dieser Bank kommuniziert. Bei dieser Auswertung der Informationen, die man *Analyse des Informationsflusses* (Traffic Analysis) nennt, interessiert sich der Angreifer nicht für den Inhalt einer Nachricht, sondern wer, wann, wie lange und mit wem kommuniziert hat.

3.1.2.2 Aktive Angriffe

Die drei Angriffskategorien Unterbrechung, Modifikation und Erzeugung kann man unter dem Begriff der *aktiven Sicherheitsangriffe* zusammenfassen. Bei dieser Art der Angriffe greift der Angreifer aktiv in einen Prozeß ein, indem er Informationen modifiziert bzw. erzeugt[8] oder den Informationsfluß unterbricht.

Ein aktiver Angriff geht nicht immer von einem *Dritten* aus. Eine Sicherheitsverletzung kann auch beispielsweise von einem der Kommunikationspartner ausgehen. Diese Arten von Sicherheitsverletzungen zielen gegen die Sicherheitsanforderung *Verbindlichkeit*. Ein Sender kann beispielsweise behaupten, eine Nachricht geschickt zu haben, obwohl dies nicht der Fall ist. Er kann ebenfalls einem Empfänger eine Nachricht schicken und später den Versand abstreiten. Eine solche Verletzung kann auch von dem Empfänger ausgehen. Er kann den Erhalt einer Nachricht abstreiten, obwohl er bereits die Nachricht vom Absender erhalten hat. Außerdem kann er behaupten, er habe eine Nachricht, die er selbst produziert hat, vom angeblichen Absender erhalten. Daher muß man bei einer Infrastruktur auch die Kommunikationspartner voreinander schützen.

Ein passiver Angriff ist im Gegensatz zu einem aktiven Angriff sehr schwer nachvollziehbar; bei einem passiven Angriff werden keine Daten verändert. Trotzdem bieten die Maßnahmen gegen die passiven Angriffe einen direkten Schutz vor Mißbrauch. Im Gegensatz zu diesen Maßnahmen bieten die meisten Maßnahmen gegen aktive Angriffe keinen direkten Schutz, sondern helfen, einen Mißbrauch zu entdecken.

3.1.3 Bösartige Programme

Ein Angreifer agiert bei seinen Angriffen meistens über ein Hilfsprogramm. Ein solches Programm nennt man auch *bösartiges Programm*[9]. Diese Programme gibt es in unterschiedlichen Arten. Die wichtigsten Arten solcher Programme sind:

- **Viren** Ein Virus ist kein selbständiges Programm, sondern er kann nur mit Hilfe eines Wirtsprogramms existieren. Er wird durch den Aufruf dieses Wirtsprogramms aktiviert und kann somit wie ein normales Programm alle Dienste des Rechners in Anspruch nehmen, die auch anderen Programmen dieses Rechners gewährt sind. Ein Virus besteht aus zwei Funktionen: der Weiterverbreitungsfunktion und der Schädigungsfunktion.

- **Hintertüren** Eine Hintertür ist eine versteckte und nicht dokumentierte Funktion eines Programms. Mit Hilfe solcher Hintertüren können sich die Eingeweihten über diese Programme Zugriff auf die Ressourcen und Daten der Anwender verschaffen.

- **Logische Bomben** Eine logische Bombe ist ein Programmsegment, das erst dann aktiv wird, nachdem eine bestimmte logische Bedingung erfüllt wurde. Ein Beispiel für eine solche Bedingung kann eine zeitliche Bedingung sein.

[8] Die aktiven Angriffe *Tarnung* und *Wiederholung* kann man in die Kategorie „Erzeugung“ einordnen. Diese beiden Angriffe zielen auf die Sicherheitsanforderung „Authentizität“.
[9] Bösartiger Code := malicious code [engl.]

- **Trojanische Pferde** Ein Trojanisches Pferd ist ein scheinbar nützliches Programm, das aber in Wirklichkeit etwas anderes im Schilde führt. Nach dem Aufruf des Programms kann es beispielsweise Daten über den Anwender sammeln und diese heimlich dem Angreifer zusenden.

- **Würmer** Ein Wurm besteht wie ein Virus aus einem sich selbst vervielfältigenden Programmteil und aus einem Programmteil, das eine Schädigungsfunktion beinhaltet. Ein Wurm verwendet zur Weiterverbreitung meistens die Infrastruktur eines Netzwerks. Aus diesem Grund nennt man ihn häufig auch Netzwerkwurm. Der Hauptunterschied zwischen einem Wurm und einem Virus liegt darin, daß ein Wurm im Gegensatz zu einem Virus ein selbständiges Programm ist.

- **Bakterien** Eine Bakterie zielt hauptsächlich gegen die Ressourcen eines Rechners. Sie ist kein eindeutiger Schädling, sondern versucht, durch Neugenerierung oder Duplizieren den Speicherplatz bzw. die Rechenleistung eines Rechners zu belasten. Ein einfaches Beispiel für eine solche Bakterie ist ein einfaches Programm, das zwei Kopien von sich selbst erzeugt und diese beiden wieder parallel aufruft und anschließend terminiert.

Man kann diese Arten von bösartigen Programmen in zwei unterschiedliche Gruppen unterteilen: Bakterien und Würmer sind selbständige Programme. Die übrigen sind entweder Funktionen eines Programms oder können nur existieren, wenn sie sich an andere Programme anhängen.

Viele der bösartigen Programme, die zur Zeit existieren, beinhalten die Eigenschaften von unterschiedlichen Arten. Der „Michelangelo“ von 1991 ist beispielsweise ein Boot-Virus, der sich nur über Disketten verbreiten kann. Die Schädigungsfunktion dieses Virus wird aber nur jedes Jahr am 6. März (am Geburtstag des Künstlers Michelangelo) aktiv, so daß dieser Virus gleichzeitig auch eine logische Bombe ist.

3.1.4 Sicherheitsmaßnahmen

Um die Sicherheitsanforderungen zu erfüllen, werden bestimmte Maßnahmen getroffen, sogenannte *Sicherheitsmaßnahmen*. Diese Maßnahmen sind keine allgemein definierten Vorgehensweisen. Die Entscheidung über die verschiedenen Maßnahmen können in Bezug auf das Einsatzgebiet des Systems getroffen werden. Einige dieser Sicherheitsmaßnahmen und Grundfunktionen der Sicherheit sind:

- Identifikation,
- Authentifikation,
- digitale Unterschrift,
- Beweissicherung,
- Verschlüsselung,
- Rechteverwaltung,
- Rechteprüfung,

- Wiederherstellung,
- Fehlerüberbrückung durch Redundanz,
- Datensicherung,
- Verpflichtungskontrolle,
- Übertragungssicherheit.

Manche dieser Sicherheitsmaßnahmen kann man durch den Einsatz von nicht-kryptographischen Sicherheitsmaßnahmen – wie beispielsweise Firewalls[10], Paketfilter oder einfach durch eine sicherheitsorientierte Konfiguration des Routers – realisieren. Andere Sicherheitsmaßnahmen, wie beispielsweise die Verschlüsselung oder die digitale Signatur, erreicht man mit Hilfe der kryptographischen Sicherheit. Sicherheitsvorkehrungen werden im Bezug auf die kryptographische Sicherheit hauptsächlich in drei unterschiedlichen Kommunikationsschichten getroffen: in der Internetschicht, in der Transportschicht und in der Anwendungsschicht (siehe Abbildung 3.2).

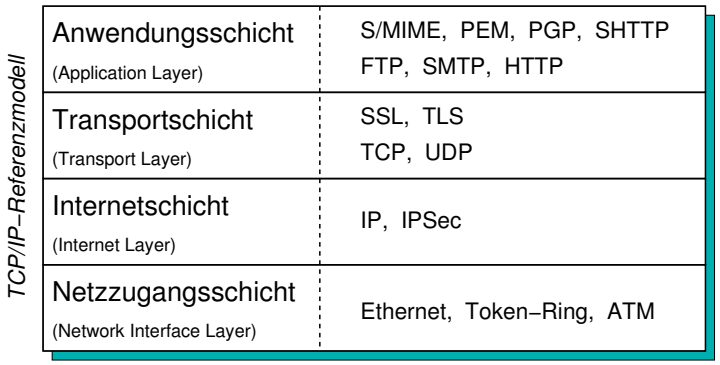

Abbildung 3.2 Sicherheitsvorkehrungen in den Kommunikationsschichten

In den „unteren Schichten“ gibt es zwei wichtige Protokolle für sichere Kanäle. Das Ergänzungsprotokoll IPSec [RFC2401] zum Internetprotokoll (IP) bietet in der Internetschicht einerseits sichere Kanäle zwischen den Vermittlungsknoten an; andererseits bietet das Sicherungsprotokoll SSL in der Transportschicht (siehe Abschnitt 3.3.4) oberhalb des Transportprotokolls TCP sichere Kanäle zwischen den Transportknoten der Anwendungskomponenten an. In der „oberen Schicht“ – der Anwendungsschicht – sind die Sicherheitsprotokolle der Internetanwendungen wie Email (z. B. S/MIME [SMIME], PEM [RFC1421] und PGP [Zimm96]) und WWW (z. B. SHTTP [RFC2660]) angesiedelt.

[10] Firewall ist ein Rechner bzw. ein Programm, das das Netzwerk einer Infrastruktur gegen die Angriffe aus einem fremden angeschlossenen Netz schützt. Zu diesem Zweck überprüft er beispielsweise die Absender der Datenpakete und läßt nur bestimmte Datenpakete passieren.

3.2 Verfahren zur Datenverschlüsselung und Einweg-Hashfunktionen

Bei der Realisierung der meisten Sicherheitsmaßnahmen werden häufig kryptographische Verfahren eingesetzt. Die meisten kryptographischen Verfahren dienen der Wahrung der Vertraulichkeit. Darunter fallen die meisten symmetrischen Verfahren. Manche der asymmetrischen Verfahren garantieren außer Vertraulichkeit auch andere Sicherheitsanforderungen wie Authentizität und Integrität. In diesem Abschnitt werden die wichtigsten Eigenschaften und Einsatzgebiete der am häufigsten verwendeten kryptographischen Verfahren, die auch zum Teil bei dieser Diplomarbeit zum Einsatz kommen, beschrieben. Eine genaue Beschreibung dieser Verfahren findet sich beispielsweise bei Stallings [Sta00].

3.2.1 Symmetrische Verfahren

Bei einem *symmetrischen Kryptographieverfahren* verwendet man zur *Verschlüsselung* und *Entschlüsselung* (fast[11]) denselben *Schlüssel*[12]. Hierbei müssen beispielsweise bei einer Nachricht der Sender und der Empfänger den gleichen Schlüssel besitzen. Aus diesem Grund darf der Schlüssel auch nur diesen beiden Personen bekannt sein.

3.2.1.1 Grundlagen der symmetrischen Verfahren

Eine der ältesten Verschlüsselungsverfahren ist die *Cäsar-Verschlüsselung*. Sie beruht darauf, daß jeder Buchstabe in einem Text durch seinen dritten Nachfolger vertauscht wird[13]. Bei diesem Verfahren ist der verwendete Schlüssel gleich „3“. Man kann aber statt drei auch eine andere Zahl verwenden. *ROT13* verwendet beispielsweise den gleichen Algorithmus, aber als Schlüssel die Zahl „13“. Dieses Verfahren hat den Vorteil, daß bei der nochmaligen Wiederholung dieses Vorgangs aus dem Chiffrat[14] der Klartext erzeugt wird. Man könnte diesen Algorithmus verallgemeinern, indem man einen Buchstaben durch einen beliebigen Buchstaben ersetzt[15], d.h. A durch D, B durch X usw.. Die von diesen Verfahren benutzten Algorithmen basieren auf einer rein *monoalphabetischen Substitution*.

Die monoalphabetischen Substitutionsverfahren sind mit der Kenntnis über die Verteilung der Buchstaben im Klartext recht schnell zu knacken (Häufigkeitsanalyse). Um diesem

[11] Die *mathematische Definition* der symmetrischen Verschlüsselungsverfahren:
K und K' sind Schlüsselräume. E ist die Verschlüsselungsfunktion und D ihre zugehörige Entschlüsselungsfunktion. h ist die Schlüsselrelation $h : K \to K'$, die K nach K' abbildet. Das Verschlüsselungsverfahren (E, D, H) ist *symmetrisch*, wenn sich die Schlüsselrelation h mit einem polynomialen Aufwand berechnen läßt. Das heißt, daß man leicht von dem Verschlüsselungsschlüssel k zu dem Entschlüsselungsschlüssel k' kommt. Wenn zur Verschlüsselung und zur Entschlüsselung derselbe Schlüssel verwendet wird, stellt dies einen Sonderfall dar, bei dem $K = K'$ und $h = id_K$ ist.

[12] Es geht hierbei um den Schlüssel und nicht um den Algorithmus. Bei den meisten Verfahren unterscheiden sich die Algorithmen zur Verschlüsselung und Entschlüsselung voneinander. Nur bei wenigen Verfahren – wie beispielsweise ROT13, One-Time-Pad oder Stromchiffrierung – setzt man zur Verschlüsselung und Entschlüsselung denselben Algorithmus.

[13] Mathematisch kann man die Cäsar-Verschlüsselung durch „Modulo“ erzeugen: `Ausgabebuchstabe = (Eingabebuchstabe + 3) mod (26)`.

[14] Die unverschlüsselte Information nennt man *Klartext*; den Vorgang des Verschlüsselns bezeichnet man als *Chiffrieren* und das Ergebnis der Verschlüsselung als *Chiffrat*, *Geheimtext* oder *Kryptogramm*. *Dechiffrieren* ist der Vorgang des Rückführens von Klartext aus dem Chiffrat.

[15] Diese Art von Abbildung des Alphabets auf sich selbst nennt man auch *Permutation*.

Problem zu begegnen, wurde die *polyalphabetische Chiffrierung* entwickelt. Bei der polyalphabetischen Substitution macht man die Substitutionsvorschrift von der Position im Text abhängig. Bei diesen Verfahren werden `n`-Zeichen durch `m`-Zeichen ersetzt, wobei für jeden Ersetzungsschritt unterschiedliche `m`- und `n`-Zeichen gewählt werden dürfen. Diese Ersetzung stellt eine Verallgemeinerung der monoalphabetischen Substitution dar, denn bei der Wahl von `m=n=1` erhält man wieder die monoalphabetische Substitution.

Ein Beispiel für polyalphabetische Chiffrierung ist die *Vignère-Verschlüsselung*. Bei dieser Chiffrierung wird als Schlüssel ein Schlüsselwort wie beispielsweise „`ZYX`“ gewählt. Zu diesem Schlüsselwort wird wiederholt der Klartext addiert[16]:

Schlüsselwort:	`ZYXZYXZYXZYXZ...` $\oplus$
Klartext:	`HOCHLEBELINUX...`
Chiffrat:	`GMZGJCACJHLRW...`

Tabelle 3.1 Beispiel für die Vignère-Verschlüsselung

Eine konsequente Weiterführung der Vignère-Chiffrierung ist die *Vernam-Chiffrierung*. Bei dieser Chiffrierung wird nicht in Buchstaben-Ebenen gearbeitet, sondern in Bit-Ebenen. Zu diesem Zweck wird anstatt `(modulo 26)` XOR, was `(modulo 2)` darstellt, verwendet. Eine weitere Verbesserung des Verfahrens stellt das *One-Time-Pad* dar. Ein One-Time-Pad hat verglichen mit der Vernam-Chiffrierung folgende zusätzliche Eigenschaften [Gri98]:

- Bei einem One-Time-Pad ist der Schlüssel bezogen auf die Bitlänge genauso groß wie der Klartext.
- Der Schlüssel besteht aus einer zufällig gewählten Bitfolge.
- Dieser Schlüssel darf nur einmal zur Verschlüsselung eingesetzt werden.

Der One-Time-Pad ist absolut sicher. Dies wurde bereits von Claude Shannon bewiesen. Trotzdem wird dieses Verfahren in der Realität kaum eingesetzt. Dies liegt wohl an der Länge des Schlüssels, der zum Chiffrieren verwendet wird.

Eine völlig andere Technik zur Chiffrierung stellt die *Transposition* (Verwürfelung oder Vertauschen) dar. Die Zeichen eines Klartextes bleiben bei der Transposition erhalten. Hier werden die Zeichen – im Gegensatz zur Substitution – nicht durch andere Zeichen ersetzt, sondern ihre Position wird lediglich verändert. Ein einfaches Beispiel für eine solche Transformation ist das Rückwärtsschreiben eines Textes („hallo“ wird zu „ollah“).

Die Transposition entspricht dem Grundprinzip der *Diffusion*, bei der die im Klartext enthaltene Information über das Chiffrat verteilt wird. Die anderen Verfahren – wie beispielsweise die polyalphabetische Substitution – basieren auf dem Prinzip[17] *Konfusion*. Bei der Konfusion wird versucht, den Zusammenhang zwischen dem Klartext und dem Chiffrat zu

[16] Bei der Addition muß man zuerst die Buchstaben auf Zahlen abbilden; also `A=0`,..., `X=23`, `Y=24`, `Z=25`. Nach der Addition zweier Zahlen wird wie bei einer Cäsar-Chiffrierung die modulo 26 gebildet. Anschließend wird das Ergebnis zurück auf Buchstaben abgebildet.

[17] Diese beiden Grundprinzipien der Chiffrierung, also Diffusion und Konfusion, wurden von Claude Shannon eingeführt.

verwischen [Wobst98]. Die meisten modernen symmetrischen Verschlüsselungsverfahren, wie DES oder IDEA, benutzen eine Kombination dieser beiden Verfahren.

Die modernen symmetrischen Verschlüsselungsverfahren werden in zwei Gruppen unterteilt:

- **Stromchiffrierung** Bei einer Stromchiffrierung erzeugt der Chiffrieralgorithmus abhängig von einem Schlüssel zuerst eine Folge von Nullen und Einsern. Diese Bitfolge – auch *Schlüsselstrom* genannt – addiert er anschließend mit Hilfe von XOR mit dem Klartext (wie bei der Vernam-Chiffrierung). Die Hauptaufgabe eines solchen Verschlüsselungsalgorithmus liegt also darin, von einem Schlüssel eine Bitfolge zu erzeugen, die fast einem Schlüssel bei einem One-Time-Pad-Verfahren ähnelt.

  Die Stromchiffrierung eignet sich gut zur *Online-Verschlüsselung* der Daten. Aus diesem Grund wird sie beispielsweise zur Verschlüsselung der Festplatten verwendet (siehe auch B.1). Wichtige Vertreter dieses Verfahrens sind die Verschlüsselungsalgorithmen *RC4* und *SEAL*.

- **Blockchiffrierung** Bei dieser Chiffrierung[18] wird der Klartext in Blöcken von Bits unterteilt. Anschließend werden die Bits eines Blocks zusammen verschlüsselt. Ein Beispiel für eine solche Chiffrierung ist die Cäsar-Chiffrierung, bei der acht Bits (also ein ASCII-Zeichen) als ein Block zusammengefaßt verschlüsselt werden. Beispiele für moderne Blockchiffrierung sind *RC5*, *IDEA*, *AES* und *DES*.

Die am häufigsten eingesetzten Verschlüsselungsalgorithmen sind Blockchiffrierungen. Die Blockchiffrierung erlaubt die Möglichkeit der Kombination von Konfusion und Diffusion. Im Gegensatz dazu bietet die Stromchiffrierung meistens nur Konfusion. Bei einer Blockchiffrierung ist es außerdem nicht nötig, im voraus einen Schlüsselstrom vorrätig zu erzeugen, was zuviel Zeit erfordert.

3.2.1.2 DES und Triple-DES

Der *Data Encryption Standard* (DES) wurde von einem Team von IBM-Mitarbeitern entwickelt [DES]. Dieses Verfahren wurde 1977 durch das US National Bureau of Standards (NBS, heute NIST) als offizieller Verschlüsselungsstandard festgelegt. Es ist seitdem eines der am häufigsten untersuchten Verschlüsselungsverfahren.

DES ist eine symmetrische Blockchiffrierung, die mit einer Blockgröße von 64 Bit arbeitet. Zu diesem Zweck wird der Klartext in Blöcken, die 64 Bit lang sind, aufgeteilt. Sie verwendet zur Verschlüsselung einen 64 Bit langen Eingabeschlüssel, von dem effektiv nur 56 Bit benutzt werden. Die restlichen acht Bit werden entweder nicht verwendet oder als Paritätsbits benutzt. Die Verschlüsselung mit diesem 56 Bit langen Schlüssel passiert in 16 Arbeitsschritten (auch *Runden* genannt), wobei in jedem Schritt der DES-Algorithmus eine Kombination von Diffusion mit Konfusion anwendet. Damit entspricht der DES einer polyalphabetischen Substitution auf der Basis von 64 Bit Blöcken, der außer Substitution auch Transposition zur Verschlüsselung einsetzt.

[18] Blockchiffrierung := block cipher [engl.]

Wie bereits erwähnt, stellt das DES-Verfahren eine Blockchiffrierung dar. Dieses Verfahren läßt sich aber in unterschiedlichen Betriebsarten betreiben, so daß es auch als Stromchiffrierung eingesetzt werden kann. Die vier Betriebsarten von DES sind:

- **Electronic Codebook (ECB)** Bei ECB wird jeder Block einzeln und unabhängig von den anderen Blöcken verschlüsselt. Dieser Modus ist nützlich, wenn man beispielsweise einen Schlüssel verschlüsselt seinem Partner verschicken möchte. Dieser Modus eignet sich außerdem für Datensätze, die bei einer Datenbank gespeichert sind. Mit Hilfe von ECB ist man in der Lage, bestimmte Datensätze zu modifizieren, ohne die gesamte Datenbank zu entschlüsseln. Dieses Verfahren ist jedoch verglichen mit den anderen Betriebsarten der unsicherste Modus.
- **Cipher Block Chaining (CBC)** Bei diesem Modus wird das Ergebnis der Verschlüsselung des i-ten Blockes, also das Chiffrat für den i-te Block, mit dem Block i+1 addiert (mit XOR). Um das Chiffrat für diesen i+1-te Block zu bekommen, wird anschließend das Ergebnis der Addition mit DES verschlüsselt. Durch diese zusätzliche Addition wird das DES-Verfahren erheblich sicherer. Dieser Modus wird meistens bei einer allgemeinen blockorientierten Übertragung verwendet.
- **Cipher Feedback (CFB)** Der CFB-Modus verwendet genau wie der CBC-Modus zur Erzeugung des Chiffrats für den i+1-te Block das Chiffrat des i-ten Blocks. Er verschlüsselt aber diesmal mit DES zuerst das Chiffrat des i-ten Blocks; das Ergebnis addiert er anschließend mit dem Block i+1. Das Resultat dieser Addition ist das Chiffrat für den i+1-te Block. Der CFB-Modus wird meistens dort eingesetzt, wo eine flußorientierte Übertragung erwünscht ist, beispielsweise bei *Secure Shell*.
- **Output Feedback (OFB)** Der OFB-Modus ähnelt sehr dem CFB-Modus. Bei dem OFB-Modus ist aber das DES-Ergebnis, das mit dem i-ten Block addiert wird, ein Produkt aus der i-maligen Verwendung von DES auf eine Zufallszahl (genannt auch *Initialisierungsvektor*). Im OFB-Modus agiert eine Blockchiffrierung als synchrone Stromchiffrierung. Dieser Modus wird häufig bei der Online-Verschlüsselung und bei einem wahlfreien Zugriff eingesetzt.

Diese Betriebsarten sind nicht nur bei DES, sondern auch bei allen anderen Blockchiffrierungen anwendbar. Eine detailierte Beschreibung dieser Betriebsarten findet man bei [Wobst98].

Die DES-Verschlüsselung ist seit einiger Zeit wegen ihre Schlüssellänge Gegenstand der Kritik. Der bei DES verwendete 56 Bit Schlüssel kann bei einem gezielten *brute-force-Angriff* in einer „akzeptablen“ Zeit geknackt werden [Sta95]. Um diesem Problem Herr zu werden, entwickelte man *Triple-DES (3DES)*. Bei diesem Verfahren wählt man einen Schlüssel, der 112 Bit lang ist. Diesen Schlüssel teilt man in zwei Teilschlüssel K_1 und K_2. Anschließend werden diese beiden Teilschlüssel wie folgt verwendet:

$$\texttt{Chiffrat} \quad := \quad DES_{K_1}(DES^{-1}_{K_2}(DES_{K_1}(\texttt{Klartext})))$$

Der Klartext wird also zuerst mit dem Schlüssel K_1 verschlüsselt. Das Ergebnis wird bei dem zweiten Schritt mit dem zweiten Teilschlüssel K_2 entschlüsselt. Anschließend wird nochmal das Ganze mit dem K_1 verschlüsselt. Es ist wichtig zu erwähnen, daß dieses Verfahren nur einen Schutz gegen den kleinen Schlüssel des DES-Verfahrens bietet. Falls der DES-Algorithmus irgendwelche unentdeckte Schwachstellen bzw. Hintertüren hat, hilft die Mehrfachverschlüsselung gegen diese Schwachstellen nicht.

3.2.1.3 IDEA, RC4, RC5 und AES

IDEA, der *International Data Encryption Algorithmus*, gibt es unter diesem Namen erst seit 1992. Bei IDEA handelt es sich wie bei DES um eine Blockchiffrierung, die in acht Runden verschlüsselt. Er arbeitet mit den Blöcken, die 64 Bit lang sind. Zur Verschlüsselung und Entschlüsselung verwendet er einen 128 Bit langen Schlüssel. Durch diesen längeren Schlüssel und durch seine interne Struktur ist IDEA verglichen mit DES widerstandsfähiger gegen Kryptoanalyse [Sta00]. IDEA kann genauso wie DES in vier Betriebsarten, nämlich in ECB, CBC, CFB und OFB, eingesetzt werden.

Eine andere symmetrische Blockchiffrierung ist das *Rivest Cipher 5* Verfahren, genannt *RC5*, das 1994 von seinem Entwickler *Ron Rivest* veröffentlicht wurde. Das Verfahren versucht, durch optimale Ausnutzung der Systembeschaffenheit sehr schnell zu sein. Falls der Algorithmus beispielsweise bei einem 32 Bit Prozessor verwendet wird, benutzt er auch 32 Bit lange Klartextblöcke. Diese Anpassung kann der Algorithmus vornehmen, weil bei seiner Konstruktion größter Wert auf Flexibilität gelegt wurde. Der Anwender kann bei der Verwendung von RC5 Blocklänge, Schlüssellänge und Rundenzahl festlegen. Damit kann der Anwender mit Blick auf die Sensibilität seiner Daten bei jeder Verschlüsselung zwischen der Geschwindigkeit und der Sicherheit selbstständig entscheiden.

Das *Rivest Cipher 4* Verfahren wurde ebenfalls von Ron Rivest entwickelt. Es unterscheidet sich aber vollkommen von den oben erwähnten Verfahren; RC4 ist eine Stromchiffrierung. Abhängig von einem Schlüssel mit variabler Länge wird ein Stromschlüssel erzeugt. Anschließend addiert der Algorithmus mit XOR diesen Stromschlüssel mit dem Klartext. Wie bei den meisten Verfahren mit einer variablen Schlüssellänge hängt bei RC4 die Sicherheit des Verfahrens direkt von der Länge des gewählten Schlüssels ab.

Einer der neuesten blockorientierten Verschlüsselungsalgorithmen ist *Rijndael*[19]. Dieser Algorithmus verdankt seine Bezeichnung den Nachnamen seiner beiden Entwickler, *Vincent Rijmen* und *Joan Daemen*. Bei diesem Verfahren handelt es sich um ein Verschlüsselungsverfahren mit einer variablen Schlüssellänge von $N * 32$ (N ist ein Integer); es arbeitet auf Blöcken der Länge $128, 192$ oder 256. Das amerikanische National Institute of Standards und Technology rief 1997 auf der Suche nach einem Nachfolger für den DES-Algorithmus einen Wettbewerb aus. Im Oktober 2000 stand der Gewinner fest: Rijndael war der Gewinner des Wettbewerbs und wurde vom NIST zum *Advanced Encrypton Standard (AES)* erklärt [AES].

3.2.2 Asymmetrische Verfahren

Bei den *asymmetrischen* Verfahren – bekannt auch unter dem Namen „*Public-Key-Kryptosysteme*“ – wird ein Schlüsselpaar aus zwei verschiedenen Schlüsseln verwendet[20]. Zum Verschlüsseln der Daten verwendet man einen Schlüssel dieser Schlüsselpaare, den

[19] Rijndael, ausgesprochen wie „Reign Dahl“ oder „Rain Doll“.

[20] Die *mathematische Definition* der asymmetrischen Verschlüsselungsverfahren:
K und K' sind Schlüsselräume. E ist die Verschlüsselungsfunktion und D ihre zugehörige Entschlüsselungsfunktion. h ist die Schlüsselrelation $h : K \to K'$, die K nach K' abbildet. Das Verschlüsselungsverfahren (E, D, H) ist *asymmetrisch*, wenn es keine zu h äquivalente Schlüsselrelation gibt, die sich mit einem polynomialen Aufwand berechnen läßt. Das heißt, daß es schwer ist, von dem Verschlüsselungsschlüssel k zu dem Entschlüsselungsschlüssel k' zu kommen.

sogenannten *öffentlichen Schlüssel*. Dieses Chiffrat kann nur mit dem anderen Schlüssel, dem *privaten Schlüssel*, entschlüsselt werden. Der Grundgedanke dieses Verfahrens ist, daß jeder Anwender einen privaten und einen öffentlichen Schlüssel besitzt. Den privaten Schlüssel kennt nur dieser Anwender, und er muß ihn geheimhalten. Im Gegensatz dazu ist der öffentliche Schlüssel jedem zugänglich. Falls eine andere Person diesem Anwender eine geheime Information zukommen lassen möchte, geht er wie folgt vor:

1. Er einigt sich mit dem Anwender über den Verschlüsselungsalgorithmus.
2. Er besorgt sich den öffentlichen Schlüssel dieses Anwenders.
3. Er verschlüsselt die Information mit dem vereinbarten Algorithmus und mit dem öffentlichen Schlüssel des Anwenders.
4. Anschließend schickt er dem Anwender dieses Chiffrat.
5. Der Anwender erhält das Chiffrat und entschlüsselt es mit seinem privaten Schlüssel.

Da nur dieser Anwender den entsprechenden privaten Schlüssel besitzt, kann er sicher sein, daß diese Information nicht von einem Dritten gelesen wurde.

Die Idee von einem asymmetrischen Verschlüsselungsverfahren stammt von *Whitfield Diffie* und *Martin Hellman* [DH76]. Den Algorithmus, der die Anforderungen für ein asymmetrisches Verschlüsselungsverfahren erfüllt, nennt man eine *Einwegfunktion mit Hintertür*[21]. Eine injektive Abbildung f ist eine *Einwegfunktion*, wenn:

$y = f(x)$	einfach zu berechnen ist, und
$x = f^{-1}(y)$	(fast) unlösbar ist.

Eine Einwegfunktion ist also eine Funktion, die deutlich leichter zu berechnen ist als die zugehörige Umkehrfunktion.

Eine Einwegfunktion mit Hintertür stellt eine Erweiterung einer solchen Funktion dar, wobei für diese Funktion f gilt:

$y = f_k(x)$	ist beim bekannten k und x einfach zu berechnen.
$x = f_k^{-1}(y)$	ist (fast) unlösbar, wenn y zwar bekannt, aber k unbekannt ist.
$x = f_k^{-1}(y)$	ist einfach zu berechnen, wenn y und k bekannt sind.

Eine Einwegfunktion mit Hintertür ist somit eine Einwegfunktion, bei der die Umkehrfunktion nur dann leicht zu berechnen ist, wenn eine zusätzliche Information (die Hintertür) bekannt ist. Bezogen auf die asymmetrische Verschlüsselung ist diese Hintertür der private Schlüssel. Ohne diesen Schlüssel ist es fast unmöglich, aus dem Chiffrat die geheime Information zu gewinnen.

Die asymmetrischen Verschlüsselungsverfahren besitzen den Vorteil, daß der Schlüssel, der zur Verschlüsselung eingesetzt wird, nicht geheimgehalten werden muß. Die asymmetrischen Verfahren werden außer zur *Verschlüsselung* auch zum *Schlüsselaustausch* und zur Erzeugung der *digitalen Signatur* verwendet. Mit der digitalen Signatur unterschreibt ein Anwender eine Information. Der Vorgang der Unterzeichnung und der Verifikation dieser Signatur funktioniert wie folgt:

[21] Einwegfunktion mit Hintertür := trap-door one-way function [engl.]

1. Der Anwender verschlüsselt (genauer gesagt: entschlüsselt) die Information mit seinem privaten Schlüssel zum Zweck der Signierung.
2. Anschließend schickt er diese verschlüsselte Information dem zweiten Anwender.
3. Der Empfänger braucht zur Verifikation dieser Signatur den öffentlichen Schlüssel. Er muß außerdem wissen, mit welchem Algorithmus diese Signatur erzeugt wurde.
4. Mit dem öffentlichen Schlüssel des Absenders entschlüsselt (bzw. verschlüsselt) er das bereits erhaltene Chiffrat und überprüft die Signatur auf ihre Echtheit.

Da der private Schlüssel des Absenders nur seinem Besitzer bekannt ist, kann der Empfänger nach einer erfolgreichen Verifikation der Signatur sicher sein, daß die Information auch wirklich von diesem Absender stammt. Eine genauere Beschreibung der wichtigsten Ansätze bei der digitalen Signatur wird im Abschnitt 3.3.2 erläutert.

Bei einem Schlüsselaustauch kommunizieren zwei Anwender miteinander mit dem Ziel, einen *Sitzungsschlüssel*[22] auszutauschen, um damit später ihre gesamte Kommunikation zu verschlüsseln. Der Vorgang des Schlüsselaustauschs ist im Zusammenhang mit der Anwendung eines asymmetrischen Verfahrens sehr wichtig. Dies liegt an der Natur dieser Verfahren. Sie haben – verglichen mit den symmetrischen Verfahren – den Nachteil, daß sie sehr viel langsamer arbeiten. Dies fällt besonders bei der Verschlüsselung großer Datenmengen auf.

Um die Daten so schnell wie bei einem symmetrischen Verfahren zu verschlüsseln und trotzdem auch die Vorzüge von asymmetrischen Verfahren zu nutzen, entwickelte man die *hybriden Kryptosysteme*. Man geht bei einem hybriden Kryptosystem wie folgt vor:

1. Der erste Anwender besorgt sich den öffentlichen Schlüssel des zweiten Anwenders.
2. Er verschlüsselt mit diesem Schlüssel einen zufällig gewählten Sitzungsschlüssel. Dieser Sitzungsschlüssel ist der Schlüssel für ein symmetrisches Verfahren wie beispielsweise DES.
3. Er schickt dem zweiten Anwender diesen verschlüsselten Sitzungsschlüssel.
4. Der zweite Anwender entschlüsselt den Sitzungsschlüssel mit seinem privaten Schlüssel.
5. Beide Anwender haben jetzt den gleichen Sitzungsschlüssel und können von diesem Moment an ihre gesamte Kommunikation mit einem schnellen symmetrischen Verfahren und mit Hilfe des Sitzungsschlüssels verschlüsseln.

Dieses hybride Kryptosystem kommt beispielsweise beim SSL-Protokoll zum Einsatz.

Die drei Hauptaufgabenbereiche der asymmetrischen Verfahren, also

- die Verschlüsselung der Daten,
- die Erzeugung der Digitalen Signatur und

[22] Sitzungsschlüssel := session key [engl.]

- der Schlüsselaustausch,

werden nicht von jedem Algorithmus abgedeckt. Die Tabelle 3.2 faßt die Einsatzgebiete der verschiedenen kryptographischen Verfahren, darunter auch einige der wichtigsten asymmetrischen Verfahren, zusammen.

Verfahren	Verschlüsselung	Digitale Signatur	Schlüsselaustausch
Diffie-Hellman	Nein	Nein	Ja
RSA	Ja	Ja	Ja
ElGamal	Ja	Ja	Nein
DSA	Nein	Ja	Nein
DES	Ja	Nein	Nein
IDEA	Ja	Nein	Nein

Tabelle 3.2 Anwendungsmöglichkeiten der kryptographischen Verfahren

3.2.2.1 RSA

Das asymmetrische Verschlüsselungsverfahren *RSA* verdankt seinen Namen seinen Entwicklern Ron **R**ivest, Adi **S**hamir und Len **A**dleman [RSA]. RSA wurde am MIT entwickelt und 1978 veröffentlicht. RSA beruht auf dem Problem der Faktorisierung[23] sehr großer Zahlen. Bei diesem Verfahren handelt es sich um eine Blockchiffrierung mit einer variablen Schlüssellänge, bei dem zur Verschlüsselung und zur Entschlüsselung derselbe Algorithmus verwendet wird. Typische Schlüssellängen sind 512, 1024 und 2048 Bit. Man sollte aber aus Sicherheitsgründen eine Schlüssellänge von mindestens 1024 Bit wählen [Wobst98].

Bevor man mit dem RSA-Verfahren Daten verschlüsseln bzw. entschlüsseln kann, braucht man einen öffentlichen und einen privaten Schlüssel. Diese Schlüsselpaare erzeugt der RSA-Algorithmus wie folgt:

1. Es werden zwei große (beispielsweise 512 Bit lange) Primzahlen p und q gewählt.
2. Aus diesen beiden Primzahlen wird die Zahl $n = p * q$ errechnet, wobei die Zahl n N-Bit lang ist.
3. Die Eulerische Funktion für n, also $\phi(n) = \phi(p*q) = (p-1)*(q-1)$, wird berechnet.
4. Abhängig von dieser Zahl $\phi(n)$ wird anschließend die Zahl e mit $\phi(n) > e > 1$ gewählt, wobei e zu der Zahl $\phi(n)$ teilerfremd[24] ist.
5. e und n bilden zusammen den öffentlichen Schlüssel.
6. Man berechnet die Zahl d mit der Bedingung, daß $de = 1(mod\ \phi(n))$ ist.
7. d und n sind die privaten Schlüssel.

[23] Bei der Faktorisierung geht es um die Zerlegung einer Integer-Zahl in ihre Primzahlen (Primfaktorzerlegung).
[24] Also gilt: $ggT(\phi(n), e) = 1$.

Wie bei jeder Blockchiffrierung wird vor der Verschlüsselung zuerst der Klartext in Blöcke P_i zerlegt, wobei sie bei RSA eine Länge von „$N-1$“ Bit besitzen. Das Chiffrat C_i eines Blockes P_i erhält man mit:

$$C_i = (P_i)^e (mod\ n)$$

Zur Entschlüsselung wird der gleiche Algorithmus verwendet, aber diesmal wird das private Schlüsselpaar $\{d, n\}$ eingesetzt:

$$P_i = (C_i)^d (mod\ n)$$

Zum besseren Verständnis des RSA-Verfahrens befindet sich im Anhang A.1 ein Beispiel für dieses wichtige Verfahren. RSA wird in vielen Verschlüsselungsprotokollen wie SSL, PGP oder S/MIME verwendet und ist damit Teil vieler Programme. Das RSA-Verfahren kann zur Verschlüsselung, zur Erzeugung der Digitalen Signatur und zum Schlüsselaustausch verwendet werden.

3.2.2.2 Diffie-Hellman und ElGamal

Das Diffie-Hellman, bekannt auch unter dem Namen *Diffie-Hellman-Schlüsselaustausch*, stammt von Whitfield Diffie und Martin Hellman; es ist das erste veröffentliche asymmetrische Kryptoverfahren (von 1976) [DH76]. Das Verfahren nutzt das Problem der Berechnung diskreter Logarithmen[25].

Das Diffie-Hellman-Verfahren kann im Gegensatz zum RSA nur zur Schlüsselübermittlung eingesetzt werden. Bei diesem Verfahren erzeugen zwei Kommunikationspartner durch die Einigung auf zwei große Primzahlen und durch den Austausch von zwei Restzahlen einen Sitzungsschlüssel. Dieses Verfahren ist kein übliches Public-Key-Verfahren, denn es existieren keine privaten und keine öffentlichen Schlüssel. Dies kann man aber auch als Vorteil sehen, denn es gibt keinen privaten Schlüssel, den man nachträglich vor anderen schützen muß.

Das Diffie-Hellman-Verfahren hat aber auch den Nachteil, daß die Kommunikationspartner direkt miteinander den Sitzungsschlüssel aushandeln müssen. Das heißt, daß dieses Verfahren für Bereiche wie beispielsweise den Email-Verkehr, bei dem nur ein Kommunikationspartner aktiv ist, nicht geeignet ist.

Ein weiteres asymmetrisches Verfahren, das auch auf der Schwierigkeit der Berechnung diskreter Logarithmen beruht, ist das *ElGamal-Verfahren*. Es wurde 1985 von Taher ElGamal veröffentlicht [ElGa85]. Es kann zur Verschlüsselung und zur Erzeugung der digitalen Signatur eingesetzt werden. Im Gegensatz zum RSA-Verfahren unterscheidet sich der Algorithmus, der zur Verschlüsselung eingesetzt wird, von dem Algorithmus, der zur Entschlüsselung bzw. zum Signieren benötigt wird.

3.2.3 Einweg-Hashfunktionen

Die in der Kryptographie verwendeten *Einweg-Hashfunktionen* sind Funktionen, die einen beliebigen Klartext auf einen *Hashwert* abbilden. Die Einweg-Hashfunktion nennt man

[25] Bei der Gleichung

$$a^x = g (mod\ n)$$

heißt x **diskreter Logarithmus** von g zur Basis a.

auch *Kompressionsfunktion*, *Konzentrationsfunktion*, *Message-Digest* oder auch *Message-Integrity-Check* (*MIC*); der Hashwert wird auch als *Hashsumme*, *Quersumme* oder *Komprimat* bezeichnet. Die Einweg-Hashfunktion verwendet zum Erzeugen des Hashwertes keinen Schlüssel. Der Hashwert eines Klartextes sollte von jedem Anwender berechnet werden können. Die Einweg-Hashfunktionen besitzen weiterhin noch die folgenden Eigenschaften:

- Aus einem Klartext P wird ein Hashwert H mit einer festen vorgegebenen Länge erzeugt.
- Die Hashfunktion h muß den Hashwert leicht berechnen können.
- Es muß (fast) unmöglich sein, aus einem Hashwert den ursprünglichen Klartext zu bekommen. Das heißt, daß die Hashfunktion *unumkehrbar* sein sollte. Sie sollte eine *Einweg-Funktion* sein, daher auch der Name.
- Es muß (fast) unmöglich sein, für einen Hashwert H irgendeinen Klartext P zu finden, für dessen Hashwert gilt: $h(P) = H$.
- Wenn zwei Klartexte den gleichen Hashwert haben spricht man auch von *Kollision*. Es muß schwer sein, zu einem *bestimmten* Klartext P mit dessen Hashwert $h(P)$ irgendeinen anderen Klartext P' mit $P \neq P'$ zu finden, dessen Hashwert $h(P')$ gleich $h(P)$ ist (*schwache Kollisionsfreiheit*).
- Es soll schwer sein, *irgendein* Klartextpaar (P, P') zu finden, deren Hashwerte gleich sind (*starke Kollisionsfreiheit*[26]).
- Die kleinste Veränderung des Klartextes soll zu einem völlig anderen Hashwert führen. Dies bedeutet, daß beispielsweise jedes Bit des Hashwertes von jedem Bit des Klartextes abhängig sein sollte (*Lawineneffekt*).

Die Einweg-Hashfunktionen werden zum Schutz der Integrität eingesetzt. Zwei der wichtigsten Einsatzgebiete der Einweg-Hashfunktionen sind die *digitalen Signaturen* und die *kryptographischen Prüfsummen* (vgl. Abschnitte 3.3.1 und 3.3.2).

Ein einfaches Beispiel für eine Hashfunktion erhält man, indem man den Klartext in Blöcke P_i der Länge n zerlegt, und jeweils das j-te Bit dieser Blöcke miteinander addiert (mit XOR), um das j-te Bit des Hashwertes zu bekommen. Der Hashwert H für diesen Klartext P wäre dann:

$$h(P) = H = H_1 H_2 \ldots H_n$$

wobei

$$H_i = P_{1,i} \oplus P_{2,i} \oplus \cdots \oplus P_{m,i}$$

Bei diesem Verfahren erhält man einen Hashwert, der n-Bit lang ist. Dieses Verfahren stellt aber eine sehr einfache Hashfunktion dar. Die am häufigsten verwendeten Hashverfahren sind die Verfahren *MD5*, *SHA-1* und *RIPEMD-160*.

[26] Die starke Kollisionsfreiheit bietet Schutz vor dem *Geburtstagsangriff* (Birthday-Angriff) [Sta95].

3.2.3.1 MD5

Message Digest 5 wurde 1991 von *Ron Rivest* als Nachfolger für *MD4*, das ebenfalls von ihm stammt, entwickelt [MD5]. Das MD5-Verfahren arbeitet mit Klartexten von beliebiger Länge. Es zerlegt den Klartext in Blöcke der Länge 512 Bit. Diese werden wiederum in 16 Teilblöcke von je 32 Bit Länge unterteilt; jeder dieser Teilblöcke wird anschließend durch vier Runden geschickt (also insgesamt 64 Schritte). In jeder Runde werden die Bits mit Hilfe der binären Funktionen UND, ODER, XOR und NICHT miteinander verknüpft. Als Endergebnis liefert das MD5-Verfahren einen 128 Bit langen Hashwert.

Bei MD5 wurden Schwachstellen entdeckt, die man verwenden kann, um Kollisionen zu berechnen [Wobst98]. Aus diesem Grund gilt dieses Verfahren nicht als sehr sicher; trotzdem bietet es für normale und nicht hochsicherheitsrelevante Anwendungen eine ausreichende Sicherheit.

3.2.3.2 SHA-1 und RIPEMD-160

Zwei weitere Hash-Verfahren, die ebenfalls auf MD4 basieren, sind *SHA-1* und *RIPEMD-160*. Der *Secure Hash Algorithmus*, genannt SHA, wurde 1993 von *NSA* entwickelt und vom *NIST* veröffentlicht. Zwei Jahre später erschien die erste revidierte Version des Algorithmus, der *SHA-1* [SHA]. Der Algorithmus arbeitet mit Klartexten, die eine maximale Länge von 2^{64} Bit ($2^{64} Bit > 10^7 GByte$) haben und liefert als Ergebnis einen 160 Bit langen Hashwert. Der Algorithmus zerlegt wie MD5 den Klartext in Blöcke der Länge 512 Bit. Beim SHA werden aber aus diesem 512 Bit 80 Teilblöcke erzeugt, die 32 Bit lang sind; die Bearbeitung dieser 512 Bits geschieht in insgesamt 80 Schritten.

Der RIPEMD-Algorithmus wurde im Rahmen des europäischen Projekts *RIPE*[27] entwickelt. Dieser Algorithmus, auch *RIPEMD-128* genannt, liefert 128 Bit lange Hashwerte. Um RIPEMD-128 gegen kryptoanalytische Angriffe resistenter zu machen, entwickelten *Hans Dobbertin*, *Antoon Bosselaers* und *Bart Preneel* 1996 den Algorithmus *RIPEMD-160*, der Hashwerte mit einer Länge von 160 Bit liefert [RIPE160]. RIPEMD-160 arbeitet mit Klartexten von beliebiger Länge. Er zerlegt den Klartext in Blöcke, die 512 Bit lang sind. Diese werden anschließend in 16 Teilblöcke der 32-Bit Länge aufgeteilt und durch 160 Schritte bearbeitet. Für den RIPEMD-Algorithmus gibt es noch die optionalen Erweiterungen auf RIPEMD-256 und RIPEMD-320. Diese liefern einen Hashwert mit einer Länge von 256 bzw. 320 Bit.

Die beiden Algorithmen SHA-1 und RIPEMD-160 gelten zur Zeit als sehr sicher. Nach dem Stand der aktuellen öffentlichen Kryptoanalyse sind diese beiden Verfahren sicherer als MD5 und resistenter gegen Kollisionen und entsprechend resistenter gegen Geburtstagsangriffe. Diese beiden Verfahren sind jedoch langsamer als MD5 [RIPE160].

[27] Der Name *RIPEMD* steht für *RACE Integrity Primitives Evaluation Message Digest.*

3.3 Sicherheitsprotokolle

In [DU93] wird das *Protokoll* wie folgt definiert:

> „*Protokoll*: Vereinbarung über den geordneten Ablauf einer Kommunikation, wobei die Vereinbarung in der Informatik diktatorischen Charakter besitzt: Wer sich nicht an sie hält, wird von der Kommunikation ausgeschlossen. . . “

Ein Protokoll regelt somit die Syntax, die Semantik und die zeitliche Aufeinanderfolge von Datensegmenten, die die Instanzen zur Diensterbringung austauschen. Weiterhin regelt es noch die Art und Weise der Benutzung der unterliegenden Schicht [Dro99].

Bei den *Sicherheitsprotokollen* handelt es sich wiederum um Protokolle, die das Problem der sicheren Kommunikation und des sicheren Datenflusses zwischen den unterschiedlichen Instanzen adressieren. In diesem Abschnitt werden einige der wichtigsten Sicherheitsprotokolle geschildert, die bei dieser Arbeit zum Einsatz kommen.

3.3.1 Kryptographische Prüfsummen (MAC)

Eine *kryptographische Prüfsumme*, auch als *Nachrichtenfreigabecode* oder *Message Authentication Code* (MAC) bezeichnet, ist ein Datenstück, das an eine bestimmte Information angehängt wird und anhand dessen die Integrität und die Authentizität dieser Information überprüft werden können.

Die kryptographische Prüfsumme wird mit der Funktion C folgendermaßen berechnet:

$$MAC = C_k(P)$$

P ist ein Klartext mit einer variablen Länge, k ein geheimer Schlüssel; die Funktion C liefert MAC mit einer festen Länge. Der Schlüssel k ist nur dem Absender und dem Empfänger bekannt. Der Empfänger berechnet zur Verifikation der Richtigkeit der empfangenen Nachricht erneut den MAC der Nachricht und vergleicht diesen Wert mit dem empfangenen MAC (siehe Abbildung 3.3).

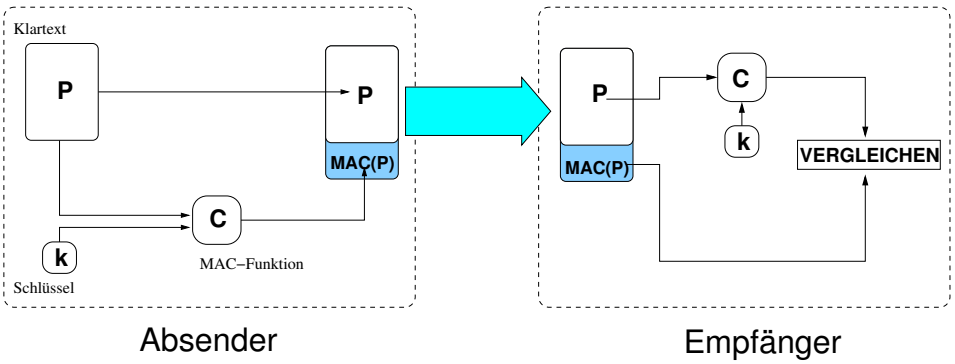

Abbildung 3.3 Erzeugung und Verifikation der kryptographischen Prüfsumme

Eine kryptographische Prüfsumme kann man beispielsweise berechnen, indem man an eine Nachricht eine geheime Information anhängt, die nur dem Sender und dem Empfänger bekannt ist. Als ein solches Geheimnis kann zum Beispiel der *Sitzungsschlüssel* selbst

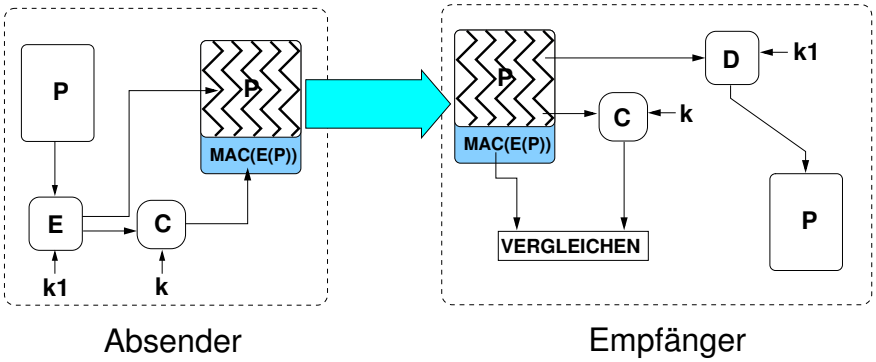

Abbildung 3.4 Verschlüsseln und MAC-Erzeugung

verwendet werden. Für die Nachricht plus Geheimnis wird anschließend mittels einer *Hash-Funktion* ein *Hashwert* (Prüfsumme) berechnet. Der Hashwert wird als kryptographische Prüfsumme mit der Nachricht zum Empfänger übermittelt. Aufgrund seiner Eigenschaften schützt der Hashwert die Nachricht vor unbemerkten Veränderungen. Wenn nämlich jemand die Nachricht verändert, ohne den Hashwert neu zu berechnen, wird dies sofort vom Empfänger bemerkt, weil der Hashwert nicht mehr stimmt. Eine korrekte Neuberechnung des Hashwertes ist aber niemandem möglich, der nicht das Geheimnis kennt[28].

Eine andere einfache Möglichkeit zur Erzeugung einer kryptographischen Prüfsumme besteht darin, den Hashwert für eine Nachricht mit einem symmetrischen Algorithmus zu verschlüsseln und diesen verschlüsselten Hashwert mit der Nachricht einem Empfänger zu senden.

Die Verschlüsselungsverfahren bieten (meistens) neben der Vertraulichkeit auch Authentizität und Integrität. Die kryptographische Prüfsumme bietet Authentizität und Integrität der Daten – sie werden meistens unter dem Begriff *Freigabe* zusammengefaßt – aber ohne Geheimhaltung. Dies ist jedoch ein erwünschtes Verhalten der kryptographischen Prüfsummen. Es gibt beispielsweise Transportprotokolle, die nur die Authentizität und die Integrität der transportierten Informationen sicherstellen sollten. Bei diesen Protokollen möchte man beispielsweise, daß – falls nötig – die Vertraulichkeit von einer höheren Sicherheitsebene übernommen wird.

Die Hauptaufgabe der kryptographischen Prüfsummen ist also nur die Freigabe. Man kann aber eine MAC-Funktion mit einem Verschlüsselungsverfahren kombinieren. Zu diesem Zweck kann man entweder die Daten zuerst verschlüsseln und anschließend ihre kryptographische Prüfsumme erzeugen (siehe Abbildung 3.4), oder man erzeugt zuerst die kryptographische Prüfsumme und verschlüsselt anschließend das Kombinat aus der Nachricht mit ihrer kryptographischen Prüfsumme (siehe Abbildung 3.5). Bei dem ersten Verfahren erhält der Empfänger also eine verschlüsselte Nachricht $E(P)$ und ihre kryptographische Prüfsumme $C_k(E(P))$. Die zweite Vorgehensweise liefert dem Empfänger eine verschlüsselte Nachricht $E(P + C_k(P))$, die die urspungliche Nachricht und ihre kryptographische Prüfsumme beinhaltet.

In diesem Kontext ist es wichtig zu erwähnen, daß die kryptographische Prüfsumme keinen

[28] Einen ähnlichen Algorithmus verwendet das *HMAC-Verfahren* [HMAC]. HMAC ist das am häufigsten verwendete Verfahren zur Erzeugung der kryptographischen Prüfsumme. Es wird für die Sicherheit von IP (IPSec), bei Transport Layer Security (TLS) sowie Secure Electronic Transaction (SET) verwendet [Sta00].

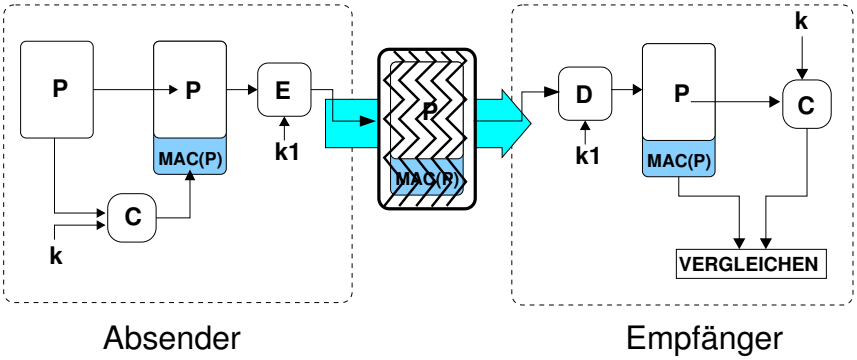

Abbildung 3.5 MAC-Erzeugung und Verschlüsseln

Schutz vor *Zurückweisung* bietet: Eine Person erzeugt beispielsweise eine Nachricht mit einem falschen Inhalt und erzeugt zu dieser Nachricht mittels eines gemeinsamen Schlüssels die kryptographische Prüfsumme. Diese Person kann anschließend behaupten, diese Nachricht sei von ihrem Kommunikationspartner erzeugt worden, mit dem sie diesen geheimen Schlüssel teilt. Die kryptographische Prüfsumme erfüllt nicht die Sicherheitsanforderung „Unabstreitbarkeit“. Zu diesem Zweck eignen sich die digitalen Signaturen.

3.3.2 Digitale Signatur

Die Verfahren zur Erzeugung der *digitalen Signatur* – auch *signierte Prüfsumme* oder *digitale Unterschrift* genannt – wurden entwickelt, um den Anwendern eine Möglichkeit zum Schutz vor *Zurückweisung* zu bieten. Die Digitale Signatur wird hauptsächlich bei der Sicherung der Verbindlichkeit verwendet; sie bietet aber auch genauso wie die kryptographische Prüfsumme Authentizität und Integrität.

Die am häufigsten verwendeten Verfahren zur Erzeugung der digitalen Signatur sind Kombinationen einer Einweg-Hashfunktion mit einem asymmetrischen Verschlüsselungsverfahren. Es gibt zwar auch Verfahren, die eine Signatur mit Hilfe symmetrischer Verschlüsselungsverfahren erzeugen; diese Verfahren sind jedoch sehr aufwendig[29].

Zur Erzeugung der digitalen Signatur mittels der asymmetrischen Verfahren kann man die gesamte Nachricht mit dem privaten Schlüssel verschlüsseln (dies wurde bereits im Abschnitt 3.2.2 erklärt). Diese Vorgehensweise hat aber einen großen Nachteil. Wie bereits im Abschnitt 3.2.2 erwähnt, sind die asymmetrischen Verfahren sehr langsam. Aus diesem Grund ist eine Erzeugung und Verifikation einer solchen Signatur bei großen Dateien sehr zeitaufwendig. Um dieses Problem zu umgehen, kombiniert man zu diesem Zweck die asymmetrischen Verfahren mit den Einweg-Hashfunktionen. Zwei der bekanntesten Ansätze, die auf einer solchen Kombination basieren, sind der *RSA-Ansatz* und der *DSS-Ansatz*.

[29] Bei der Verwendung der symmetrischen Verfahren für die Erzeugung der digitalen Signatur braucht man für jede Kommunikation einen vertrauenswürdigen Dritten, der jede Nachricht bestätigt. Zu diesem Zweck muß jede Nachricht bei dieser Vertrauensperson gespeichert werden. Diese Person tritt jedes Mal als Vermittler zwischen den Kommunikationspartnern auf.

3.3.2.1 RSA-Ansatz

Beim RSA-Ansatz erzeugt man die digitale Signatur in zwei aufeinander folgenden Schritten (siehe Abbildung 3.6):

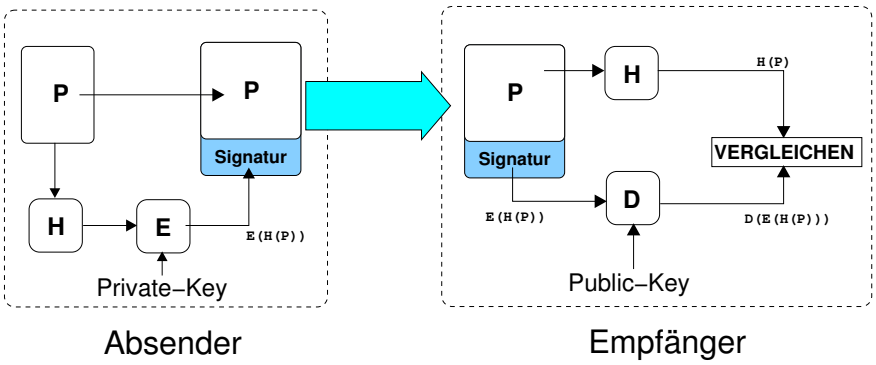

Abbildung 3.6 Erzeugung und Verifikation der digitalen Signatur beim RSA-Ansatz

- Zuerst erzeugt man mit einer Einweg-Hashfunktion den Hashwert des Klartextes (in diesem Zusammenhang spricht man auch von *Fingerprint*).
- Anschließend wird nur dieser Hashwert mit dem *privaten Schlüssel* eines asymmetrischen Verschlüsselungsverfahrens verschlüsselt.

Dieser Ansatz nutzt die Tatsache, daß die Einweg-Hashfunktionen besonders bei großen Datenmengen als Eingabe schneller als die asymmetrischen Verschlüsselungsverfahren sind. Die zu verschlüsselnden Hashwerte sind höchstens 320 Bit lang (siehe Abschnitt 3.2.3.2).

Nach der Erzeugung einer solchen Signatur kann sie dem Klartext angehängt und beispielsweise als Nachricht einer anderen Person übermittelt werden. Zur Verifikation dieser Signatur geht der Empfänger wie folgt vor:

- Er extrahiert aus der Nachricht den Klartext und die digitale Signatur.
- Er erzeugt aus dem Klartext den Hashwert.
- Anschließend entschlüsselt er mit dem öffentlichen Schlüssel des Absenders die empfangene digitale Signatur und
- vergleicht das Ergebnis mit dem zuvor errechneten Hashwert.

Den öffentlichen Schlüssel des Absenders kann der Empfänger beispielsweise aus einem Zertifikat, das er ebenfalls zugeschickt bekam, entnehmen. Außerdem muß die Nachricht die Information über die verwendeten Hash- und Verschlüsselungsverfahren beinhalten.

Bei diesem Ansatz kann zur Erzeugung des Hashwertes eine beliebige Einweg-Hashfunktion – wie beispielsweise SHA-1, MD5 oder RIPEMD-160 – verwendet werden. Zur Verschlüsselung setzt man *RSA* ein (es kann aber auch ein anderes asymmetrisches Verfahren, wie beispielsweise *LUC*[30], verwendet werden).

[30] *LUC* ist ein asymmetrisches Verschlüsselungsverfahren, das in seinem Aufbau sehr dem RSA-Algorithmus ähnelt. Es kann genauso wie RSA zum Verschlüsseln, zum Signieren oder zum Schlüsselaustausch verwendet werden (weitere Information findet man in [Sta95]).

3.3.2.2 DSS-Ansatz

Der *DSS-Ansatz* basiert auf dem *Digital Signature Algorithm* (DSA), einem Verfahren zur Erzeugung der digitalen Signatur. DSA stammt vom *Federal Information Processing Standard*, der wiederum vom NIST als *Digital Signature Standard* veröffentlicht wurde. Der DSS-Ansatz ist auf die Verwendung des DSA zur Erzeugung der digitalen Signatur und SHA als Hash-Funktion festgelegt.

Bei diesem Ansatz wird zuerst mittels *SHA-1* der Hashwert des Klartextes erzeugt. Anschließend wird dieser 160 Bit lange Hashwert von dem DSA bearbeitet. Beim Erzeugen der Signatur verwendet der DSA zusätzlich zum privaten Schlüssel des Signierers ebenfalls eine Zufallszahl und einen globalen öffentlichen Schlüssel. Der Algorithmus liefert als Ergebnis eine Signatur, die aus einem Zahlenpaar (r, s) besteht. Bei der Verifikation der Signatur wird r mit dem Ergebnis der Verifikationsfunktion verglichen, wobei diese Verifikationsfunktion den Hashwert des Klartextes, den globalen öffentlichen Schlüssel, den öffentlichen Schlüssel des Signierers und noch das Zahlenpaar (r, s) als Eingabe hat.

Der DSA basiert wie Diffie-Hellman oder ElGamal auf der Schwierigkeit der Berechnung diskreter Logarithmen. Er ist eine Variante der Unterschriftsalgorithmen von *C.P. Schnorr* und *Taher ELGamal*. Der größte Unterschied zwischen DSA und RSA liegt aber darin, daß DSA nur zur Erzeugung der digitalen Signatur und nicht zum Verschlüsseln der Daten oder zum Schlüsselaustausch verwendet werden kann. Dies kann man als Vorteil dieses Verfahrens ansehen. In vielen Ländern, wie beispielsweise China, ist es verboten, ein Verfahren wie RSA zur Verschlüsselung zu verwenden[31].

3.3.3 Schlüsselmanagement und Zertifizierung

Bei der Schilderung der bisherigen asymmetrischen Verfahren wurde bereits vorausgesetzt, daß die Personen zum Verschlüsseln einer Nachricht bzw. zur Verifikation einer digitalen Signatur stets den öffentlichen Schlüssel ihrer Kommunikationspartner kannten. Die Verwaltung der öffentlichen Schlüssel einer kleinen Gruppe von vertrauenswürdigen Kommunikationspartnern, deren Schlüssel man direkt bekommt, stellt keinen großen Aufwand dar. Dies ändert sich aber, wenn mit Personen kommuniziert wird, die man entweder nicht kennt, oder wenn man mit Personen über einen unsicheren Kanal kommuniziert und den öffentlichen Schlüssel austauscht: In solchen Fällen kann man nicht mehr sicher sein, daß der öffentliche Schlüssel auch wirklich von der angegebenen Person stammt. Durch einen man-in-the-middle-Angriff[32] könnte ein Angreifer sich als eine andere Person ausgeben und damit die von ihm erzeugten privaten Schlüssel verteilen.

Diese Probleme adressieren die *Public-Key Zertifikate* und die *Schlüsselmanagement Systeme*. Mit einem Zertifikat werden einer sozialen Person bestimmte Merkmale, wie beispielsweise Namen, Körpergröße, Wohnort usw., zugeordnet. Hauptaufgabe eines Public-Key Zertifikats ist somit die starke Bindung eines öffentlichen Schlüssels an seinen Eigentümer [Gri98]. Bei der Ausstellung eines solchen Zertifikats wird die Zusammengehörigkeit

[31] *Electronic Privacy Information Center (EPIC)* veröffentlicht jedes Jahr einen Kryptographiebericht, der Informationen über die politische Haltung einzelner Länder zum Thema Kryptographie beinhaltet. Der Bericht ist unter der folgenden Seite erreichbar:
`http://www.epic.org/`

[32] man-in-the-middle-Angriff gehört zur Untergruppe *Tarnung*.

des öffentlichen Schlüssels, des digitalen Namens und der sozialen Identität geprüft. Das Zertifikat wird dabei von einer vertrauenswürdigen Person oder Instanz erzeugt und beglaubigt und hat eine bestimmte *Lebensdauer*. Die Verwaltung der Schlüssel und deren eindeutige Zuordnung zu bestimmten Personen (durch die Ausstellung von Zertifikaten) wird als *Schlüsselmanagement* bezeichnet und von den Schlüsselmanagement-Systemen erledigt. Man unterscheidet das *dezentrale* und das *zentrale Schlüsselmanagement*.

3.3.3.1 Dezentrales Schlüsselmanagement

Das dezentrale Schlüsselmanagement hat seinen wichtigsten Einsatz bei *Pretty Good Privacy (PGP)* [Zimm96]. PGP ist ein Programm, das von Philip Zimmermann entwickelt wurde; es hat hauptsächlich die Sicherheit beim Email-Verkehr zum Ziel. PGP erzeugt mit Hilfe von RSA Schlüsselpaare. Mit Hilfe der erzeugten Schlüssel ist der Anwender in der Lage, mittels PGP Texte und andere Dateien zu verschlüsseln bzw. zu signieren. Diese können anschließend von einem Email-Client an andere Anwender geschickt werden.

Bei PGP wurde als Lösung für das Problem der Zertifizierung das Konzept „Web of Trust“ (Netz des Vertrauens) gewählt, das ein dezentrales Schlüsselmanagement darstellt. Jeder Anwender erzeugt für sich mittels PGP ein Schlüsselpaar, wobei jeder öffentliche Schlüssel einen *Key-Id* (besteht aus den letzten acht Octets des öffentlichen Schlüssels) und einen *Fingerprint* hat. Dieser Fingerprint ist der Hashwert des öffentlichen Schlüssels, der mit MD5 erzeugt wurde. Dieser öffentliche Schlüssel kann auf zwei Arten von einem anderen Anwender zertifiziert werden:

- **Direkte Zertifizierung** Bei einer direkten Zertifizierung, genannt auch *0-stufige* Zertifizierung, erhält der Anwender den öffentlichen Schlüssel direkt von einem anderen (vertrauenswürdigen) Anwender und signiert ihn mit seinem privaten Schlüssel. Er kann diesen Schlüssel auch über einen unsicheren Kanal empfangen. Die Richtigkeit des Ursprungs dieses Schlüssels überprüft er, indem er sich über einen alternativen sicheren Kanal – wie beispielsweise das Telefon – den Fingerprint des empfangenen öffentlichen Schlüssels vorlesen läßt und diesen mit dem Fingerprint, den er selbst berechnet hat, vergleicht.
- **Indirekte Zertifizierung** Ein Anwender kann der Herkunft eines öffentlichen Schlüssels (von einem unbekannten Anwender) indirekt vertrauen, wenn dieser Schlüssel von einem vertrauenswürdigen Dritten – auch *Introducer* genannt – signiert wurde, dessen öffentlichen Schlüssel man bereits besitzt (Diesen Vorgang nennt man auch *1-stufige* Zertifizierung). Dieser öffentliche Schlüssel wird eingebettet in ein Zertifikat dem Anwender geschickt und enthält neben dem öffentlichen Schlüssel auch dessen Erzeugungsdatum und Lebensdauer sowie die digitale Signatur aller Personen, die diesem Schlüssel ihr Vertrauen ausgesprochen haben.

Bei dieser Vorgehensweise entsteht in einem bottom-up-Verfahren ein Netz des Vertrauens, das auf dem Beziehungsgeflecht von Partnern basiert. Dies ist der Vorteil eines solchen dezentralen Schlüsselmanagements, bei dem die Anwender ohne eine zentrale Organisation agieren können und dadurch unabhängig von einer Zertifizierungsinfrastruktur sind (endbenutzerorientiert).

Das dezentrale Schlüsselmanagement ist für kleine Anwendergruppen gut geeignet. Große Gruppen können bei einem dezentralen Schlüsselmanagement nur durch Transivität von

Vertrauen hergestellt werden, das in Wirklichkeit keine Nachvollziehbarkeit bietet. Außerdem lassen sich bei einem solchen Schlüsselmanagement die Schlüssel nicht mit Sicherheit zurückziehen. Ein Anwender möchte seinen Schlüssel beispielsweise dann zurückziehen, wenn er glaubt, daß ein anderer Anwender seinen privaten Schlüssel im Besitz hat.

3.3.3.2 Zentrales Schlüsselmanagement

Bei großen offenen Anwendergruppen löst man das Problem der Authentizität von Personen meistens mit einem zentralen Schlüsselmanagement. Bei einem zentralen Schlüsselmanagement existieren einige unabhängige vertrauenswürdige *zentrale Zertifizierungsstellen*, genannt auch *Certificate Authority (CA)*, die die Authentizität der einzelnen Mitglieder prüfen und zertifizieren. Ein Anwender muß somit zur Prüfung der Authentizität einer anderen Person einfach deren Zertifikat prüfen, das von einer (dem Anwender) vertrauenswürdigen Zertifizierungsstelle ausgestellt und beglaubigt wurde.

Das *Authentication Framework X.509*, bekannt auch als *X.509-Verzeichnisprüfungsdienst*, ist ein Beispiel für ein zentrales Schlüsselmanagement-System [X509]. X.509 gehört zu der Reihe der X.500 Verzeichnisdienste. Die X.500-Reihe sind Empfehlungen der *ITU* (*International Telecommunication Union* früher *CCITT* genannt)[33] und beschreiben einen Verzeichnisdienst. X.509 beschreibt in diesem Kontext die Authentikation einzelner Instanzen, das Aussehen von Zertifikaten und die Art des Schlüsselmanagements.

Ein Zertifikat wird in X.509 von einer Zertifizierungsstelle erzeugt und beglaubigt. Der im Zertifikat enthaltene öffentliche Schlüssel des Anwenders kann (und sollte) von dem Anwender selbst erzeugt werden. Der Aufbau eines Zertifikats nach X.509 wird im Anhang A.4 geschildert. Dort findet sich auch ein Beispiel für ein solches Zertifikat.

Ein Zertifikat wird nach seiner Ausstellung von der Zertifizierungsstelle oder vom Anwender persönlich im Verzeichnis (beispielsweise nach dem X.500-Standard [X500]) abgelegt. Dieses Verzeichnis muß für alle zugänglich sein. Um eine große Anzahl von Anwendern in ein solches Verzeichnis aufnehmen zu können, wurde bei X.509 eine verteilte Struktur von Verzeichnissen und Zertifizierungsstellen vorgeschlagen. Die verwendete Zertifizierungsinfrastruktur bei X.509 besteht aus einem Zertifizierungsbaum. An der Wurzel dieses Baumes ist die sogenannte *Root-Zertifizierungsstelle* angesiedelt. Jede andere Zertifizierungsstelle unter der Wurzel erzeugt ein eigenes Zertifikat und läßt dieses von den anderen Zertifizierungsstellen *unter* und *über* ihr (in diesem Baum) signieren.

Ein Beispiel für einen solchen Zertifizierungsbaum stellt die Abbildung 3.7 dar. Falls der Student „Stud1“ mit dem Studenten „Stud5“ kommunizieren möchte, braucht er zuerst dessen Zertifikat. Um dem Zertifikat von Stud5 zu vertrauen, geht er wie folgt vor: Das Zertifikat von Stud5 ist von der Zertifizierungstelle „Fachbereich-Physik“ unterschrieben. Um dieses Zertifikat zu verifizieren, braucht Stud1 den öffentlichen Schlüssel vom „Fachbereich-Physik“, wobei dessen Schlüssel wiederum von der Zertifizierungstelle „Uni-Frankfurt“ signiert ist. Der öffentliche Schlüssel von Uni-Frankfurt ist vom „Fachbereich-Informatik“ und sein Schlüssel schließlich von Stud1 selbst unterschrieben. Auf diesem Pfad kann Stud1 das Zertifikat von Stud2 verifizieren.

Ein solches zentrales Schlüsselmanagement wie das von X.509 ist sehr stark von einer zentralen Infrastruktur abhängig. Es löst trotzdem das Problem der Authentizität von

[33] ISO/IEC erklärte später diese Empfehlung zum Zertifikatsformat-Standard X.509 (ISO/IEC 9594-8).

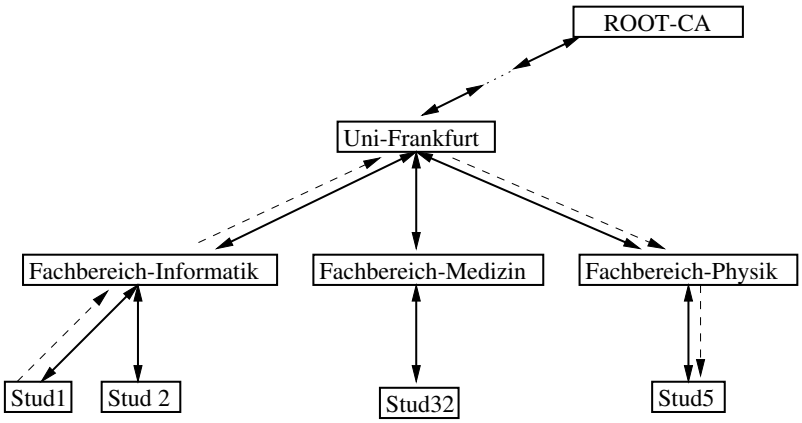

Abbildung 3.7 Beispiel für einen Zertifizierungsbaum nach X.509

Personen in großen, offenen Benutzergruppen wie im Internet, die nicht vorher in Kontakt waren [Gri98]. Aus diesem Grund wird es auch schon in vielen Protokollen verwendet. *Privacy Enhanced Mail (PEM)* [RFC1421] benutzt ein zentrales Schlüsselmanagement, das zum X.509-Schlüsselmanagement kompatibel ist. *S/MIME* [SMIME], *SET* [Dre99], *SSH* [SSH] und das *SSL-Protokoll* sind weitere Protokolle, die sich beim Schlüsselmanagement ebenfalls des X.509-Protokolls bedienen.

3.3.4 Secure Socket Layer (SSL)

Eine der bekanntesten Sicherheitsprotokolle ist das *Secure Socket Layer* (SSL) [SSL]. Das SSL-Protokoll ist ein verbindungsorientiertes bidirektionales Punkt-zu-Punkt Daten Transport Protokoll. SSL arbeitet auf der Socket-Ebene oberhalb des verbindungsorientierten TCP/IP und unterhalb einer Anwendungsschicht, wie beispielsweise dem HTTP-Protokoll (vgl. Abbildung 3.2). Das SSL-Protokoll erfüllt dabei die folgenden Sicherheitsanforderungen:

- Beidseitige Authentisierung,
- Vertraulichkeit durch Verschlüsselung und
- Datenintegrität,

wobei eine beidseitige Authentisierung nicht zwingend vorgeschrieben ist; es kann beim Verbindungsaufbau auch nur eine Server-Authentisierung durchgeführt werden.

Beim SSL-Protokoll wird vor dem tatsächlichen Datenaustausch zwischen den Kommunikationspartnern eine Vereinbarung über die Art der verwendeten Algorithmen getroffen. Bei diesem sogenannten *Handshake* findet ebenfalls die Authentisierung der Kommunikationspartner statt. Diese Vorgehensschritte werden vom *Handshake-Protokoll* beschrieben. Nach dem Handshake werden die Daten in Blöcke, sogenannte *Record Protcol Units*, zerstückelt und mittels vorher ausgehandelter kryptographischer Algorithmen behandelt. Diese Aufgabe wickelt wiederum das *Record-Protokoll* ab. Im einzelnen wird unter SSL eine sichere Verbindung auf folgende Weise hergestellt:

1. Der Client teilt dem Server mit, daß er eine Verbindung aufbauen möchte. Dabei schickt er außerdem eine Liste der unterstützten Algorithmen und Protokolle (z. B. zur Verschlüsselung und zur Komprimierung).

2. Der Server wählt aus der empfangenen Liste die Algorithmen, die er in dieser Sitzung verwenden möchte, und schickt seine Wahl dem Client. Er übermittelt dem Client ebenfalls seinen öffentlichen Schlüssel, der in einem Zertifikat (nach X.509) integriert ist. Dieser Schlüssel wird im folgenden zur Übertragung des Session Keys verwendet. Falls nötig, fordert der Server den Client auf, sich ebenfalls mittels eines Zertifikats zu identifizieren (dies ist bei einer beidseitigen Authentisierung nötig).

3. Das Zertifikat des Servers wird vom Client verifiziert. Anschließend schickt er – falls verlangt – dem Server seinen öffentlichen Schlüssel (ebenfalls in einem Zertifikat). Falls der Client zum ersten Mal mit diesem Server über SSL kommuniziert oder kein Schlüssel von der früheren Sitzung vorhanden ist, muß der Client mit Hilfe von Zufallszahlen einen neuen Sitzungsschlüssel (Session Key) generieren. Der Sitzungsschlüssel gilt für mehrere Verbindungen. Der Client verschlüsselt den Sitzungsschlüssel mit dem öffentlichen Schlüssel des Servers, so daß er nur vom Server entschlüsselt werden kann, und schickt ihn zum Server. Damit ist der Handshake-Prozeß beendet.

4. Nun beginnt der eigentliche Datenaustausch zwischen dem Client und dem Server. Alle Nachrichten dieser Verbindung werden in Blöcke zerteilt (Fragmentierung). Jeder Block wird anschließend komprimiert und um eine kryptographische Prüfsumme MAC (vgl. Abschnitt 3.3.1) ergänzt. Das Kombinat wird anschließend mit dem vereinbarten Sitzungsschlüssel mittels eines symmetrischen Verschlüsselungsverfahrens verschlüsselt. Diese werden anschließend in TCP-Pakete verpackt und dem Kommunikationspartner übermittelt.

5. Der Kommunikationspartner entschlüsselt das empfangene Datenpaket mit dem Sitzungsschlüssel und prüft die kryptographische Prüfsumme des komprimierten Fragments. Nach dieser Verifikation wird anschließend das Fragment dekomprimiert und mit den weiteren Fragmenten zusammengesetzt.

Weitere Bestandteile des SSL-Protokolls sind außer dem *Handshake-Protokoll* und dem *Record-Protokoll* das *SSL Application Data-Protokoll*, das *SSL Change Cipher Spec-Protokoll* und das *SSL Alert-Protokoll*. Das Application Data-Protokoll sorgt für die Datenübermittlung zwischen der Anwendung und dem SSL-Protokoll, während das Alert-Protokoll zur Weiterleitung der Warn- und Fehlermeldung benutzt wird. Das Change Cipher Spec-Protokoll dient wiederum zur Signalisierung der Änderungen in der Verschlüsselung. Eine Change Cipher-Nachricht wird vom Client und vom Server geschickt, um den Kommunikationspartner davon in Kenntnis zu setzten, daß von nun an alle folgenden Blöcke mit dem neuen ausgehandelten Verfahren (Cipher Spec) behandelt werden. Eine detailliertere Beschreibung der einzelnen SSL-Bestandteile findet man in [SSL].

Die aktuelle Version des SSL-Protokolls ist die Version v3.0; sie ist als Internet-Draft bei der *Internet Engineering Task Force (IETF)* erhältlich. SSL wurde ursprünglich 1994 von der *Netscape Communications Corporation* als ein offener Standard entwickelt und hatte sein größtes Einsatzgebiet beim Web-Browser (*Netscape Navigators*) und beim Web-Server (*Netscape Secure Server*) dieser Firma. Die Implementation von Netscape war jedoch für

kommerzielle Zwecke nicht frei. Außerdem fiel dieses Produkt wegen der verwendeten Verschlüsselungsverfahren (genauer gesagt: wegen der Schlüssellängen) unter die Exportbeschränkungen der USA.

Aus diesen Gründen entwickelten *Eric A. Young* und *Tim J. Hudson* 1995 die freie Implementierung *SSLeay*. Seit 1998 wird dieses Projekt unter dem Namen *OpenSSL* von der Internet-Gemeinde weiterentwickelt. Die aktuelle Version des Programms unterstützt neben *SSL-v2/v3* auch *Transport Layer Security (TLS)*, das von der IETF entwickelt wurde und in Zukunft das SSL-Protokoll ersetzen soll [RFC2246]. Das OpenSSL-Programmpaket bietet neben Bibliotheken für SSL und TLS ebenfalls viele weitere Werkzeuge, mit deren Hilfe man beispielsweise Zertifikate nach X.509 erzeugen kann. Eine Liste der möglichen kryptographischen Algorithmen und der Protokolle, die von OpenSSL angeboten werden, findet man im Anhang A.3.

Kapitel 4

Sicherheitsanalyse der INDIGO-Infrastruktur

In diesem Kapitel erfolgt eine sicherheitsorientierte Analyse der INDIGO-Infrastruktur.

Im ersten Abschnitt des Kapitels werden die an dieser Infrastruktur beteiligten Akteure und die schützenswerten Güter erläutert. Im Anschluß daran werden in den weiteren Abschnitten aus der Sicht der Akteure die Sicherheitsanforderungen beschrieben, die an die Güter gestellt werden.

In diesem Kapitel werden außerdem Fragen der Sicherheit – wie z. B. Authentisierung, Autorität, Verfügbarkeit und Integrität der Daten und Dokumentmethoden – in bezug auf die INDIGO-Infrastruktur behandelt. Daneben werden die jeweiligen Sicherheitsprobleme analysiert.

Lösungen und Gegenmaßnahmen für die in diesem Kapitel beschriebenen Sicherheitsprobleme werden in Kapitel 5 angeboten.

4.1 Sicherheitsziele

Die *Sicherheitsziele* beschreiben die *schützenswerten Güter* [Gri94]. Das Ziel der Sicherheitsvorkehrung liegt darin, die Güter eines Systems vor unerlaubten Zugriffen und Modifikationen zu schützen. In einer Infrastruktur, in der mehrere Akteure mit unterschiedlichen Interessen agieren, kann es zu einem Interessenkonflikt und dadurch zu Sicherheitsverletzungen kommen. Um die Sicherheit zu garantieren, müssen die von den Akteuren gemeinsam verwendeten Güter so geschützt sein, daß diese Interessenkonflikte nicht zu einer Störung der Funktionalität des Systems führen.

Um solche Interessenkonflikte näher betrachten zu können, muß man sich zuerst über die beteiligten Akteure und die schützenswerten Güter bei der Infrastruktur im klaren sein. In diesem Abschnitt werden bei einer intensiven Betrachtung der INDIGO-Infrastruktur die Akteure und die schützenswerten Güter abstrahiert.

4.1.1 Akteure

Akteure[1] werden in [Gri94] als „Personen in Rollen“ bezeichnet. Der Akteur repräsentiert die in der Rolle handelnde Person. Er ist also die Verbindung einer Person mit einer Rolle. Eine *Rolle* ist andererseits ein spezifiziertes Verhalten, zum Beispiel ein programmierter Prozeß. Ein Akteur kann vollständig durch ein Programm realisiert sein; allerdings bleibt dahinter immer eine *verantwortliche Person*, auch wenn sie sich vollständig automatisiert verhält.

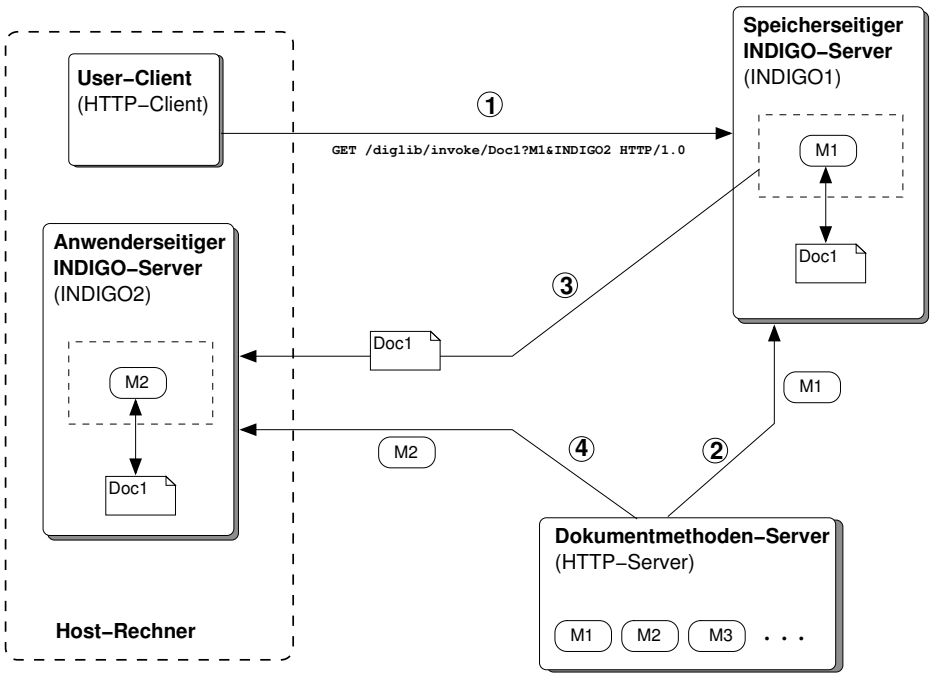

Abbildung 4.1 Akteure beim INDIGO-System

Ein allgemeines Modell für die Arbeit der INDIGO-Infrastruktur stellt die Abbildung 4.1 dar; sie ist die detailierte Beschreibung eines Teilabschnitts der Abbildung 2.1 (siehe Seite 7). Diese Abbildung zeigt zwei miteinander kommunizierende INDIGO-Server. Der Anwender schickt mittels eines HTTP-Clients eine Anfrage an den speicherseitigen Ausführungs-Server „INDIGO1“. Dieser Server ist Teil der digitalen Bibliothek. Der Anwender möchte, daß auf diesem INDIGO-Server die Dokumentmethode „M1“ des Dokuments „Doc1“ ausgeführt wird. Die Anfrage beinhaltet ebenfalls die Adresse seines anwenderseitigen Ausführungs-Servers „INDIGO2“. Der INDIGO-Server holt sich die Methode M1 von einem Dokumentmethoden-Server. Die Ausführung der Dokumentmethode M1 veranlaßt den INDIGO-Server zum Transport des Dokuments Doc1 zu dem zweiten INDIGO-Server und dem Aufruf der Dokumentmethode „M2“ auf dem zweiten INDIGO-Server. Die Dokumentmethode M2 wird genauso wie die Dokumentmethode M1 über einen Dokumentmethoden-Server bezogen.

Die Befehle und Nachrichten werden von einer Komponente zur anderen Komponente über ein Netz übertragen. Diese Komponenten arbeiten bei dieser Arbeit zusammen. Sie führen

[1] Akteur := principal [engl.]

dabei Operationen auf Ressourcen aus. Hinter diesen Komponenten stehen die Akteure, wobei diese Akteure unterschiedliche Interessen und Ziele verfolgen dürfen. Die in der INDIGO-Infrastruktur beteiligten Akteure sind:

- **Dokumentmethoden-Produzent** Er ist der Hersteller einer Dokumentmethode. Es kann sich beispielsweise um den Programmierer der Methode handeln.
- **Dokumentmethoden-Server-Betreiber** Dieser Betreiber bietet Dokumentmethoden zum Herunterladen an. Zu diesem Zweck verwendet er einen Dokumentmethoden-Server. Bei diesem Server handelt es sich meistens um einen normalen HTTP- bzw. FTP-Server.
- **Autor** Der Autor eines Dokuments ist für den Inhalt des Dokuments zuständig. Er legt außerdem die Attribute dieses Dokuments fest und weist jedem Dokument die entsprechenden Dokumentmethoden zu.
- **Bibliothek** Sie stellt die von den Autoren erstellten Dokumente den Anwendern zur Verfügung. Sie agiert zu diesem Zweck über einen Verbund von speicherseitigen INDIGO-Ausführungs-Servern.
- **Anwender** Eine Person in ihrer Rolle als „Anwender“ (oder auch Benutzer genannt) agiert über ihren User-Client; es handelt sich dabei um einen HTTP-Client, wie z. B. einen Telnet-Client oder einen Web-Browser. Über diesen Client schickt der Anwender Anfragen an den speicherseitigen INDIGO-Server. Der Anwender kann außerdem einen anwenderseitigen INDIGO-Ausführungs-Server betreiben. Somit ist er auch in der Lage, Dokumente lokal bei sich zu speichern und Dokumentmethoden aufzuführen.
- **Netzwerk-Betreiber** Dieser Akteur stellt die gesamten Transportsysteme und Zwischenstationen zur Verfügung. Die gesamte Kommunikation zwischen den unterschiedlichen Komponenten dieser Infrastruktur läuft über dessen Netze.

Wichtig ist dabei, daß beispielsweise die Dokumentmethode kein Akteur ist. Die Methode ist als Spezifikat die Rolle; in ihrer Ausführung ist sie der automatisierte Anteil des Akteurs in dieser Rolle. Der Akteur ist der Hersteller der Dokumentmethode, der indirekt über diese durch den Methodenaufruf handelt. Falls in dieser Arbeit vom Ausführungs-Server oder von der Dokumentmethode als Akteur gesprochen wird, sind immer die Personen gemeint, die mittels dieser aktiven Instanzen agieren.

4.1.2 Schützenswerte Güter

Bei jedem System existieren Bedrohungen, die sich gegen die *schützenswerten Güter* dieses Systems richten. Es kann sich bei diesen Gütern[2] um die Ressourcen oder um sensible Daten eines Systems handeln. Die Sicherheitsvorkehrungen sollten stets in der Lage sein, diese Bedrohungen durch besondere Maßnahmen so weit zu bekämpfen, daß das *verbleibende Risiko* akzeptiert werden kann. Die Güter, die man während des Betriebs der INDIGO-Infrastruktur schützen soll, sind[3]:

[2] In dem Bell-LaPadula Sicherheitsmodell werden die Güter als „Objekte“ bezeichnet.

[3] In diesem Zusammenhang werden absichtlich die Komponenten, wie beispielsweise der Dokumentmethoden-Server oder der User-Client, nicht als schützenswerte Güter beachtet; nähere Erläuterung siehe Abschnitt 4.6.

- **Metadokumente** Damit sind die Dokumente und deren Metadaten gemeint. Jeder Abschnitt der Metadokumente – Attribute, Methodenzuordnung und Inhalt – kann Opfer eines Angriffs sein. Beim Verschicken der Metadokumente kann man zum Beispiel durch einen aktiven Angriff die Datei-Inhalte manipulieren. Dies gilt ebenfalls für die gespeicherten Dokumente. Eine böswillige Dokumentmethode kann theoretisch beim Zugriff den Inhalt der Dateien ändern.
- **Dokumentmethoden** Die meisten Sicherheitsprobleme, die bei den Metadokumenten existieren, tauchen auch beim Umgang mit den Dokumentmethoden auf. Insgesamt sind bei den Dokumentmethoden die in Methoden enthaltenen Algorithmen und die sensiblen Daten zu schützen. Verglichen mit den Metadokumenten kommt hier noch hinzu, daß die Dokumentmethoden bei ihrer Ausführung ebenfalls als aktive Instanzen tätig sind. Aus diesem Grund ist die Sicherstellung der Datenintegrität und Datenauthentizität eines der wichtigsten Sicherheitsziele bezüglich der Dokumentmethoden.
- **INDIGO-Server** Der INDIGO-Server und die in ihm gespeicherten Daten sollten stets vor Angriffen der Dokumentmethoden und anderer böswilliger Komponenten geschützt sein.
- **Basis-Infrastruktur** Die Basis-Infrastruktur besteht aus den Host-Rechnern und dem Server-Verbund. Ihre Ressourcen werden von der INDIGO-Infrastruktur benutzt und müssen vor dem Mißbrauch von Seiten der INDIGO-Infrastruktur geschützt werden. Hier sind nicht nur die Systemressourcen gemeint, sondern beispielsweise auch die Daten der Benutzer auf dem Host-Rechner.

Das Ziel der Angriffe und Bedrohungen sind stets die Güter eines Systems. Diese Sicherheitsbedrohungen entstehen – wie bereits im Abschnitt 4.1 erwähnt – durch Interessenkonflikte zwischen den unterschiedlichen Akteuren. Dadurch, daß die notwendigen Anforderungen erfüllt sind, werden diese Güter geschützt.

In den folgenden Abschnitten werden die Sicherheitsanforderungen aus der Sicht einzelner Akteure beschrieben, die an die Güter der INDIGO-Infrastruktur gestellt werden. Dabei konzentrieren sich die Sicherheitsanforderungen der Akteure jeweils auf unterschiedliche Güter; eine Übersicht bietet die Tabelle 4.1. Bei dieser Beschreibung wird ausschließlich auf drei Güter – also auf Metadokumente, Dokumentmethoden und INDIGO-Server – eingegangen.

Die wichtigsten Anforderungen, die an die Basis-Infrastruktur gestellt werden, sind die Verfügbarkeit und Robustheit der Infrastruktur. Die Infrastruktur ist gegen einen überhöhten Ressourcenverbrauch – wie zum Beispiel denial-of-service-Attacken – zu schützen. Manchmal genügt es nicht, die einzelnen Anwendungen gegen Angreifer sicherer zu gestalten, während diese Anwendung auf einem unsicheren Host-Rechner läuft: Ein System ist nur so sicher wie sein schwächstes Glied. Die Basis-Infrastruktur sollte natürlich auch die Sicherheitsanforderungen nach Vertraulichkeit, Authentizität, Autorität und Integrität erfüllen.

Die Sicherheitsanforderungen bezüglich der Basis-Infrastruktur sind in diesem Kontext nicht spezifisch für die INDIGO-Infrastruktur und werden aus diesem Grund auch nicht weiter behandelt. Sie sind eher allgemeine Anforderungen an die Infrastrukturen, die mobile Codes verwenden. Die gleichen Sicherheitsanforderungen werden auch von den Infrastrukturen für *mobile Agenten* verlangt. In diesem Zusammenhang kann beispielsweise auf

Akteure	Schützenswerte Güter
Autor	Metadokumente, Dokumentmethoden
Dokumentmethoden-Produzent	Metadokumente
Dokumentmethoden-Server-Betreiber	Dokumentmethoden-Server[4]
Bibliothek	Metadokumente, Dokumentmethoden, INDIGO-Server
Anwender	Metadokumente, Dokumentmethoden, INDIGO-Server

Tabelle 4.1 Sicherheitsanforderungen an die schützenswerten Güter aus Sicht der Akteure

[AgentTCL] verwiesen werden. Hier wird bezüglich der *mobilen Agenten* beschrieben, wie der Schutz von Netzwerken durchgesetzt werden kann.

4.2 Sicherheitsanforderungen aus Sicht des Autors

Die Sicherheitsanforderungen aus Sicht des Autors konzentrieren sich auf die schützenswerten Güter *Metadokument* und *Dokumentmethoden*.

4.2.1 Metadokumente

Die Anforderungen, die von den Autoren an die Dokumentdateien gestellt werden, sind:

- Vertraulichkeit,
- Authentizität und Integrität,
- Unabstreitbarkeit und Verbindlichkeit,
- Verfügbarkeit.

Vertraulichkeit Es gibt gewisse Dokumente, die sogenannten *nicht-frei-zugänglichen* oder *proprietären* (in exklusivem Besitz befindlichen) Dokumente, die der Autor nur für einen bestimmten Kreis von Anwendern freigibt. Dies ist besonders bei kommerzielen Dokumenten der Fall, indem nur die Anwender, die bereits bezahlt haben, einen Zugriff auf den Inhalt eines Dokuments erhalten.

Diese proprietären Dokumente müssen entsprechend vertraulich behandelt werden. Dies gilt nicht nur beim Transport zwischen den Akteuren. Ein nicht-autorisierter Anwender sollte ebenfalls keine Möglichkeit haben, auf die gespeicherten Dokumente auf dem speicherseitigen bzw. anwenderseitigen INDIGO-Server zuzugreifen. Außerdem sollte der INDIGO-Server Mechanismen anbieten, die einer fremden Dokumentmethode den Zugriff auf diese Dokumente versperrt.

[4] Die Sicherheitsanforderungen des Dokumentmethoden-Server-Betreibers betreffen nur den Dokumentmethoden-Server, der aber bei dieser Arbeit nicht als schützenswertes Gut betrachtet wird (siehe 4.6). Aus diesem Grund werden die Anforderungen dieses Betreibers nur am Rande und zusammen mit den Sicherheitsanforderungen des Dokumentmethoden-Produzenten beschrieben.

Authentizität und Integrität Die Datenauthentizität ist neben der Datenintegrität die wichtigste Anforderung, die man an die Dokumente stellen muß. Das Dokument sollte Informationen beinhalten, die eindeutig auf dessen Autor verweisen. Um die Rechte des Autors vor einer Erzeugung seitens eines Angreifers zu schützen, sollte das Dokument sogar mit Mechanismen und Merkmalen ausgestattet sein, die es dem Autor trotz einer Manipulation ermöglichen, die Herkunft des Dokuments zu demonstrieren (Stichwort: Urheberrechtsschutz). Außerdem sollte der Autor jederzeit in der Lage sein, die Integrität seines Dokuments, das im Umlauf ist, nachzuprüfen. Dies bezieht sich nicht nur auf den Dokumentinhalt, sondern auch auf die Methodenzuordnung des Metadokuments.

Unabstreitbarkeit und Verbindlichkeit Die Verbindlichkeit spielt für den Autor bei den proprietären Dokumenten häufig eine wichtige Rolle, besonders bei solchen Dokumenten, bei denen der Anwender direkt mit dem Autor wegen einer Zugriffsautorisation in Verbindung steht. Es muß beispielsweise nachweisbar sein, daß ein Dokument von einem bestimmten Autor geschrieben wurde.

Für die kommerziellen Dokumente bedeutet Unabstreitbarkeit insbesondere, daß weder der Autor noch der Anwender den Versand oder den Empfang abstreiten können. Sonst könnte ein Autor behaupten, daß er beispielsweise für das Geld, das er erhalten hat, bereits das Dokument (bzw. den benötigten Autorisationscode) geschickt hat, obwohl das nicht der Wahrheit entspricht. Wie kann ein Autor sicher sein, daß der Anwender die Wahrheit sagt, wenn er behauptet, er habe das gewünschte Dokument nicht bekommen? Oder der Anwender bezahlt für ein Dokument, aber während des Ladens des Produkts stürzt bei ihm die Anwendung ab. Wie kann er beweisen, daß er dieses Dokument nicht erhalten hat? Wie kann der Autor sicher sein, daß der Anwender die Wahrheit sagt (Unabstreitbarkeit)? Ab welcher Stufe des Handels kann man von einer Verbindlichkeit ausgehen? Welche Quittungen von seiten der Akteure sind für eine verbindliche Transaktion ausreichend?

Verfügbarkeit Da der Vertrieb der Dokumente nicht direkt vom Autor, sondern von der Bibliothek erledigt wird, steht die Verfügbarkeit der Dokumente mit der Verfügbarkeit des Host-Rechners und des INDIGO-Servers bei der Bibliothek in Verbindung. Insbesondere sollen die Dokumente nicht unbefugt gelöscht werden können. Der Server sollte beispielsweise nicht-autorisierten Akteuren das Löschen der Dokumente verbieten.

4.2.2 Dokumentmethoden

Die Sicherheitsanforderungen des Autors an die Dokumentmethoden sind:

- Integrität und Authentizität,
- Verfügbarkeit.

Integrität und Authentizität Integrität und Authentizität der Dokumente und ihrer Wiedergabe hängen auch von den Methoden ab, die diesen Dokumenten zugeordnet werden. Bei der Wiedergabe eines Dokuments mit einer manipulierten Präsentationsmethode kann beispielsweise der Inhalt des Dokuments verändert wiedergegeben werden; die Methode kann sogar den Inhalt des Dokuments manipulieren. Aus diesem Grund muß

der Autor den Produzenten dieser Methoden vertrauen und sollte bei der Wahl seiner Methoden bzw. deren Adresse sicherstellen, daß der Anwender die Authentizität und die Integrität der angegebenen Dokumentmethoden prüfen kann.

Verfügbarkeit Bei näherer Betrachtung der INDIGO-Infrastruktur stellt man fest, daß man beim Umgang und bei der Benutzung der Dokumente auf die Verfügbarkeit der Dokumentmethoden angewiesen ist: Ohne diese Methoden haben die Anwender bzw. die Bibliotheken keine Möglichkeit, auf ihre gewünschten Dokumente – beispielsweise auf der Präsentationsebene – zuzugreifen.

Aus diesem Grund sollte der Autor stets ein Interesse daran haben, daß die von seinem Dokument benötigten (und im Methodenzuordnungsabschnitt angegebenen) Dokumentmethoden bzw. indirekt die Dokumentmethoden-Server, die diese Methoden zur Verfügung stellen, stets von den Anwendern und Bibliotheken erreichbar sind.

4.3 Sicherheitsanforderungen aus Sicht des Dokumentmethoden-Produzenten und des Dokumentmethoden-Server-Betreibers

Der Dokumentmethoden-Produzent stellt Dokumentmethoden für ein bestimmtes Datenformat her. Diese Methoden können aber auch speziell für ein bestimmtes Dokument zugeschnitten sein, so daß sie den Anforderungen des Autors bezüglich der Sicherheitsprüfungen entsprechen.

Die Sicherheitsanforderung, die der Produzent direkt an seine Dokumentmethoden stellt, betreffen die Vertraulichkeit. Die Dokumentmethoden, die bei dieser Arbeit verwendet werden, sind frei zugänglich; sie brauchen daher nicht vertraulich behandelt zu werden. Dennoch gibt es Dokumentmethoden-Produzenten, die ihre Programme und darin enthaltene Algorithmen geheimhalten möchten. Hier ist als Beispiel das DVD-Konsortium zu erwähnen, das versucht, seinen zum Präsentieren der Video-Filme benötigten Algorithmus geheimzuhalten [Schw01]. Wenn man solche Dokumentmethoden ebenfalls bei der INDIGO-Infrastruktur benutzt, sollte man den Transport solcher Dokumentmethoden sichern. Hier müssen die Methoden genauso wie bei den Metadokumenten nicht nur beim Transport, sondern auch bei der Speicherung und bei der Ausführung vertraulich behandelt werden.

Die Sicherheitsanforderungen des Dokumentmethoden-Server-Betreibers richten sich dagegen nur an den von ihm betriebenen Dokumentmethoden-Server. Ein solcher Server kann einfach ein HTTP- oder FTP-Server sein. Dieser Server muß gegenüber Angriffen und Überlastungen robust sein (Verfügbarkeit). Er sollte sich außerdem – falls notwendig – eindeutig authentifizieren können (z. B. mittels eines Zertifikats).

4.4 Sicherheitsanforderungen aus Sicht der Bibliothek

Die Sicherheitsanforderungen aus der Sicht der Bibliothek betreffen die *Metadokumente*, die *Dokumentmethoden* und die *INDIGO-Server*.

4.4.1 Metadokumente

In der INDIGO-Infrastruktur tritt die Bibliothek als Vermittler zwischen dem Anwender auf der einen Seite sowie dem Autor und dem Produzenten auf der anderen Seite auf. Die Bibliothek als Betreiber der speicherseitigen Ausführungs-Server gilt für den Anwender als eine Vertrauensinstanz; indem der Anwender seine Dokumente von einem Ausführungs-Server dieser Bibliothek bezieht, kann er diesen Dokumenten vertrauen. Um dieses Vertrauen der Anwender nicht zu enttäuschen, muß sie sich um einen Bestand an Dokumenten bemühen, die den folgenden Sicherheitsanforderungen entsprechen:

- Authentizität Integrität,
- Unabstreitbarkeit und Verbindlichkeit.

Authentizität und Integrität Die Bibliothek muß sich über die Herkunft der Dokumente, die sie verwaltet, sicher sein. Zum einen, weil die Anwender ihre Dokumente direkt von der Bibliothek beziehen und dadurch dieser Instanz vertrauen, zum anderen, weil die Bibliothek als Betreiber der speicherseitigen Ausführung-Server durch die Ausführung der entsprechenden Dokumentmethoden direkt auf die Authentizität der Dokumente angewiesen ist. Die Ausführung der Methoden eines Dokuments, dessen Herkunft nicht sicher bestimmt werden kann, stellt ein Sicherheitsrisiko dar.

Die gleichen Bedenken gibt es auch in Verbindung mit der Unversehrtheit der Dokumente. Ein Dokument kann beispielsweise eindeutig von einem vertrauenswürdigen Autor stammen, aber auf den Transportwegen könnten Teile des Metadokuments – besonders der Methodenzuordnungs-Abschnitt des Dokuments – von einem Angreifer manipuliert werden. Solchen Problemen kann man mit Integritätsprüfungen begegnen. Die Bibliothek muß jederzeit in der Lage sein, die Richtigkeit des vom Autor erzeugten Dokumentes festzustellen.

Unabstreitbarkeit und Verbindlichkeit Für eine Bibliothek als Anbieter muß nachweisbar sein, daß ein Dokument, das sie anbietet, auch von einem bestimmten Autor geschrieben wurde. Es sollte somit ebenfalls gewährleistet sein, daß der Autor die von ihm gesendeten Daten nicht abstreiten kann. Sonst könnte jeder Verfasser die von ihm erstellten früheren Dokumente abstreiten.

4.4.2 Dokumentmethoden

Die Bibliothek ist – wie breits oben erwähnt – als Betreiber der speicherseitigen Ausführungs-Server und somit indirekt auch als Ausführer der Dokumentmethoden von deren Verhalten betroffen. Die Dokumentmethoden befinden sich ursprünglich auf einem

Dokumentmethoden-Server. Der INDIGO-Server holt sie sich bei Bedarf von dem Dokumentmethoden-Server und speichert sie lokal unter dem Dokumentverzeichnis des jeweiligen Dokuments. Die Probleme, die die Bibliothek und die Anwender in dieser Infrastruktur mit diesen Dokumentmethoden haben, tauchen oft auch bei den anderen Systemen auf, die ebenfalls mobile Programmteile verwenden. Die Sicherheitsanforderungen an die Dokumentmethoden sind:

- Authentizität,
- Integrität,
- Verfügbarkeit.

Authentizität Das größte Problem, das bei der Ausführung der mobilen Codes auftritt, ist die Frage der Authentizität dieser Codes. Dabei stellt sich die Frage: Welche Dokumentmethoden dürfen ausgeführt werden? Welche Methoden sind vertrauenswürdig? Durch die Kenntnis über Identität und Herkunft der Methoden entsteht eine Vertrauensbasis für die Ausführung dieser Methoden. Erst die Authentizität der Herkunft schafft das nötige Vertrauen, um einer Methode zu erlauben, auf Ressourcen des Systems zuzugreifen. Die Authentizität der Dokumentmethoden ist deshalb so wichtig, weil die Authentizität des gesamten Dokuments sowohl von der Authentizität des Dokumentinhalts als auch von der Authentizität der Dokumentmethoden abhängt.

Die Probleme, die durch die Ausführung bösartiger Methoden entstehen können, beschränken sich nicht nur auf das dazugehörige Dokument. Bösartige Methoden könnten z. B. Daten des Host-Rechners, andere Dokumente oder sogar andere Dokumentmethoden ausspionieren. Sie könnten andere Dokumentmethoden angreifen und bei der Arbeit stören oder sogar den INDIGO-Server zum Absturz bringen.

Diese Gefahren und Angriffe sollten von den anderen Komponenten – wie z. B. dem INDIGO-Server – abgewehrt werden. Die Abwehr aller möglichen Angriffe ist aber nicht durchsetzbar. Bei Bibliotheken, in denen man auf einen höheren Sicherheitsstandard setzt, sollte man nur vertrauenswürdigen Methoden, deren Ursprung und Identität bekannt sind, erlauben, von dem INDIGO-Server geladen und ausgeführt zu werden.

Integrität Die Bibliothek hat außerdem ein Interesse daran, daß die Dokumentenmethoden weder auf den Dokumentenmethoden-Server noch auf den Transportwegen von einem Angreifer manipuliert werden. Es darf bei Manipulationsversuchen und bei technischen Fehlern nicht möglich sein, Adressen oder Nutzdaten zu verändern.

Verfügbarkeit Für die Ausführung der verschiedenen Operationen auf die Dokumente benötigt die Bibliothek die Verfügbarkeit der dazugehörigen Methoden bzw. die Dokumentmethoden-Server, die diese beherbergen.

4.4.3 INDIGO-Server

Die INDIGO-Ausführungs-Server, die in dieser Infrastruktur zum Einsatz kommen, werden in anwenderseitige und speicherseitige Server unterteilt. Die Anforderungen, die die

Bibliothek an die INDIGO-Server stellt, betreffen (zum größten Teil) nur die speicherseitigen Ausführungs-Server. Diese INDIGO-Server sind für die Verwaltung der Dokumente und die Ausführung der Dokumentmethoden zuständig. Sie müssen gewisse Werkzeuge zur sicheren Speicherung und Bedienung der Dokumente und Methoden bieten. Außerdem müssen sie darauf achten, daß bei der Ausführung der Dokumentmethoden keine Güter verletzt werden können.

Ein speicherseitiger INDIGO-Server ist außerdem in der Lage, mit einem anderen INDIGO-Server (meistens einem anwenderseitigen Ausführungs-Server) zu kommunizieren, um z. B. diesem Server ein Metadokument zu schicken und für dieses Dokument wiederum eine Dokumentmethode aufzurufen. Die Sicherheitsanforderungen der Bibliothek an den INDIGO-Server sind:

- Vertraulichkeit,
- Authentizität und Integrität,
- Zugriffskontrolle,
- Verfügbarkeit und Robustheit.

Vertraulichkeit Über die Kommunikationskanäle tritt der INDIGO-Server mit den anderen INDIGO-Servern, Dokumentmethoden-Servern und User-Clients in Verbindung. Diese Kommunikation könnte mitgeschnitten werden. Falls nötig, sollte diese Kommunikation so vertraulich ablaufen, daß der Angreifer die ausspionierten Daten nicht verwenden kann.

Dies gilt ebenfalls für die gespeicherten Dokumente und Methoden. Falls erforderlich, sollten diese Daten auf eine gesicherte Weise auf dem Server gespeichert sein, um die Daten vor den Angriffen des Host-Rechners und böswilligen Dokumentmethoden zu schützen.

Authentizität und Integrität Der speicherseitige Server sollte Mechanismen bieten, die die Identität der Anwender vor dem Zugriff auf seine Ressourcen eindeutig feststellen können. Zu diesem Zweck sollte er ebenfalls gewisse Werkzeuge bereitstellen, die die Identifikationsmerkmale der Anwender (wie beispielsweise Paßwörter oder Zertifikate) sicher verwalten. Vor der Aufnahme eines neuen Dokuments sollte der speicherseitige Server ebenfalls mit Hilfe dieser Mechanismen die Identität des Autors prüfen und, nachdem dieser als vertrauenswürdig erkannt wurde, dessen Dokument in die Bibliothek aufnehmen.

Manchmal genügt es nicht, daß die Anwender sich authentisieren. Die Anwender – genauer gesagt deren Clients und anwenderseitige Ausführungs-Server – wollen in manchen Fällen, daß die speicherseitigen INDIGO-Server sich ebenfalls bei ihnen eindeutig identifizieren. Ein speicherseitiger INDIGO-Server soll sich stets eindeutig als Server und außerdem als Teil eines speicherseitigen Server-Verbunds, die die Bibliothek darstellt, identifizieren können.

Eine andere Aufgabe des Servers ist – wie bereits in den Abschnitten 4.4.1 und 4.4.2 erwähnt – die Feststellung der Authentizität und Integrität der Metadokumente und der Dokumentmethoden. Der Server sollte beispielsweise die Ausführung von unbekannten Dokumentmethoden (also die Methoden, die nicht vom Autor des Dokuments autorisiert wurden) ablehnen oder zumindest vor ihrer Ausführung warnen.

Zugriffskontrolle Der INDIGO-Server sollte den Dokumentmethoden keinen Zugriff auf die lokal gespeicherten Daten bieten. Die Dokumentmethoden sollten keine Möglichkeit haben, auf dem Host-Rechner irgendwelche Dateien zu lesen oder sogar zu manipulieren. Das Starten von Programmen auf dem Rechner sollte ihnen ebenfalls nicht gestattet sein.

Der INDIGO-Server sollte den Methoden auch keinen Zugriff auf die fremden (auf dem INDIGO-Server gespeicherten) Metadokumente oder Dokumentmethoden erlauben. Die Kontrolle des Zugriffs auf die Netzwerk-Ressourcen – wie zum Beispiel die Sockets – ist ebenfalls eine der Aufgaben des INDIGO-Servers.

Auch bei der Speicherung der Dokumente entstehen Sicherheitsrisiken. Ein böswilliger Autor oder Anwender könnte über einen Client den Speicherplatz des INDIGO-Servers (genauer gesagt den Speicherplatz des Host-Rechners, auf dem der Server läuft) zum Überlaufen bringen, indem er (mit Hilfe des `POST`-Befehls) den Server mit irgendwelchen Dokumenten bombardiert. In diesem Fall stellt sich die Frage nach der Autorität. Wem sollte man die Möglichkeit bieten, seine Dokumente bei dem Server zu speichern und zu archivieren?

Man kann einer bestimmten Gruppe besondere *Rechte* bezüglich ihres Umgangs mit den Dokumenten oder anderen Ressourcen einräumen. Diese autorisierte Gruppe dürfte sogar die Dokumente modifizieren. Die Frage, ob es überhaupt privilegierte Anwender geben und wie weit ihre Befugnis gehen soll, ist eine Frage der Sicherheitsrichtlinien von Systemen.

Bei der Betrachtung der Abbildung 4.1 sieht man eine andere sicherheitsrelevante Schwachstelle dieser Infrastruktur: Über einen Client erhält ein speicherseitiger Ausführungs-Server die Aufforderung, ein Dokument zu einem anwenderseitigen Ausführungs-Server zu schicken, wobei anschließend nach dem Transport des Dokuments bei diesem anwenderseitigen Server eine Dokumentmethode ausgeführt wird. Der Anwender hinter diesem Client muß bei dieser Infrastruktur nicht zwingend derselbe Anwender sein, der den anwenderseitigen Ausführungs-Server betreibt. Dieses Verhalten der Infrastruktur wird unter dem Begriff der *indirekten Autorisierung* zusammengefaßt, da sich der Client indirekt über den speicherseitigen Ausführungs-Server bei dem anwenderseitigen Ausführungs-Server identifizieren sollte, bevor er bei diesem Server das Zugriffsrecht zum Ausführen einer Methode auf ein übermitteltes Dokument erhält. Diese Art von Autorisation setzt aber eine sichere *indirekte Authentisierung* voraus.

Eine ungeschützte indirekte Autorisierung – wie es bei dieser Infrastruktur der Fall ist – birgt aber gewisse Sicherheitsrisken. Wie der Abbildung 4.2 zu entnehmen ist, kann beispielsweise ein Angreifer sich diese Eigenschaft der Infrastruktur zunutze machen, um mit Hilfe von mehreren speicherseitigen Ausführungs-Servern einen bestimmten Ausführungs-Server mit Dokumenten – zumindest mit Anfragen – zu bombardieren. Die verwendeten speicherseitigen Ausführungs-Server sollten daher so gestaltet sein, daß sie nicht für diese Art von Angriffen – den sogenannten *distributed-denial-of-service-Angriffen* – mißbraucht werden können.

Verfügbarkeit und Robustheit Eine der Forderungen, die die Bibliothek allgemein an die Ausführungs-Server stellt, ist die Forderung nach deren Robustheit und Absturzsicherheit. Der INDIGO-Server darf nicht bei jeder Überlastung abstürzen. Er sollte den Platzverbrauch der Dokumentmethoden kontrollieren und die Anzahl der Anwender bei dem Server beschränken.

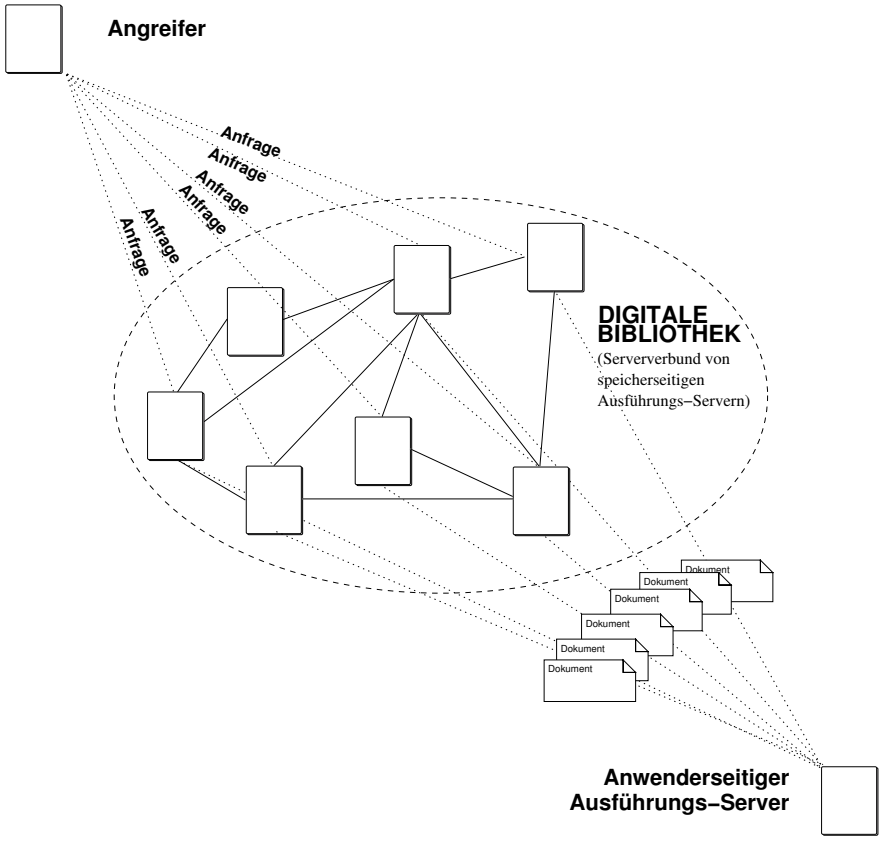

Abbildung 4.2 Beispiel für den Mißbrauch der indirekten Autorisierung

Beim INDIGO-Server ist genauso wie beim Dokumentmethoden-Server die Verfügbarkeit des Servers abhängig vom Host-Rechner, auf dem der INDIGO-Server läuft. Falls man auf die Verfügbarkeit des Servers Wert legt, sollte man bei der Wahl des Host-Rechners ebenfalls auf seine Verfügbarkeit achten.

4.5 Sicherheitsanforderungen aus Sicht des Anwenders

Die Sicherheitsinteressen des Anwenders stimmen in vielen Bereichen mit den Interessen der Bibliothek überein. Die Gemeinsamkeit der Interessen hat ihre Wurzel insbesondere bei dem verwendeten INDIGO-Ausführungs-Server, der von den beiden Akteuren gleichzeitig verwendet wird. Trotzdem gibt es Bereiche, die vom Anwender bezüglich der Sicherheit eher als Schwerpunkte angesehen werden. Aus diesem Grund werden in diesem Abschnitt nur die Anforderungen des Anwenders beschrieben, die sich von den Anforderungen der Bibliothek unterscheiden[5].

[5]Da die Anforderungen der Anwender bezüglich der Dokumentmethoden mit den Anforderungen der Bibliothek fast übereinstimmen, werden diese hier nicht erwähnt.

4.5.1 Metadokumente

Bei proprietären Dokumenten hat der autorisierte Anwender genauso wie der Autor ein Interesse daran, daß die Dokumente auf den Transportwegen und lokal – bei der Bibliothek und bei ihm selbst – vertraulich behandelt werden.

Damit sich der Anwender über die Herkunft und die Unversehrtheit eines Dokuments sicher sein kann, sollte dieses Dokument zusätzliche Informationen beinhalten, mit deren Hilfe der Anwender deren Datenauthentizität und Datenintegrität prüfen kann. Diese zusätzlichen Informationen sollten das Dokument so mit dem Autor verbinden, daß er später die Herkunft dieses Dokuments nicht abstreiten kann (Verbindlichkeit).

Genauso wie der Autor oder die Bibliothek ist der Anwender auf die Verfügbarkeit der Dokumente angewiesen, wobei für den Anwender die Verfügbarkeit eines Dokuments von dessen Bibliothek (genauer gesagt von dem speicherseitigen Ausführungs-Server, die dieses Dokument vertreibt) und von dem Dokumentmethoden-Server abhängt, der die dazugehörigen Methoden vertreibt.

4.5.2 INDIGO-Server

Die Sicherheitsinteressen des Anwenders stimmen bezüglich der Vertraulichkeit, Authentizität und Integrität mit den Sicherheitsinteressen der Bibliothek überein. Natürlich richten sich diese Interessen nicht nur an den speicherseitigen Ausführungs-Server, sondern zum größten Teil an den anwenderseitigen Ausführungs-Server.

Da der anwenderseitige Ausführungs-Server primär als eine lokale Anwendung – bei Bedarf auf der Anwenderseite – eingesetzt wird, wird bei diesem Server kein so großer Wert auf dessen Verfügbarkeit gelegt, wie es bei dem speicherseitigen Ausführungs-Server der Fall ist. Trotzdem sollte dieser Ausführungs-Server auch gegenüber den Belastungen robust sein.

Die wichtigsten Anforderungen, die der Anwender an den Ausführungs-Server stellt, betreffen die Überprüfung der Datenintegrität und der Datenauthentizität der Dokumente und der benötigten Dokumentmethoden. Der anwenderseitige Ausführungs-Server darf manipulierte oder unbekannte Dokumentmethoden nicht ausführen. Ferner sollte er den Methoden nur Zugriff auf die dazugehörigen Dokumente erlauben. Der anwenderseitige Ausführungs-Server sollte außerdem die Identität der Anwender und vor allem der Bibliotheken vor deren Zugriff auf seine Ressourcen eindeutig bestimmen und nur denen, die er für vertrauenswürdig hält, den Zugriff gewähren.

4.6 Weitere Sicherheitsanforderungen an die INDIGO-Infrastruktur

Sicherheitsanforderungen aus der Sicht des Netzwerk-Betreibers richten sich nicht explizit an die in der INDIGO-Infrastruktur beteiligten Güter, sondern sind allgemeingültige Anforderungen, die für alle beteiligten Güter dieses Systems gelten. Aus diesem Grund wird bei dieser Arbeit auf die Sicherheitsanforderungen des Netzwerk-Betreibers nur am Rande eingegangen.

Aus dem selben Grund werden bei der vorliegenden Arbeit die in dieser Infrastruktur beteiligten Komponenten – wie beispielsweise der Dokumentmethoden-Server oder der User-Client – nicht als schützenswerte Güter betrachtet. Trotzdem sollte man bei deren Einsatz darauf achten, daß sie gewisse Anforderungen erfüllen.

Verfügbarkeit und Robustheit des Dokumentmethoden-Servers gehören zu den wichtigsten Anforderungen, die man an den Server stellt. Dabei ist auch das Zusammenspiel mit dem Betriebssystem, auf dem der Server läuft, von Bedeutung. Außerdem sollte der Dokumentmethoden-Server auch die Forderung nach Authentizität erfüllen. Manchmal (siehe Abschnitt 5.2.2) ist es notwendig, daß der Dokumentmethoden-Server sich gegenüber den INDIGO-Server eindeutig identifizieren kann. Hierbei sind Mechanismen gefragt, mit deren Hilfe er sich authentifizieren kann.

Die User-Clients werden von den Akteuren (z. B. vom Anwender oder vom Autor) verwendet, um mit dem INDIGO-Server zu kommunizieren. Mit deren Hilfe wird beispielsweise ein Dokument zu einem INDIGO-Server verschickt. Außerdem werden über diese Komponente Anweisungen einem Server übermittelt, die wiederum diesen Server zur Ausführung einer Methode oder einer Basis-Operation bewegen. Diese Anweisungen können sensible Daten wie beispielsweise Paßwörter zur Autorisationszwecke enthalten. Für den Fall, daß die übertragenen Dokumente bzw. die Anweisungen (allgemein die übertragenen Informationen) vertraulich behandelt werden müssen, sollte der Client dazu die entsprechenden Mechanismen bieten, die beispielsweise auf der Protokollebene den gesamten Datenverkehr vor einem Angreifer absichern.

Natürlich sollte man bei der Wahl des Clients darauf achten, daß er auch den Sicherheitsanforderungen wie Integrität und Robustheit genügt. Er sollte außerdem gewisse Sicherheitsprotokolle beherrschen, die die Unabstreitbarkeit und die Verbindlichkeit der Transaktionen garantieren.

Kapitel 5

Sicherheitskonzepte für die INDIGO-Infrastruktur

In Kapitel 4 wurde eine Sicherheitsanalyse der INDIGO-Infrastruktur durchgeführt. In diesem Kapitel werden, basierend auf dieser Analyse, Sicherheitskonzepte für die INDIGO-Infrastruktur entwickelt.

Bei der Entwicklung der Sicherheitskonzepte werden zunächst Sicherheitsmaßnahmen vorgeschlagen, mit denen die im letzten Kapitel analysierten Sicherheitsanforderungen erfüllt werden können. Zu diesem Zweck werden in den ersten drei Abschnitten unterschiedliche Regeln und Maßnahmen zum Schutz der wichtigsten schützenswerten Güter, also Metadokumente, Dokumentmethoden und INDIGO-Server, beschrieben. Die Tabelle 5.1 bietet eine Übersicht der Sicherheitsanforderungen, auf denen sich in diesem Kapitel das Augenmerk bei den jeweiligen schützenswerten Gütern richtet. Auf den Schutz der Basis-Infrastruktur wird nur am Rande im Zusammenhang mit dem Schutz des INDIGO-Servers eingegangen. In diesen drei Abschnitten werden jeweils die Vor- und Nachteile der Maßnahmen in dem jeweiligen Kontext erörtert. Hierbei werden auch die Annahmen und Einschränkungen, die bei der Implementierung der Infrastruktur vorausgesetzt werden, erläutert. Diese Annahmen fließen automatisch in die Betrachtung der Sicherheitsmaßnahmen und Sicherheitsvorkehrungen ein.

Aus der Menge der diskutierten Sicherheitsmaßnahmen werden anschließend einige Verfahren zur Realisierung ausgewählt. Diese Maßnahmen, die die Sicherheitsrichtlinie der INDIGO-Infrastruktur darstellen, werden im letzten Abschnitt dieses Kapitels beschrieben. Die Konkretisierung dieser ausgewählten Maßnahmen erfolgt anschließend im nächsten Kapitel 6.

In diesem Kapitel werden die im Abschnitt 4.6 beschriebenen Sicherheitsanforderungen nicht behandelt. Dies bedeutet konkret, daß einige Komponenten, wie der Dokumentmethoden-Server oder der HTTP-fähige Client, nicht neu implementiert werden. Es gibt schon viele Produkte, die die im Abschnitt 4.6 geforderten Sicherheitsanforderungen erfüllen. Hier ist als Beispiel für einen sicheren Client der *Mozilla Web-Browser* oder als ein sicherer Server der *Apache* zu erwähnen, den man als sicheren Dokumentmethoden-Server verwenden kann. Bei diesen beiden Produkten kommt noch hinzu, daß die Quellcodes dieser Produkte frei zugänglich (*Open-Source*) sind, was das Vertrauen in diese Produkte zusätzlich steigert.

Schützenswerte Güter	Sicherheitsanforderungen
Metadokumente	Vertraulichkeit, Authentizität, Integrität, Verbindlichkeit
Dokumentmethoden	Vertraulichkeit, Authentizität, Integrität
INDIGO-Server	Vertraulichkeit, Authentizität, Integrität, Verbindlichkeit, Verfügbarkeit, Zugriffskontrolle

Tabelle 5.1 Übersicht über die Vorgehensweise beim Schutz der Güter

5.1 Sicherheitsmaßnahmen zum Schutz der Metadokumente

Die drei Bereiche eines Dokuments können unterschiedlich behandelt werden. Die Methodenzuordnung eines Dokuments ist immer statisch. Dieser Bereich wird vom Autor erstellt und darf nicht geändert werden.

Die Attribute eines Dokuments haben ebenfalls einen statischen Teil, der vom Autor erstellt wird. Sie dürfen aber auch einen Bereich beinhalten, der dynamisch ist. Dieser Bereich beinhaltet beispielsweise die vom Autor oder von einer Vertrauensinstanz erstellte digitale Signatur des Dokumentinhalts. Bei der Vertrauensinstanz handelt es sich beispielsweise um Verlage, Bibliotheken, Universitäten oder wissenschaftliche Einrichtungen, die durch ihre Unterschrift dieses Dokument eindeutig dem Autor des Dokuments zuweisen.

Der Dokumentinhalt kann entweder statisch oder dynamisch sein. Die meisten Sicherheitsmaßnahmen ergeben in dieser Infrastruktur aber nur einen Sinn, wenn man von einem statischen Inhalt ausgeht. Aus diesem Grund werden in diesem Kapitel meistens Dokumente mit einem statischen Dokumentinhalt betrachtet.

5.1.1 Schutz der Vertraulichkeit bei den Metadokumenten

Beim Transport der proprietären Dokumente über einen unsicheren Kanal kann die Vertraulichkeit mit Hilfe von zwei unterschiedlichen Verfahren durchgesetzt werden. Bei der ersten Methode wird nur der *Kanal*, der zum Transport der Dokumente verwendet wird, gesichert. Diese Art der Vertraulichkeit nennt man auch *Server-zu-Server* Vertraulichkeit. Bei dem zweiten Verfahren, der *Ende-zu-Ende* Vertraulichkeit, werden die Daten direkt verschlüsselt und dann verschlüsselt dem Anwender übermittelt.

In diesem Abschnitt wird auch auf die vertrauliche Speicherung und Verwaltung der Dokumente bei einem Ausführungs-Server und deren Schutz vor den böswilligen Dokumentmethoden eingegangen.

Server-zu-Server Vertraulichkeit bei den Metadokumenten: Eine Server-zu-Server Vertraulichkeit kann man sich zwischen allen aktiven Komponenten vorstellen. Zur sicheren Kommunikation zwischen den Clienten, den INDIGO-Servern und den Dokumentmethoden-Servern kann man beispielsweise den Diffie-Hellman-Algorithmus oder den RSA-Algorithmus verwenden. Diese Algorithmen können zum Austausch eines

gemeinsamen Schlüssels genutzt werden. Nach dem sogenannten „Handshake“ besitzen die beiden Kommunikationspartner einen zufällig gewählten Schlüssel (für ein symmetrisches Verschlüsselungsverfahren), mit dem sie dann ihre Kommunikation sicher über das Netz führen können. Diese Verfahren schützen das System gegen einen *passiven Angriff*, wie z.B. das *Lauschen*.

Bei einem *aktiven Angriff*, wie z.B. einer *Verfälschung* oder einer *Maskerade*, bei dem der Angreifer an der Kommunikation zwischen zwei Gesprächspartnern aktiv beteiligt ist, zeigen die oben erwähnten Schlüsselaustausch-Verfahren keine Wirkung. Der gleichzeitige Schutz vor aktiven und passiven Angriffen fordert einen größeren Aufwand als der alleinige Schutz vor passiven Angriffen. Für den Schutz vor aktiven Angriffen ist die Authentifikation durch ein sicheres Schlüsselmanagement erforderlich. Aus Gründen der Skalierbarkeit verwendet man zu diesem Zweck in einem Server-Verbund mit einer unbegrenzten Teilnehmeranzahl Zertifikate. In solchen Fällen besitzen die kommunizierenden Komponenten die öffentlichen Schlüssel der Zertifizierungsstellen. Einen Standard für die Zertifikate stellt die im X.509-Verzeichnisprüfungsdienst (siehe Abschnitt 3.3.3) festgelegte Zertifikatsstruktur dar. Bei dieser Arbeit wird die Existenz einer solchen Zertifizierungs-Infrastruktur vorausgesetzt.

Ein Sicherheitsprotokoll, das die Kommunikation vor einem aktiven Angriff schützt, ist das SSL-Protokoll. Das SSL-Protokoll wurde bereits im Abschnitt 3.3.4 erläutert. SSL verwendet signierte Zertifikate zur Übertragung der öffentlichen Schlüssel. Nach dem Handshake verläuft die Kommunikation zwischen den Instanzen verschlüsselt. Aus diesem Grund werden die Daten beim Transport vertraulich behandelt.

Einer der Vorteile des SSL-Protokolls liegt darin, daß bei dessen Einsatz nicht nur die Dokumentmethoden und Metadokumente verschlüsselt transportiert werden, sondern die gesamte Kommunikation (also auch die Anfragen) über einen sicheren Kanal abläuft. Dieses Verfahren hat außerdem den Vorteil, daß hier auch Dokumente mit einem dynamischen Dokumentinhalt sicher zum Anwender transportiert werden können.

Das SSL-Protokoll ist ein im Internet sehr verbreitetes Protokoll. Die am häufigsten eingesetzten HTTP-Server im Internet – wie beispielsweise Apache – unterstützen ebenfalls dieses Protokoll, so daß diese Server ohne jegliche Änderung als Dokumentmethoden-Server eingesetzt werden können. Dadurch ist ein sicherer Transport der Dokumentmethoden garantiert.

Die Server-zu-Server Vertraulichkeit bietet Schutz nur gegen einen Angriff auf den Transportkanal. Bei diesem Verfahren muß der Autor automatisch den speicherseitigen Ausführungs-Servern, die sein Dokument anbieten, vertrauen. Denn bei diesen Servern liegen die Dokumente unverschlüsselt vor.

Ende-zu-Ende Vertraulichkeit bei den Metadokumenten: Falls der Autor seine proprietären Dokumente nur den autorisierten Anwendern – also auch nicht den Bibliotheken – zugänglich machen will oder gar diese Dokumente auch über nicht vertrauenswürdige oder anonyme Server verbreiten möchte, sollte er ein Verfahren verwenden, das eine Ende-zu-Ende Vertraulichkeit garantiert: Sie stellt sicher, daß nur die End-Akteure – also in unserem Fall nur der Autor und der autorisierte Anwender – auf den Inhalt des proprietären Dokuments zugreifen dürfen.

Eine Ende-zu-Ende Vertraulichkeit kann bei der INDIGO-Infrastruktur auf folgende Weise realisiert werden: Dokumentinhalte werden z. B. direkt vom Autor verschlüsselt und anschließend verschickt; sie werden beim speicherseitigen und anschließend beim anwenderseitigen INDIGO-Server – ohne entschlüsselt zu werden – gespeichert. Das Dokument kann auf dem anwenderseitigen Ausführungs-Server beispielsweise bei seiner Präsentation von einer bestimmten Dokumentmethode entschlüsselt werden, wobei der Anwender das für die Entschlüsselung zutreffende Paßwort eingeben muß. Dieses Verfahren wird eigentlich nur zum Zweck der Autorisierung für die Benutzung eines Dokuments oder für die Ausführung einer Dokumentmethode von einem autorisierten Kreis verwendet. Bei dieser Vorgehensweise wird die Entschlüsselung des Dokuments von den Dokumentmethoden – und somit indirekt vom Anwender – erledigt.

Im Gegensatz zu einem serverbasierten Transport, bei dem der gesamte Dokumentinhalt vollständig zum Anwender geschickt wird, ist beim methodenbasierten Transport ein wichtiger Punkt in bezug auf die Ende-zu-Ende Vertraulichkeit zu beachten: Viele Verschlüsselungsverfahren – genauer gesagt deren Betriebsarten (siehe Seite 30) – unterstützen kein „Streaming“. Hier muß man bei der Wahl des Verschlüsselungsverfahrens darauf achten, daß es sich um einen Online-Algorithmus handelt. Zu diesem Zweck eignen sich beispielsweise die Stromchiffrierungen oder Verschlüsselungsalgorithmen, die zumindest eine stromorientierte Betriebsart bieten [Wobst98].

Die Ende-zu-Ende Vertraulichkeit hat für den Autor den Vorteil, daß er ohne jegliche Abhängigkeit von der INDIGO-Infrastruktur den Verschlüsselungsalgorithmus, der ihm als sicherer erscheint, zur Verschlüsselung seiner Daten verwenden kann. Durch diese Art der Vertraulichkeit hat der Benutzer auf der anderen Seite den Vorteil, daß er Dokumente auch von einer unsicheren und nicht vertrauenswürdigen Bibliothek (speicherseitiger Server) beziehen kann (mehr zu diesem speziellen Fall siehe Anhang B.2).

Die Verfahren, die eine Ende-zu-Ende Vertraulichkeit sichern, weisen aber auch Schwachpunkte auf: Da der Dokumentinhalt vom Autor verschlüsselt wird, kann man bei diesen Methoden keine Dokumente mit einem dynamischen Inhalt zulassen.

Außerdem verschlüsseln diese Verfahren nur den Dokumentinhalt. Hier wäre ein Angreifer sogar durch einen passiven Angriff in der Lage, die Attribute und die Methodenzuordnung des jeweiligen Dokuments herauszufinden. Noch schlimmer wäre es, wenn der Angreifer die Methodenzuordnung manipulieren könnte. Dies würde anschließend dazu führen, daß der Benutzer beim Aufruf der Methoden eine feindliche Dokumentmethode lädt. Diese Methode wäre dann in der Lage, das Paßwort des Benutzers auszuspionieren und dem Angreifer zu schicken. Ein solcher Angriff würde das Ende-zu-Ende Verfahren ad absurdum führen. Um diese Sicherheitslücke zu schließen, muß der Autor die Methodenzuordnung um die digitale Signatur der benötigten Methoden ergänzen bzw. die benötigten Dokumentmethoden direkt signieren. Genauer genommen hat eine Ende-zu-Ende Vertraulichkeit nur Sinn, wenn auch die digitalen Signaturen der Dokumentmethoden vorliegen. Die Sicherstellung der Authentizität von Dokumentmethoden wird im Abschnitt 5.2.2 erläutert.

Viele der bekannten Anwendungen und Infrastrukturen setzten zum vertraulichen Umgang mit ihren digitalen Dokumenten auf die ähnlichen Ende-zu-Ende Verschlüsselungsverfahren, die in diesem Abschnitt beschrieben wurden. Hier sind als Beispiele die *eBooks* von *Adobe*[1] und *Liquid Audio* von *Liquid*[2] zu erwähnen.

[1] Adobe siehe: http://www.adobe.com/
[2] Liquid Audio siehe: http://www.liquidaudio.com/

Adobe hat ein Verfahren zum sicheren Umgang mit elektronischen Büchern, die sogenannten eBooks, entwickelt. Bei diesem Verfahren kann der Herausgeber sein Dokument, das im PDF-Format vorliegt, verschlüsseln (hierbei kommt eine 56-Bit-Verschlüsselung zum Einsatz) und direkt mit der Adresse des Verkäufers im Internet verlinken. Zum Präsentieren dieser Art von Dokumenten braucht der Anwender den Viewer *Acrobat eBook Reader* von Adobe. Falls der Anwender ein auf diese Weise verschlüsseltes Dokument ohne einen Zahlungsbeleg (*voucher*) eröffnet, wird er zur Internet-Adresse des Verkäufers weitergeleitet.

Der hier benötigte Zahlungsbeleg wird vom Verkäufer erstellt und ist eindeutig einem oder mehreren Anwendern – etwa mittels deren CPU-Kennzeichen oder Disk-Kennzeichen – zugeordnet; dabei darf der Anwender (in diesem Fall also der Käufer) selbst bestimmen, welches Kennzeichen er dem Verkäufer senden möchte. Diesen Informationen muß der Verkäufer aber auch zustimmen. Nach der Einigung dieser beiden Geschäftspartner über die benötigten Informationen kommt es zum eigentlichen Kauf, indem der Käufer beispielsweise mit seiner Kreditkarte für das Dokument bezahlt. Der Käufer erhält als Gegenleistung den elektronischen Zahlungbeleg (und falls nötig das verschlüsselte Dokument) vom Verkäufer zugeschickt und kann mit dessen Hilfe das Dokument anschauen. Die eBooks von Adobe unterstützen zusätzlich zur Vertraulichkeit auch digitale Signaturen und digitale Wasserzeichen. Bei dieser Art von Dokumenten kann sogar der Verfasser den Anwendern unterschiedliche Zugriffsrechte gewähren. Er kann beispielsweise festlegen, ob ein Dokument ausgedruckt werden darf oder Teile des Dokuments kopiert, kommentiert oder gar modifiziert werden dürfen[3].

Von Liquid stammt *Liquid Audio*, ein Musik-Distributions-System für die Verbreitung von Audiodaten via Internet. Zum Abspielen der Audio-Daten von Liquid benötigt man den *Liquid Audio-Player* dieser Firma. Die Dateien im Liquid Audio-Format werden mit Hilfe von RSA-Verfahren verschlüsselt. Eine solche Datei kann nur vom Player des Käufers dieser Datei entschlüsselt und abgespielt werden. Bei diesem Format kommt noch hinzu, daß der Umgang der Anwender mit diesen Dateien protokolliert wird. Außerdem kann der Urheber bei diesem Format festlegen, wie oft und wie lange ein Anwender ein Musikstück abspielen darf. Dadurch kann der Urheber bzw. der Eigentümer dieser Dateien durch die entsprechenden Player die Anzahl der möglichen Abspielungen dieser Audio-Dateien oder die Dauer der Abspielgültigkeit von Anwendern bestimmen. Basierend auf diesem Konzept kann anschließend beispielsweise die Höhe der Lizenzgebühr berechnet werden.

Schutz der Metadokumente vor den böswilligen Dokumentmethoden: Für proprietäre Dokumente existieren andere Sicherheitsrisiken, die durch die Ausführung der Dokumentmethoden entstehen können. Die fremden Dokumentmethoden können bei ihrer Ausführung das in sie gesetzte Vertrauen mißbrauchen, um sensitive Daten des Anwenders auszuspionieren und weiterzugeben (solche Programme bezeichnet man auch als „Trojanische Pferde“ [Muss89]). Sie könnten auch andere Dokumente oder Dokumentmethoden

[3]Die Sicherheit der eBooks von Adobe ist sehr umstritten. Die russische Firma *ElcomSoft* verkaufte beispielsweise Programme, mit deren Hilfe sich die Kopier- und Drucksperren von geschützten PDF-Dateien entfernen ließen. Außerdem waren diese Produkte in der Lage, die Verschlüsselung einiger eBooks vollkommen zu überwinden. Näheres zu dem Crack der eBooks von Adobe siehe:
http://www.elcomsoft.com/
http://www.heise.de/newsticker/data/daa-28.06.01-001/
http://www.heise.de/newsticker/data/wst-22.12.00-001/

ausspionieren oder diese sogar zerstören. Um diesen Gefahren vorzubeugen, die von den Dokumentmethoden ausgehen, wurde bei der Implementierung des INDIGO-Servers das „Sandbox-Konzept“ (Sandkasten-Konzept) von Java verwendet. Mit Hilfe dieses Konzepts kann sichergestellt werden, daß die Dokumentmethoden nur auf die Dokumentdateien des entsprechenden Dokuments zugreifen dürfen. Ein lesender bzw. schreibender Zugriff auf andere Dateien wird damit verhindert.

5.1.2 Schutz der Authentizität bei den Metadokumenten

Server-zu-Server Authentizität bei den Metadokumenten: Um die Authentizität der Kommunikationspartner zu sichern, kann man das in dem letzten Abschnitt 5.1.1 vorgeschlagene SSL-Verfahren verwenden. Das SSL-Protokoll unterstützt eine beidseitige Authentifikation; dies bedeutet, daß sich nicht nur der Server, sondern auch der Client beim Handshake authentifizieren kann. Dies ist besonders bei der indirekten Autorisierung (siehe Seite 79) oder beim Versenden neuer Dokumente zu einem speicherseitigen Ausführungs-Server nötig. Beim Handshake authentifizieren sich die Kommunikationspartner durch Zertifikate. Im Laufe dieses Handshakes wird auch ein Sitzungsschlüssel vereinbart. Nach dem Handshake verläuft die Kommunikation mit Hilfe dieses Schlüssels, der nur diesen beiden Partnern bekannt ist.

Um die Authentizität jedes Daten-Pakets noch zusätzlich zu sichern, wird vor dem Verschicken die kryptographische Prüfsumme (MAC) jedes Pakets errechnet und ebenfalls mitgeschickt. Zum Bilden dieser kryptographischen Prüfsumme verwendet man die MAC-Funktion (siehe Abschnitt 3.3.1), wobei diese Funktion den beim Handshake erzeugten Sitzungsschlüssel ebenfalls verwendet.

Hier wird der gesamte Authentisierungsprozeß von dem SSL-Protokoll übernommen, so daß sogar bei einer Verwendung des methodenbasierten Transports eine Authentisierung stattfindet. Dies betrifft ebenfalls die Verwendung der Dokumente mit einem dynamischen Inhalt.

Die Server-zu-Server Authentizität hat in diesem Kontext den Nachteil, daß man nur über die Identität des Kommunikationspartners sicher sein kann. So kann beispielsweise der Anwender sicher sein, daß er seine Dokumente von einer bestimmten Bibliothek, mit der er kommuniziert, bezieht. Bei dieser Art der Authentizität ist es nicht möglich, das Dokument *nachträglich* direkt einem bestimmten Autor zuzuordnen.

Ende-zu-Ende Authentizität bei den Metadokumenten: Um die Herkunft eines Dokuments direkt zu bescheinigen, wird es von seinem Autor signiert. Zu diesem Zweck erzeugt der Autor die *digitale Signatur* des Dokumentinhaltes und der Methodenzuordnung. Hierfür gibt es zwei wichtige Ansätze: den DSS-Ansatz und den RSA-Ansatz. Diese beiden Ansätze wurden bereits im Abschnitt 3.3.2 beschrieben.

Bei diesen Ansätzen berechnet der Autor zuerst den Hashwert des Dokumentinhalts und signiert anschließend diesen Hashwert mit seinem privaten Schlüssel. Zu diesem Zweck verwendet er einen der Hash-Algorithmen MD5 oder SHA-1, kombiniert mit dem asymmetrischen Verschlüsselungsverfahren RSA oder DSA. Die Methodenzuordnung wird auch der gleichen Prozedur unterzogen. Diese beiden Prüfsummen werden anschließend bei den Attributen des Dokuments gespeichert.

Der Empfänger bildet zur Prüfung der Authentizität des Dokumentinhalts zuerst ebenfalls den Hashwert des Inhalts. Anschließend entschlüsselt er mit dem öffentlichen Schlüssel des Autors die digitale Signatur und vergleicht das Ergebnis mit dem vorher errechneten Hashwert des Dokumentinhalts. Genauso verifiziert er die digitale Signatur der Methodenzuordnung. Den öffentlichen Schlüssel extrahiert der Anwender dabei von dem Zertifikat des Autors, das bei den Attributen des Dokuments gespeichert sein darf.

Beim Schutz der Dokumente vor einer Verfälschung geht es nicht nur um den Schutz der Interessen des Autors. Hier stehen ebenfalls die Interessen der Anwender und der Bibliotheken im Vordergrund, die manchmal mit den Interessen der Autoren nicht übereinstimmen. Bei einer Verfälschung geht man in diesem Zusammenhang nicht immer von einem Dritten aus: Auch der wahre Autor kann beispielsweise durch Angriffe seine eigenen Dokumente nachträglich modifizieren. In diesem Zusammenhang kann man als Lösung für solche Probleme den Einsatz der *Zeitstempel* beim Signieren der Dokumente einführen. Eine andere Lösung wäre der Einsatz von *Signaturen der Dritten*. Hierbei handelt es sich um *Vertrauensinstanzen*, wie beispielsweise öffentliche Bibliotheken, wissenschaftliche Einrichtungen oder namhafte Firmen. Dies bedeutet, daß beispielsweise eine Diplomarbeit nicht nur von deren Autor signiert wird, sondern auch von dem Dekanat, von der Universitätsbibliothek oder von dem Professor, der diese Arbeit betreut hat. Diese Lösung ist zusätzlich auch dann sinnvoll, wenn ein Anwender ein Dokument, dessen Autor er nicht kennt, lädt und seine Methoden ausführt. Wenn der Anwender dem Autor nicht vertraut, kann er auch dessen Dokumenten und den entsprechenden Dokumentmethoden nicht vertrauen.

Eine Vertrauensinstanz kann zu diesem Zweck eine digitale Signatur des statischen Bereichs von Attributen erstellen und sie anschließend dem dynamischen Bereich der Attribute anhängen. Hier setzt man voraus, daß Dokumente verwendet werden, deren Attribute einen dynamischen und einen statischen Bereich haben. Man könnte sogar die Metadokumente um einen weiteren Abschnitt ergänzen, der die gesamten digitalen Signaturen der Vertrauensinstanzen beinhaltet.

Es ist sinnvoll, daß hier die digitalen Signaturen der Attribute erstellt werden und nicht die des Dokumentinhalts. Denn dadurch signiert man die Attribute des Dokuments und indirekt auch den Dokumentinhalt und die Methodenzuordnung.

Die in diesem Abschnitt vorgeschlagenen Verfahren bieten eine Ende-zu-Ende Authentizität. Es ist ersichtlich, daß bei einer Ende-zu-Ende Authentizität keine Dokumente mit einem dynamischen Inhalt verwendet werden dürfen. Denn jede Änderung des Dokumentinhalts macht dessen digitale Signatur ungültig. Hier können nur Dokumente mit einem statischen Inhalt verwendet werden.

Bei der Verwendung eines methodenbasierten Transports kann man die hier erwähnten digitalen Signaturen auch nicht mehr verwenden. Für die Berechnung des Hashwerts eines Dokumentinhalts muß das gesamte Datum vorliegen. Dies ist aber ein Widerspruch zur Verwendung eines methodenbasierten Transports.

Eine Ende-zu-Ende Sicherheit bietet in bezug auf Authentizität – genauso wie bei der Vertraulichkeit – den Vorteil, daß man nicht auf die Existenz einer vertrauenswürdigen Bibliothek angewiesen ist. Man könnte seine Dokumente auch von einer unbekannten Bibliothek beziehen; wichtig ist nur, daß man den Instanzen und dem Autor, die das Dokument signiert haben, vertraut. Falls man aber seine Dokumente direkt von einer vertrauenswürdigen Bibliothek beziehen kann, braucht man diese Signaturen nicht. Hier würde eine Server-zu-Server Authentizität ausreichen.

Urheberrechtsschutz bei den Metadokumenten: Die Signatur der Vertrauensinstanzen, die im letzten Abschnitt erläutert wurde, stärkt das Vertrauen der Anwender in die Echtheit der Dokumente. Falls ein Dokument beispielsweise von einem namhaften und vertrauenswürdigen Verlag oder einer wissenschaftlichen Einrichtung signiert ist, kann der Anwender (fast) sicher sein, daß dieses Dokument nicht gefälscht wurde und den Urheberrechten entspricht.

Eine solche digitale Signatur ist trotzdem kein ausreichendes Instrument, um die Verfälschung eines Dokuments zu verhindern oder zumindest bemerkbar zu machen. Ein Angreifer kann dieses signierte fremde Dokument als sein eigenes ausgeben. Dafür muß er nur minimal den Dokumentinhalt manipulieren (z. B. seine Identität einfügen), die Attribute dieses Dokuments ändern und auch eine neue digitale Signatur des Dokumentinhalts erzeugen. Eine digitale Signatur des Autors gibt dem Anwender nur die Sicherheit, daß dieses Dokument von dem angeblichen Autor unterzeichnet wurde. Ob diese Person auch wirklich der tatsächliche Autor des Dokuments ist, kann man anhand einer solchen digitalen Signatur nicht feststellen, da man sie ohne weiteres abtrennen kann.

Damit der Autor nachweisen kann, daß ein bestimmtes Dokument von ihm stammt, gibt es unterschiedliche Verfahren. Eines dieser Verfahren ist das *digitale Wasserzeichen*[4]. Mit dessen Hilfe möchte man die Authentizität der Daten gewährleisten, um die Identität des Eigentümers – in unserem Fall des Autors – zu garantieren, um so beispielsweise *Urheberrechte* durchzusetzen. Dies geschieht, indem spezifische Informationen des Autors direkt in das Dokument *unsichtbar*[5] eingebracht werden.

Ein digitales Wasserzeichen muß einige Voraussetzungen erfüllen: Es muß qualitätserhaltend sein. Dies bedeutet, daß die Qualität des Ursprungsdokuments unter dem Wasserzeichen nicht leiden darf. Das Wasserzeichen muß außerdem von seinem Eigentümer – im Gegensatz zu dem anderen Anwender – eindeutig identifizierbar sein. Es ist somit ein transparentes, nicht wahrnehmbares Muster[6], das in die Daten – meist unter Verwendung eines geheimen Schlüssels – integriert wird. Eine weitere Voraussetzung betrifft dessen Robustheit; ein digitales Wasserzeichen darf weder verändert noch gelöscht werden und muß gegenüber den Skalierungen bzw. Konvertierungen (sogar DAD-Konvertierungen[7]) robust sein. Dies bedeutet beispielsweise bezogen auf Bilder, daß deren digitale Wasserzeichen verlustbehaftete Bildkompressionen – wie JPEG oder PNG – sowie das Ausdrucken und Wiedereinscannen überstehen müssen.

[4] Digitales Wasserzeichen := digital watermarking or copyright marking system [engl.]

[5] Es gibt zwei Arten der digitalen Wasserzeichen: sichtbare und unsichtbare. Bei den Fernsehsendern werden beispielsweise deren Logos in der oberen Ecke ihrer gesendeten Programme als ein sichtbares Wasserzeichen verwendet. Da die sichtbaren Wasserzeichen leicht zu entfernen sind, konzentrieren sich die meisten aktuellen Verfahren auf die Erstellung der unsichtbaren Wasserzeichen.

[6] Digitale Wasserzeichen sind nahe verwandt mit der *Steganographie*: Beide Verfahren verwenden Methoden zum Verstecken der Informationen in Daten. In der Steganographie wird beschrieben, wie eine bestimmte geheime Information – *Klartext* genannt – in einer harmlosen Nachricht – *Cover* genannt – versteckt werden kann. Das Verfahren zur Steganographie nennt man *Stegosystem* und das Ergebnis eines Stegosystems *Stegoobjekt*. Ziel eines Stegosystems ist die Produktion eines Stegoobjekts, das mit dem Cover fast identisch ist. Die Stegosysteme können wie die Kryptosysteme schlüsselabhängig sein.
In [Sta95] wird die Steganographie gar als Überbegriff für das digitale Wasserzeichen definiert. Die beiden Verfahren verfolgen aber unterschiedliche Ziele. Das Ziel der digitalen Wasserzeichen ist der Urheberschutz. Hingegen ist das Ziel der Steganographie nur das Verstecken wichtiger Informationen.

[7] DAD-Konvertierung steht für Digital-Analog-Digital-Konvertierung; dabei wird ein digitales Dokument zuerst zu einem analogen Dokument konvertiert und das Ergebnis anschließend wieder zu einem digitalen Dokument überführt. Man kann beispielsweise ein digitales Bild ausdrucken und anschließend das ausgedruckte Bild mit einem Scanner wieder digitalisieren.

Zusätzlich zu den Urheberinformationen können über die Wasserzeichen auch andere Informationen – wie beispielsweise Informationen über Nutzungsrechte, Besitzer oder Anwender – in ein Dokument eingebracht werden. Solche anwenderbezogene Wasserzeichen nennt man auch *Fingerprint*. Durch diese Fingerprintings-Verfahren erhält jeder Anwender eine Kopie eines Dokuments mit einem anderen Wasserzeichen. Der Sinn dieser speziellen Wasserzeichen liegt darin, daß der Urheber bei jeder Lizenz-Verletzung die Quelle dieser Verletzung leicht ausmachen kann (*Identifikation der Piraten*). Somit dient ein solcher Fingerprint außer der Urheberidentifizierung auch der Kundenidentifizierung.

Das in [Pfitz97] beschriebene Protokoll stellt ein Beispiel für ein digitales Wasserzeichen-Verfahren dar. Die digitalen Wasserzeichen werden häufig bei Bildern, Videos und Audio-Dateien verwendet. Die European Broadcasting Union (EBU) hat beispielsweise erstmals zur Übertragung der Fußballweltmeisterschaft 1998 unsichtbare und robuste Wasserzeichen verwendet. Sie setzte die patentierten Verfahren der in Darmstadt ansässigen Firma *SysCoP*[8] ein. Zur Zeit beschäftigen sich viele Firmen mit der Entwicklung sicherer Verfahren zur Erzeugung von Wasserzeichen. Viele dieser Firmen haben bereits ihre eigenen Produkte entwickelt. Die meisten verwendeten Verfahren sind aber anwendungsspezifisch und bieten teilweise geringe Sicherheitsniveaus. Diese Verfahren sind somit nicht robust genug [Schw01]. Die meisten Wasserzeichen bei Bildern werden beispielsweise durch die geometrischen Verzerrungen ungültig (*StirMark-Angriff*).

Viele der wichtigsten Verfahren, die in letzter Zeit von großen Firmen und Institutionen entwickelt wurden, waren Opfer der Angriffe. Die Angreifer waren sogar in der Lage, die entsprechenden Wasserzeichen – ohne das Ursprungsdokument zu verletzen – zu beschädigen oder gar zu entfernen. In diesem Zusammenhang sind beispielsweise die erfolgreichen Hacker-Angriffe auf die von der *Secure Digital Music Initiative (SDMI)* entwickelten Schutzmechanismen für digitale Musikstücke, auf die Sicherheitsmechanismen des *Verance Watermark* oder auf die *Microsofts Digital Rights Management Version 2 (MS DRM-2)* zu erwähnen[9].

5.1.3 Schutz der Integrität bei den Metadokumenten

Die oben erwähnten Verfahren zum Schutz der Authentizität bieten auch Schutz der Dokumente vor Manipulationen. Die RSA- bzw. die DSA-basierte digitale Signatur verwendet Hash-Algorithmen. Jede Änderung des Dokumentinhalts macht dessen Hashwert und entsprechend dessen digitale Signatur ungültig. Dadurch ist der Benutzer in der Lage, jede Änderung festzustellen. Bei der Verwendung der digitalen Signatur des Autors wird nur der Dokumentinhalt und die Methodenzuordnung berücksichtigt. Eine Manipulation der Attribute könnte dadurch nicht nachgewiesen werden. Falls der Autor sein gesamtes Dokument sichern möchte, sollte er eine digitale Signatur des statischen Bereichs seiner Dokumentattribute dem dynamischen Bereich anhängen. Die Ergänzung der Dokumente um die digitalen Signaturen der Vertrauensinstanzen bieten ebenfalls die Möglichkeit der Integritätsprüfung der Attribute.

Das SSL-Protokoll bietet auch mehrere Möglichkeiten zur Integritätsprüfung der verschickten Daten. Hier ist als Beispiel das verwendete MAC-Verfahren zu erwähnen, das die Mani-

[8] SysCoP siehe: http://syscop.igd.fhg.de/
[9] Näheres zu diesem Thema siehe:
http://www.heise.de/newsticker/data/vza-22.04.01-000/ und
http://www.heise.de/newsticker/data/vza-19.10.01-000/

pulation jedes Daten-Pakets zum Vorschein bringt. Beim SSL-Protokoll wird die Prüfung der Integrität von dem Protokoll selbst übernommen, so daß der Benutzer davon nichts bemerkt. Hier werden außerdem die gesamten Dokumentbereiche, also auch die Attribute und die Methodenzuordnungen, geschützt. Das Protokoll bietet aber nur eine Integritätsprüfung der Daten während der Kommunikation zwischen den zwei Kommunikationspartnern und nicht direkt zwischen dem Autor und dem Anwender.

5.1.4 Schutz der Unabstreitbarkeit und Verbindlichkeit bei den Metadokumenten

Durch die digitale Signatur kann neben der Echtheit (Authentizität) und der Vollständigkeit (Integrität) von digitalen Dokumenten auch die Verbindlichkeit sichergestellt werden. Bei der Verwendung eines signierten Dokuments hat der Anwender die Gewißheit, daß dieses Dokument vom Autor stammt. Dadurch wird jede Distanzierung des Unterzeichners dieses Dokuments unterbunden, denn es kann nur von dieser Person unterzeichnet sein (Hier wird davon ausgegangen, daß der private Schlüssel des Unterzeichners nicht durch einen Angriff ungültig gemacht wurde). Man kann mit Hilfe solcher nicht-abstreitbaren Dokumente verbindliche Telekooperationen entwickeln [Gri94].

Beim Schutz der Integrität und der Verbindlichkeit gilt wie beim Schutz der Authentizität, daß man einen Mißbrauch nicht verhindern, sondern nur entdecken kann. Falls eine Person sich zu Unrecht als Autor eines Dokuments ausgibt, kann man mit Hilfe ihrer Signatur den Mißbrauch nicht feststellen. Man kann jedoch, nachdem die Person des Mißbrauchs überführt ist, eindeutig nachweisen, daß der Mißbrauch von dieser Person ausging. Sie kann den Mißbrauch nicht „erfolgreich“ abstreiten.

5.2 Sicherheitsmaßnahmen zum Schutz der Dokumentmethoden

Die Dokumentmethoden müssen genauso wie die Dokumente vor Angriffen geschützt sein. Da die Dokumentmethoden bei ihrer Ausführung (als aktive Komponente) Dokumente oder sogar andere Dokumentmethoden manipulieren oder ausspionieren können, muß man stets den Wirkungskreis der Methoden eingrenzen.

In diesem Abschnitt werden Maßnahmen zum Schutz der Dokumentmethoden vorgestellt. Außerdem werden hier Verfahren geschildert, die die anderen Komponenten in der Infrastruktur vor den böswilligen Dokumentmethoden schützen.

5.2.1 Schutz der Vertraulichkeit bei den Dokumentmethoden

Um die Dokumentmethoden vertraulich zu behandeln, gibt es verschiedene Ansätze. Man kann beispielsweise die Methoden beim Transport vor einem Dritten schützen, indem man die Dokumente über einen sicheren Kanal verschickt. Zu diesem Zweck muß sowohl der Dokumentmethoden-Server, der diese Methoden zur Verfügung stellt, als auch der Dokument-Server ein verbindungsorientiertes Verfahren – wie beispielsweise das SSL-Protokoll (siehe Abschnitt 3.3.4) – beherrschen.

Die bei dieser Arbeit verwendeten Dokumentmethoden müssen von jedem Anwender verwendet werden können. Hier wird vorausgesetzt, daß die Dokumentmethoden frei zugänglich sind. Aus diesem Grund können in diesem Kontext die Methoden offen über das Netz transportiert werden, so daß eine vertrauliche Behandlung der Methoden überflüssig wird.

Schutz der Dokumentmethoden vor der Analyse: Verglichen mit dem Schutz der Dokumente stellt sich der direkte Schutz der Dokumentmethoden und der darin enthaltenen Algorithmen als sehr viel schwieriger dar. Sobald eine Methode von einem INDIGO-Server ausgeführt werden soll, muß sie spätestens zu diesem Zeitpunkt unverschlüsselt vorliegen. Hier zeigen die im Abschnitt 5.1.1 erwähnten Ende-zu-Ende Verschlüsselungsverfahren, die man bei den Dokumenten einsetzt, keine Wirkung. Die einzigen Methoden, die sich in einem solchen Kontext als einigermaßen effektiv gegen eine Ausspähung erweisen, sind die *Verwürfelungsalgorithmen*.

Ein Verwürfelungsalgorithmus erhält als Eingabe ein Quellprogramm. Er erzeugt anschließend ein semantisch äquivalentes Programm zu seiner Eingabe. Das neu erzeugte Programm hat die Eigenschaft, daß zu dessen Analyse eine längere Zeitspanne benötigt wird als für die Analyse des ursprünglichen Programms. Bei der Analyse eines Programms möchte man beispielsweise die Aufgabe des Programms und dessen Strategie zum Lösen dieser Aufgabe erfahren. Man kann auch gezielt nach bestimmten Daten, wie z. B. Paßwörtern, in dem Programm suchen [Roe97].

Die Aufgabe beim Entwurf eines effektiven Verwürfelungsalgorithmus besteht also darin, einen Algorithmus zu entwickeln, der diese Analysezeit verlängert. Ein Verwürfelungsalgorithmus ist umso besser, je größer diese Zeitspanne ist. Der Verwürfelungsalgorithmus darf die Programmiersprache des Eingabeprogramms nicht ändern. Dies bedeutet, daß (beispielsweise im Gegensatz zu einem Compiler) das Ausgabeprogramm in der gleichen Programmiersprache wie das Eingabeprogramm vorliegen muß.

Bei der INDIGO-Infrastruktur könnte man Verwürfelungsalgorithmen einsetzen, um die Dokumentmethode vor jedem Versenden neu zu verwürfeln. Dieses Verfahren hätte hinsichtlich der Authentifikation der Dokumentmethoden einen großen Nachteil. Wie im nächsten Abschnitt 5.2.2 gezeigt wird, werden zum Nachweis der Authentizität von verwendeten Dokumentmethoden deren digitale Signaturen an die Methodenzuordnung angehängt. Bei der Verwendung der Verwürfelungsverfahren würde der Code eines Programms geändert, was zur Änderung seiner Prüfsumme führen würde. Dies hätte wiederum zur Folge, daß die digitale Signatur dieser Dokumentmethoden ungültig wird.

Beim Einsatz der Verwürfelungsalgorithmen sollte also der Benutzer in dieser Umgebung damit rechnen, daß er eventuell eine bösartige Dokumentmethode anwendet, weil der Anwender keine Möglichkeit der Echtheitsprüfung einer solchen Methode hat. Die Verwendung einer solchen vom Autor nicht signierten Dokumentmethode kann der Anwender höchstens dann riskieren, wenn er seine Dokumentmethoden von einem sicheren und vertrauenswürdigen Server bezieht. Natürlich sollte in einem solchen Kontext der Transportkanal durch geeignete Maßnahmen (wie beispielsweise SSL) gegen Manipulation geschützt sein. Außerdem sollte der Methodenzuordnungsabschnitt des Dokuments vom Autor signiert sein, also zwar nicht jede einzelne Dokumentmethode, in jedem Fall aber deren Adresse (hier wird anstatt einer direkten Authentizität eine Server-zu-Server Authentizität verwendet).

Manche Programmiersprachen geben Programmen die Möglichkeit, ihren eigenen Code zu verändern. Diese Programme nennt man auch *selbstmodifizierend*. Sie können sich also selbst modifizieren. Diese Möglichkeit kann man mit der Idee der Verwürfelungsalgorithmen kombinieren, so daß für die Veränderung des Codes ein Verwürfelungsalgorithmus verwendet wird. Man könnte beispielsweise Algorithmen entwickeln, bei denen Teile des Programms vom Programm selbst ver- bzw. entschlüsselt werden. Ein Beispiel für einen solchen Algorithmus ist in [Auc96] beschrieben. Eine Dokumentmethode könnte dann so programmiert sein, daß sie ihren Code bei jedem Programmaufruf variiert. Dies würde natürlich die Analyse des Programmcodes weiter erschweren. Bei den meisten Programmiersprachen, so etwa auch bei Java, ist die Erzeugung eines selbstmodifizierenden Codes nicht möglich. Außerdem sind die oben erwähnten Verfahren einfach zu neu, um eine genaue Einschätzung über deren Sicherheit abgeben zu können. Diese Verfahren sind bis jetzt nicht hinreichend analysiert worden.

Abgesehen von den Sicherheitsbedenken in bezug auf die Authentizitäts- und Integritätsprüfung der Methoden bei dieser Infrastruktur weisen die Verwürfelungsalgorithmen auch andere Nachteile auf. Die von den Verwürfelungsalgorithmen behandelten Programme sind langsamer als ihre Ursprungsprogramme. Es kommt noch hinzu, daß diese Programme einen größeren Platzbedarf haben. Außerdem bieten diese Verfahren keinen großen Schutz vor der Analyse. Sie versuchen lediglich, den Aufwand für eine Analyse so zu erhöhen, daß eine Analyse keinen Gewinn bringen würde (weil beispielsweise das Programm bereits veraltet ist).

Wegen der oben genannten Gründe verwenden die Methoden, die (zur Zeit) für diese Infrastruktur entwickelt wurden, solche Verfahren wie die Verwürfelung nicht.

5.2.2 Schutz der Authentizität und Integrität bei den Dokumentmethoden

Server-zu-Server Authentizität und Integrität bei den Dokumentmethoden: Die Verwendung der verbindungsorientierten SSL-Verfahren sichert neben der Verbindlichkeit auch die Authentizität und die Integrität der zu transportierenden Daten-Pakete. Dieses Verfahren ist beispielsweise dann sinnvoll, wenn ein Dokument keine Ende-zu-Ende Signatur der verwendeten Methoden oder der Methoden-Produzenten besitzt – wenn also ein Dokument bei der Methodenzuordnung nur die Adresse der jeweiligen Dokumentmethoden beinhaltet. Durch die Verwendung einer Server-zu-Server Authentizität ist bei einem solchen Dokument sichergestellt, daß man die benötigten Methoden auch wirklich von der beabsichtigten Adresse bezieht. Die Vor- und Nachteile dieses Verfahrens wurden bereits im Abschnitt 5.1.2 und 5.1.3 im Zusammenhang mit Metadokumenten erläutert.

Ende-zu-Ende Authentizität und Integrität bei den Dokumentmethoden: Eine Ende-zu-Ende Authentizität der Dokumentmethoden kann bei der INDIGO-Infrastruktur durch diese beiden Methoden erreicht werden:

- Indirekte Ende-zu-Ende Authentizität,
- Direkte Ende-zu-Ende Authentizität.

Bei der *indirekten Authentizität* der Methoden spezifiziert der Autor die Produzenten, denen er vertraut. Dies könnte so funktionieren, daß der Autor die Zertifikate der jeweiligen Produzenten signiert und das Ergebnis in die Attribute des Dokuments einfügt. Die Produzenten müßten entsprechend ihre Methoden vor dem Verschicken signieren. Dieses Verfahren bezeichnet man als *Code Signing*. Es wird von verschiedenen Programmiersprachen wie auch von Java unterstützt. In Java kann man beispielsweise mit dem Hilfsprogramm „jarsigner“ (vgl. A.2) ein Java-Archiv[10] signieren. Ein ähnliches Verfahren zum Code Signing stammt von Microsoft und heißt *Microsoft Authenticode* [MSAT]. Microsoft Authenticode kann zur Signierung und Signaturprüfung von ausführbaren Dateien (exe-Dateien), dynamischen Bibliotheken (dll-Dateien) usw. verwendet werden. Ein anderes Verfahren ist das *XML-Signature-Verfahren*[11], das Werkzeuge zum Integritätsnachweis, Herkunftsnachweis und Identitätsnachweis jeder Art von Daten bereitstellt. Bei diesem Verfahren wird eine solche Signatur in einer XML-Datei untergebracht, die entweder direkt diese Daten oder nur einen Verweis (z. B. ein URL) auf sie beinhaltet.

Diese Verfahren zur indirekten Ende-zu-Ende Authentifizität haben den Vorteil, daß der Benutzer in der Lage ist, immer die aktuelle und verbesserte Version der benötigten Dokumentmethoden zu verwenden. Falls der Produzent neue Versionen für eine solche Methode produziert hat, kann er die alten Methoden einfach durch diese ersetzen, ohne daß der Benutzer von dieser Versionsänderung etwas bemerkt.

Bei einer *direkten Ende-zu-Ende Authentizität* spezifiziert der Autor anstelle der Produzenten direkt die benötigten Dokumentmethoden. Zu diesem Zweck testet er die benötigten Dokumentmethoden vor dem Verschicken – also vor der Veröffentlichung – seines Dokuments. Anschließend fügt er die digitale Signatur dieser Methoden der Methodenzuordnung hinzu oder signiert direkt die benötigten Dokumentmethoden (beipielsweise mit Hilfe von jarsigner). Zum Erstellen der digitalen Signatur der Dokumentmethoden kann der Autor die gleichen Verfahren verwenden, die er auch zur Erstellung der digitalen Signatur des Dokumentinhalts verwendet hat (siehe Abschnitt 5.1.2).

Dieses Verfahren bietet den Vorteil, daß hier der öffentliche Schlüssel des Produzenten (bzw. das Zertifikat des Produzenten) dem Benutzer nicht bekannt sein muß. Es reicht, wenn der Benutzer das Zertifikat des Autors kennt. Auf der anderen Seite hat dieses Verfahren gegenüber der indirekten Ende-zu-Ende Authentizität den Nachteil, daß der Benutzer auf eine bestimmte Version der Methode, die bei der Methodenzuordnung angegeben wurde, festgelegt ist. Denn eine neue Version der Dokumentmethode würde auch eine neue digitale Signatur dieser Methode notwendig machen.

Die hier beschriebenen Verfahren zur direkten und indirekten Ende-zu-Ende Authentizität verwenden digitale Signaturen zur Sicherstellung der Authentizität, die wiederum – wie bereits im Abschnitt 5.1.3 erwähnt – auch die Integrität der übertragenen Daten sichern.

[10] Die Java-Archive, die sogenannten Jar-Archive, werden in Java verwendet, um unterschiedliche Dateien – wie beispielsweise Klassen, Grafiken etc. – in eine komprimierte und kompakte Form zu packen.
[11] Extensible Markup Language (XML) siehe [XML] und [XML1.0] und XML-Signature-Verfahren siehe [XMLSig] und [RFC3275].

5.3 Sicherheitsmaßnahmen zum Schutz der INDIGO-Server

5.3.1 Schutz der Vertraulichkeit bei den INDIGO-Servern

Wie bereits im Abschnitt 5.1.1 erwähnt, müssen beim Transport der sensiblen Dokumente die Transportkanäle mit Hilfe der Server-zu-Server Verfahren gesichert werden. Dazu greift man auf der Server-Seite auf ein verbindungsorientiertes Verfahren, wie beispielsweise das SSL-Protokoll, zurück. Der Bedarf nach einem vertraulichen Kommunikationskanal könnte beispielsweise auch dann bestehen, wenn sich ein Anwender bei einem INDIGO-Server – beispielsweise mittels Benutzername und Paßwort – authentifiziert, um Autorisationsrechte zur Ausführung der Operationen und der Methoden zu erhalten.

Neben dem Schutz der Kommunikationskanäle muß ebenfalls der Server selbst vor Angriffen geschützt werden. Der Schutz der Vertraulichkeit beim INDIGO-Server bezieht sich dabei nicht auf seinen Programm-Code, denn der Quellcode des Programms ist jedem frei zugänglich. Der Schutz der Vertraulichkeit bezieht sich eher auf die gespeicherten Daten des Servers, wie beispielsweise die Liste der zugelassenen Anwender, sowie auf den Schutz der zu verwaltenden Dokumente.

Der Schutz der Daten eines INDIGO-Servers vor dem Host-Rechner, auf dem der Server läuft, ist sehr aufwendig. Es gibt zwar Methoden wie das *Krypto-Filesystem* (siehe Anhang B.1), die einen Schutz vor einem böswilligen Host-Rechner ermöglichen sollen. Diese Methoden bieten aber nur bedingt Sicherheit vor dem Host-Rechner. Aus diesem Grund wird bei dieser Arbeit vorausgesetzt, daß der INDIGO-Server und sein Host-Rechner entweder vollständig vertrauenswürdig oder überhaupt nicht vertrauenswürdig sind. Es kann also der Fall vernachlässigt werden, in dem der INDIGO-Server vertrauenswürdig, der Host-Rechner aber nicht vertrauenswürdig ist; denn dies würde dazu führen, daß auch der Server nicht vertrauenswürdig ist. Die Sicherheit der Daten eines INDIGO-Servers ist indirekt abhängig von der Sicherheit des Host-Rechners.

Im Gegensatz zum Schutz der sensiblen Daten eines Servers vor dem Host-Rechner ist deren Schutz vor den Dokumentmethoden leicht zu realisieren. Dies geschieht über die Einschränkung der Zugriffsrechte der Methoden. Die Methoden dürfen beispielsweise nur auf die Daten eines bestimmten Dokuments zugreifen. Dies wurde bereits im Abschnitt 55 im Zusammenhang mit dem Schutz der Dokumente vor den böswilligen Dokumentmethoden geschildert. Die Durchsetzung und die Überwachung dieser Aufgabe erfolgt durch den INDIGO-Server.

5.3.2 Schutz der Authentizität, Integrität und Verbindlichkeit bei den INDIGO-Servern

Ein INDIGO-Server muß sich – falls nötig – bei einer Kommunikation eindeutig als ein bestimmter INDIGO-Server identifizieren können. Ein anwenderseitiger bzw. speicherseitiger INDIGO-Server sollte sich zusätzlich noch als Mitglied einer Bibliothek bzw. als Teil eines bestimmten Server-Verbunds, der eine Bibliothek darstellt, ausweisen können. Dieser Identitätsnachweis ist im Falle einer Server-zu-Server Authentizität – beispielsweise bei einer Kommunikation mittels SSL-Protokoll – nötig. Die Identifikation des INDIGO-Servers geschieht mit Hilfe eines Zertifikats. In diesem ist der öffentliche Schlüssel des Servers

integriert, wobei dieser vom Betreiber dieses Servers und von der zentralen Verwaltungseinrichtung des Bibliothek-Verbunds signiert ist.

Bei einer Server-zu-Server Authentizität soll der Server ebenfalls in der Lage sein, die Identität seiner Kommunikationspartner zu bestimmen. Beim Einsatz von SSL bedeutet dies, daß beispielsweise auch der Client sich mit einem Zertifikat identifizieren muß. Die Client-Identifikation kann auch (bzw. zusätzlich noch) mit Hilfe von Benutzernamen und Paßwörtern geschehen. Falls man Paßwörter zuläßt, sollte man diese Paßwörter nur auf einer gesicherten Leitung – also verschlüsselt – den Servern übermitteln. Auf der anderen Seite sollten diese Paßwörter bei den Servern auf eine gesicherte Art – beispielsweise im Shadow-Modus – gespeichert werden.

Im Falle einer Ende-zu-Ende Authentizität bei den Metadokumenten kann die Überprüfung der Echtheit und Unversehrtheit ebenfalls vom INDIGO-Server übernommen werden. Zu diesem Zweck muß er die digitale Signatur des Dokumentinhalts, der Dokumentmethodenzuordnung und des statischen Bereichs der Attribute verifizieren. Der Server muß außerdem vor dem Ausführen der Dokumentmethoden deren digitale Signatur, die vom Autor des Dokuments erstellt wurde, überprüfen. Die Integrität der Daten und die Verbindlichkeit der Kommunikation werden bei der Anwendung der hier erwähnten Verfahren ebenfalls gesichert.

5.3.3 Schutz der Verfügbarkeit bei den INDIGO-Servern

Beim INDIGO-Ausführungs-Server stellt sich wie bei jedem Server der Wunsch nach Verfügbarkeit und Robustheit; da der speicherseitige Ausführungs-Server im Gegensatz zu einem anwenderseitigen Server stets erreichbar sein muß, ist dieser Wunsch bei den speicherseitigen Servern größer. Die meisten Angriffe gegen die Informationsdienste richten sich in letzter Zeit gegen diese Anforderung; als Beispiel ist der *denial-of-service-Angriff* zu erwähnen. Eine der möglichen Vorkehrungen gegen solche Angriffe kann die Beschränkung der Anzahl der möglichen parallel aufgebauten Verbindungen zum Server sein. Bezogen auf den INDIGO-Server bedeutet dies, daß nur eine beschränkte Anzahl an Dokument-Methoden und Aufrufen gleichzeitig bearbeitet werden darf. Der INDIGO-Server startet zum Ausführen jedes Aufrufs einen neuen leichtgewichtigen Prozeß (*Thread*). Um die Robustheit des Servers zu erhöhen, muß der INDIGO-Server nur einer bestimmten Anzahl an Threads erlauben, gleichzeitig zu existieren. Sobald die maximale Anzahl an gleichzeitig laufenden Threads erreicht wurde, dürfen keine neuen Verbindungen zu diesem Server aufgebaut werden.

Man kann außerdem jedem Thread nur eine beschränkte Lebensdauer zubilligen (*Timeouts*). Nach dem Ablauf dieser Zeit werden dieser Thread und die dazugehörigen Dokumentmethoden einfach zur Terminierung gezwungen. Man kann diese Ablaufszeit sogar allgemein von den Benutzern und nicht von den Threads abhängig machen, so daß nach dem Ablauf dieser Zeit der Anwender zum erneuten Einloggen gezwungen wird. Es ist auf jeden Fall sinnvoll, daß diese Anzahl der Threads und die entsprechenden Timeouts vom Server-Betreiber über eine Konfigurationsdatei geändert werden können. Der INDIGO-Server-Betreiber sollte ebenfalls die Möglichkeit haben, beim laufenden Betrieb jeden einzelnen Anwender aus dem System zu entfernen, ohne den Server gleichzeitig zur Terminierung zu bringen.

Diese Art der Thread- und Benutzer-Verwaltung stellt keine ausreichende Sicherheitsmaßnahme für die Verfügbarkeit des Servers dar. Durch diese Maßnahmen wird versucht, einem möglichen Absturz des Systems wegen Überlastung zuvorzukommen, was ein Minimum an Verfügbarkeit darstellt. Diese Maßnahmen zielen genauer genommen auf die Robustheit des Servers.

In diesem Zusammenhang sollte man nochmals erwähnen, daß die Verfügbarkeit der Dokumente außer von der Verfügbarkeit der INDIGO-Server auch von der Verfügbarkeit der benötigten Dokumentmethoden abhängt. Aus diesem Grund müssen die Dokumentmethoden und indirekt auch die Dokumentmethoden-Server, die diese Methoden verwalten, stets für die Anwender und die Bibliotheken verfügbar sein. Aus dem selben Grund müssen bei der Archivierung der Dokumente unbedingt auch die benötigten Dokumentmethoden mitgespeichert werden.

5.3.4 Schutz der Zugriffskontrolle bei den INDIGO-Servern

Die Zugriffskontrolle kann je nach Infrastruktur und abhängig vom Einsatzkontext sehr unterschiedlich behandelt werden. Die Zugriffskontrolle unterliegt direkt der Sicherheitspolitik: Ein Anwender möchte beipielsweise auf seine lokal gespeicherten Dokumente auf jegliche Art zugreifen und falls nötig, diese auch modifizieren. Auf der anderen Seite darf in einer öffentlichen Bibliothek mit Sammelauftrag ein Dokument nach seiner Speicherung bei einem speicherseitigen Ausführungs-Server nachträglich weder modifiziert noch gelöscht werden. Aus diesem Grund dürfen die Dokumente, die bei einem solchen Server gespeichert sind, nur einen statischen Inhalt besitzen. Kein Anwender hat ein schreibendes bzw. löschendes Zugriffsrecht auf die Dokumente dieses Servers. Diese Einschränkung des Zugriffsrechts gilt in einem solchen Kontext sogar für den Verfasser des Dokuments.

Im Hinblick auf diese Überlegungen kann man bei der Implementierung das Zugriffsrecht bei der INDIGO-Infrastruktur sehr unterschiedlich auslegen: Es kann beispielsweise eine sehr feine Granulierung gewählt werden, indem man für jedes Dokument einen „Eigentümer“ definiert. Er wäre dann in der Lage, die Zugriffsrechte auf sein Dokument zu bestimmen. Natürlich wäre er ein neuer Akteur in dieser Infrastruktur. Bei dieser Arbeit wird aber vorausgesetzt, daß der Betreiber eines INDIGO-Servers gleichzeitig auch der Eigentümer aller Dokumente ist, die bei ihm auf seinem Server gespeichert sind. Dadurch fallen diese beiden Akteure zusammen, so daß ihre getrennte Behandlung überflüssig wird.

Bei dieser Infrastruktur sollte einer Autorisierung stets eine erfolgreiche Authentisierung vorausgehen. Nach der Authentisierung eines Anwenders beim INDIGO-Server müssen im Zusammenhang mit der Zugriffskontrolle folgende Entscheidungen getroffen werden:

- Welche Dienste (Server-Befehle) des INDIGO-Servers darf dieser Anwender in Anspruch nehmen?
- Auf welche Dokumente darf dieser Anwender (mit Hilfe der Dienste) zugreifen?
- Falls dieser Benutzer auf ein bestimmtes Dokument Zugriff hat: Welche Dokumentmethoden dieses Dokuments darf er aufrufen?
- Auf welche Daten und Ressourcen (und auf welche Art) dürfen diese Dokumentmethoden bei ihrer Ausführung zugreifen? Dürfen diese Methoden beispielsweise ein

Dokument manipulieren bzw. löschen? Welcher Bereich dieses Dokuments darf manipuliert werden?

Zur Behandlung dieser Fragen (und allgemein zur Zugriffssteuerung) kann der Server eine *Capability-Liste* über die Zugriffsrechte der einzelnen Anwender bzw. der Anwendergruppen führen.

Die Entscheidung über die gesamten Zugriffsfragen muß aber nicht zwingend direkt nur vom INDIGO-Server getroffen werden. Man kann eine Aufteilung vornehmen, bei der nur Teile der Zugriffssteuerung vom Server entschieden werden; die Entscheidung über den Rest könnte auf die anderen Komponten übertragen werden: Die Zugriffssteuerung auf den Inhalt der einzelnen Dokumente und deren Dokumentmethoden kann beispielsweise von Dokumenten selbst übernommen werden. So wird sichergestellt, daß Teile der Zugriffssteuerung vom Autor des Dokuments geregelt werden. Eine mögliche Vorgehensweise ist in diesem Kontext der Einsatz von Verfahren, die eine Ende-zu-Ende Vertraulichkeit bieten (siehe Abschnitt 5.1.1). Beim Einsatz eines solchen Verfahrens würde die Bibliothek jedem Anwender den Zugriff auf die entsprechenden Dokumente gewähren. Zum Zugriff auf den Inhalt des Dokuments müßte dem Anwender beispielsweise das richtige Kennwort bekannt sein.

Ein anderer Ansatz, der auch auf Aufteilung der Zugriffssteuerung und die Eingliederung der Autorisationskontrolle beim Autor setzt, wird in [Moe01, Seite 69] beschrieben. Bei diesem Verfahren führt das Dokument eine *dokumentspezifische Autorisierung* des Anwenders zur Operationsausführung durch. Zu diesem Zweck kommt das Zertifikat des Anwenders und die Liste vertrauenswürdiger Zertifizierungsstellen, die vom Autor in das Dokument eingetragen sind, zum Einsatz. Abbildung 5.1 zeigt ein Beispiel für dieses Verfahren.

Bei diesem Beispiel führt der Anwender nach seiner erfolgreichen Authentisierung (mittels seines Zertifikats) beim speicherseitigen Ausführungs-Server auf ein Dokument die Present-Operation aus. Diese führt anschließend die private Operation „`private.authorize`“ mit dem Namen der anderen privaten Operation „`private.safePresent`“ als Argument aus. Die Operation `privat.authorize` hat die Aufgabe, das Zertifikat des Present-Aufrufers – also des Anwenders – mit der Liste der vertrauenswürdigen Zertifizierungsstellen für die Operation `private.safePresent` zu vergleichen. Falls das Zertifikat des Anwenders von einer vertrauenswürdigen Instanz signiert bzw. erstellt wurde, identifiziert `privat.authorize` diesen Anwender als einen autorisierten Anwender und gibt diese Information an die Present-Operation zurück. Diese Operation führt nach dieser erfolgreichen Authentisierung die sichere Präsentiermethode `private.safePresent` auf das gewünschte Dokument aus.

Bei der Verwendung dieses Verfahrens geht der Besitz des Dokuments zum Teil vom Betreiber des INDIGO-Servers auf den Autor über. Eine solche dokumentspezifische Autorisation kann aber korrekt funktionieren, wenn dieser Autorisierungsvorgang – also auch die Authentisierung des Anwenders – auf einem sicheren und dem Autor vertrauenswürdigen Ausführungs-Server vollzogen wird.

Indirekte Autorisierung: Um den Sicherheitsrisiken zu begegnen, die von einer ungesicherten indirekten Autorisierung ausgehen (siehe Abschnitt 4.4.3), kann man unterschiedliche Verfahren einsetzen. Eine mögliche Lösung ist der Einsatz von *Cookies*. Zu diesem Zweck generiert der Ausführungs-Server in seiner Initialisierungsphase einen

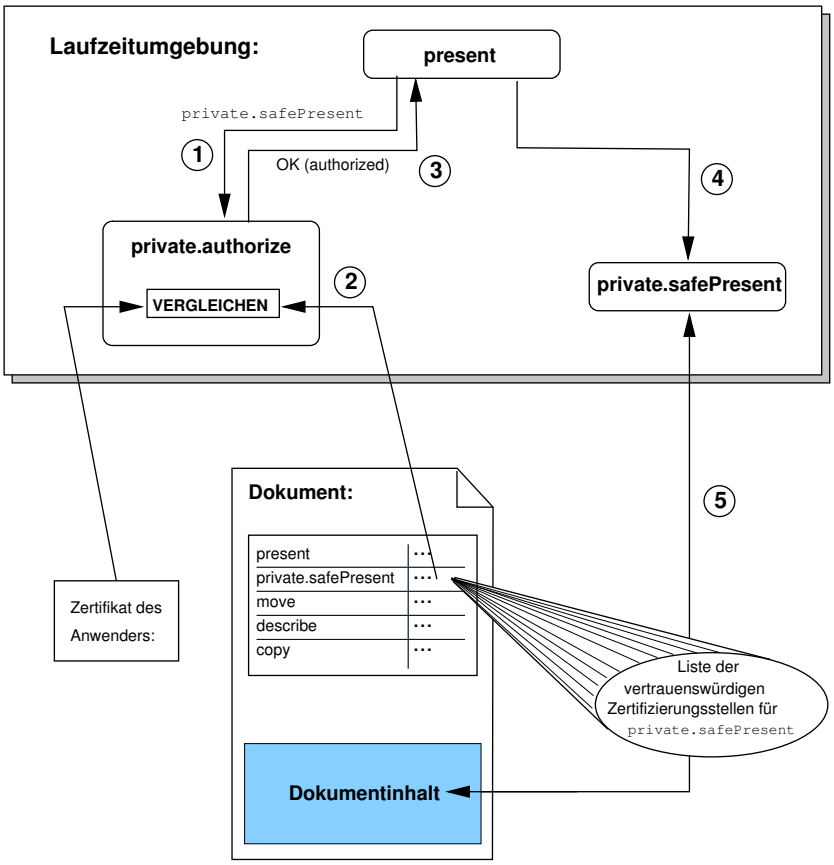

Abbildung 5.1 Dokumentspezifische Autorisation

Schlüssel, das sogenannte Cookie. Der Client würde bei der indirekten Autorisierung dieses Cookie – beispielsweise als ein Argument – an den speicherseitigen Ausführungs-Server schicken. Dieser Server würde anschließend bei der Kommunikation mit dem anwenderseitigen Ausführungs-Server ebenfalls dieses Cookie verwenden, um Zugriffsrechte für das Ausführen von Operationen zu erlangen (siehe Abbildung 5.2). Anfragen, die sich mit dem Cookie nicht identifizieren können, verweigert der anwenderseitige Ausführungs-Server den Zugriff auf seine Ressourcen. Diese Art von Autorisierung lehnt sich an den *X-Authority-Zugriffsmechanismus* von *X-Windows-Systems*[12].

Dieses Verfahren weist aber auch gewisse Schwachstellen auf. Bei ihm werden die Cookies unverschlüsselt zwischen den beteiligten Komponenten transportiert, so daß jeder Angreifer mit einem passiven Angriff in der Lage wäre, dieses Cookie zu ermitteln. Ähnliche Probleme existieren auch bei dem in HTTP/1.0 verwendeten Autorisierungs-Mechanismus, der sogenannten *Basic Authentication* [HTTP]. Bei diesem Verfahren schickt der

[12]Beim X-Windows-System werden die Ressourcen – also Bildschirm, Tastatur und Maus – von einem X-Server verwaltet. Der Zugriff der X-Anwendungen – also die Clients – auf diese Ressourcen werden über verschiedene Schutzmechanismen geregelt. Einer dieser Schutzmechanismen ist der *X-Authority*, auch *MIT-MAGIC-COOKIE-1* genannt. Hierbei wird ein 128 Bit langes Cookie generiert (meistens von *xdm*) und dem X-Server nach dessen Start mitgeteilt. Nach X-Authority muß jeder X-Client bei seiner Startphase den Server dieses Cookie schicken. Der X-Server autorisiert nur solche Anwendungen, die sich genau mit dem bekannten Cookie bei ihm identifizieren.

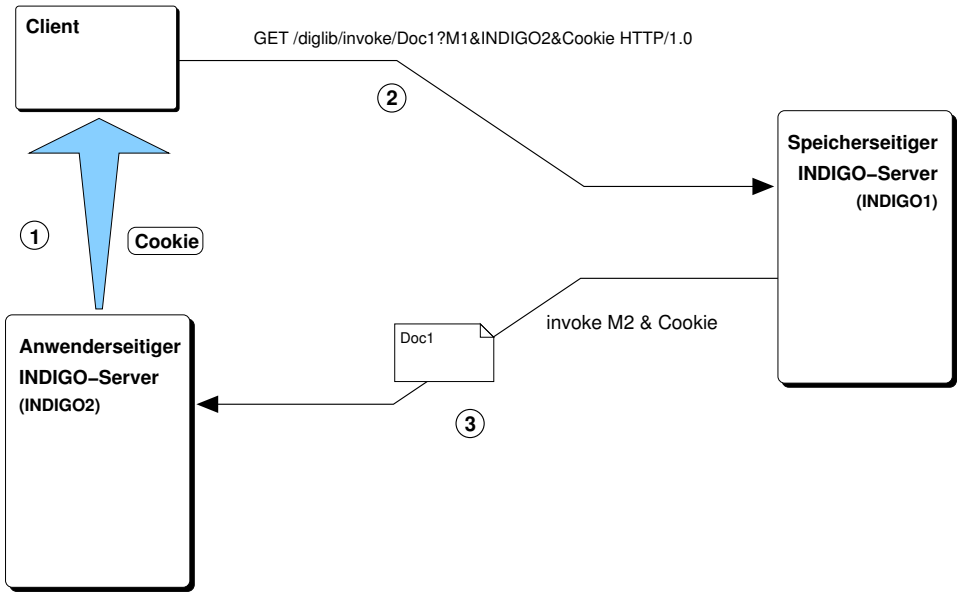

Abbildung 5.2 Indirekte Autorisierung mittels Cookies

Client die zur Autorisierung benötigten Daten unverschlüsselt[13] zum Server. Um diesem Problem zu begegnen, wurde in HTTP/1.1 [HTTP1.1] der Autorisierungs-Mechanismus *Digest Access Authentication* eingeführt[14]. Bei diesem Verfahren kombiniert der Client sein Paßwort mit einer Buchstabenkette (*Nonce*[15]), die er vom Server erhalten hat. Anschließend wendet er auf dieses Kombinat ein Hashverfahren an (meistens MD5) und schickt diesen Hashwert dem Server im Authorization-Header der HTTP-Anfrage. Durch die Verwendung von Digest Access Authentication wird sichergestellt, daß die Paßwörter nicht im Klartext übermittelt werden (siehe [RFC2069, RFC2617]). Ähnlich wie bei der Digest Access Authentication kann man in der INDIGO-Infrastruktur mit den Cookies umgehen, in dem man sie vor deren Transport mit einem Hashverfahren behandelt.

Ein anderes Verfahren zum vertraulichen Transport von Cookies ist der Einsatz von SSL zur Verschlüsselung des gesamten Kommunikations-Kanals. Zu diesem Zweck muß die Kommunikation zwischen dem Client und dem speicherseitigen Server sowie zwischen diesem Server und dem anwenderseitigen Server verschlüsselt ablaufen. Es ist ersichtlich, daß hier nur sichere und vertrauenswürdige speicherseitige Ausführungs-Server betrachtet werden können. Denn bei dieser Methode liegen die Cookies bei dem speicherseitigen Server unverschlüsselt vor.

Ein völlig anderer Ansatz für die Sicherung der indirekten Autorisierung ist der Aufbau eines *verbindlichen Kommunikationskanals* zwischen dem Client und dem anwenderseitigen Ausführungs-Server. Einen solchen Kanal kann man mit Hilfe des SSL-Protokolls wie folgt realisieren (siehe Abbildung 5.3):

[13] Bei Basic Authentication wird an den Benutzernamen ein „:“ und anschließend das Paßwort angehängt. Diese Buchstabenkette wird anschließend nur mit dem Base64-Verfahren eingepackt und unverschlüsselt im Header „Authorization“ der HTTP-Anfrage dem Server übermittelt.

[14] In HTTP/1.1 wird empfohlen, stets die Digest Access Authentication einzusetzen. Trotzdem wird aus Kompatibilitätsgründen mit dem HTTP/1.0 auch die Basic Authentication angeboten.

[15] Nonce ist eine Zahl oder eine Buchstabenkette, die nur einmal verwendet wird. Bei deren Verwendung will man in diesem Kontext den Replay-Attacken vorbeugen.

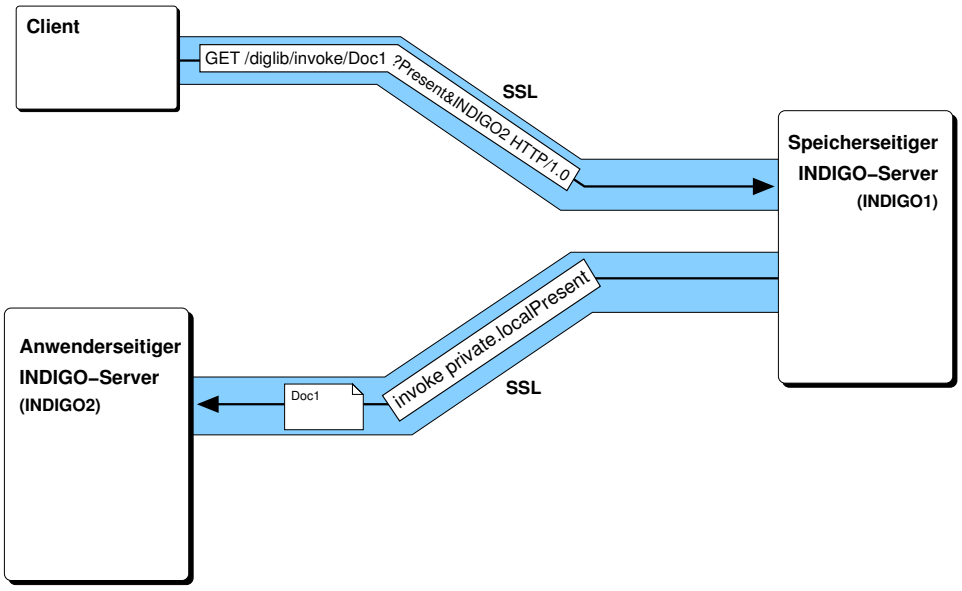

Abbildung 5.3 Indirekte Autorisierung mittels verbindlicher Kommunikationskanäle

1. Der Anwender baut über seinen Client eine Verbindung mittels SSL zu dem speicherseitigen Ausführungs-Server auf. Dabei findet bei der Handshake-Phase eine beidseitige Authentisierung statt. Falls der Anwender sich nicht als *Mitglied* dieses Bibliotheken-Verbunds ausweisen kann, bricht der Server die Verbindung ab.

2. Nach dem Handshake schickt der Client dem Server seine Anfragen, z. B.
   `GET /diglib/invoke/Doc1?Present&INDIGO2 HTTP/1.0`

3. Der speicherseitige Ausführungs-Server führt auf das Dokument `Doc1` die Präsentiermethode mit der Adresse des anwenderseitigen Ausführungs-Servers als Argument aus. Das Ausführen dieser Methode führt wiederum dazu, daß dieses Dokument zu dem anwenderseitigen Ausführungs-Server transportiert wird.

4. Zum Transport des Dokuments baut der speicherseitige Server eine Verbindung zum anwenderseitigen Server auf. Diese Verbindung wird über SSL hergestellt, so daß wieder eine beidseitige Authentisierung der Kommunikationspartner stattfinden kann. Hierbei ist wichtig, daß der anwenderseitige Server seinen Kommunikationspartner eindeutig als Teil des vertrauenswürdigen Bibliotheken-Verbunds identifizieren kann.

5. Nach dem Aufbau des Kommunikationskanals transportiert der speicherseitige Server das Dokument zum anwenderseitigen Server mit der Aufforderung, auf dieses die lokale Präsentiermethode auszuführen.

Bei diesem Ansatz basiert die Sicherheit auf der Verbindlichkeit und Unabstreitbarkeit. Hier kann es zwar zu einem Mißbrauch der indirekten Autorisierung kommen; der Verursacher kann aber eindeutig identifiziert werden, und er könnte sogar seinen Angriff nicht abstreiten. Der Angreifer kann eindeutig identifiziert werden, weil er sich für diesen Angriff bei dem speicherseitigen Server identifizieren muß. Um einen solchen Angriff zu einem späteren Zeitpunkt nachvollziehen zu können, muß der speicherseitige Server stets Protokoll (*Log-Datei*) über die akzeptierte Verbindung führen.

5.4 Sicherheitsrichtlinien für die INDIGO-Infrastruktur

Nach [Gri94] entstehen die Sicherheitsrichtlinien durch die geeignete Zuordnung von Sicherheitsmaßnahmen zu Sicherheitsanforderungen. In diesem Abschnitt erfolgt diese Zuordnung, indem aus der Menge der bisher in diesem Kapitel beschriebenen Maßnahmen einige ausgewählt werden. Hier werden somit die Sicherheitsmaßnahmen beschrieben, die dann bei der modifizierten sicherheitsorientierten INDIGO-Infrastruktur zum Einsatz kommen. Diese Maßnahmen, die die Sicherheitsrichtlinien dieser Arbeit darstellen, stellen die Grundgedanken für die im nächsten Kapitel 6 beschriebenen Implementierungen dar.

Wichtig ist in diesem Kontext, daß von den in diesem Kapitel beschriebenen Sicherheitsanforderungen nur die Vertraulichkeit a priori durchgesetzt werden kann. Bei den anderen Sicherheitsanforderungen ist nur eine a posteriori Nachvollziehbarkeit von Aktionen möglich (siehe [Gri94] Seite 44).

Sicherheitsrichtlinien hinsichtlich der Vertraulichkeit: Die Güter, die bei dieser Infrastruktur vertraulich behandelt werden müssen, sind die Dokumente und die Transportkanäle. Bei der Sicherstellung der Vertraulichkeit zwischen den verschiedenen aktiven Komponenten wird bei der modifizierten Infrastruktur das „SSL-Protokoll“ – ein Server-zu-Server Verfahren – verwendet. Zu diesem Zweck benötigen die Kommunikationspartner Zertifikate, wobei die Existenz einer Zertifizierungsinfrastruktur vorausgesetzt wird. Da bei dieser Infrastruktur die Anwender ihre Dokumente von einem sicheren Server-Verbund (Bibliothek) beziehen, kann man sicher sein, daß die über einen SSL-gesicherten Kanal transportierten Dokumente vertraulich behandelt werden. Ein positiver Nebeneffekt bei der Verwendung des SSL-Protokolls ist, daß zusätzlich zu den Dokumenten auch die gesamte Kommunikation zwischen den Akteuren vor einem passiven und aktiven Angriff gesichert abläuft.

Die Verwendung des SSL-Protokolls schließt aber die Verwendung einer Ende-zu-Ende Vertraulichkeit nicht aus. Die Implementierung einer solchen Vertraulichkeit kann aber vollständig aus dem INDIGO-Server herausgenommen werden. Aus diesem Grund entscheiden die Autoren der Dokumente direkt über den Einsatz dieser Art von Sicherheitsmaßnahmen, so daß deren Realisierung den Autoren bzw. den Dokumentmethoden-Produzenten überlassen wird.

Im Gegensatz zu den Dokumenten ist es in dieser Arbeit nicht erforderlich, die Dokumentmethoden vertraulich zu behandeln. Diese liegen für jeden Anwender zugänglich auf den Dokumentmethoden-Servern. Außerdem werden bei der modifizierten Infrastruktur Verfahren wie die Verwürfelungsalgorithmen, die dem Schutz der Methoden vor einer Analyse dienen sollen, aus Gründen der Inkompatibilität mit den Verfahren zur Authentizitäts- und Integritätssicherung nicht unterstützt.

Sicherheitsrichtlinien hinsichtlich der Authentizität und Integrität: Bei den modifizierten INDIGO-Servern verwenden die Kommunikationspartner beim Verbindungsaufbau mittels SSL-Protokoll zur Authentifikation X.509-konforme Zertifikate. Authentizität und Integrität der einzelnen Transportpakete werden wiederum während der Kommunikation bei SSL durch die Verwendung der kryptographischen Prüfsumme (MAC) sichergestellt.

Da bei dieser Infrastruktur – wie bereits erwähnt – die Dokumente von einem sicheren Server-Verbund (Bibliothek) geholt werden, können die Anwender sicher sein, daß diese Dokumente von dem angegebenen Autor stammen. Um eine spätere Überprüfung der Herkunft und Vollständigkeit der Dokumente zu ermöglichen, sollten diese trotzdem die digitale Signatur ihrer jeweiligen Autoren beinhalten. Eine solche Signatur wird aber bei dieser Arbeit nicht zwingend vorgeschrieben. Aus diesem Grund wird bei dem modifiziert sicherheitsorientierten INDIGO-Server die Verifikation der digitalen Signatur der Dokumente zum Teil aus dem INDIGO-Server ausgegliedert. Der Server wird diesen Verifikationsprozeß nicht direkt durchführen; er wird stattdessen den Dokumentmethoden Schnittstellen (z. B. über Packages) bieten. Über diese Schnittstellen können die Dokumentmethoden die Signatur der entsprechenden Dokumente prüfen. Zur Erzeugung der digitalen Signatur wird der Einsatz einer Kombination von MD5 bzw. SHA-1 mit RSA oder DSA empfohlen.

Der Urheberrechtsschutz bleibt den Autoren überlassen. Es wird ihnen freigestellt, gewisse Mechanismen – wie beispielsweise digitale Wasserzeichen – zu verwenden. Der bei dieser Arbeit modifizierte INDIGO-Server wird aber keine zusätzlichen Werkzeuge zur Durchsetzung der Urherberrechte bieten.

Bei der Implementierung wird ausschließlich von Dokumenten mit einem statischen Inhalt ausgegangen. Dadurch soll sichergestellt sein, daß alle Dokumente auch die digitale Signatur des jeweiligen Autors beinhalten dürfen.

Während eine digitale Signatur bei den Dokumenten nicht zwingend notwendig ist, muß diese bei den Dokumentmethoden vorhanden sein. Bei der vorliegenden Arbeit wird das Verfahren zur direkten Ende-zu-Ende Authentizität (siehe Seite 75) bevorzugt. Bei dem eingesetzten Verfahren handelt es sich um eine Variante der direkten Ende-zu-Ende Authentizität; bei dieser Variante signiert der Autor direkt die benötigten Dokumentmethoden.

Sicherheitsrichtlinien hinsichtlich der Verbindlichkeit: Die Verbindlichkeit wird bei der modifizierten INDIGO-Infrastruktur durch zwei Verfahren sichergestellt: Zum einem wird die Verwendung der Zertifikate und asymmetrischen Kryptowerkzeuge bei der beidseitigen Authentifikation beim SSL-Protokoll sicherstellen, daß ein Kommunikationspartner wirklich mit der von ihm gewünschten Person kommuniziert. Zum anderen wird der Anwender durch die Authentisierung der digitalen Signatur von Dokumenten sicher sein, daß diese Dokumente (bzw. die entsprechenden Dokumentmethoden) auch wirklich vom angegebenen Autor stammen (bzw. gebilligt wurden).

Sicherheitsrichtlinien hinsichtlich der Zugriffskontrolle und Verfügbarkeit: Um eine feingranulierte Zugriffskontrolle bis hin zu den einzelnen Dokumenten zu erreichen, wird die Zugriffssteuerung beim sicherheitsorientierten INDIGO-Server auf unterschiedlichen Ebenen[16] vorgenommen:

- Auf der Protokollebene,
- auf der Serverebene und
- auf der Dokumentebene.

[16] Die Zugriffskontrolle beim modifizierten INDIGO-Server und die einzelnen Zugriffssteuerungsebenen werden detailliert im nächsten Kapitel in Abschnitt 6.2 erläutert.

Nach seiner Authentifizierung wird ein Anwender bei dem modifizierten INDIGO-Server einem bestimmten Anwendertyp zugeordnet. Diese Zuordnung erfolgt auf der Protokollebene. Auf dieser Ebene wird außerdem abhängig vom Anwendertyp entschieden, welche Dienste des Servers der jeweilige Anwender in Anspruch nehmen darf.

Wie bereits im Abschnitt 5.1.1 (siehe Seite 67) beschrieben, dürfen die Dokumentmethoden nur Operationen auf ihre Dokumente ausführen. Auch der Zugriff auf die Ressourcen des INDIGO-Servers und allgemein des Host-Rechners muß kontrolliert ablaufen. Der modifizierte INDIGO-Server wird auf der Serverebene diesen Zugriff der Dokumentmethoden kontrollieren und falls nötig unterbinden. Dabei wird vorausgesetzt, daß die Dokumentmethoden in einer beschränkten Umgebung arbeiten können. Da bei dieser Infrastruktur nur Dokumente mit einem statischen Inhalt verwendet werden, wird in dieser Ebene außerdem darauf geachtet, daß keine Dokumentmethoden zum Einsatz kommen, die den Inhalt manipulieren können.

Von den drei genannten Zugriffssteuerungsebenen betreffen nur die Protokollebene und die Serverebene direkt den INDIGO-Server. Auf der Dokumentebene wird die Zugriffssteuerung von den Dokumenten behandelt. Dadurch wird es möglich sein, mit Hilfe der auf Seite 79 beschriebenen Verfahren beispielsweise dokumentspezifische Autorisation vorzunehmen.

Der modifizierte INDIGO-Server prüft vor dem Ausführen einer Dokumentmethode deren Integrität und Echtheit. Nur wenn diese Integritäts- und Authentizitätsprüfung positiv ausfällt, darf ein autorisierter Anwender diese Methode überhaupt ausführen.

Um den Gefahren zu begegnen, die von einer ungesicherten indirekten Autorisierung ausgehen, setzt die modifizierte Infrastruktur auf den Aufbau eines verbindlichen Kommunikationskanals zwischen den beteiligten Akteuren (siehe Seite 81).

Die Robustheit des INDIGO-Servers wird verstärkt, indem der modifizierte Server nur eine bestimmte Zahl an gleichzeitig laufenden Threads zuläßt. Außerdem wird jedem Thread nur eine begrenzte Lebensdauer zugebilligt.

Kapitel 6

Prototypische Implementierung

Im letzten Kapitel wurden zum Erfüllen der Sicherheitsanforderungen, die an diese Infrastruktur gestellt werden, konkrete Maßnahmen vorgeschlagen. Im Laufe dieser Diplomarbeit wurden einige dieser Maßnahmen und Sicherheitsmechanismen, die die Sicherheitsrichtlinien dieser Infrastruktur darstellen, realisiert. In diesem Kapitel wird auf die Realisierung dieser Maßnahmen eingegangen.

Die Realisierung der Sicherheitsmaßnahmen betrifft die Modifikation und die Erweiterung des INDIGO-Servers und dessen Packages sowie die Implementierung einiger neuer Clients. Zur Demonstration dieser Erweiterungen wurden zusätzlich für die Infrastruktur weitere Dokumente und Dokumentmethoden, die von diesen Dokumenten benötigt werden, erstellt.

Ausgangspunkt der Implementierung war der in der Programmiersprache Java erstellte Prototyp der INDIGO-Infrastruktur, die im Rahmen der Dissertation [Moe01] realisiert wurde. Bei der Implementierung kamen *Java Software Development Kit (JDK)* in Version 1.4 und *OpenSSL* in Version 0.96 zum Einsatz.

6.1 Erweiterungen beim Ausführungs-Server INDIGO

Die wichtigsten sicherheitsrelevanten Modifikationen beim INDIGO-Server kann man wie folgt zusammenfassen:

- Der INDIGO-Server wurde verändert, so daß er über gesicherte Kanäle mit seinen Kommunikationspartnern kommunizieren kann (HTTPS über SSL bzw. TLS).
- Der INDIGO-Server kann je nach Konfiguration auch über nicht gesicherte Kanäle erreicht werden (HTTP).
- Der Server kann sich mittels eines Zertifikats eindeutig identifizieren.
- Er ist außerdem in der Lage, (optional) eine Client-Authentisierung durchzuführen.
- Die Konfiguration des Servers wurde um neue Konfigurationsdienste und Variablen erweitert.

- Der Server kann die benötigten Dokumentmethoden auch von einem HTTPS-Server beziehen.
- Er kann außerdem signierte Dokumentmethoden vor ihrer Ausführung verifizieren.
- Der INDIGO-Server verfügt jetzt über einen feingranulierten konfigurierbaren Zugriffsmechanismus, über den er – basierend auf der Identität der Anwender – den Zugriff auf seine Ressourcen steuern kann.

In diesem Abschnitt werden diese Erweiterungen – mit Ausnahme der letzten beiden Punkte – erläutert. Da die Implementierung der Zugriffssteuerung beim INDIGO-Server sehr umfangreich ausfiel, erfolgt ihre Beschreibung im nächsten Abschnitt 6.2.

6.1.1 Implementierung sicherer Kommunikationskanäle

Die Kommunikation zwischen den einzelnen aktiven Komponenten verlief bei der ursprünglichen INDIGO-Infrastruktur über die normalen *TCP-Sockets*. Über diese Kanäle läuft die Kommunikation aber vollkommen ungeschützt. Um diese Kommunikationskanäle sicherer – in bezug auf die Vertraulichkeit, Integrität, Authentizität und Verbindlichkeit – zu gestalten, wurden die einfachen Sockets durch die mittels SSL-Protokoll gesicherten Sockets ersetzt. Dabei kam die *Java Secure Socket Extension (JSSE)* – die SSL-Implementierung der Firma *Sun Microsystems* – zum Einsatz, die in Java SDK ab der Version 1.4 enthalten ist.

6.1.1.1 INDIGO-spezifische Socket-Klassen

Um die Arbeit mit den Sockets so einfach wie möglich zu gestalten, wurden die zwei neuen Socket-Klassen „`IndigoSSLServerSocket`“ und „`IndigoSSLSocket`“ entwickelt. Die Klasse `IndigoSSLServerSocket` liefert Objekte vom Typ `ServerSocket`, die von den Serverseitigen Komponenten benutzt werden, während die Klasse `IndigoSSLSocket` Objekte vom Typ `Socket` liefert. Auf einem `ServerSocket`-Objekt wartet eine Server-seitige Komponente auf eine Verbindung. Ein `Socket`-Objekt wird hingegen von einer Client-seitigen Komponente verwendet, um mit einer auf einem Server-Socket wartenden Anwendung in Verbindung zu treten.

Der große Vorteil dieser beiden Klassen liegt darin, daß bei ihnen während der Instanziierung die Konfigurationsvariable `indigo.secureExtension` des Servers (über die statische Methode `IndigoConfig.secureExtension()`) ausgewertet wird. Abhängig vom Wert dieser Variable liefern die beiden Klassen entweder SSL-basierte oder normale TCP-Sockets. Dies bedeutet im konkreten Fall für den Server, daß er abhängig von dieser Variable entweder über einen normalen Socket (im Normalmodus) oder über einen SSL-Socket (im Sicherheitsmodus) erreichbar ist. Ein weiterer Vorteil dieser beiden Klassen ist, daß die Objekte dieser Klassen bei ihrer Instanziierung auf eine Instanz der Klasse „`IndigoSSLContext`“ zugreifen (mehr dazu siehe Abschnitt 6.1.1.2).

Die Instanziierung eines `ServerSocket`-Objekts kann über die Klasse `IndigoSSLServerSocket` wie folgt durchgeführt werden:

```
IndigoSSLServerSocket iSSS   = new IndigoSSLServerSocket();
ServerSocket          server = iSSS.createServerSocket(443);
```

bzw.

```
IndigoSSLServerSocket iSSS   = new IndigoSSLServerSocket();
SSLServerSocket       server = iSSS.createServerSocket(443);
```

falls die Konfigurationsvariable `indigo.secureExtension` den Wert „`true`“ besitzt[1]. Somit erhält man über die Klasse `IndigoSSLServerSocket` den Zugriff auf ein `ServerSocket`- bzw. `SSLServerSocket`-Objekt, das den Port 443[2] verwendet. Eine noch elegantere Art für die Erzeugung eines solchen `ServerSocket`-Objekts ist die Verwendung der statischen Methode der Klasse `IndigoSSLServerSocket`:

```
ServerSocket    server = IndigoSSLServerSocket.getServerSocket(443);
```

bzw.

```
SSLServerSocket server = IndigoSSLServerSocket.getServerSocket(443);
```

Über dieses `ServerSocket`- bzw. `SSLServerSocket`-Objekt kann man anschließend über die `accept()`-Methode auf eine Verbindung warten:

```
Socket serverClient    = server.accept();
```

Dabei ist das im Erfolgsfall von der `accept()`-Methode gelieferte `Socket`-Objekt ein `SSLSocket`-Objekt, falls das dazugehörige Objekt (also in diesem Fall „`server`“) ein `SSLServerSocket`-Objekt ist:

```
SSLSocket serverClient = server.accept();
```

Auf der Client-Seite könnte man die `IndigoSSLSocket`-Klasse verwenden, um `Socket`- bzw. `SSLSocket`-Objekte zu erzeugen, die mit einem Server verbunden sind:

```
IndigoSSLSocket iSS = new IndigoSSLSocket();
Socket  client      = iSS.createServerSocket("www.indigo.de", 443);
```

bzw. mit Hilfe der statischen Methoden:

```
Socket client       = IndigoSSLSocket.IndigoSSLSocket("www.indigo.de",443);
```

[1] Da die Klasse `SSLServerSocket` von der Klasse `ServerSocket` abgeleitet ist, kann man dem mittels der Methode `createServerSocket()` erzeugten Objekt immer eine ServerSocket-Referenz zuweisen (also wie das erste Beispiel). Später im Programmcode kann man dann je nach Bedarf mittels *Cast-Operator* den Typ dieses Objekts in `SSLServerSocket` umwandeln.

[2] Man kann hier jede beliebige Portnummer verwenden. Wichtig ist zu erwähnen, daß „433“ der *wellknown-Port* für das HTTPS darstellt.

Die hier erhaltenen `Socket`-Objekte können – wie bei der Klasse `IndigoSSLServerSocket` – eine Instanz der Klasse `SSLSocket` sein, wenn der Server im Sicherheitsmodus läuft.

Ein weiterer wichtiger Punkt im Zusammenhang mit den in diesem Abschnitt erwähnten Socket-Klassen betrifft die Konfigurationsvariable „`indigo.needClientAuth`“. Über diese Variable kann der Server-Betreiber festlegen, ob sich ein Client bei seinem Verbindungsaufbau authentifizieren muß bzw. soll oder gar eine Authentifikation überhaupt nicht nötig wäre. Das Ergebnis der Auswertung dieser Variable wird in der Klasse `IndigoSSLServerSocket` jedem neu erzeugten `SSLServerSocket`-Objekt mitgeteilt[3]. Diese Variable hat eine weitgehende Wirkung auf die Zugriffmechanismen des Servers und wird auch als *Sicherheitsmodus-Variable* bezeichnet (siehe Abschnitt 6.1.2 und 6.2).

6.1.1.2 INDIGO-spezifische SSLContext-Klasse

Ein signifikantes Merkmal der beiden Klassen `IndigoSSLServerSocket` und `IndigoSSLSocket` ist – wie bereits erwähnt –, daß die Objekte dieser Klassen bei ihrer Instanziierung auf eine Instanz der Klasse „`IndigoSSLContext`“ zugreifen. Diese Klasse, die ebenfalls im Laufe dieser Arbeit entwickelt wurde, spielt in bezug auf die implementierten Sicherheitsmechanismen eine sehr wichtige Rolle.

Die Hauptaufgabe der Klasse `IndigoSSLContext` ist die Interpretation und Zurverfügungstellung des *Server-Keystores* und des *Server-Truststores* für andere Klassen. Der Keystore beinhaltet den privaten und den öffentlichen Schlüssel eines Anwenders, wobei der öffentliche Schlüssel auch signiert sein darf (beispielsweise von einer Zertifizierungsstelle). Der Truststore beinhaltet dagegen die Liste und den öffentlichen Schlüssel – in Form von Zertifikaten – der vertrauenswürdigen Anwender und Zertifizierungsstellen (mehr zur Erzeugung dieser Trust- bzw. Keystores siehe [Lipp00] und [JSSE]).

Ein Objekt der Klasse `IndigoSSLContext` erzeugt bei seiner Instanziierung ein Objekt der Klasse „`SSLContext`“. Die Klasse `SSLContext` ist die Kernklasse bei der JSSE-Implementierung. Sie repräsentiert ein bestimmtes Protokoll, wie beispielsweise SSL oder TLS. Mit Hilfe der `SSLContext`-Klasse kann man `SSLServerSocketFactory`- bzw. `SSLSocketFactory`-Objekte erzeugen, die wiederum verwendet werden, um `SSLServerSocket`- bzw. `SSLSocket`-Objekte zu generieren[4].

Die Klasse `IndigoSSLContext` wertet während der Initialisierung des `SSLContext`-Objekts mehrere Konfigurationsvariablen aus und gibt diese an dieses Objekt weiter. Die wichtigsten dieser Variablen betreffen das Protokoll, das vom Server bei einem gesicherten Kanal verwendet werden muß, sowie den zu verwendenden Keystore und Truststore. Somit wird für jeden Server ein spezielles systemweites `SSLContext`-Objekt erzeugt, das von den unterschiedlichen Komponenten dieses Servers – wie beispielsweise von dem Server selbst, von der Base-Package und von der NetSSL-Package – verwendet werden kann. Auf dieses Server-spezifische `SSLContext`-Objekt kann über die `IndigoSSLContext`-Klasse wie folgt zugegriffen werden:

[3] Dies geschieht über die statische Methode „`IndigoConfig.clientAuth(SSLServerSocket socket)`“.
[4] Dieselben `SSLServerSocketFactory`- bzw. `SSLSocketFactory`-Objekte kommen auch in der `IndigoSSLServerSocket`- und `IndigoSSLSocket`-Klasse beim Erzeugen von `SSLServerSocket`- bzw. `SSLSocket`-Objekte zum Einsatz.

```
IndigoSSLContext iContext = new IndigoSSLContext();
SSLContext sslContext     = iContext.getSSLContext();
```

Die `IndigoSSLContext`-Klasse bietet ebenfalls über die Methode „`getSSLServerSocketFactory()`“ bzw. „`getSSLSocketFactory()`“ eine Schnittstelle zum Zugriff auf die Server-spezifischen `SSLServerSocketFactory`- bzw. `SSLSocketFactory`-Objekte. Zu diesem Zweck bietet diese Klasse zusätzlich noch zwei statische Methoden, wobei dadurch die Instanziierung eines `IndigoSSLContext`-Objekts entfällt:

```
SSLServerSocketFactory serverSF =
                              IndigoSSLContext.getStaticSSLServerSocketFactory();
```

bzw.

```
SSLSocketFactory       clientSF =
                              IndigoSSLContext.getStaticSSLSocketFactory();
```

Diese statische Methode wird beispielsweise bei der Klasse `MetaDocumentStore` verwendet. Die Methode kommt dann zum Einsatz, wenn der INDIGO-Server eine Dokumentmethode von einem HTTPS-Server beziehen muß:

```
URLConnection aConnection;
URL           location;

aConnection = location.openConnection();
((HttpsURLConnection) aConnection).
               setSSLSocketFactory(IndigoSSLContext.getStaticSSLSocketFactory());
```

In diesem Kapitel wird zwar immer von SSL-Sockets gesprochen. Trotzdem bedeutet dies nicht unbedingt, daß zur Kommunikationssicherung auch tatsächlich das SSL-Protokoll eingesetzt wird. Hier kann auch ein „verwandtes“ Protokoll – wie beispielsweise das TLS-Protokoll – eingesetzt werden. Das gewünschte Protokoll wird über die Konfigurationsvariable „`indigo.httpsProtocols`“ festgelegt. Dieses wird anschließend bei der Instanziierung eines `SSLServerSocket`- bzw. `SSLSocket`-Objekts über die Klasse `IndigoSSLContext` berücksichtigt. Eine Liste der unterstützten Protokolle findet sich im Abschnitt 6.1.2.2.

6.1.2 Konfigurationsvariablen

Im Laufe der Implementierung wurde der Server um neue Konfigurationsvariablen erweitert. In diesem Abschnitt werden die wichtigsten Konfigurationsvariablen des Servers näher beschrieben.

Die Konfigurationsvariablen werden dem Server über eine Konfigurationsdatei mitgeteilt. Diese können jedoch beim Starten des Servers über die „-D“-Option des Java-Interpreters überschrieben werden; beispielsweise kann durch den Aufruf

```
java -Dindigo.port="443" Indigo
```

die Konfigurationsvariable `indigo.port` auf 443 gesetzt werden; dies führt dazu, daß der INDIGO-Server auf der Port-Adresse 443 läuft.

Die Konfigurationsvariablen werden beim INDIGO-Server von der Klasse `IndigoConfig` ausgewertet. Für die Bearbeitung der neuen Variable mußte diese Klasse auch modifiziert werden. Diese Klasse wurde außerdem um weitere Methoden ergänzt, die den anderen Klassen des Servers den Zugriff auf diese Variablen ermöglichen.

Die Konfigurationsvariablen des INDIGO-Servers können in vier Kategorien unterteilt werden:

- Allgemeine Konfigurationsvariablen,
- Allgemeine sicherheitsrelevante Konfigurationsvariablen,
- Truststore- und Keystore-Konfigurationsvariablen,
- Konfigurationsvariablen für die Zugriffssteuerung.

Die allgemeinen sicherheitsrelevanten Konfigurationsvariablen, die Truststore- und Keystore-Konfigurationsvariablen und die Konfigurationsvariablen für die Zugriffssteuerung bilden zusammen die *sicherheitsrelevanten Konfigurationsvariablen* des Servers.

6.1.2.1 Allgemeine Konfigurationsvariablen

Die *allgemeinen Konfigurationsvariablen* des Servers sind in der Tabelle 6.1 aufgezählt.

Konfigurationsvariable	**Belegung**
`indigo.port`	*Integer*
`indigo.documentdir`	*FileURI*
`indigo.serverdir`	*FileURI*
`indigo.timeout`	*Integer*
`indigo.verboseMode`	$\{false, true\}$

Tabelle 6.1 Allgemeine Konfigurationsvariablen

Über die Variable `indigo.port` legt der INDIGO-Server-Betreiber die Port-Adresse des Servers fest, unter welcher der Server zu erreichen ist. Das Verzeichnis des Servers bestimmt der Betreiber über die `indigo.serverdir`-Variable und das Verzeichnis für die lokal gespeicherten Dokumente und ihre Methoden über die `indigo.documentdir`-Variable.

Falls die `indigo.verboseMode`-Variable beim Programmstart des Servers den Wert *true* besitzt, erhält man während des Programmverlaufs Ausgaben (auch Fehlermeldungen) über den Programmzustand. Über die `indigo.timeout`-Variable wird das Timeout des Servers festgelegt[5].

[5] Eine genauere Beschreibung der Variablen `indigo.documentdir`, `indigo.serverdir` und `indigo.timeout` findet man in [Moe01].

6.1.2.2 Allgemeine sicherheitsrelevante Konfigurationsvariablen

Die *allgemeinen sicherheitsrelevanten Konfigurationsvariablen* des Servers und ihre möglichen Belegungen sind in der Tabelle 6.2 zusammengefaßt:

Konfigurationsvariable	Belegung
`indigo.secureExtension`	$\{false, true\}$
`indigo.needMethodVerify`	$\{false, true\}$
`indigo.httpsProtocols`	$\{SSL, SSLv2, SSLv3, TLS, TLSv1\}$

Tabelle 6.2 Allgemeine sicherheitsrelevante Konfigurationsvariablen

Über die Variable `indigo.secureExtension` kann der Betreiber des INDIGO-Servers bestimmen, ob der Server im *Normalmodus* oder im *Sicherheitsmodus* laufen soll. Falls diese Variable wahr (`true`) ist, läuft der Server im Sicherheitsmodus. In diesem Modus benutzt der Server SSL-Sockets anstelle von normalen TCP-Sockets. Dadurch ist der Server auch über eine HTTPS-Adresse erreichbar (siehe Abschnitt 6.1.1.1). Falls diese Variable aber auf `false` gesetzt wird, ist der Server über eine HTTP-Adresse zu erreichen. Außerdem verlieren die folgenden sicherheitsrelevanten Variablen ihre Bedeutung, denn dadurch werden die wichtigsten Sicherheitsvorkehrungen ausgeschaltet. Dies führt beispielsweise automatisch dazu, daß bei den Dokumentmethoden keine *Verifizierung* mehr stattfindet. Außerdem werden weder der Keystore noch der Truststore verwendet. Hinzu kommt, daß auch die Zugriffssteuerung des Servers ausgeschaltet wird. Dadurch wird jeder Anwender in die Lage versetzt, alle Dienste des Servers über einen nicht gesicherten Kanal in Anspruch zu nehmen.

Für die Verifizierung der Dokumentmethoden ist die Konfigurationsvariable `indigo.needMethodVerify` zuständig (zur Verifikation der Methoden siehe Abschnitt 6.2.7). Falls diese Variable den Wert `true` besitzt, werden alle Dokumentmethoden verifiziert, bevor sie ausgeführt werden. Falls eine Methode bei ihrer Verifikation durchfällt, wird sie nicht mehr ausgeführt. Diese Eigenschaft des Servers wird aber abgeschaltet, sobald die Variable `indigo.needMethodVerify` auf `false` gesetzt wird.

Über die Variable `indigo.httpsProtocols` kann man das Protokoll festlegen, das im Sicherheitsmodus zur Kommunikation verwendet wird. Die hier erwähnten Protokolle sind genau die Protokolle die auch von dem JSSE-Paket unterstützt werden. Dies bedeutet auch, daß falls bei den zukünftigen Versionen von JSSE weitere Protokolle unterstützt werden, diese auch automatisch von dem INDIGO-Server verwendet werden können.

6.1.2.3 Truststore- und Keystore-Konfigurationsvariablen

Die *Truststore- und Keystore-Konfigurationsvariablen* werden hauptsächlich von der Klasse `IndigoSSLContext` verwendet. Die Konfigurationsvariablen bezüglich des Keystore und des Truststore des Servers sind in Tabelle 6.3 zu finden. Über die Konfigurationsvariablen `indigo.keyStore` und `indigo.trustStore` kann der Server-Betreiber dem Server seine individuelle Keystore-Datei bzw. Truststore-Datei übergeben. Die `indigo.keyStoreType` und die `indigo.trustStoreType`-Variable beschreiben den Typ der entsprechenden Stores, während `indigo.keyManagerFactoryAlgorithm` bzw. `indigo.trustManagerFactoryAlgorithm` deren Algorithmen beschreiben.

Konfigurationsvariable	Belegung
`indigo.keyStore`	*FileURI*
`indigo.keyStorePassword`	*String*
`indigo.keyStorePrivateKeyPassword`	*String*
`indigo.keyStoreType`	$\{JKS\}$
`indigo.keyManagerFactoryAlgorithm`	$\{SunX509\}$
`indigo.trustStore`	*FileURI*
`indigo.trustStorePassword`	*String*
`indigo.trustStoreType`	$\{JKS\}$
`indigo.trustManagerFactoryAlgorithm`	$\{SunX509\}$

Tabelle 6.3 Truststore- und Keystore-Konfigurationsvariablen

Da der private Schlüssel in der Keystore-Datei verschlüsselt aufbewahrt wird, braucht der Server den Schlüssel, um mit diesem den privaten Schlüssel zu entschlüsseln. Der benötigte Schlüssel wird über die Konfigurationsvariable `indigo.keyStorePrivateKeyPassword` dem Server überreicht. Die anderen beiden Schlüssel `indigo.keyStorePassword` und `indigo.trustStorePassword` werden hauptsächlich gebraucht, um die Integrität der beiden Stores zu prüfen[6]. Die in diesem Abschnitt erwähnten Variablen orientieren sich stark an denen des JSSE-Pakets. Eine detaillierte Beschreibung für die möglichen Belegungen dieser Variablen finden sich aus diesem Grund in [JSSE].

6.1.2.4 Konfigurationsvariablen für die Zugriffssteuerung

In diesem Abschnitt wird hauptsächlich die Syntax der Variablen zur Zugriffssteuerung erläutert. Eine nähere Beschreibung der eigentlichen Zugriffssteuerung des Servers folgt anschließend im Abschnitt 6.2. Tabelle 6.4 faßt die Konfigurationsvariablen zur Zugriffssteuerung des INDIGO-Servers zusammen.

Konfigurationsvariable	Belegung
`indigo.superUserName`	*String*
`indigo.superUserPassword`	*String*
`indigo.needClientAuth`	$\{false, maybe, true\}$
`indigo.trueAccesscontrolllist`	*TRIPEL*
`indigo.maybeAccesscontrolllist`	*TRIPEL*
`indigo.falseAccesscontrolllist`	*TRIPEL*

Tabelle 6.4 Konfigurationsvariablen für die Zugriffssteuerung

Wenn man auf der Serverseite den Client authentisieren will, muß man die Konfigurationsvariable `indigo.needClientAuth` auf `true` setzen. In diesem Betriebsmodus, dem *True-Sicherheitsmodus*, wird jede Verbindung, bei der keine beidseitige Authentifizierung durchgeführt wird, von der Serverseite abgebrochen[7]. In diesem Modus *muß* sich der Cli-

[6] Dieser Schlüssel kommt meistens zum Einsatz, wenn man in den Schlüsselring (Store) einen neuen Schlüssel einfügen oder irgendwelche Schlüssel aus dem Schlüsselring entfernen möchte. Durch diesen Schlüssel soll eine fremde Person daran gehindert werden, diesen Schlüsselring zu manipulieren (*Schutz der Authentizität und Integrität des Schlüsselrings*).

[7] Falls diese Variable auf `true` gesetzt ist, ruft die statische Methode `clientAuth(SSLServerSocket sslSS)` der `IndigoConfig`-Klasse intern für das `SSLServerSocket`-Objekt die Methode `setNeedClientAuth(true)`.

ent authentifizieren. Falls aber keine Client-Authentifizierung nötig ist, kann man diese Variable auf `false` setzten (*False-Sicherheitsmodus*).

Man kann den Server in dem dritten Betriebsmodus – dem *Maybe-Sicherheitsmodus* – betreiben, indem man die `indigo.needClientAuth`-Variable auf `maybe` setzt[8]. In diesem Modus *möchte* der Server, daß sich der Client authentifiziert. Falls der Client dazu nicht in Lage ist, wird die Verbindung trotzdem nicht unterbrochen. Der Server kann somit den Clients, die sich authentifizieren können, mehr Zugriffsrechte zuteilen, während die Clients, die sich nicht authentifizieren, trotzdem einen minimalen Zugriff auf gewisse Dienste des Servers erhalten.

Die Konfigurationsvariablen `indigo.trueAccesscontrolllist`, `indigo.maybeAccesscontrolllist` und `indigo.falseAccesscontrolllist` stellen die *Zugriffskontrollisten* für die unterschiedlichen Sicherheitsmodi des Servers dar. Die Wahl einer dieser Listen als Zugriffskontrolliste des Servers wird wiederum über die Wahl des Sicherheitsmodus festgelegt. Dies bedeutet: Wenn die `indigo.needClientAuth`-Variable den Wert `true` bzw. `false` oder `maybe` hat, wählt der Server automatisch den Wert von `indigo.trueAccesscontrolllist` bzw. `indigo.falseAccesscontrolllist` oder `indigo.maybeAccesscontrolllist` als seine aktuelle Zugriffskontrolliste aus.

Die Zugriffskontrollisten stellen eine Art *Capability-Liste* für die Server-Befehle dar. Der Wert einer solchen Liste ist ein $TRIPEL$ (vgl. Tabelle 6.4). Die Syntax des $TRIPEL$ lautet:

$$TRIPEL \quad := \quad \underbrace{(X_{1,1}, \cdots, X_{1,l})}_{Superuser} \underbrace{(X_{2,1}, \cdots, X_{2,m})}_{Group} \underbrace{(X_{3,1}, \cdots, X_{3,n})}_{Others}$$

wobei für $l, m, n, X_{1,j}, X_{2,j}$ und $X_{3,j}$ gilt:

$$\begin{aligned} l, m, n &\geq 1 \quad \text{und} \\ X_{i,j} &\in \{all, quit, deliver, invoke, attributes, documents, packages, none\} \end{aligned}$$

Die Zugriffskontrollisten bestehen somit aus drei Klammerpaaren. Jedes dieser Klammerpaare bezieht sich auf einen bestimmten *Anwendertyp*. Die Anwender werden vom INDIGO-Server in drei Typen eingeteilt[9]:

- **Superuser** Zu diesem Anwendertyp gehören Anwender, die sich mittels gültiger Zertifikate und des Superuser-spezifischen Benutzernamens und Paßworts authentifizieren.
- **Group** Die Anwender, die sich lediglich mit ihren Zertifikaten authentisieren können, gehören zu diesem Typ.
- **Others** Der Anwendertyp „Others“ bezieht sich hingegen auf die anonymen Anwender.

[8] Falls diese Variable auf `maybe` gesetzt ist, wird für das `SSLServerSocket`-Objekt die Methode `setWantClientAuth(true)` aufgerufen.

[9] Diese Einteilung ist sehr stark von dem Sicherheitsmodus des Servers abhängig. Eine detaillierte Beschreibung dieser Einteilung erfolgt in Abschnitt 6.2.6, 6.2.5 und 6.2.4.

In dem ersten Klammerpaar (von links) der Zugriffskontrollisten stehen die Operationen, die ein Superuser ausführen darf, während das mittlere Klammerpaar die Operation beinhaltet, die ein Anwender vom Typ Group ausführen darf. Alle anderen Anwender – *Others* – dürfen hingegen die Operationen ausführen, die im rechten Klammerpaar stehen. Ein Beispiel für eine solche Zugriffskontrolliste könnte wie folgt aussehen:

$$\texttt{indigo.maybeAccesscontrolllist} = \underbrace{(all)}_{SU}\,\underbrace{(invoke, attributes, documents)}_{Group}\,\underbrace{(documents)}_{Others}$$

Wenn ein Anwender sich als Superuser authentifizieren muß, braucht er dafür die Superuser-spezifische *Username-Phrase* und die *Password-Phrase*. Auf der Serverseite müssen diese beiden Phrasen auch dem Server mitgeteilt werden; dies geschieht über die beiden Konfigurationsvariablen `indigo.superUserName` und `indigo.superUserPassword`.

In diesem Kontext ist zu erwähnen, daß man die sicherheitskritischen Konfigurationsvariablen – wie die beiden Superuser-spezifischen Konfigurationsvariablen oder die anderen Keystore- bzw. Truststore-Paßwörter – aus Sicherheitsgründen dem Server nur über eine Konfigurationsdatei mitteilen sollte. Wenn diese dem Server über die „-D“-Option des Java-Interpreters mitgeteilt werden, können die anderen Klassen des Servers direkt auf diese Konfigurationsvariablen zugreifen. Außerdem kann man beim Aufruf der Programme zur Statusanzeige der aktiven Prozesse – wie beispielsweise „*ps*“ – diese Konfigurationsvariablen, die als Argumente dem Java-Interpreter übergeben werden, sichtbar machen.

6.1.3 Weitere Modifikationen des INDIGO-Servers

In diesem Abschnitt werden die weiteren wichtigen Modifikationen des Servers, die nicht direkt die Zugriffssteuerung betreffen, beschrieben.

Eine der wichtigsten Änderungen des Servers betrifft die Kommunikation des INDIGO-Servers mit den Dokumentmethoden-Servern. Durch die Modifikationen der `MetaDocumentStore`-Klasse ist der Server – wie bereits im Abschnitt 6.1.1.2 geschildert – in der Lage, auch mit HTTP-fähigen Dokumentmethoden-Servern zu kommunizieren, die das SSL-Protokoll zur Sicherung ihrer Kommunikationskanäle verwenden (*HTTPS-Server*).

Eine weitere Modifikation des Servers betrifft die `Attributes`-Klasse. Diese Klasse wurde dahingehend modifiziert, daß sie auch mit den Dokumentattributen – wie beispielsweise Zertifikaten – umgehen kann, die über mehrere Zeilen gehen. Diese Zeilen müssen von Multiline-Tags eingeschlossen sein.

Base64Converter Die Klasse `Base64Converter` wurde für die Konvertierung der Binärdaten in das Base64-Zeichensatzformat entwickelt. Base64 ist allgemein ein textbasiertes Format, das zur Übertragung von Binärdaten dient. Dieses Format wird in RFC 2045 [RFC2045] beschrieben und gehört zu den MIME-Richtlinien. Das Base64-Format verwendet einen eingeschränkten Zeichensatz von folgenden 64 Zeichen:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

Die `Base64Converter`-Klasse wird in der Implementierung des Servers unter anderem zum Kodieren und Dekodieren der Superuser-spezifischen Daten (bei der Basic Authentication), beim Dekodieren der Zertifikate der Autoren und beim Dekodieren der digitalen Signatur der Dokumentinhalte, die im Base64-Format vorliegen, eingesetzt.

In JDK existiert im Paket `sun.misc` die Klasse `BASE64Encoder`, die die gleiche Funktionalität wie die `Base64Converter`-Klasse bietet. Trotzdem wurde diese Klasse bei der Implementierung nicht verwendet, da die Entwickler der Klasse von deren Einsatz abraten.

IndigoConfig Die Hauptaufgabe der `IndigoConfig`-Klasse, die auch im Laufe dieser Implementierung erweitert wurde, ist das Lesen und Auswerten der Konfigurationsvariablen. Die Klasse stellt ebenfalls diese Konfigurationsvariablen über ihre Methoden den anderen Klassen des Servers zur Verfügung. In diesem Zusammenhang ist es wichtig zu erwähnen, daß die `IndigoConfig`-Klasse aus Sicherheitsgründen den Zugriff auf die Superuser-spezifischen Daten nur von ihr abgeleiteten Klassen erlaubt.

IndigoAuthorization Die `IndigoAuthorization`-Klasse ist für die Autorisierung der Anwender bezüglich der Server-Befehle zuständig. Der INDIGO-Server legt außerdem über diese Klasse den Anwendertyp der Clients fest. Zu diesem Zweck erzeugt der INDIGO-Server für jeden Client – in der `IndigoConnectionHandler`-Klasse – eine Instanz dieser Klasse. Dies geschieht, nachdem der Client die Request-Zeile und die Header-Zeilen zum Server geschickt und der Server das Zertifikat des Clients ausgewertet hat. Die Instanziierung geschieht wie folgt:

```
IndigoAuthorization clientAuthorization;
clientAuthorization = new IndigoAuthorization(boolHasClientValidCert,
                                              strAuthorizationValueBASE64);
```

Über das erste Argument (vom Typ `boolean`) des Konstruktors wird die Instanz der Klasse über die Gültigkeit des Client-Zertifikats informiert. Das zweite Argument ist die vom Client gesendete Superuser-spezifische Information (vom Typ `String`), wobei diese im Base64-Format der Instanz zur Verifikation[10] übergeben wird. Mit Hilfe dieser Client-spezifischen Instanz entscheidet anschließend der Server darüber, ob der Client zur Ausführung eines Server-Befehls prädestiniert ist. Dies geschieht, indem für dieses Client-spezifische Objekt die Methode `authorizedFor` – mit dem entsprechenden Server-Befehl als Argument – aufgerufen wird. Falls der vom Client geforderte Server-Befehl „invoke“ ist, könnte eine solche Autorisierung wie folgt aussehen[11]:

```
if (clientAuthorization.authorizedFor("invoke")) {
  // Fuehre invoke aus!
  ...
}
```

[10] Da der Zugriff auf die Superuser-spezifischen Informationen des Servers nur den abgeleiteten Klassen der `IndigoConfig`-Klasse erlaubt ist, wurde aus diesem Grund bei der Implementierung die `IndigoAuthorization`-Klasse von der Klasse `IndigoConfig` abgeleitet.

[11] Die Methode `authorizedFor` liefert als Ergebnis einen booleschen Wert.

IndigoOut Diese Klasse wurde für eine kontrollierte Ausgabe der Server-Meldungen, abhängig von der `indigo.verboseMode`-Konfigurationsvariable des Servers, erstellt. Die anderen Klassen des INDIGO-Servers verwenden beispielsweise anstelle der Methode `println(String)` der Klasse `java.lang.System.err` die statische Methode `errPrintln(String)` der `IndigoOut`-Klasse. Diese Methode gibt die als Argument erhaltene Fehlermeldung nur dann – über `err.println(String)` – aus, wenn die `indigo.verboseMode`-Konfigurationsvariable den Wert `true` besitzt. Diese Klasse besitzt eine weitere Methode „`outPrintln`“, die das Pendant zur Methode `java.lang.System.out.println` darstellt.

6.2 Zugriffssteuerung

Eine der wichtigsten Sicherheitsanforderungen bezüglich des INDIGO-Servers ist die *Zugriffskontrolle*. Viele der Modifikationen des Servers betreffen bei der Implementierung diese Sicherheitsanforderung. Aus diesem Grund werden diese Modifikationen des INDIGO-Servers getrennt von den anderen in diesem Abschnitt gesondert behandelt.

Zuerst werden die Zugriffssteuerungsebenen des Servers beschrieben. Anschließend werden die Initialisierungsphase des Servers aus der Sicht der Zugriffskontrolle sowie der Authentisierungs- und Autorisierungsablauf für die Anwender in unterschiedlichen Betriebsmodi des Servers erläutert. Dabei wird besonders auf die Eigenheiten der unterschiedlichen Betriebsmodi des Servers – vor allem auf der Protokollebene – eingegangen. Zum Schluß folgt die Beschreibung des Verifikationsvorgangs bei den Dokumentmethoden. In diesem Zusammenhang kommen auch die näheren Bedeutungen der im Abschnitt 6.1.2 erwähnten sicherheitsrelevanten Konfigurationvariablen zur Sprache.

6.2.1 Zugriffssteuerungsebenen

Die Sicherheitsmaßnahmen bezüglich der Sicherheitsanforderung Zugriffskontrolle werden – wie bereits im Abschnitt 5.4 erwähnt – bei der Implementation in drei unterschiedlichen Ebenen realisiert:

- Zugriffssteuerung auf der Dokumentebene,
- Zugriffssteuerung auf der Serverebene und
- Zugriffssteuerung auf der Protokollebene.

Der Vorteil dieser Aufteilung liegt darin, daß man dadurch eine viel feingranuliertere Zugriffssteuerung – verteilt auf unterschiedliche Ebenen – erreichen kann. Das Zustandsdiagramm 6.1 zeigt die gesamte Zugriffssteuerung inklusive der drei Zugriffssteuerungsebenen. Dabei wird besonders der Fall betrachtet, bei dem der Server im Sicherheitsmodus läuft und der Server-Befehl „invoke“ ausgeführt wird; in diesem speziellen Fall sind die meisten Zugriffssteuerungsmechanismen aktiv.

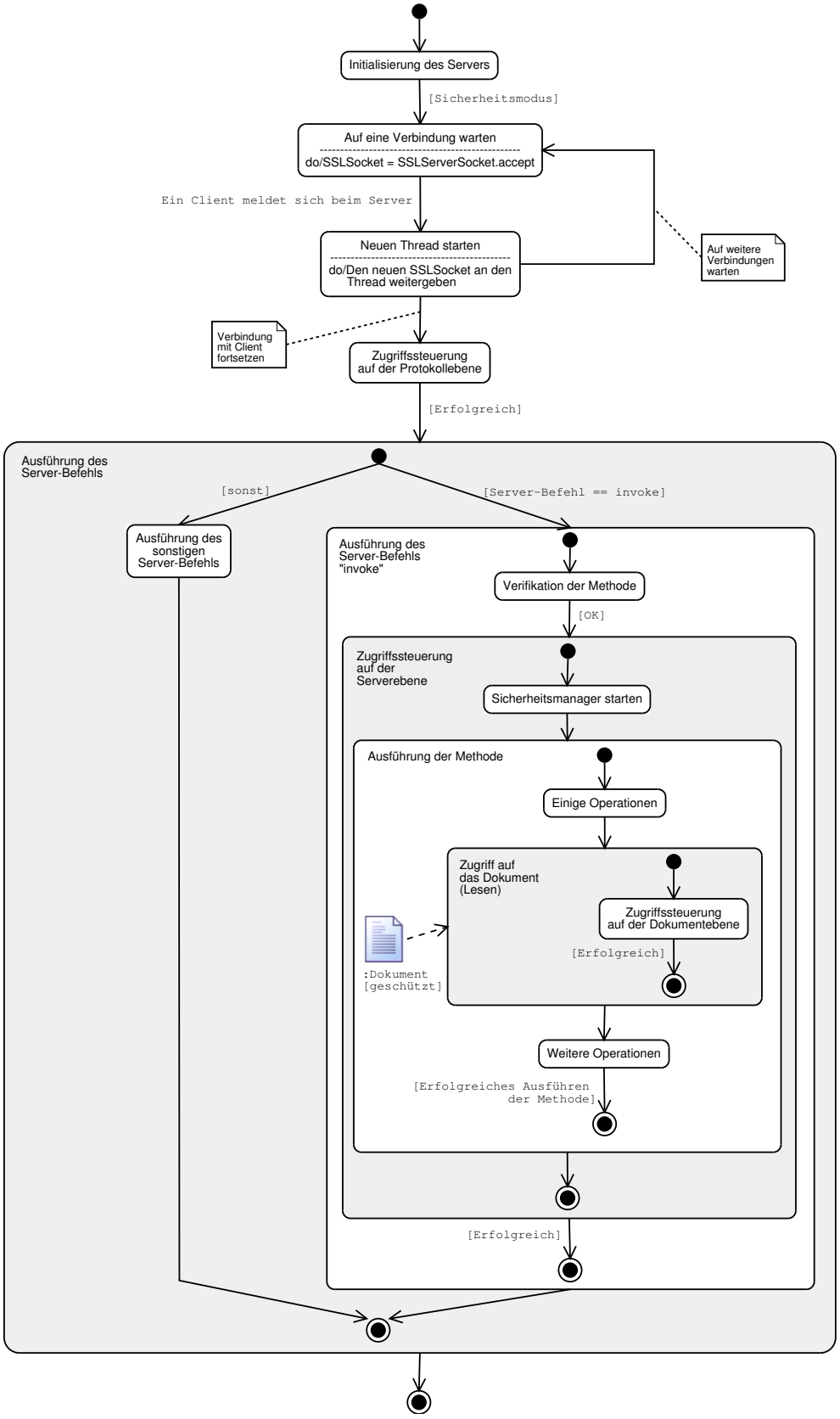

Abbildung 6.1 Zugriffssteuerungsebenen beim INDIGO-Server

Zugriffssteuerung auf der Dokumentebene Die Zugriffssteuerung auf der Dokumentebene bezieht sich auf den Zugriff der Anwender auf die einzelnen Dokumente. Für diese Art von Zugriffssteuerung kann man beispielsweise die dokumentspezifische Autorisierung, die im Abschnitt 5.3.4 (siehe Seite 79) erläutertet wurde, verwenden. Auf dieses Verfahren wurde aber bei der Implementierung nicht eingegangen, da bei diesem Verfahren die gesamten Sicherheitsvorkehrungen bezüglich der Zugriffssteuerung von den Autoren bzw. von den Dokumentmethoden-Produzenten der entsprechenden Dokumente getroffen werden.

Zugriffssteuerung auf der Serverebene Die Zugriffssteuerung auf der Serverebene bezieht sich auf die Sicherheitsvorkehrungen, die benutzt werden, um den Dokumentmethoden den Zugriff auf die fremden Ressourcen zu verwehren. Um den Gefahren zu begegnen, die von den böswilligen Dokumentmethoden ausgehen, wird beim INDIGO-Server – wie bereits im Abschnitt 5.1.1 (siehe Seite 67) geschildert – das *Sandbox-Konzept* von Java verwendet. Bei der Implementierung dieses Konzepts kommt der *Sicherheitsmanager (Security Manager)* zum Einsatz. Dabei geht man bei der Implementierung des INDIGO-Servers wie folgt vor:

Beim Start eines INDIGO-Servers wird eine Thread-Gruppe erzeugt; in dieser Gruppe instantiiert der Server jede Dokumentmethode als ein Thread. Der Zugriff dieser Threadgruppe auf die Ressource des Systems wird durch einen Sicherheitsmanager geregelt. Der Sicherheitsmanager überwacht zur Laufzeit einer virtuellen Maschine alle potentiell gefährlichen Operationen. Unter gefährlichen Operationen versteht man z. B. den Zugriff auf das Dateisystem des Rechners oder das Öffnen von Netzwerkverbindungen mittels Sockets. Der Sicherheitsmanager wird als abstrakte Basisklasse im Package `java.lang` deklariert. Die Basisklasse definiert check-Methoden, die überprüfen, ob eine gefährliche Operation ausgeführt werden darf.

Die Klasse „`IndigoSecurityManager`“ ist beim INDIGO-Server die Hauptklasse für die Zugriffssteuerung auf der Serverebene. Diese Klasse wird von der Klasse `java.lang.SecurityManager` abgeleitet. Die Klasse stellt beispielsweise sicher, daß die Dokumentmethoden nur auf die Dokumentdateien, also nur auf die `attributes`-, `content`- und `methods`-Datei des entsprechenden Dokuments zugreifen dürfen. Ein lesender bzw. schreibender Zugriff auf andere Dateien wird damit verhindert.

Diese Sicherheitsvorkehrungen wurden bereits bei der Implementierung des ursprünglichen INDIGO-Servers beachtet, so daß während dieser Arbeit in dieser Hinsicht keine Modifikationen vorgenommen werden mußten. Aus diesem Grund wird die Zugriffssteuerung auf der Serverebene hier nicht näher erläutert, sondern auf die genaue Beschreibung bei [Moe01] hingewiesen.

Zugriffssteuerung auf der Protokollebene Ein INDIGO-Server kommuniziert mit den anderen INDIGO-Servern und mit den Clients über das INDIGO-spezifische Protokoll. Dieses HTTP-basierte Protokoll wurde bereits im Abschnitt 2.3.1 auf Seite 12 erläutert.

Die Zugriffssteuerung auf der Protokollebene erfolgt, indem ein Anwender nach dem Verbindungsaufbau mit dem Server in einen der Anwendertypen eingeteilt wird. Die Anfrage dieses Anwenders (die *Server-Befehle*) wird anschließend vom Server analysiert; anhand

der Zugriffskontrolliste wird entschieden, ob dieser Anwender zur Ausführung dieser Aktion berechtigt ist.

Die in dieser Arbeit realisierten Sicherheitsvorkehrungen bezüglich der Zugriffskontrolle betreffen hauptsächlich die Zugriffssteuerung auf dieser Protokollebene. Die detaillierte Beschreibung dieser Zugriffssteuerungsebene folgt in den Abschnitten 6.2.4, 6.2.5 und 6.2.6.

6.2.2 Initialisierung des Servers aus Sicht der Zugriffskontrolle

Der INDIGO-Server kann in vier unterschiedlichen *Betriebsmodi* betrieben werden:

- *Normalmodus*,
- Sicherheitsmodus; dieser wird wiederum in drei spezielle Sicherheitsmodi unterteilt:
  - *False-Sicherheitsmodus*,
  - *Maybe-Sicherheitsmodus*,
  - *True-Sicherheitsmodus*.

Nach dem Start des INDIGO-Servers wird nach der Auswertung der sicherheitsrelevanten Konfigurationsvariablen – während der Initialisierungsphase – der Betriebsmodus des Servers festgelegt. Zu diesem Zweck sind besonders zwei Konfigurationsvariablen von Bedeutung: die `indigo.secureExtension-` und die `indigo.needClientAuth`-Variable.

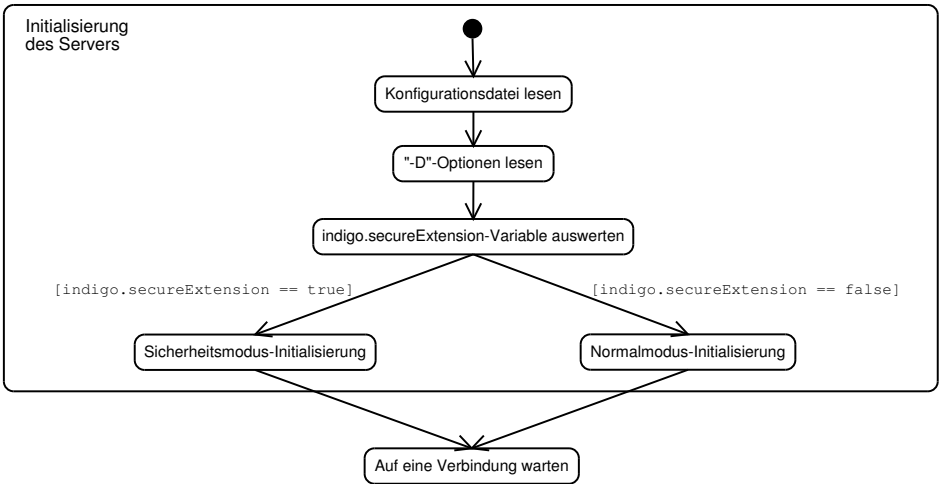

Abbildung 6.2 Initialisierung des Servers im Detail

Die Initialisierung des INDIGO-Servers aus der Sicht der Zugriffssteuerung ist in Abbildung 6.2 dargestellt. Diese Abbildung stellt die detaillierte Beschreibung des Zustands „Initialisierung des Servers“ aus der Abbildung 6.1[12] dar. Der Server liest nach seinem Start die Konfigurationsdatei und anschließend die „-D“-Optionen, die beim Starten des Servers dem Java-Interpreter übergeben werden. Um zu bestimmen, in welchem Modus er laufen soll, wertet er zuerst die Konfigurationsvariable `indigo.secureExtension` aus. Falls diese

[12] Es ist zu beachten, daß die Abbildung 6.1 nur den Spezialfall betrachtet, in dem der Server im Sicherheitsmodus läuft.

Variable den Wert `false` besitzt, läuft der Server im *Normalmodus* und muß in den Zustand „Normalmodus-Initialisierung“ übergehen. Falls sie aber `true` ist, geht der Server in den *Sicherheitsmodus* über und muß noch den Zustand „Sicherheitsmodus-Initialisierung“ durchlaufen. In diesem Zustand wird die `indigo.needClientAuth`-Variable ausgewertet, um den speziellen Sicherheitsmodus zu ermitteln. Die beiden Zustände Normalmodus- und Sicherheitsmodus-Initialisierung sind somit Unterzustände des Zustands „Initialisierung des Servers“.

Normalmodus-Initialisierung In diesem Unterzustand werden alle Sicherheitsmechanismen des Servers ausgeschaltet. Zu diesem Zweck werden alle weiteren sicherheitsrelevanten Konfigurationsvariablen auf `false` bzw. auf `Nihil` gesetzt.

Die Ausschaltung der Sicherheitsmechanismen hat beispielsweise zur Folge, daß der Server in diesem Modus zur Kommunikation mit den anderen Akteuren die normalen TCP-Sockets verwendet. Zu diesem Zweck wird während der Initialisierungsphase des Servers ein `ServerSocket` erzeugt, auf den (später) der Server mittels `accept()` auf die Anfragen der Clients wartet (siehe Abbildung 6.3).

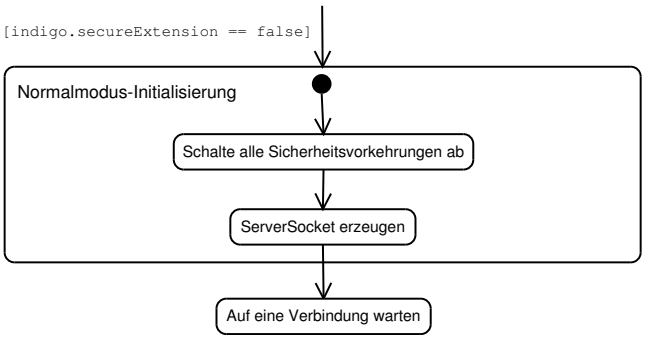

Abbildung 6.3 Normalmodus-Initialisierung im Detail

Sicherheitsmodus-Initialisierung In dem Unterzustand „Sicherheitsmodus-Initialisierung“ werden die restlichen sicherheitsrelevanten Konfigurationsvariablen auf die vom Server-Betreiber angegebenen Werte gesetzt. Anschließend wird die Variable `indigo.needClientAuth` ausgewertet, um den speziellen Sicherheitsmodus zu bestimmen (siehe Abbildung 6.4). Abhängig von dem Wert – `true`, `maybe` und `false` – dieser Variable läuft der Server in einem der Sicherheitsmodi True-Sicherheitsmodus, Maybe-Sicherheitsmodus oder False-Sicherheitsmodus.

Der Hauptunterschied zwischen diesen Modi liegt während der Initialisierungsphase des INDIGO-Servers darin, daß abhängig von diesen Modi der Server jeweils mit einer anderen *Zugriffskontrolliste* (siehe Abschnitt 6.1.2.4) gestartet wird. Das bedeutet: Wenn der INDIGO-Server beispielsweise im True-Sicherheitsmodus läuft, wählt er auch automatisch die *True-Zugriffskontrolliste* als die aktuelle Zugriffskontrolliste (`currentAccesscontrolllist=indigo.trueAccesscontrolllist`).

Ein weiterer wichtiger Unterschied zwischen den Modi liegt in der Initialisierung des `SSLServerSocket`-Objekts. Dieses Objekt wird während der Initialisierung des Servers erzeugt (siehe Abschnitt 6.1.1.1). Auf diesem Socket wartet der Server auf Clients. Wenn

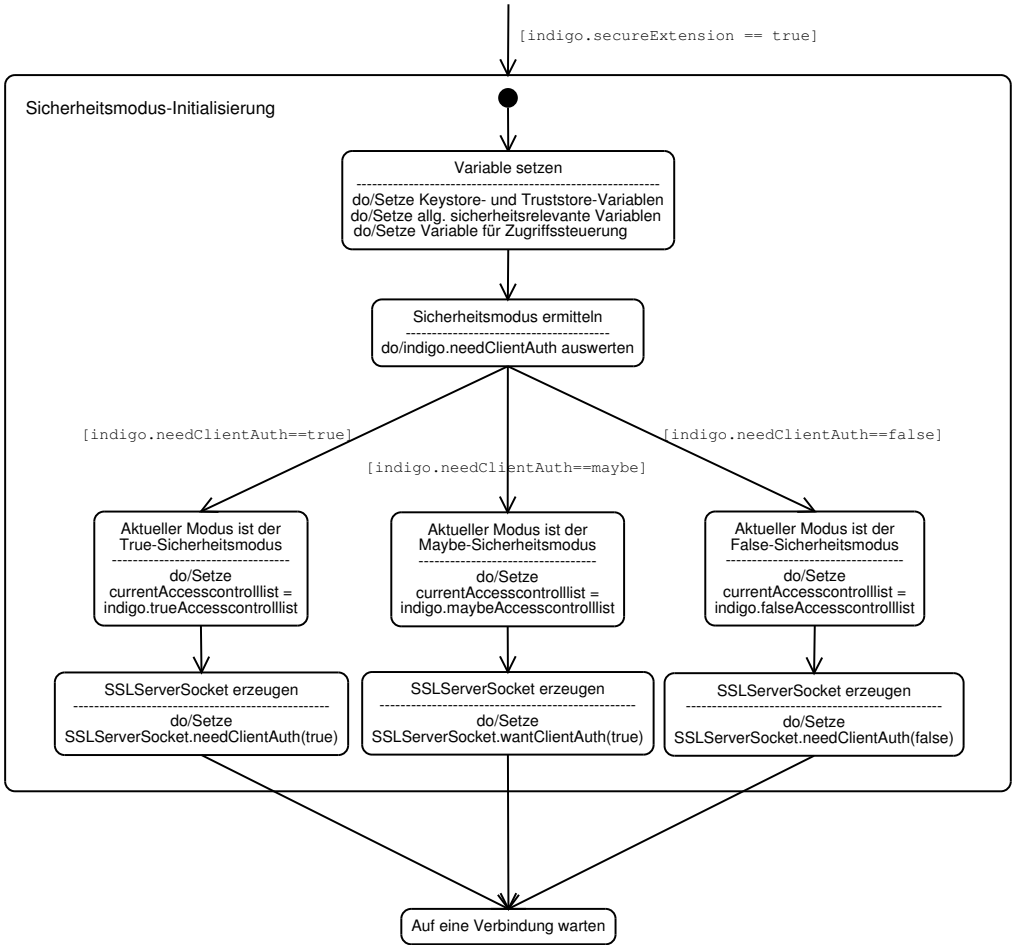

Abbildung 6.4 Sicherheitsmodus-Initialisierung im Detail

der Server im True-Sicherheitsmodus läuft, müssen die Clients, die mit diesem Server kommunizieren möchten, sich mittels gültiger Zertifikate identifizieren. Falls diese Client-seitige Authentifizierung fehlschlägt, wird die Verbindung von dem `SSLServerSocket`-Objekt unterbrochen. Im Maybe-Sicherheitsmodus wird hingegen beim Fehlschlagen der Client-Authentifizierung die Verbindung nicht unterbrochen; trotzdem kennzeichnet der Server diesen Client als einen anonymen Anwender. Im Gegensatz zu diesen beiden Modi wird beim False-Sicherheitsmodus überhaupt keine Client-Authentifizierung verlangt. Somit ist der True-Sicherheitsmodus – gefolgt vom Maybe-Sicherheitsmodus – der Modus mit den strengsten Sicherheitsanforderungen an den Anwender.

6.2.3 Autorisation im Normalmodus

Im Normalmodus ist der Server über normale TCP-Sockets erreichbar. In diesem Modus existieren keine unterschiedlichen Anwendertypen. Da alle Sicherheitsvorkehrungen ausgeschaltet sind, können die Anwender uneingeschränkt alle Server-Befehle des Servers in Anspruch nehmen. Der Normalmodus bietet keine Möglichkeit zur Verifizierung der Dokumentmethoden. Außerdem können die Dokumentmethoden nicht von einem HTTPS-Dokumentmethoden-Server bezogen werden.

Die Ausschaltung der Sicherheitsvorkehrungen bezüglich der Zugriffskontrolle bezieht sich aber lediglich auf die Protokollebene, so daß die Dokumentmethoden weiterhin keinen Zugriff auf die fremden Ressourcen haben. Außerdem kann eine Dokumentmethode sogar im Normalmodus über die NetSSL-Package (siehe Abschnitt 6.3.2) die digitale Signatur ihres Dokumentinhaltes prüfen.

Auf den Normalmodus wird in dieser Arbeit nicht weiter eingegangen, denn er stellt keine besondere Änderung des ursprünglichen INDIGO-Servers in [Moe01] dar.

6.2.4 Autorisation im True-Sicherheitsmodus

Nach der Initialisierungsphase wartet der INDIGO-Server im True-Sicherheitsmodus auf eine Verbindung (am `SSLServerSocket`). Nachdem ein Client sich bei ihm gemeldet hat, erzeugt dieser Server einen neuen Thread und übergibt an diesen Thread den – mittels der `accept`-Methode – neu erzeugten `SSLSocket` (siehe Abbildung 6.1 auf Seite 99). Von diesem Zeitpunkt an agiert der Server – wie bei den Servern allgemein üblich – in zwei unterschiedlichen leichtgewichtigen Prozessen: Während im ersten Thread – am `SSLServerSocket` des Servers – wieder auf eine neu ankommende Verbindung gewartet wird, wird bei dem zweiten Thread die Kommunikation mit dem Client über den neuen `SSLSocket` fortgesetzt. Dieser zweite Thread landet dabei bei der Zugriffssteuerung auf der Protokollebene. Die Abbildung 6.5 stellt die detaillierte Beschreibung des Zustands „Zugriffssteuerung auf der Protokollebene“ aus der Abbildung 6.1 dar, wobei dieser hier speziell im True-Sicherheitsmodus betrachtet wird.

Wie bereits in diesem Kapitel erwähnt, muß sich ein Client im True-Sicherheitsmodus mittels eines gültigen Zertifikates bei dem INDIGO-Server authentifizieren. Zu diesem Zweck fordert der Server den Client beim Verbindungsaufbau auf, sich zu identifizieren. Dieser Vorgang wird während der Handshake-Phase des Verbindungaufbaus automatisch vom SSL-Protokoll erledigt. Bei der Verifikation des Client-Zertifikats beachtet der Server mehrere Punkte, um über die Gültigkeit des Zertifikats zu entscheiden: Er prüft beispielsweise, ob der Client überhaupt ein Zertifikat verwendet. Außerdem muß dieses Zertifikat von einer dem Server-Betreiber vertrauenswürdigen Instanz signiert sein[13], und das Gültigkeitsdatum des Zertifikats darf nicht abgelaufen sein. Bei der Überprüfung der Gültigkeit des Client-Zertifikats wird bei jedem Auftreten einer Unregelmäßigkeit die Verbindung automatisch durch das SSL-Protokoll abgebrochen. Dies führt dazu, daß man in diesem Modus keinen Anwender des Typs „Others“ hat.

Nach der erfolgreichen Verifikation des Client-Zertifikats wird die Anfrage des Clients aus dem Socket gelesen und bearbeitet; hierbei wird beispielsweise aus der Request-Zeile der Server-Befehl extrahiert. In dieser Phase werden ebenfalls die restlichen Header-Zeilen ausgewertet.

In der nächsten Autorisierungsphase wird der Anwender in einer der beiden Typen „Group“ oder „Superuser“ eingeteilt. Zu diesem Zweck durchsucht der Server die Header-Zeilen der Anfrage nach der „Authorization-Zeile“. Über diese Zeile kann der Anwender dem Server den Superuser-spezifischen Benutzernamen und das Paßwort übermitteln (siehe Seite 80). Hierbei kommt die *Basic Authentication* [HTTP] zum Einsatz. Diese Art der Authentifikation ist unsicher. Da im Sicherheitsmodus der INDIGO-Infrastruktur das HTTP-Protokoll

[13] Die Liste der vertrauenswürdigen Instanzen legt der Server-Betreiber mit der Truststore-Datei fest, die er über die `indigo.trustStore`-Konfigurationsvariable an den Server überreicht.

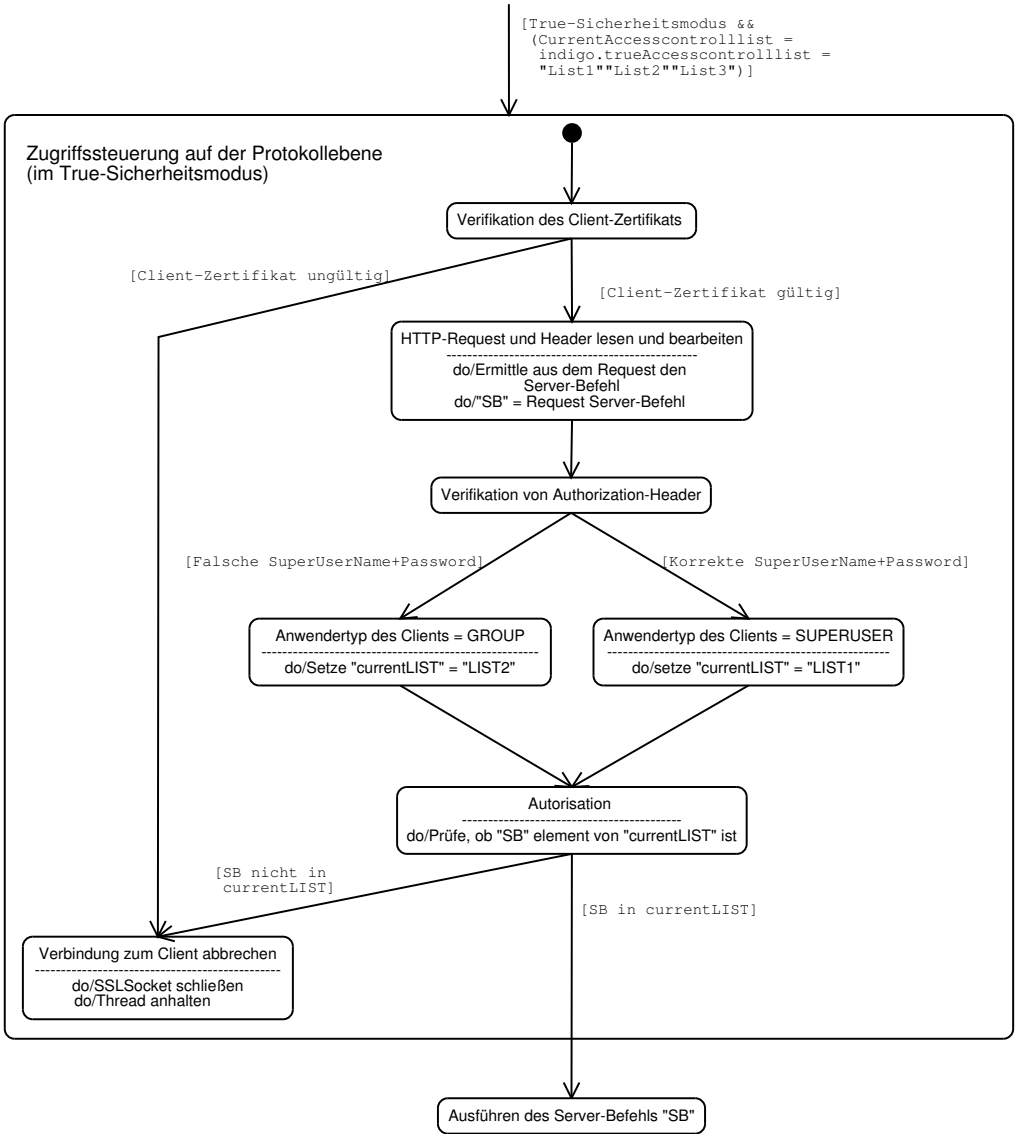

Abbildung 6.5 Zugriffssteuerung auf der Protokollebene im True-Sicherheitsmodus

durch das darunter liegende SSL-Protokoll geschützt vor jedem Zugriff verwendet wird, kann man in diesem Kontext trotzdem diese Authentifikation einsetzen. Eine solche Authentifikation könnte wie folgt aussehen:

```
GET /diglib/documents HTTP/1.1
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
```

Hierbei wird der Benutzername und das Paßwort getrennt durch „:“ mit Base64 kodiert[14]; das Ergebnis der Kodierung wird nach dem Schlüsselwort „`Basic`“ an die Anfrage angehängt.

Wenn die vom Client gesendeten Header-Zeilen eine solche Authorization-Zeile beinhalten, vergleicht der Server den in dieser Zeile enthaltenen Benutzernamen und das Paßwort

[14]Für dieses Beispiel gilt: *Base64(*`guest:guest`*)* = `Z3Vlc3Q6Z3Vlc3Q=`

mit seinen Superuser-spezifischen Informationen. Falls diese Informationen übereinstimmen, wird dieser Anwender als Superuser markiert. Andernfalls wird er zum Anwendertyp „Group“ gezählt (auch dann, wenn die Anfrage keine Authorization-Zeile beinhaltet).

Die *aktuelle Zugriffskontrolliste* des Servers, die während der Initialisierungsphase festgelegt wird, besteht aus drei *Zugriffslisten* (siehe Abschnitt 6.1.2.4). Nachdem der Anwendertyp des Clients festgelegt ist, wird diesem abhängig von seinem Typ einer dieser Zugriffslisten zugewiesen. In jeder Zugriffsliste sind die Server-Befehle zusammengefaßt, auf denen der entsprechende Anwendertyp zugreifen darf. Falls der vom Client geforderte Server-Befehl zu den in der Zugriffsliste aufgezählten Server-Befehlen gehört, wird dieser ausgeführt. Sonst schickt der Server dem Client eine HTTP-konforme Fehlermeldung. Falls der Anwender beispielsweise zum Typ Group zählt und einen Server-Befehl ausführen möchte, der ihm nicht erlaubt ist, schickt ihm der Server die folgende Status-Zeile mit den entsprechenden Headers und Fehlermeldungen (ausführliche Informationen zur Syntax und zur Semantik der HTTP-konformen Antworten (Response) siehe [HTTP]):

```
HTTP/1.1 401 Unauthorized
Server: Java-Indigo Server V1.9 with Secure Extensions
Date: Sat Apr 06 02:25:09 CEST 2002
Content-type: text/plain
WWW-Authenticate: Basic realm="Welcome To INDIGO - The Home Of Documents"

Not Authorized for this method: documents
```

Die Zugriffskontrolle im True-Sicherheitsmodus ist somit auf der Protokollebene bereits abgeschlossen. Zwar finden während der Ausführung des Server-Befehls weitere Kontrollen – wie die Verifikation der Dokumentmethoden (siehe Abschnitt 6.2.7) – statt; diese zählen aber nicht zu den Zugriffskontrollen, die auf der Protokollebene durchgeführt werden.

6.2.5 Autorisation im Maybe-Sicherheitsmodus

Im Maybe-Sicherheitsmodus wartet der Server – wie im True-Sicherheitsmodus – an einem `SSLServerSocket` auf eine ankommende Verbindung. Beim Vergleich der beiden UML-Zustandsdiagramme 6.6 und 6.5 wird schnell deutlich, daß die Unterschiede auf der Protokollebene zwischen den beiden Sicherheitsmodi bezüglich der Autorisierung nicht so gravierend ausfallen.

Im Maybe-Sicherheitsmodus wird – im Gegensatz zum True-Sicherheitsmodus – während der Verifikation eines Client-Zertifikats die Verbindung zum Client vom SSL-Protokoll nicht automatisch unterbrochen, wenn das Zertifikat ungültig ist; die Kommunikation mit diesem Client wird fortgesetzt. In diesem Modus wird dieser Client lediglich als ein anonymer Anwender behandelt, indem er als Anwendertyp „Others“ markiert wird.

Zwar werden bei Others – im Maybe-Sicherheitsmodus – wie bei den anderen Anwendertypen die Request- und die Header-Zeilen gelesen; trotzdem findet bei diesem Anwendertyp keine Auswertung der Authorization-Zeile statt.

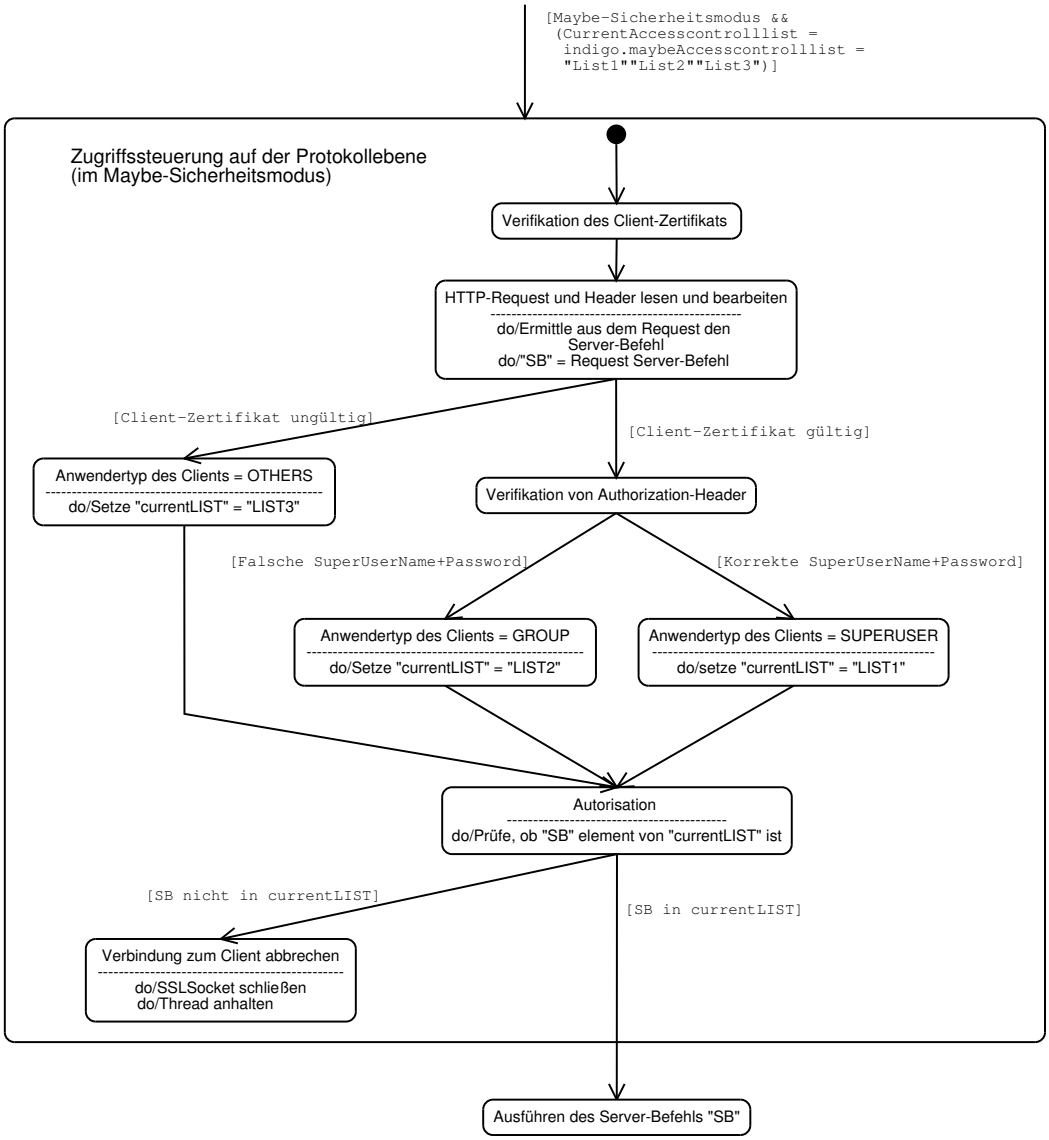

Abbildung 6.6 Zugriffssteuerung auf der Protokollebene im Maybe-Sicherheitsmodus

6.2.6 Autorisation im False-Sicherheitsmodus

Im False-Sicherheitsmodus findet nur eine Server-seitige Authentifikation statt. Dadurch hat der Server keine Möglichkeit zur Überprüfung der Client-Zertifikate. Aus diesem Grund gibt es in diesem Modus keine Anwender des Typs „Group“. Trotzdem können die Anwender sich mittels der HTTP-basierten „Basic Authentication“ authentifizieren, um sich weitere Zugriffsrechte zu sichern. Die Anwender, die sich mittels der Basic Authentication authentifizieren, werden in diesem Modus als Superuser behandelt, während die restlichen anonymen Anwender als Anwendertyp Others betrachtet werden.

Da es im False-Sicherheitsmodus nur die beiden Anwendertypen „Superuser“ und „Others“ gibt, unterscheidet sich dieser Modus in einigen wichtigen Punkten von den beiden anderen Sicherheitsmodi (siehe Abbildung 6.7): Die Clients besitzen in diesem Modus keine Zertifikate. Aus diesem Grund findet während der Autorisierung im False-Sicherheitsmodus

keine Verifikation der Client-Zertifikate statt. Außerdem werden die Anwender nach der Verifikation der Authorization-Zeile entweder im Anwendertyp Others oder im Anwendertyp Superuser eingeteilt. Das Vorgehen des Servers im False-Sicherheitsmodus ist in den restlichen Autorisationsschritten fast identisch mit dem Vorgehen des Servers in den anderen Modi.

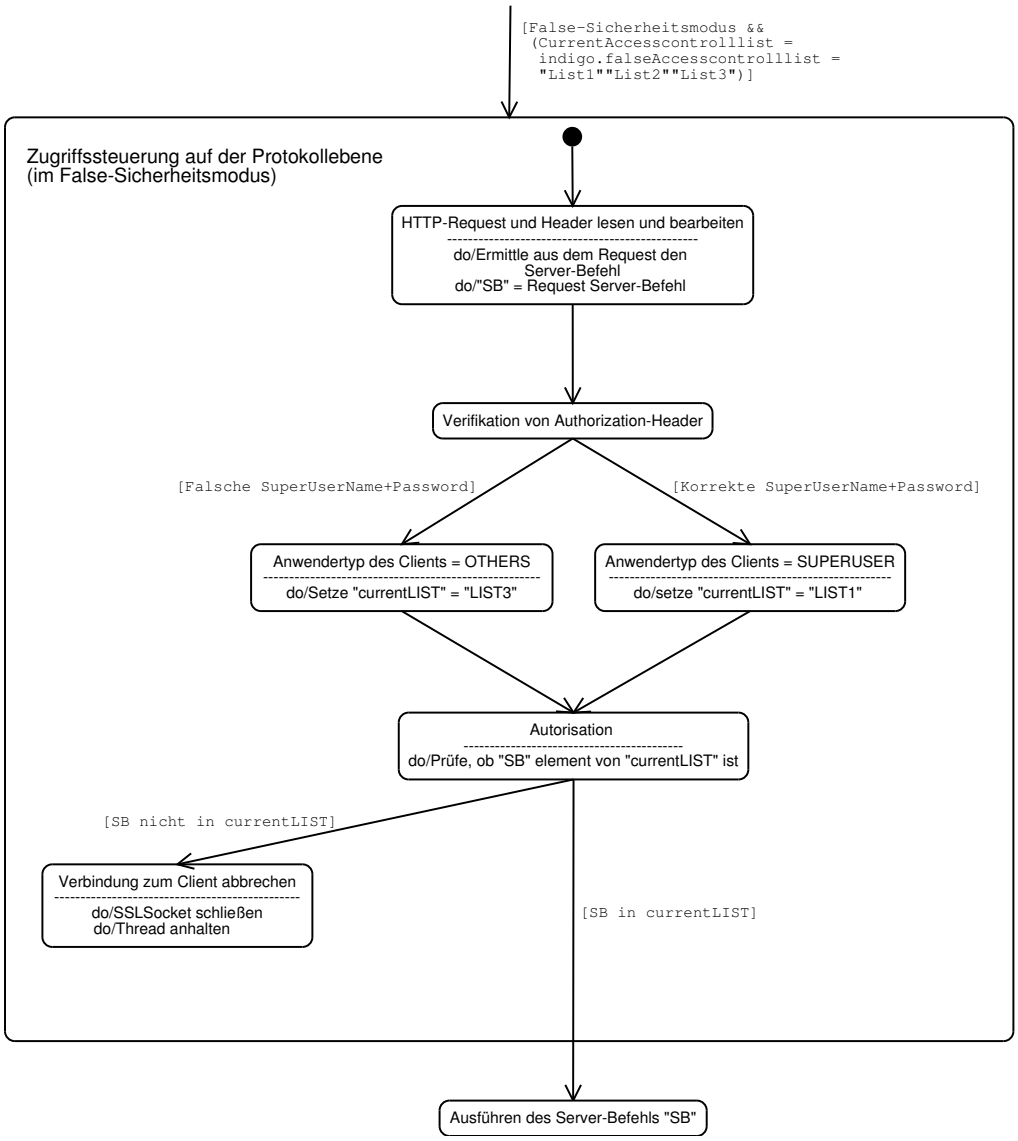

Abbildung 6.7 Zugriffssteuerung auf der Protokollebene im False-Sicherheitsmodus

6.2.7 Verifizierung der Dokumentmethoden

Unter *Verifizierung der Dokumentmethoden* versteht man den Vorgang, bei dem man die digitale Signatur einer signierten Dokumentmethode verifiziert. Die Verifikation der Dokumentmethoden kann nur bei den INDIGO-Servern stattfinden, die im Sicherheitsmodus laufen.

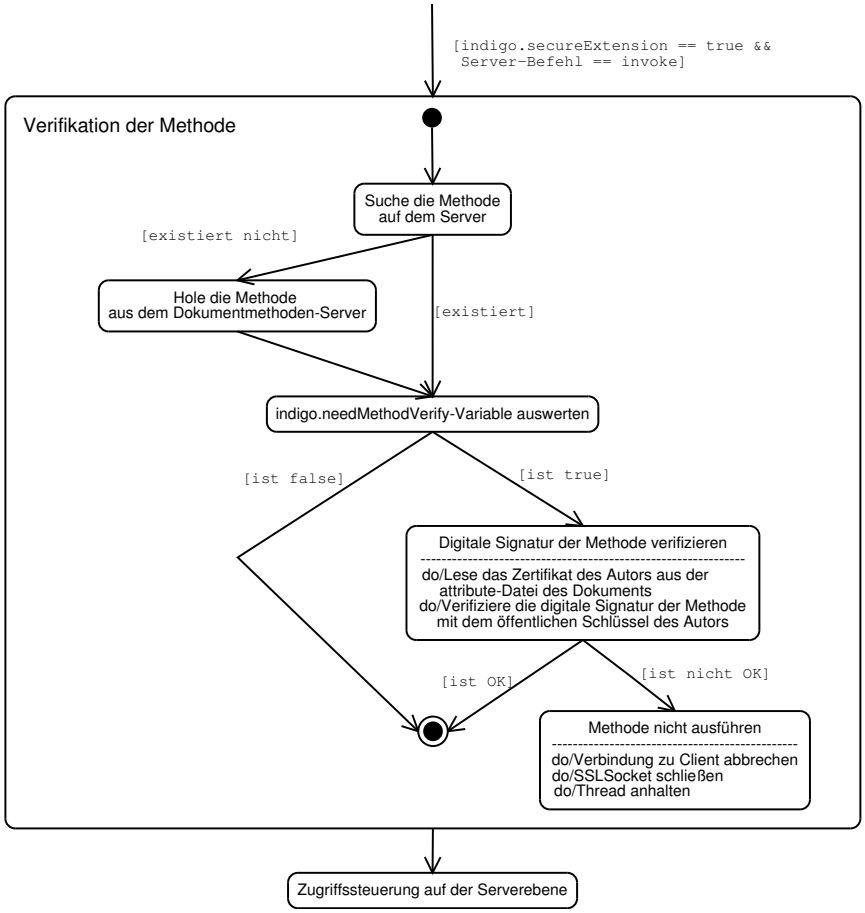

Abbildung 6.8 Verifikation der Dokumentmethoden im Detail

Der Vorgang der Methoden-Verifizierung ist in Abbildung 6.8 dargestellt. Dieses UML-basierte Zustandsdiagramm stellt die detaillierte Beschreibung des Zustands „Verifikation der Methode“ der Abbildung 6.1 (siehe Seite 99) dar.

Bevor eine Dokumentmethode – verursacht durch invoke – ausgeführt wird, muß diese auf dem Server gespeichert sein. Zu diesem Zweck wird – falls die Methode nicht beim Server existiert – zuerst die Methode von einem fremden Dokumentmethoden-Server geladen und beim Server gespeichert. Falls die Konfigurationsvariable `indigo.needMethodVerify` den Wert `true` besitzt, muß der Server jede Methode vor deren Ausführung verifizieren.

Die zu verifizierenden Dokumentmethoden müssen in dieser Infrastruktur als *Jar-Dateien* vorliegen. Eine Jar-Datei ist fast identisch mit einer Zip-Datei, mit dem Unterschied, daß die Jar-Datei neben den gespeicherten Dateien weitere zusätzliche Informationen speichert (*Manifest*). Diese Dateien können außerdem von mehreren Personen signiert sein [Fla00], wobei die Signatur und die privaten Schlüssel der Signierer in die Jar-Datei eingefügt werden. Somit kann in dieser Infrastruktur eine Dokumentmethode von ihrem Produzenten und von irgendwelchen Autoren unterschrieben sein. Diese Eigenschaft der Jar-Archive wurde bei dieser Arbeit benutzt, indem vorausgesetzt wurde, daß die Methoden von den Autoren der entsprechenden Dokumente, die diese Methoden verwenden, unterschrieben sein müssen.

Zum Signieren der Dokumentmethoden kann das Programm „*jarsigner*“ verwendet werden, das im JDK-1.4 mitgeliefert wird[15]. Ein Dokument, das eine solche signierte Methode verwendet, muß lediglich das Zertifikat seines Autors beinhalten. Dies geschieht, indem das Zertifikat – kodiert in Base64 – in den Attribute-Abschnitt des Dokuments nach dem Schlüsselwort „`AuthorCertificate`“ eingefügt wird. Ein Beispiel für einen solchen Attribute-Abschnitt könnte wie folgt aussehen:

```
Autor J.W. von Goethe
Owner goethe@pen.de
AuthorCertificate <MULTILINE>
-----BEGIN CERTIFICATE-----
MIICRjCCAa8CBDyFlfgwDQYJKoZIhvcNAQEEBQAwajELMAkGA1UEBhMCREUxDzANBgNVBAgTBkhl
c3NlbjEVMBMGA1UEBxMMRnJhbmtmdXJ0LO0uMQwwCgYDVQQKEwNQZW4xDzANBgNVBAsTBlBvZXNp
ZTEUMBIGA1UEAxMLSi5XLiBHb2V0aGUwHhcNMDIwMzA2MDQwNzIwWhcNMTIwMzAzMDQwNzIwWjBq
MQswCQYDVQQGEwJERTEPMA0GA1UECBMGSGVzc2VuMRUwEwYDVQQHEwxGcmFua2Z1cnQvTS4xDDAK
BgNVBAoTA1BlbjEPMA0GA1UECxMGUG9lc2llMRQwEgYDVQQDEwtKLlcuIEdvZXRoZTCBnzANBgkq
hkiG9w0BAQEFAAOBjQAwgYkCgYEArzcflEVLY4prtPhTvz4ogYJzjkhJZuQ1Rr6we3hqOyQcSc7K
8tYviWAXIkmI3RbnBUakoOqQYnjlUY0pLvGXFgQ7jmn295/roRh2zN/h2/D1uX0R16fPkLdJUPGz
wBcdFYEk2SJ8DDr4ezm+9YJ63R2Is4/s3vomzAfoD4esI60CAwEAATANBgkqhkiG9w0BAQQFAAOB
gQBvB+8OZwhExcnRtmyt4Tt5i4YO6SL1q/3O9uuKaJ2ef2L0cpv5GPIbANdj5j3R9G4eTYMO+Use
JkGzIOvOGBdsGsqMouv5m5PfBul53dDagK1MVKlzxujmnqwtYc1nCF1i2prbg3aptmGcS0m5sgDp
QB2jvnUP78z+GoxBDkjKbQ==
-----END CERTIFICATE-----
</MULTILINE>
AuthorCertificateType X.509
```

Wie in diesem Beispiel zu sehen ist, muß ein solcher Attribut-Wert (wie das Zertifikat), der über mehrere Zeilen geht, stets von Multiline-Tags markiert sein. Über die `AuthorCertificateType`-Variable muß außerdem der INDIGO-Server über den Typ des Zertifikats informiert werden.

Bei der Verifikation einer Dokumentmethode wird zuerst die Signatur dieser Datei – in der Jar-Datei – verifiziert. Dabei werden die öffentlichen Schlüssel der Signierer verwendet, die in der Jar-Datei mitgespeichert sind. Anschließend wird der öffentliche Schlüssel des Autors, der aus seinem Zertifikat extrahiert wurde, mit den öffentlichen Schlüsseln verglichen, die bei der Verifikation der entsprechenden Methode zum Einsatz kamen. Falls diese Verifikation erfolgreich war, wird diese Methode ausgeführt; andernfalls wird sie nicht ausgeführt und der Client erhält eine Fehlermeldung.

Aus der Sicht der Zugriffskontrolle stellt die Verifikation der Dokumentmethoden ein Bindeglied zwischen der Zugriffskontrolle auf der Protokollebene und der Zugriffskontrolle auf der Serverebene dar: Vor jeder Verifikation einer Methode findet stets eine Zugriffskontrolle auf der Protokollebene statt. Nach der erfolgreichen Verifikation einer Methode wird diese unter der Aufsicht des Sicherheitsmanagers INDIGO-Server ausgeführt (siehe Abbildung 6.1 auf Seite 99).

[15] Eine kurze Beschreibung des Programms *jarsigner* findet sich im Anhang A.2.

6.3 Erweiterungen der Packages

Die *Packages* stellen beim INDIGO-Server eine Schnittstelle für die Dokumentmethoden dar. Über diese Schnittstellen haben die Dokumentmethoden den Zugriff auf die Ressourcen des Servers. Sie sind die Implementierung der im Abschnitt 2.3.3 erwähnten *Server-*, *Dokument-* und *Traversierungs-Funktionen*.

Die Erweiterungen der Packages beziehen sich auf die sicherheitsrelevante Modifikation der bestehenden Packages *Base* und *Self* sowie die Erstellung des neuen Packages *NetSSL*. In diesem Abschnitt werden die entsprechenden Modifikationen erläutert.

6.3.1 Base-Package

Im Base-Package sind die Server-Funktionen implementiert. Über das Base-Package kann beispielsweise eine Präsentationsmethode ihr Dokument mittels „`s_transport`“ zu einem fremden INDIGO-Server schicken und anschließend auf dieses Dokument – auf dem fremden Server – eine lokale Präsentationsmethode ausführen (siehe Abbildung 6.9).

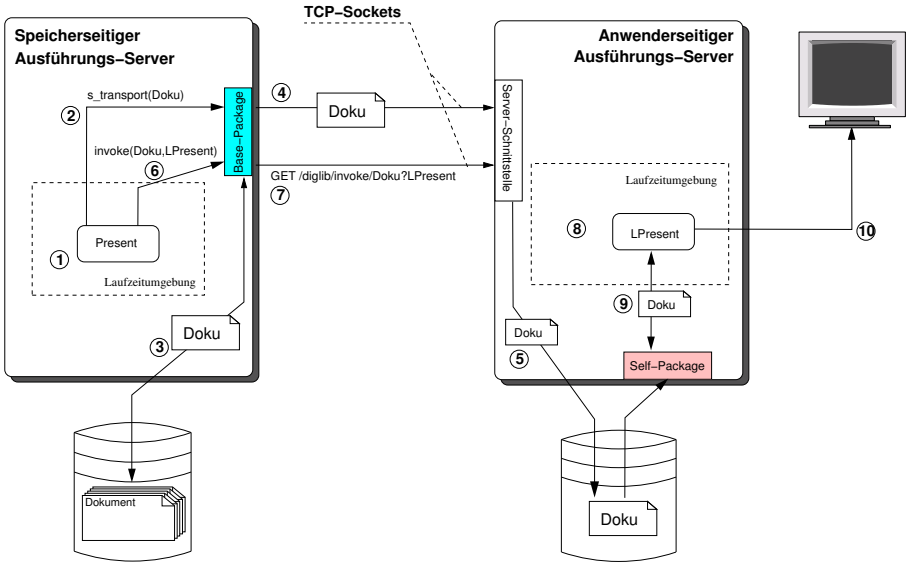

Abbildung 6.9 Base-Package mit TCP-Sockets

Das ursprüngliche Base-Package unterstützte die Kommunikation einer Dokumentmethode mit einem INDIGO-Server über die TCP-Sockets (siehe Abbildung 6.9). Bei der Modifikation wurde dieses Package hauptsächlich um die SSL-Eigenschaften erweitert. Die Modifikationen dieses Packages betreffen fast alle seine Methoden: Sowohl die Methoden zum Transport der Metadokumente (die Methode `s_transport` für den serverbasierten Transport und die Methode `d_transport` für den methodenbasierten Transport) als auch die Methoden zum Aufrufen der Dokumentmethoden (`invoke` und `invokeAsync`) wurden modifiziert.

Diese Modifikationen sind aber – im Betrieb – nur dann wirksam, wenn der Server im Sicherheitsmodus läuft. In diesem Modus benutzen die Methoden des Base-Packages SSL-gesicherte Kanäle zur Kommunikation (siehe Abbildung 6.10). Dies wird erreicht, indem im

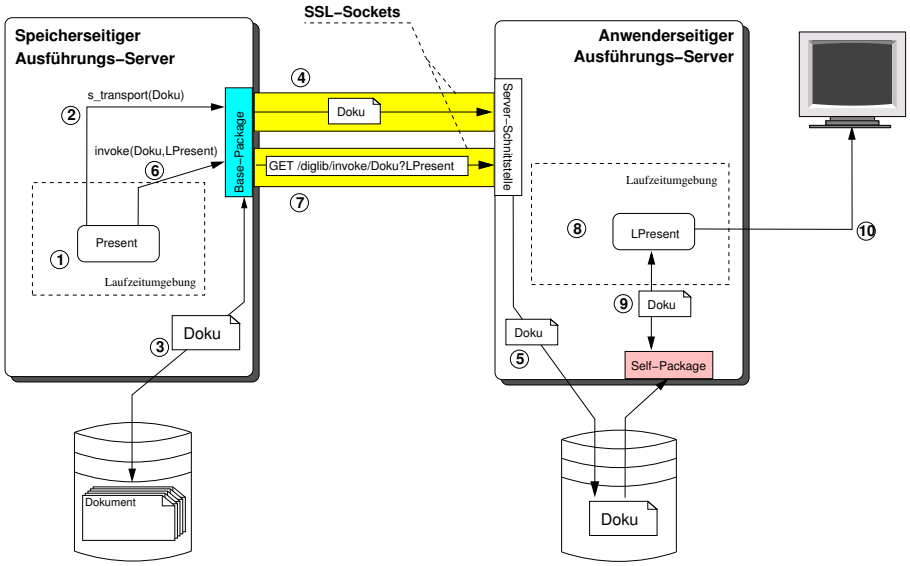

Abbildung 6.10 Base-Package mit SSL-Sockets

Sicherheitsmodus beispielsweise bei der `s_transport`-Methode anstatt der Anweisungen

```
URL url = new URL("http", serverAdress, serverPort, "/diglib/deliver");
HttpURLConnection aConnection = (HttpURLConnection)url.openConnection();
```

die folgenden Anweisungen verwendet werden:

```
URL url = new URL("https", serverAdress, serverPort, "/diglib/deliver");
HttpURLConnection aConnection = (HttpsURLConnection)url.openConnection();
((HttpsURLConnection)aConnection).
          setSSLSocketFactory(IndigoSSLContext.getStaticSSLSocketFactory());
```

Dies führt dazu, daß im Sicherheitsmodus eine Dokumentmethode, die eine solche Methode wie `s_transport` verwendet, anstelle des HTTP-Protokolls das HTTPS-Protokoll zur Kommunikation mit einem fremden Server verwendet. Durch die Verwendung dieses Packages setzt außerdem eine solche Dokumentmethode zur Authentifizierung während des Handshake das Zertifikat des Servers – auf dem sie läuft – ein (siehe dritte Zeile des Programmcodes; detailliertere Informationen zu diesen Anweisungen findet man im Abschnitt 6.1.1.2).

Bei der Modifikation der Methoden der Base-Packages wurde stets darauf geachtet, daß die Dokumentmethoden von den Modifikationen dieser Methoden unberührt bleiben. Dies bedeutet, daß die existierenden Dokumente und deren Dokumentmethoden, die dieses Package benutzen, weiterhin auch mit dem modifizierten INDIGO-Server – sogar im Sicherheitsmodus – verwendet werden können. Eine Ausnahme bilden die Dokumentmethoden, die von einem methodenbasierten Transport Gebrauch machen. Für diese Art von Methoden wurde speziell das NetSSL-Package entwickelt.

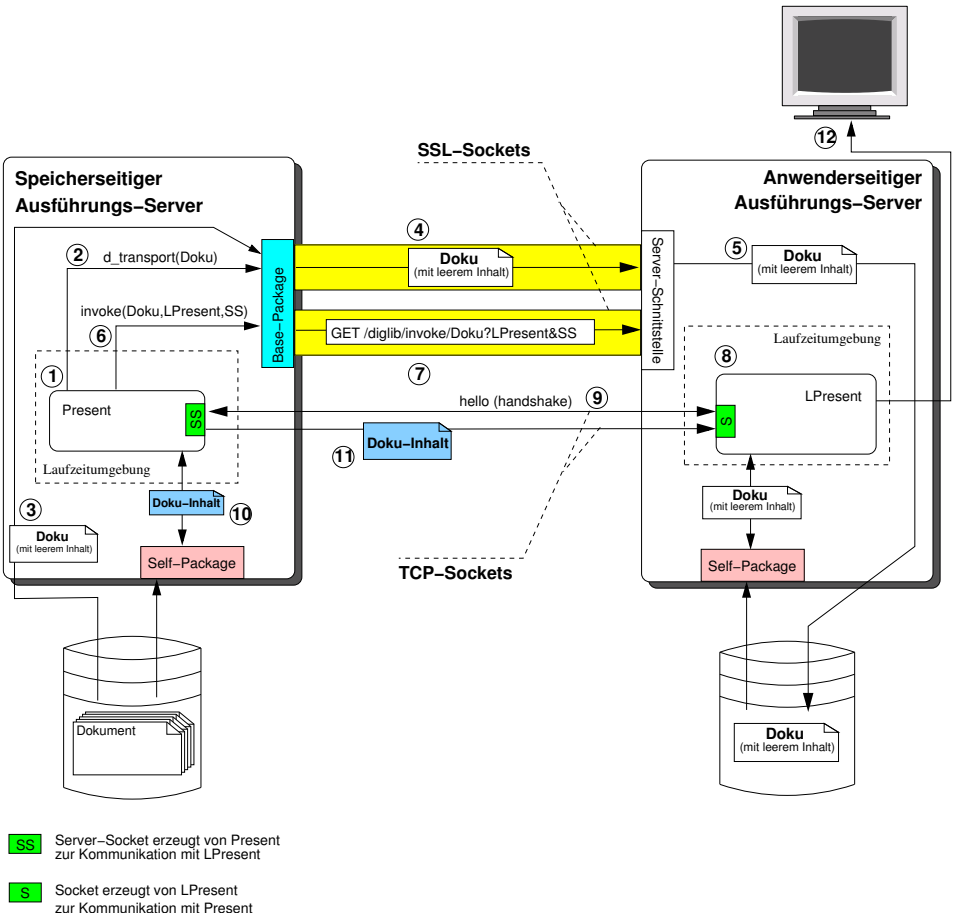

Abbildung 6.11 Online-Präsentation ohne NetSSL

6.3.2 NetSSL-Package

Bei einer Kommunikation, bei der der serverbasierte Transport verwendet wird – wie beispielsweise bei einer Offline-Präsentation (siehe Abschnitt 2.4.1) – schickt ein Ausführungs-Server[16] das gesamte Metadokument zu einem anderen Ausführungs-Server. Bei dieser Transportart kann man sich darauf verlassen, daß der Dokumentinhalt *sicher* über ein unsicheres Netzwerk übertragen wird, da beim serverbasierten Transport im Gegensatz zu einem methodenbasierten Transport die Metadaten eines Dokuments mit seinem gesamten Dokumentinhalt über den SSL-geschützten Kanal des Base-Packages – bei der Verwendung der `s_transport`-Methode – verschickt werden (siehe Abbildung 6.10 - Punkt 4).

Anders sieht es bei der Kommunikation mittels des methodenbasierten Transports aus: Bei dieser Art der Kommunikation – wie beispielsweise bei einer Online-Präsentation (siehe Abschnitt 2.4.2) – wird nur das Metadokument ohne den Dokumentinhalt übertragen. Eine Übertragung des Dokumentinhalts geschieht zu einem späteren Zeitpunkt nach den Verhandlungen zwischen der Dokumentmethode und der lokalen Dokumentmethode.

Wie im Beispiel der Abbildung 6.11 dargestellt, wird zur Online-Präsentation zuerst beim speicherseitigen Ausführungs-Server die Präsentationsmethode `Present` aufgerufen. Diese

[16] Der Ausführungs-Server agiert in diesem Zusammenhang als Client.

Dokumentmethode ruft die Methode `d_transport` des Base-Package (Punkt 2). Diese Methode besorgt sich das Metadokument `Doku` (3) und schickt es über einen SSL-gesicherten Kanal zum anwenderseitigen Ausführungs-Server (4). Hierbei werden nur die Metadaten des Dokuments ohne den Dokumentinhalt übertragen, wobei diese beim anwenderseitigen Server gespeichert werden (5). Anschließend ruft die Präsentationsmethode die `invoke`-Methode des Base-Packages (6). Diese Methode erhält neben dem Dokumentbezeichner `Doku` und dem Methodennamen `LPresent` auch die Port-Adresse eines Server-Sockets `SS`. Dieser Socket wird von der Präsentationsmethode erzeugt und dient der Kommunikation mit der lokalen Präsentationsmethode `LPresent`. Das Aufrufen dieser `invoke`-Methode führt beim anwenderseitigen Server zum Ausführen der lokalen Präsentationsmethode (8). Dieser Dokumentmethode wird bei ihrer Initialisierung die Port-Adresse `SS` übereicht. Diese Methode erzeugt anschließend ein Socket `S`, der mit dem `SS`-Server-Socket der `Present`-Methode verbunden ist. Nach einer kurzen Handshake-Phase über diesen Kommunikationskanal (9), schickt die `Present`-Methode der `LPresent`-Methode den Dokumentinhalt `Doku-Inhalt` (11).

Wie aus diesem Beispiel hervorgeht, entstehen trotz Verwendung des modifizierten Base-Packages große Sicherheitsschwachstellen, da der Dokumentinhalt beim methodenbasierten Transport nicht mit Hilfe des Base-Packages – und somit nicht über die SSL-geschützten Kanäle – übertragen wird. Um diese Sicherheitslücke zu schließen, wurde das *NetSSL*-Package entwickelt.

Das NetSSL-Package bietet den Dokumentmethoden einfache Schnittstellen zur Erzeugung der SSL-basierten Server-Sockets bzw. Sockets. Das wichtigste Merkmal dieses Packages ist, daß die erzeugten SSL-Socket-Objekte das Zertifikat des jeweiligen INDIGO-Servers anwenden.

Ein `SSLServerSocket`-Objekt kann mit Hilfe des NetSSL-Packages durch die beiden folgenden Anweisungen erzeugt werden:

```
NetSSL netSSL          = new NetSSL();
SSLServerSocket server = netSSL.getSSLServerSocket();
```

In diesem Beispiel wird ein `SSLServerSocket`-Objekt erzeugt, wobei die Portnummer dieses Sockets automatisch von dem Betriebssystem gewählt wird. Man kann durch die Anweisung

```
NetSSL netSSL          = new NetSSL(i);
SSLServerSocket server = netSSL.getSSLServerSocket();
```

ein Server-Socket-Objekt erzeugen, der auf dem Port „`i`“ auf eine Anfrage wartet.

Wenn aber auf der Client-Seite ein SSL-Socket erzeugt werden muß, der beispielsweise mit einem bestehenden Server-Socket auf der Adresse „`141.23.40.1:433`“ kommunizieren möchte, kann das durch die folgenden Anweisungen erreichen werden:

```
NetSSL netSSL    = new NetSSL("141.23.40.1",433);
SSLSocket client = netSSL.getSSLSocket();
```

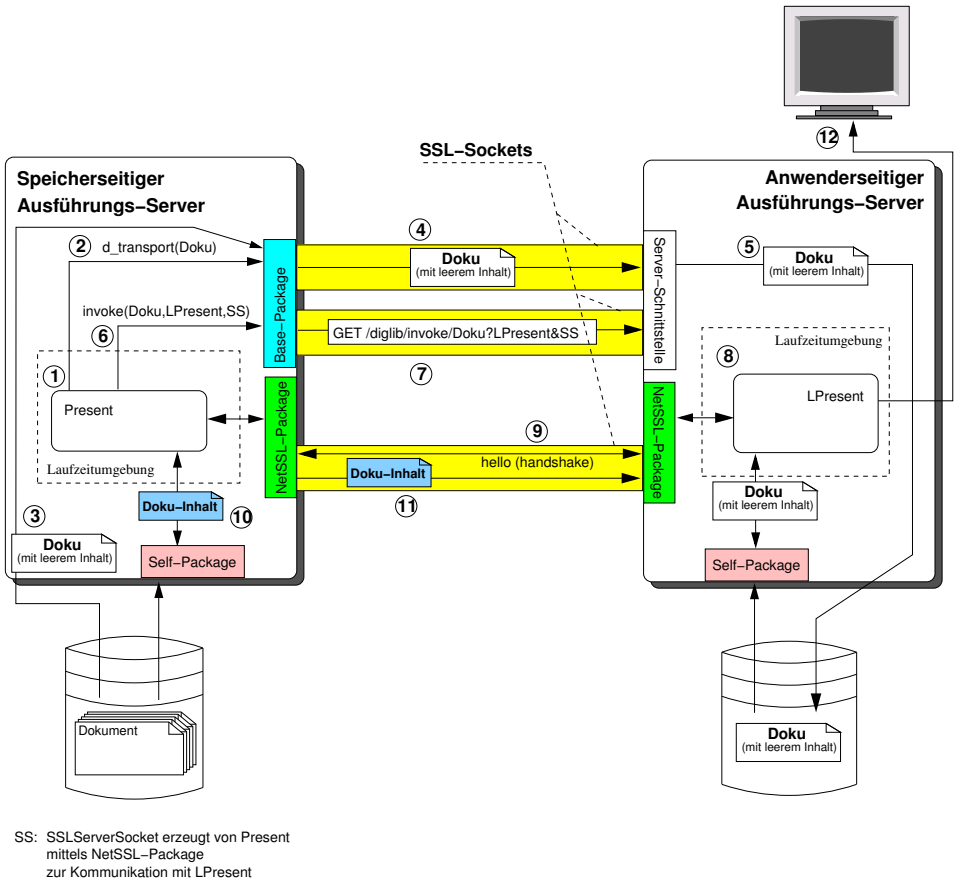

Abbildung 6.12 Online-Präsentation mit NetSSL

Die Abbildung 6.12 zeigt die modifizierte Version des Beispiels aus der Abbildung 6.11, wobei bei diesem das NetSSL-Package benutzt wird. Wie der Abbildung 6.12 zu entnehmen ist, wird bei diesem Beispiel ausschließlich über die SSL-gesicherten Kanäle – anstatt über die TCP-Sockets – kommuniziert.

Bei diesem Beispiel wird wie bei dem letzten Beispiel das Metadokument mit dem leeren Dokumentinhalt zum anwenderseitigen Server transportiert. Anschließend erzeugt die `Present`-Methode mittels NetSSL-Packages ein SSL-Server-Socket-Objekt `SS` und schickt dessen Adresse mittels der `invoke`-Methode zum anwenderseitigen Server. Dieser ruft die lokale Präsentationsmethode `LPresent` auf. Die `LPresent`-Methode erzeugt wiederum mit Hilfe des NetSSL-Packages einen SSL-Socket, über den sie mit der `Present`-Methode kommuniziert. Dieser gesicherte Kanal wird anschließend auch dazu genutzt, den Dokumentinhalt zur `LPresent`-Methode zu schicken.

Mit der Verwendung des NetSSL-Packages und des modifizierten Base-Packages kann man sogar bei einem methodenbasierten Transport – wie bei der Online-Präsentation in Abbildung 6.12 – sicher sein, daß das gesamte Metadokument, also auch der Dokumentinhalt, *geschützt* zwischen den Instanzen transportiert wird.

6.3.3 Self-Package

Das Self-Package realisiert die *Dokument-Funktionen*. Über die Methoden dieses Packages können die Dokumentmethoden beispielsweise auf den Dokumentinhalt, auf die Dokumentattribute oder auf die Methodenzuordnung eines Dokuments zugreifen. Die Modifikationen dieses Packages beziehen sich unter anderem auf die Erstellung neuer Methoden; einige der wichtigsten Methoden werden in diesem Abschnitt erläutert:

getAuthorCertificate Das Zertifikat des Autors, kodiert im Base64-Format, ist im Attribute-Abschnitt eines Dokuments gespeichert (mehr dazu siehe Abschnitt 6.2.7). Über diese Methode kann eine Methode auf dieses Zertifikat zugreifen. Die Methode liefert dabei ein Objekt vom Typ „`java.security.cert.Certificate`“.

getAuthorPublicKey Diese Methode liest das Zertifikat des Autors und extrahiert daraus den öffentlichen Schlüssel des Autors. Die `getAuthorPublicKey`-Methode liefert als Ergebnis ein „`java.security.PublicKey`“-Objekt.

verifyContentFileDigitalSignature Die digitale Signatur eines Dokumentinhalts wird unter dem Attribute-Abschnitt seines Dokuments gespeichert. Sie kann beispielsweise wie folgt aussehen:

```
...
DocumentContentDigitalSignature <MULTILINE>
-----BEGIN SIGNATURE-----
LINiL/4OT1kWFrnQZYzRFfg3TBmoUcwPlbXeb1+ZYEuFPhyytWg2GpjkZf2iei+rV4kI2uMNFuFt
EftH2BIObjx+tlTAlpuTc8882TOApEroyp4m1squjfLv6MiTnSJaT//S5OhSHt2MYRgj+zF6CUOU
8gQnBNRT916JTwcND7E=
-----END SIGNATURE-----
</MULTILINE>
DocumentContentDigitalSignatureType SHA1withRSA
...
```

Die Signatur wird nach dem Schlüsselwort `DocumentContentDigitalSignature`, eingeschlossen in die Multiline-Tags, den Attributen hinzugefügt. Auf jeden Fall muß auch der Name des Algorithmus, der zum Signieren verwendet wurde, dabeistehen; der Name wird durch das Schlüsselwort `DocumentContentDigitalSignatureType` gekennzeichnet. Die `verifyContentFileDigitalSignature`-Methode liest die digitale Signatur eines Dokumentinhalts und den öffentlichen Schlüssel des Autors. Sie entschlüsselt die digitale Signatur mittels des öffentlichen Schlüssels und vergleicht dieses Ergebnis mit dem Hashwert des Dokumentinhalts, den sie vorher unter Berücksichtigung vom `DocumentContentDigitalSignatureType` errechnet hatte. Falls diese beiden Werte übereinstimmen, ist die digitale Signatur korrekt, und die Methode liefert `true`; falls die Verifikation fehlschlägt, liefert sie `false` (mehr zu den digitalen Signaturen siehe Abschnitt 3.3.2).

Zur Erstellung der digitalen Signatur eines Dokumentinhalts kann beispielsweise das Programm *CreatDigSig* verwendet werden, das später im Abschnitt 6.4.2 beschrieben wird.

getContentFileDigitalSignatureToString Diese Methode liefert die im Attributen-Abschnitt gespeicherte digitale Signatur des Dokumentinhaltes als `String`-Objekt.

6.4 Zusätzliche Erweiterungen der Infrastruktur

Im Laufe der Implementierung wurden für die INDIGO-Infrastruktur zwei neue Clients, ein neues Programm zum Signieren von Dokumentinhalten und einige neue Metadokumente entwickelt. Sie werden in diesem Abschnitt erläutert.

6.4.1 Clients

Die zwei neu entwickelten Clients heißen *SSLClient* und *SSLClientWithClientAuth*. Mit SSLClient kann man über die SSL-gesicherten Kanäle mit INDIGO-Servern kommunizieren. Dieser Client kann aber ausschließlich für `GET`-HTTP-Anfragen verwendet werden. Dies bedeutet, daß er für alle Server-Befehle außer für den deliver-Befehl benutzt werden kann.

SSLClient verwendet bei der Kommunikation kein Zertifikat zur Authentifizierung. Somit kann man mit diesem Client nur mit solchen INDIGO-Servern kommunizieren, die im Maybe- bzw. False-Sicherheitsmodus betrieben werden. Trotzdem bietet er optional die Möglichkeit des Transports der Superuser-spezifischen Informationen zum Server.

Der Client SSLClientWithClientAuth ist eine Erweiterung des Clients SSLClient. Der Hauptunterschied zwischen diesen beiden Clients liegt darin, daß SSLClientWithClientAuth im Gegesatz zum SSLClient sich mittels eines Zertifikates ausweisen kann. Somit kann man diesen Client auch dann verwenden, wenn eine beidseitige Authentifizierung vorgeschrieben ist. Zu diesem Zweck muß er Zugriff auf einen Keystore haben. Diesen erhält der Client bei seinem Aufruf durch die `-Dindigo.keyStore`-Option. Der Aufruf dieser beiden Clients mit den entsprechenden Optionen wird im Anhang C.2 beschrieben.

6.4.2 Anwendung zur Erstellung der digitalen Signatur

Mit dem Programm „CreatDigSig“ kann man eine Datei – beispielsweise einen Dokumentinhalt – mit einem beliebigen privaten Schlüssel, der sich in einem Keystore befindet, signieren (*Erstellen von digitalen Signaturen*). Dabei wird das Programm wie folgt aufgerufen:

```
java CreatDigSig \
     "./my.keystore" "keystorePasswd" "privateKeyPasswd" "myAlias" \
     "./documentContent"
```

Das Programm erhält bei seinem Aufruf einen Keystore (in diesem Beispiel die Datei „`./my.keystore`“), die benötigten Paßwörter für diesen Keystore, den Alias-Namen des Signierers und anschließend den Namen der Datei, die signiert werden soll.

Dieses Programm kann nur mit den Keystores des Typs „JKS“ umgehen. Zum Signieren der Dateien verwendet es außerdem einen der beiden Algorithmen: Entweder den „`SHA1withRSA`“- oder den „`SHA1withDSA`“-Algorithmus, wobei die Entscheidung vom Typ des im Keystore befindlichen privaten Schlüssels abhängt. Falls der private Schlüssel für den RSA- bzw. für den DSA-Algorithmus erzeugt wurde, kommt zum Signieren automatisch der „`SHA1withRSA`“- bzw. der „`SHA1withDSA`“-Algorithmus zum Einsatz.

Das Programm liefert neben einigen Informationen die digitale Signatur der entsprechenden Datei im Base64-Format. Die Ausgabe des Programms könnte wie folgt aussehen:

```
Private Key Serial Version UID: 6034044314589513430
Private Key Format:             PKCS8
Private Key Algorithm:          RSA
Signature Algorithm:            SHA1withRSA

-----BEGIN SIGNATURE-----
LINiL/4OT1kWFrnQZYzRFfg3TBmoUcwPlbXeb1+ZYEuFPhyytWg2GpjkZf2iei+rV4kI2uMNFuFt
EftH2BIObjx+tlTAlpuTc8882TOApEroyp4m1squjfLv6MiTnSJaT//S5OhSHt2MYRgj+zF6CUOU
8gQnBNRT916JTwcND7E=
-----END SIGNATURE-----
```

6.4.3 Metadokumente

Zur Prüfung und Demonstration der erweiterten Funktionsmöglichkeit des INDIGO-Servers wurden im Laufe der Implementierung einige neue Metadokumente erstellt, wobei in diesem Zusammenhang stets die Implementierung der benötigten Dokumentmethoden im Vordergrund stand. In diesem Abschnitt werden drei der wichtigsten dieser Metadokumente vorgestellt.

6.4.3.1 NettextSSL

Das Metadokument *NettextSSL* mit den entsprechenden Präsentationsmethoden `Present` und `LPresent` wurde entwickelt, um die Funktionalität des NetSSL-Packages (siehe Abschnitt 6.3.2) zu demonstrieren[17]. Die beiden Dokumentmethoden `Present` und `LPresent` realisieren eine Online-Präsentation, die in Abbildung 6.12 dargestellt ist. Bei dieser Präsentation wird der Inhalt eines Textes über einen vom NetSSL-Package erzeugten SSL-Socket von der `Present`-Methode zur `LPresent`-Methode übermittelt.

Nachdem die Präsentationsmethode `Present` auf dem speicherseitigen Server gestartet wurde, erzeugt sie mittels NetSSL-Packages einen SSL-Server-Socket. Mit Hilfe des Base-Packages schickt sie unmittelbar danach (mittels des methodenbasierten Transports) das Metadokument mit dem leeren Inhalt zu einem anwenderseitigen Server und ruft auf dieses Dokument (auf dem anwenderseitigen Server) die lokale Präsentationsmethode `LPresent` auf. Diese Methode erzeugt auf dem anwenderseitigen Server die in Abbildung 6.13 dargestellte grafische Benutzeroberfläche und kontaktiert über einen SSL-Socket, der wieder mittels des NetSSL-Packages erzeugt wird, die `Present`-Methode bzw. den von ihr erzeugten SSL-Server-Socket. Über diesen gemeinsamen SSL-Socket verläuft anschließend der tatsächliche Transport des Dokumentinhalts. Über diesen Socket wartet auch die Dokumentmethode `Present` auf die Interaktion des Anwenders bei `LPresent`: Falls der Anwender auf den „more“-Knopf der grafischen Benutzeroberfläche drückt, übermittelt

[17] Diese Dokumentmethoden sind den Dokumentmethoden des bereits existierenden Metadokuments *Nettext* sehr ähnlich; der Hauptunterschied liegt darin, daß bei NettextSSL – zur Online-Präsentation – sogar der Dokumentinhalt über SSL-gesicherte Kanäle transportiert wird.

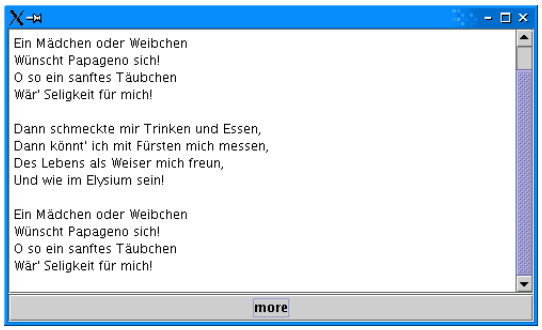

Abbildung 6.13 Benutzeroberfläche für die lokale Präsentation des NettextSSL-Metadokuments

`LPresent` der `Present`-Methode über den SSL-Socket eine Meldung, worauf sie wiederum der `LPresent`-Methode über diesen Socket eine Zeile des Dokumentinhalts schickt[18].

6.4.3.2 NettextSSLJar

Das Metadokument *NettextSSLJar* ist weitgehend identisch mit NettextSSL. Bei diesem Metadokument sind die verwendeten Dokumentmethoden lediglich vom Autor des Dokuments signiert. Dieses Metadokument wurde entwickelt, um die im Abschnitt 6.2.7 beschriebenen Verifikationsmechanismen prüfen zu können. Zum Signieren der Methoden wird der *jarsigner* verwendet, der ebenfalls im Abschnitt 6.2.7 erläutert wurde.

Zusätzlich zur Signatur der Dokumentmethoden beinhaltet dieses Metadokument eine weitere Modifikation: Die Dokumentattribute dieses Dokuments wurden um zwei Attribute, `AuthorCertificate` und `AuthorCertificateType`, ergänzt.

6.4.3.3 NettextSSLJarDocuSig

Dieses Metadokument wurde hauptsächlich zur Demonstration der Mechanismen zur Verifikation der digitalen Signatur der Dokumentinhalte entwickelt. Bei dem Inhalt dieses Metadokuments handelt es sich um ein Text-Dokument. Dieses Metadokument bietet wie die letzten beiden Metadokumente zwei Methoden: die Präsentationsmethode `Present` und die `LPresent`-Methode zur lokalen Präsentation. Obwohl es sich auch bei diesen Methoden um Präsentationsmethoden handelt und diese außerdem ebenfalls signiert sind, unterscheiden sie sich gravierend von denen der letzten beiden Metadokumente. Um eine Verifikation des Dokumentinhalts zu ermöglichen, benutzt die Präsentationsmethode einen Server-basierten Transport. Somit wird mittels des Base-Packages das gesamte Metadokument – und somit auch der Dokumentinhalt – gleichzeitig zu einem anwenderseitigen Server transportiert.

Nach dem erfolgreichen Transport des Metadokuments wird auf diese Kopie des Dokuments die lokale Präsentationsmethode aufgerufen. Während der Ausführung dieser Methode erscheint auf der Anwenderseite die in Abbildung 6.14 dargestellte grafische Benutzeroberfläche der `LPresent`-Methode.

[18] Die `Present`-Methode schickt den Dokumentinhalt zeilenweise, wobei sie der `LPresent`-Methode nach jedem Klick nur eine Zeile sendet.

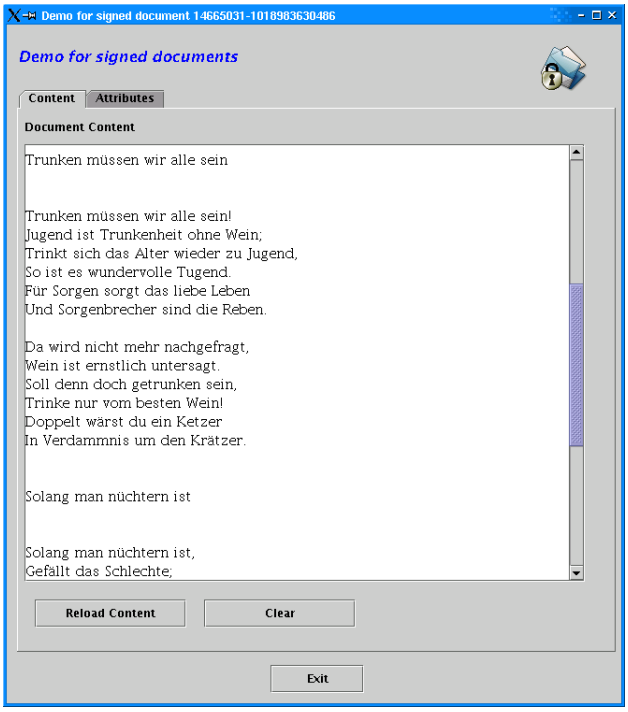

Abbildung 6.14 Content-Panel der Benutzeroberfläche für die lokale Präsentation

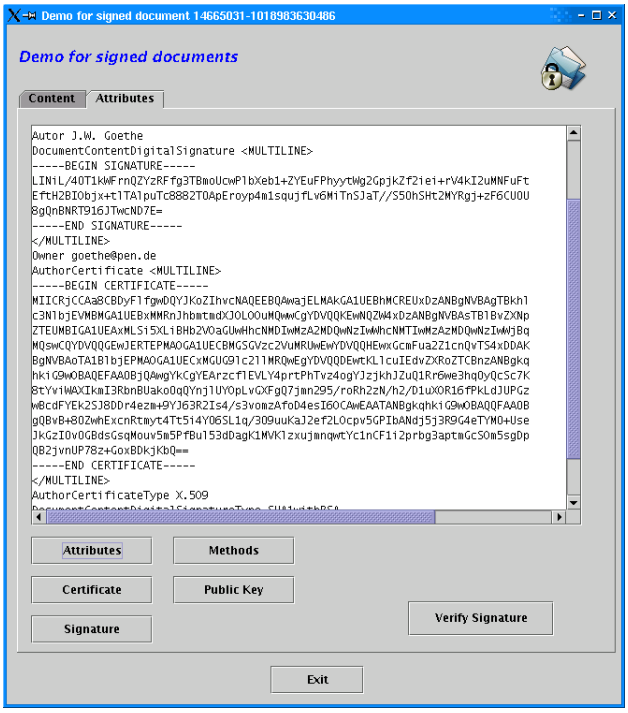

Abbildung 6.15 Attribute-Panel der Benutzeroberfläche

Über diese grafische Oberfläche – im Panel „Content“ – kann man den Inhalt des Dokuments anzeigen lassen, während im Panel „Attributes“ die Möglichkeit zur Anzeige der dokumentspezifischen Metadaten – wie beispielsweise das Zertifikat des Autors – besteht (siehe Abbildung 6.15).

In dem Attributes-Panel befindet sich auch der Knopf „Verify Signature“, über den die digitale Signatur des Dokumentinhalts geprüft wird. Um überhaupt eine solche Verifikation der digitalen Signatur des Inhalts durchführen zu können, beinhaltet das Dokument in seinem Attribute-Abschnitt die folgenden Attribute, deren Bedeutung in den vorhergehenden Abschnitten bereits beschrieben wurde:

- `AuthorCertificate`,
- `AuthorCertificateType`,
- `DocumentContentDigitalSignature`,
- `DocumentContentDigitalSignatureType`.

Zur Verifikation des Dokumentinhalts verwendet `LPresent` die im Abschnitt 6.3.3 erläuterte statische Methode `verifyContentFileDigitalSignature` des Self-Packages. Dies geschieht wie folgt:

```
if (Self.verifyContentFileDigitalSignature()) {
    // Die Signatur ist korrekt!
    // Gib OK aus.
    ...
}
else {
    // Die Signatur ist fehlerhaft!
    // Gib Fehlermeldung aus.
    ...
}
```

Falls die Verifikation des Dokumentinhalts erfolgreich verläuft, erhält der Anwender die Meldung, die in Abbildung 6.16 dargestellt ist.

Die in Abbildung 6.17 dargestellte Fehlermeldung erhält man dagegen nur dann, wenn die digitale Signatur des Dokumentinhalt nicht korrekt ist.

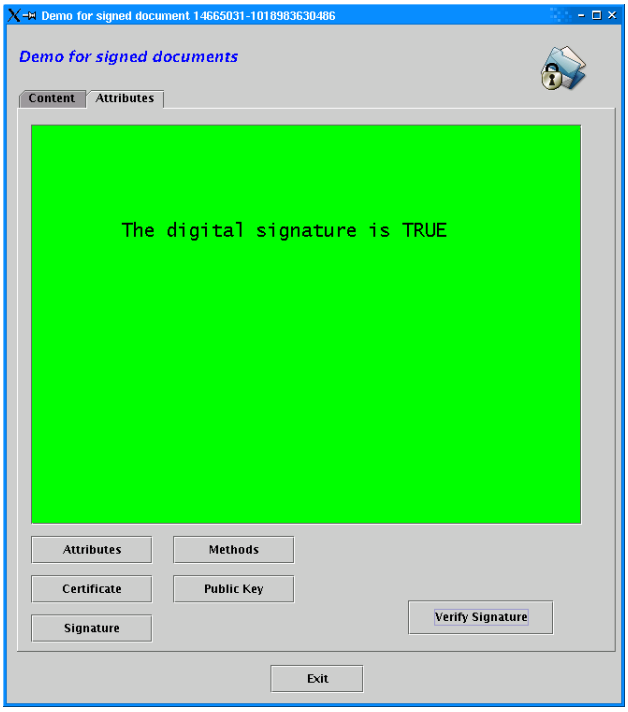

Abbildung 6.16 Verifikation des Dokumentinhalts bei nettextSSLJarDocuSig

Abbildung 6.17 Fehlermeldung bei der Verifikation des Dokumentinhalts

Kapitel 7

Zusammenfassung und Ausblick

Zusammenfassung

Angesichts der überragenden Bedeutung der modernen Kommunikationstechnik in allen Lebensbereichen kommt auch den digitalen Bibliotheken ein wachsendes Gewicht zu. Dabei spielen nicht nur die platzsparende Speicherung, sondern auch die schnelle Datenübermittlung und der unmittelbare Zugang zu den Dokumenten eine wichtige Rolle. Da eine solche Bibliothek über ein offenes Netz betrieben wird, erhalten in diesem Zusammenhang Sicherheitsaspekte ein essentielles Gewicht. Die vorliegende Diplomarbeit geht diesen Fragen nach und zeigt Wege auf, wie die bestehenden Sicherheitsrisiken minimiert werden können.

Ziel dieser Arbeit war daher der Entwurf und die Realisierung von Sicherheitsmechanismen für eine Infrastruktur für digitale Bibliotheken. Dabei wurde speziell auf die INDIGO-Infrastruktur eingegangen; sie stellt eine verteilte Infrastruktur für digitale Bibliotheken dar.

Der erste Teil dieser Diplomarbeit enthält eine Einführung in die Grundlagen der INDIGO-Infrastruktur und der Sicherheit. In Kapitel 2 wurden die INDIGO-Infrastruktur und ihre Komponenten erläutert; in Kapitel 3 folgte anschließend die Beschreibung einiger kryptographischer Verfahren und Sicherheitsprotokolle.

Im zweiten Teil dieser Arbeit wurden Sicherheitsmechanismen für die INDIGO-Infrastruktur entworfen. In dieser Entwurfsphase erfolgte zunächst in Kapitel 4 die Sicherheitsanalyse der Infrastruktur. Basierend auf dieser Analyse wurden in Kapitel 5 Sicherheitskonzepte für diese Infrastruktur entwickelt. Während der gesamten Entwurfsphase standen die Sicherheitsanforderungen Vertraulichkeit, Authentizität, Integrität, Verbindlichkeit und die Autorität stets im Mittelpunkt des Interesses.

Im dritten und letzten Teil der Arbeit wurden die Sicherheitsmechanismen für die INDIGO-Infrastruktur realisiert. Dabei wurden die in Abschnitt 5.4 beschriebenen Sicherheitsrichtlinien der Infrastruktur implementiert. Die Beschreibung der Implementierung erfolgte in Kapitel 6.

Die wichtigsten Modifikationen des INDIGO-Servers betrafen folgende Punkte:

- Sicherung und Aufbau der verbindlichen Kommunikationskanäle durch den Einsatz von SSL- bzw. TLS-basierten Server-zu-Server Verfahren.

- Realisierung von Sicherheitsmechanismen zur Verifikation der digital signierten Dokumente und Dokumentmethoden.
- Erweiterung des INDIGO-Servers um feingranuliert konfigurierbare Zugriffsmechanismen, die verteilt auf drei unterschiedliche Ebenen den Zugriff der Anwender (bzw. Dokumentmethoden) auf seine Ressourcen kontrollieren.

Neben den Modifikationen des INDIGO-Servers wurden zwei neue Clients zur Kommunikation mit dem INDIGO-Server und eine Anwendung zur Erzeugung der digitalen Signatur der Dokumente entwickelt. Ferner wurden einige neue Metadokumente und Dokumentmethoden erstellt, um die neuen Eigenschaften der Infrastruktur zu demonstrieren.

Bei der Realisierung der Sicherheitsmechanismen wurde größter Wert auf die Abwärtskompatibilität, Konfigurierbarkeit und Modularität gelegt. Die Abwärtskompatibilität zur ursprünglichen Infrastruktur wird beispielsweise erreicht, indem die bereits existierenden Metadokumente und Dokumentmethoden bei dem modifizierten Server auch verwendet werden können. Diese müssen – falls nötig – minimal um die digitale Signatur der Autoren ergänzt werden.

Das Sicherheitsverhalten des INDIGO-Servers läßt sich beliebig über seine Konfigurationsdatei ändern (Konfigurierbarkeit). Alle wichtigen Sicherheitsmechanismen des modifizierten Servers lassen sich den Wünschen des Betreibers anpassen. Dadurch ist sichergestellt, daß jeder Betreiber den Server seinen jeweiligen Sicherheitsbedürfnissen entsprechend betreiben kann. Der Betreiber kann beispielsweise über die Einstellung seiner Konfigurationsdatei bestimmen, ob die Clients sich bei der Kommunikation mit seinem Server identifizieren müssen. Zudem kann er beispielsweise festlegen, ob die Dokumentmethoden, die keine korrekte digitale Signatur besitzen, ausgeführt werden dürfen oder nicht. Die Konfigurierbarkeit des Servers hinsichtlich der Sicherheitsmechanismen geht sogar so weit, daß man den Server im Normalmodus betreiben kann; in diesem Modus sind alle Sicherheitsmechanismen des Servers ausgeschaltet.

Die Modularität hinsichlich der Sicherheitsmechanismen wurde bei der Implementierung durch die Verteilung dieser Mechanismen auf die unterschiedlichen und eigenständigen Klassen erzielt, die jeweils eine wohldefinierte Eigenschaft und Aufgabe besitzen. Diese Vorgehensweise führt dazu, daß bei einer Weiterentwicklung des Servers um neue Sicherheitsdienste nur die wenigen betroffenen Klassen modifiziert werden müssen, ohne daß der gesamte Server davon betroffen ist. So kann der INDIGO-Server beispielsweise um den Authentisierungsdienst Kerberos [Stei88] erweitert werden, in dem nur die entsprechende Authentisierungsklasse des Servers (`IndigoAuthorization`-Klasse) ergänzt wird.

Ausblick

Die modifizierte INDIGO-Infrastruktur verfügt über einen feingranulierten Zugriffsmechanismus. Dies wird vor allem auf der Dokumentebene – beispielsweise durch den Einsatz der dokumentspezifischen Autorisation – erreicht, und betrifft nur den Server-Befehl „invoke“ (somit das Ausführen der Dokumentmethoden). Um eine feingranulierte Zugriffssteuerung bezüglich der weiteren Server-Befehle zu erzielen, muß man die Authentisierungsmöglichkeit der Anwender auf der Protokollebene ausbauen. Hierbei würde die Integration eines Authentisierungsdienstes, wie beispielsweise Kerberos, in die Protokollebene dienlich sein.

Eine weitere mögliche sicherheitsrelevante Erweiterung des Servers wäre die Ergänzung der Verifikationsmechanismen hinsichtlich der Dokumentmethoden, so daß der Server auch Verfahren zur indirekten Ende-zu-Ende Authentizität (vgl. Seite 5.2.2) unterstützen würde. Zusätzlich könnte der Server um Sicherheitsmechanismen zur Durchsetzung der Urheberrechte erweitert werden. Dies setzt aber die Existenz von sicheren und nichtproprietären Verfahren zum Urheberschutz voraus.

Ferner wird empfohlen, die Metadokumente um einen weiteren Abschnitt, der die digitale Signatur des gesamten Metadokuments beinhaltet, zu erweitern. Bei der modifizierten INDIGO-Infrastruktur wird lediglich der Dokumentinhalt, aber nicht der Methodenzuordnungsabschnitt oder die Attribute des Metadokuments signiert. In diesem Zusammenhang wäre sogar eine Umstellung der gesamten Metadokumente von MIME auf S/MIME [SMIME] oder auf XML [XML] ratsam, denn diese unterstützen implizit viele der gewünschten Sicherheitsmechanismen, wie beispielsweise Mechanismen zur Durchsetzung der Vertraulichkeit und der Authentizität (vgl. [RFC2632, RFC2633] und [XMLSig, RFC3275]). Diese Umstellung würde auf jeden Fall eine viel tiefergehende Modifikation, gar eine Reimplementierung der vorhanden Infrastruktur notwendig machen.

Anhang A

Ergänzung der Sicherheitsgrundlagen

A.1 Beispiel für das RSA-Verfahren

Um mit dem RSA-Verfahren einen Klartext zu verschlüsseln, muß man zunächst ein Schlüsselpaar erzeugen: Den *privaten* Schlüssel, mit dem der Klartext verschlüsselt wird, und den *öffentlichen* Schlüssel, mit dem das Chiffrat entschlüsselt wird. Zu diesem Zweck wird zuerst den Variablen p und q jeweils eine Primzahl zugewiesen und aus diesen beiden Zahlen anschließend n errechnet[1]:

$$p = 47 \qquad und \qquad q = 71$$

$$n = p * q = 47 * 71 = 3337$$

Daraus resultiert für die Eulerische Funktion $n\phi(n)$:

$$\phi(n) = \phi(p * q) = (p - 1) * (q - 1) = 3220$$

Abhängig von dieser Zahl $\phi(n)$ wird anschließend eine zufällige Zahl e mit $\phi(n) > e > 1$ gewählt, wobei e zu der Zahl $\phi(n)$ teilerfremd sein muß:

$$e = 79 \quad \wedge \quad ggT(79, 3220) = 1$$

Aus e und $\phi(n)$ kann man schließlich d ausrechnen:

$$de = 1(mod\ \phi(n)) \qquad \Leftrightarrow \qquad d = 79^{-1}(mod\ 3220) = 1019$$

Somit hat man die beiden Schlüssel:

$$\begin{array}{rcccc} \text{Privater Schlüssel} & = & (d, n) & = & (1019, 3337) \\ \text{Öffentlicher Schlüssel} & = & (e, n) & = & (79, 3337) \end{array}$$

[1] Um dieses Beispiel verständlicher zu machen, werden die Belegungen der Variablen p und q absichtlich so klein gewählt. Dieses Beispiel stammt von [Lipp00].

Angenommen ein Klartext $P = 6882326879666683$ wird mit Hilfe dieser Schlüssel verschlüsselt. Zu diesem Zweck muß der Klartext in „N-1“-stellige Blöcke P_i zerteilt werden, wobei $N = 4$ die Länge von n ist:

$$P = P_1P_2P_3P_4P_5P_6 \quad \wedge$$

$$P_1 = 688 \quad \wedge \quad P_2 = 232 \quad \wedge \quad P_3 = 687 \quad \wedge \quad P_4 = 966 \quad \wedge \quad P_5 = 668 \quad \wedge \quad P_6 = 003$$

Das Teilchiffrat C_1 kann mit Hilfe des öffentlichen Schlüssels wie folgt berechnet werden:

$$C_1 \quad = \quad (P_1)^e (mod\ n) \quad = \quad (688)^{79} (mod\ 3337) \quad = \quad 1570$$

Durch die Anwendung des RSA-Verfahrens auf den Rest des Klartextes erhält man:

$$C \quad = \quad 1570\ 2756\ 2091\ 2276\ 2423\ 158$$

Falls man dieses Chiffrat entschlüsseln möchte, verwendet man den privaten Schlüssel. Man kann beispielsweise das Teilchiffrat C_1 wie folgt entschlüsseln:

$$P_1 \quad = \quad (C_1)^d (mod\ n) \quad = \quad (1570)^{1019} (mod\ 3337) \quad = \quad 688$$

A.2 Jarsigner

Mit Hilfe von `jarsigner` kann man Dateien signieren; diese müssen lediglich als Jar-Archiv vorliegen. Mit dem `jarsigner` kann man beispielsweise eine Datei „`file.jar`“ wie folgt signieren:

```
jarsigner -keystore ./Autor.keystore -storepass "StorePassword" \
          -keypass "PKPassword" ./file.jar "goethe"
```

Dabei wird dem `jarsigner`-Programm ein Keystore (`Autor.keystore`) übergeben, der den privaten Schlüssel des Signierers beinhaltet. Ein Keystore funktioniert wie ein Schlüsselbund und kann mehrere Schlüssel von unterschiedlichen Personen beinhalten. Aus diesem Grund muß beim Signieren auch der Alias-Name (in diesem Beispiel „`goethe`“) des Signierers dem Programm `jarsigner` übergeben werden. Da ein privater Schlüssel verschlüsselt im Keystore gespeichert wird, muß man dem `jarsigner` über die `storepass`- und `keypass`-Option den erforderlichen Schlüssel mitteilen (Näheres zu diesen beiden Optionen siehe Abschnitt 6.1.2.3).

Bei diesem Vorgang wird die digitale Signatur direkt in die Jar-Datei eingefügt. Falls man die ursprüngliche Datei nicht überschreiben möchte, kann man über die Option „`-signedjar <file>`“ einen Dateinamen angeben, unter dem die neue Datei gespeichert werden soll.

A.3 OpenSSL

OpenSSL ist eine freie Implementierung der Protokolle *Secure Sockets Layer (SSL)* und *Transport Layer Security (TLS)*. Das Software-Paket bietet zusätzlich dazu weitere Werkzeuge zum Verschlüsseln, zum Signieren sowie zur Zertifikat-Erzeugung und Verwaltung an. Eine kurze Liste der wichtigsten unterstützten kryptographischen Algorithmen und der Protokolle findet sich in der Tabelle A.1:

Kategorien	**Beispiele für enthaltene Verfahren**
Symmetrische Verfahren	Blowfish, CAST, CAST5, DES, Triple-DES, IDEA, RC2, RC4 und RC5
Asymmetrische Verfahren	RSA, DSA und Diffie-Hellman
Einweg-Hashfunktionen	MD2, MD4, MD5, MDC2, RMD-160, SHA und SHA-1
Weitere Features	Base64, S/MIME, PKCS#7, PKCS#8, PKCS#12, Erzeugung und Umgang mit X.509-Zertifikaten, X.509 Certificate Revocation List (CRL) Management und X.509 Certificate Signing Request (CSR) Management

Tabelle A.1 Wichtige Features von OpenSSL

Das OpenSSL-Paket basiert auf dem *SSLeay-Paket*. SSLeay wurde von Eric A. Young und Tim J. Hudson entwickelt; es wird aber nicht mehr weiterentwickelt. Das OpenSSL-Paket kann von der folgenden Adresse geladen werden:

`http://www.openssl.org`

Auf dieser Seite findet man auch sehr hilfreiche Dokumentation für dieses Software-Paket.

Das bei dieser Arbeit verwendete OpenSSL-Paket trägt die Versionsnummer „0.9.6a“. OpenSSL wurde hauptsächlich beim Erzeugen der X.509-basierten CA-Zertifikate und beim Signieren der Zertifikate der Anwender von diesen CAs eingesetzt.

A.4 Zertifikate nach X.509

Ein Zertifikat nach X.509-Verzeichnisprüfungsdienst besteht aus mehreren Feldern. Diese Felder sind in Abbildung A.1 dargestellt (die Abbildung stammt aus [Sta00]).

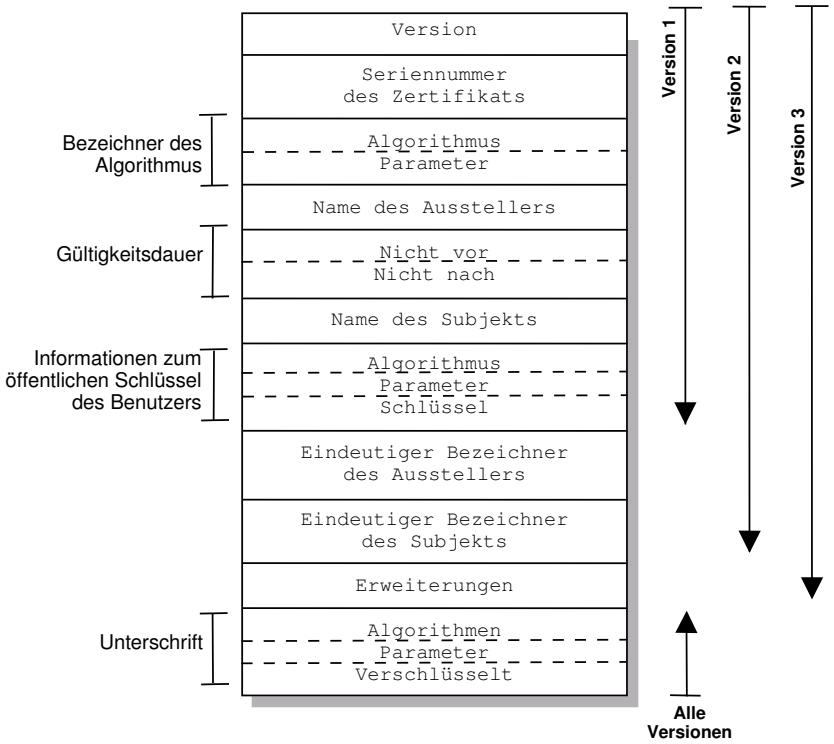

Abbildung A.1 Zertifikat nach X.509

Das Feld „Version“ bestimmt die Versionsnummer des Zertifikates. Je nach Versionsnummer ändern sich auch Inhalt und Anzahl der Felder des Zertifikates. Die „Seriennummer“ verknüpft mit dem Ausstellernamen ist ein eindeutiges Kennzeichen für jedes Zertifikat und bezieht sich nur auf ein einziges Zertifikat. Der „Name der Aussteller“ bezieht sich auf den Namen der Zertifizierungsstelle (CA), die das Zertifikat erzeugt und signiert hat. Der „Name des Subjekts“ bezieht sich hingegen auf den Benutzer, der der Eigentümer des Zertifikats ist. Um eine eindeutige Bezeichnung dieser beiden Akteure zu erreichen, beinhalten die Zertifikate seit der zweiten Version noch zusätzlich zwei weitere Felder: „Eindeutiger Bezeichner des Ausstellers“ und „Eindeutiger Bezeichner des Subjekts“.

Das Zertifikat besitzt außerdem ein Feld, das das Gültigkeitsintervall des Zertifikates angibt. Außerdem informiert es über die Namen der Algorithmen, mit denen der öffentliche Schlüssel des Eigentümers signiert wurde. Zu guter Letzt beinhaltet das Zertifikat ein Feld, das alle anderen Felder des Zertifikats abdeckt. Dieses stellt eine digitale Signatur der anderen Felder dar, die von der CA erzeugt und dem Zertifikat angehängt wird.

Ein mittels *OpenSSL* nach X.509-Standard erzeugtes Zertifikat sieht im Klartext beispielsweise wie folgt aus:

```
Certificate:
   Data:
      Version: 3 (0x2)
      Serial Number: 259 (0x103)
      Signature Algorithm: md5WithRSAEncryption
      Issuer: C=DE, ST=Hessen, L=Frankfurt/M., O=J.W. Goethe-Universitaet, \
              OU=Userverwaltung der Universitaetsbibliotheken, \
              CN=Herr der User/Email=herr.der.user@uni-frankfurt.de
      Validity
         Not Before: Feb 18 16:08:45 2002 GMT
         Not After : Feb 18 16:08:45 2003 GMT
      Subject: C=DE, ST=Hessen, O=J.W. Goethe-Universitaet, \
               OU=User der Universitaetsbibliotheken, CN=Razi Lotfi
      Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
         RSA Public Key: (1024 bit)
            Modulus (1024 bit):
               00:df:4d:3b:b5:d9:6b:ad:ac:71:85:e7:c4:34:98:
               d8:aa:54:0a:31:9e:fb:db:85:80:e3:2d:81:d6:73:
               c2:04:5e:25:a2:ef:b4:58:c6:88:7f:9c:6d:05:97:
               85:f7:bc:70:a4:0a:95:98:2c:2a:96:48:0d:07:0a:
               49:f6:f0:63:8d:35:68:33:d5:78:93:40:04:ef:9a:
               b1:a4:c6:02:fd:fa:8d:d9:b0:aa:ee:5a:de:60:f4:
               99:14:7b:8a:d8:26:f8:f7:8e:6e:c5:78:c9:84:f8:
               bb:22:bc:6d:3a:65:c1:35:39:1a:88:8f:65:a3:81:
               da:b6:a2:e7:66:0c:58:6c:e5
            Exponent: 65537 (0x10001)
      X509v3 extensions:
         X509v3 Basic Constraints:
            CA:FALSE
         Netscape Comment:
            OpenSSL Generated Certificate
         X509v3 Subject Key Identifier:
            1A:90:2C:6E:88:D0:9F:52:97:3C:9F:7D:35:0C:97:44:4D:EF:11:BB
         X509v3 Authority Key Identifier:
            keyid:5B:9D:C5:B4:96:39:CE:79:66:58:6B:DD:62:32:3B:E3:BF:A9:F9:20
            DirName:/C=DE/ST=Hessen/L=Frankfurt/M./O=J.W. Goethe-Universitaet/\
                     OU=Userverwaltung der Universitaetsbibliotheken/\
                     CN=Herr der User/Email=herr.der.user@uni-frankfurt.de
            serial:00

   Signature Algorithm: md5WithRSAEncryption
      66:dd:f1:b4:0b:74:a5:70:89:6a:34:cb:a8:56:70:b0:31:b4:
      9b:12:04:e9:a1:e9:d2:77:cc:8b:d9:d6:b6:69:3b:96:bb:e1:
      91:04:e0:b9:d2:8f:cc:36:aa:ee:94:82:0b:d0:33:e6:47:41:
      c8:66:df:57:1f:06:e4:fe:46:6b:44:7f:0c:8a:18:9b:e1:28:
      bd:18:07:bd:df:b4:fe:cc:a9:f2:85:b7:03:db:9e:bb:de:9b:
      8a:dc:4c:1a:4c:7a:8c:6e:ca:f4:c0:5c:20:a5:4f:05:92:09:
      06:22:2c:c6:96:73:1a:03:0f:54:29:05:2a:38:0d:c8:f2:de:
      ca:db:2e:22:a2:a7:f9:79:a0:ec:97:84:17:eb:d4:13:bd:33:
      98:dc:5c:d1:e8:b6:25:d1:72:aa:92:de:8d:81:a3:43:df:b1:
      54:5f:ba:e9:f6:8a:bd:d9:7f:9a:2d:d9:9e:ca:1f:57:d3:ab:
      2e:1c:eb:aa:b9:14:a9:08:bb:69:ef:7b:c5:75:e1:06:56:31:
      91:66:d8:f8:7b:c1:fb:df:fd:f1:b7:f8:a0:ca:14:cf:7c:5c:
      63:20:1f:85:42:32:1d:45:33:97:79:2c:24:d7:e7:b0:af:0b:
      58:d9:18:52:72:34:96:99:ae:5e:b6:00:9c:fd:aa:53:ec:75:
      53:d3:8f:d9
-----BEGIN CERTIFICATE-----
MIIEvjCCA6agAwIBAgICAQMwDQYJKoZIhvcNAQEEBQAwgdYxCzAJBgNVBAYTAkRF
```

MQ8wDQYDVQQIEwZIZXNzZW4xFTATBgNVBAcTDEZyYW5rZnVydC9NLjEhMB8GA1UE
ChMYSi5XLiBHb2V0aGUtVW5pdmVyc2l0YWV0MTUwMwYDVQQLEyxVc2VydmVyd2Fs
dHVuZyBkZXIgVW5pdmVyc2l0YWV0c2JpYmxpb3RoZWtlbjEWMBQGA1UEAxMNSGVy
ciBkZXIgVXNlcjEtMCsGCSqGSIb3DQEJARYeaGVyci5kZXIudXNlckB1bmktZnJh
bmtmdXJ0LmRlMB4XDTAyMDIxODE2MDg0NVoXDTAzMDIxODE2MDg0NVowgYkxCzAJ
BgNVBAYTAkRFMQ8wDQYDVQQIEwZIZXNzZW4xITAfBgNVBAoTGEouVy4gR29ldGhl
LVVuaXZlcnNpdGFldDErMCkGA1UECxMiVXNlciBkZXIgVW5pdmVyc2l0YWV0c2Jp
Ymxpb3RoZWtlbjEZMBcGA1UEAxMQQ2hyaXN0aWFuIE1vZW5jaDCBnzANBgkqhkiG
9w0BAQEFAAOBjQAwgYkCgYEA3007tdlrraxxhefENJjYqlQKMZ7724WA4y2B1nPC
BF4lou+0WMaIf5xtBZeF97xwpAqVmCwqlkgNBwpJ9vBjjTVoM9V4k0AE75qxpMYC
/fqN2bCq7lreYPSZFHuK2Cb4945uxXjJhPi7IrxtOmXBNTkaiI9lo4HatqLnZgxY
bOUCAwEAAaOCAWMwggFfMAkGA1UdEwQCMAAwLAYJYIZIAYb4QgENBB8WHU9wZW5T
U0wgR2VuZXJhdGVkIENlcnRpZmljYXRlMB0GA1UdDgQWBBQakCxuiNCfUpc8n301
DJdETe8RuzCCAQMGA1UdIwSB+zCB+IAUW53FtJY5znlmWGvdYjI747+p+SChgdyk
gdkwgdYxCzAJBgNVBAYTAkRFMQ8wDQYDVQQIEwZIZXNzZW4xFTATBgNVBAcTDEZy
YW5rZnVydC9NLjEhMB8GA1UEChMYSi5XLiBHb2V0aGUtVW5pdmVyc2l0YWV0MTUw
MwYDVQQLEyxVc2VydmVyd2FsdHVuZyBkZXIgVW5pdmVyc2l0YWV0c2JpYmxpb3Ro
ZWtlbjEWMBQGA1UEAxMNSGVyciBkZXIgVXNlcjEtMCsGCSqGSIb3DQEJARYeaGVy
ci5kZXIudXNlckB1bmktZnJhbmtmdXJ0LmRlggEAMA0GCSqGSIb3DQEBBAUAA4IB
AQBm3fG0C3SlcIlqNMuoVnCwMbSbEgTpoenSd8yL2da2aTuWu+GRBOC50o/MNqru
lIIL0DPmR0HIZt9XHwbk/kZrRH8Mihib4Si9GAe937T+zKnyhbcD25673puK3Ewa
THqMbsr0wFwgpU8FkgkGIizGlnMaAw9UKQUqOA3I8t7K2y4ioqf5eaDsl4QX69QT
vTOY3FzR6LYl0XKqkt6NgaND37FUX7rp9oq92X+aLdmeyh9X06suHOuquRSpCLtp
73vFdeEGVjGRZtj4e8H73/3xt/igyhTPfFxjIB+FQjIdRTOXeSwk1+ewrwtY2RhS
cjSWma5etgCc/apT7HVT04/Z
-----END CERTIFICATE-----

Eine detailliertere Beschreibung der X.509-Zertifikate findet man beispielsweise in [Sta00].

Anhang B

Weitere Sicherheitsmaßnahmen für die INDIGO-Infrastruktur

B.1 Schutz vor dem böswilligen Host-Rechner

Der Schutz des INDIGO-Ausführungs-Servers vor dem Host-Rechner ist nur bei einem Multi-User-System von Bedeutung; bei einem Single-User-System – wie beispielsweise „DOS“ – ist dies ohnehin irrelevant. Der Schutz vor dem Host-Rechner beinhaltet den Schutz des INDIGO-Servers vor den anderen Anwendern (bzw. Anwendungen).

Bei den sicheren Betriebssystemen – wie zum Beispiel den UNIX-Systemen – existieren Mechanismen der *Rechtevergabe*. Mit ihrer Hilfe ist man in der Lage, gewisse Bereiche und Verzeichnisse für die unbefugten Anwender zu sperren, so daß diese Personen keine lesenden oder schreibenden Zugriffsrechte auf diese Bereiche haben. Der Nachteil dieser Systeme liegt darin, daß der Superuser (also Administrator) trotzdem Zugriff auf diese Bereiche hat. Für ihn gelten keine Einschränkungen von Zugriffsrechten. Diese erweiterten Zugriffsrechte für den Superuser sind auch der Grund dafür, daß die meisten Hacker-Angriffe sich auf das Knacken des Superuser-Paßworts konzentrieren. Sicherheitslücken in UNIX-Systemen beruhen meist darauf, daß jemand unbefugt Superuser werden kann [Wobst98].

Um die Daten bei den digitalen Bibliotheken vor dem Host-Rechner zu schützen, müssen diese Daten verschlüsselt auf dem Host-Rechner gespeichert werden. Zu diesem Zweck kann man das *Krypto-Filesystem* verwenden. Bei der Verwendung dieses Systems wird die Verschlüsselung und Entschlüsselung der Datei bei der Speicherung völlig aus dem INDIGO-Server ausgegliedert. Hier wird diese Aufgabe von dem Betriebssystem und dem Filesystem übernommen. Der Zugriff auf die Dateien erfolgt bei den Krypto-Filesystemen scheinbar wie immer. Die Anwendung merkt von den Veränderungen nichts, jedoch werden die Daten auf dem Weg zwischen den Platten und der Anwendung ver- bzw. entschlüsselt.

Ein Beispiel für ein Krypto-Filesystem ist *CFS*, das *Cryptographic File System for UNIX* [CFS93]. Hier ist die Verschlüsselung in einem NFS-Dämon eingebaut. Der Systemaufruf der Dateisystem-Schnittstelle wird über einen NFS-Dämon geleitet, der die Verschlüsselung vornimmt. Dies gilt auch bei lokalen Dateizugriffen. Weder für die Anwendung noch für den Anwender ändert sich dadurch etwas. Die verschlüsselten Verzeichnisse müssen lediglich vor der Benutzung mit dem *mount*-Kommando zugänglich gemacht werden; dabei wird ein Paßwort abgefragt, das in den kryptographischen Schlüssel umgewandelt

wird. Zur Verschlüsselung wird eine Kombination aus Quasi-Stromchiffrierung und der Blockchiffrierung eingesetzt. Diese Kombination ist effektiv und bietet einen wahlfreien[1] Zugriff (*random access*) auf die chiffrierten Dateien. Das System ist verträglich mit der Standard-UNIX-Vernetzung. Es bietet weiterhin den Vorteil, daß auch Dateinamen geschützt sind. Durch die Verwendung des Krypto-Filesystems können unbefugte Personen außerdem durch das Ausbauen und Analysieren der Festplatte wenig Nutzen ziehen[2].

Dieses Verfahren hat in diesem Kontext wie alle anderen Verfahren aber auch Nachteile und Schwachpunkte. Die Verwendung dieses Systems ergibt nur einen Sinn, wenn der Benutzer seiner eigenen Maschine trauen kann. Falls auf dem System eine manipulierte Software läuft, die das Paßwort beim mount-Kommando ausspioniert, ist die ganze Mühe natürlich umsonst. Dieses Verfahren bietet keine hundertprozentige Sicherheit; es erhöht lediglich die Sicherheit bei der Speicherung der Daten. Sobald ein Angreifer sich administrative Rechte erkämpft hat, erhält er eine fast unbeschränkte Macht. Der Superuser kann zum Beispiel den Speicher und den Swap-Bereich[3] lesen. In dem Augenblick, in dem ein Dokumentbereich oder sogar ein ganzes Dokument entschlüsselt auf dem Speicher vorliegt, kann der Angreifer natürlich auf diese Daten zugreifen. Mit Hilfe der obigen Verfahren kann der Zugriff auf die Daten auf jeden Fall erschwert werden. Trotzdem ist der Schutz vor dem Host-Rechner nur sehr bedingt garantiert.

[1] Die meisten kryptographischen Algorithmen arbeiten sequentiell. Dies stellt zwar bei der Verschlüsselung einzelner Dateien kein Problem dar, aber bei der Plattenverschlüsselung sollte ein Programm ohne Probleme nacheinander das 200., das 123. und danach das 17. Byte einer Datei lesen können. Aus diesem Grund wird in solchem Kontext ein Algorithmus, der einen wahlfreien Zugriff garantiert, eingesetzt.

[2] Außer CFS gibt es andere Systeme, die ebenfalls eine sichere Speicherung der Dateien versuchen; das *Secure File System* (SFS) [SFS], das Transparent Cryptographic File System (TCFS) [TCFS], das Satan File System oder das Microsoft Encrypted File System (EFS) sind einige dieser Systeme, die sich diese Aufgabe auf ihre Fahnen geschrieben haben.

[3] Swap-Bereich sind Teile des Speichers, die auf der Platte abgelegt und dort verwaltet werden.

B.2 Schutz vor dem böswilligen INDIGO-Server

Der Schutz der Dokumente vor einem INDIGO-Server, auf dem sie gespeichert sind, ist sehr schwer zu realisieren. Ein Verfahren, das einen minimalen Schutz vor einem nicht-vertrauenswürdigen INDIGO-Server bietet, ist das Ende-zu-Ende Verfahren, das im Abschnitt 5.1.1 erläutert wurde.

Der Inhalt eines Dokumentes, das mit einem Ende-zu-Ende Verfahren behandelt wurde, liegt verschlüsselt vor und kann nur von einer autorisierten Person entschlüsselt werden. Somit ist er auf dem Weg vom Autor zur autorisierten Person sicher vor jedem unerlaubten lesenden Zugriff. Falls der INDIGO-Server, auf dem sich dieses Dokument befindet, nicht gleichzeitig der anwenderseitige Ausführungs-Server ist, ist sein Dokumentinhalt vor diesem Server geschützt. Solche Server – wie die speicherseitigen INDIGO-Ausführungs-Server – sind nur zur Archivierung und Verbreitung der Dokumente zuständig[4].

Falls aber der anwenderseitige Ausführungs-Server nicht vertrauenswürdig ist, ist der Schutz des Dokumentinhalts vor diesem INDIGO-Server fast unmöglich. Wie bereits im Abschnitt 5.1.1 und in B.1 erwähnt, bieten Verfahren zur Ende-zu-Ende Vertraulichkeit nur einen bedingten Schutz vor einem Angreifer, da diese Verfahren keine Sicherheit gegen ein Speicher- bzw. Swap-Ausspähen bieten. Es kommt noch hinzu, daß der Betreiber eines solchen anwenderseitigen Ausführungs-Servers die Möglichkeit besitzt, eine manipulierte Dokumentmethode anstelle der vom Autor angegebenen Dokumentmethode aufzurufen. Eine solche manipulierte Dokumentmethode kann beispielsweise das Paßwort des Anwenders, das die ursprüngliche Methode zum Entschlüsseln des Dokumentinhalts bräuchte, an den Angreifer weiterleiten.

Beim Umgang mit den großen Dokumenten ist es außerdem noch üblich – je nach Bedarf – nur einen Teil der Datei auf den Speicher zu laden. Der Rest des Dokuments wird auf der Festplatte ausgelagert. Falls diese Datei mit Hilfe eines üblichen Verschlüsselungsverfahrens (wie beispielsweise DES, IDEA oder RSA) verschlüsselt wurde, muß man natürlich die gesamte Datei vor der Verwendung vollkommen entschlüsseln. Danach liegt die Datei ganz oder teilweise unverschlüsselt auf der Platte, und der Angreifer hat freien Zugriff auf sie.

Bei der Verschlüsselung solcher großen Dokumente kann man dieses Dilemma durch die Wahl eines Verschlüsselungsverfahrens, das einen wahlfreien Zugriff bietet, umgehen. Wie bereits im Abschnitt 3.2.1.2 erwähnt, kann das DES-Verfahren, das im wesentlichen eine Blockchiffretechnik ist, in Betriebsarten *Cipher Feedback* (*CFB*) oder *Output Feedback* (*OFB*) in eine Stromchiffre umgewandelt werden. Das Krypto-Filesystem CFS [CFS93] verwendet ebenfalls das OFB-Modus, um bei der Verwendung des DES-Verschlüsselungsverfahrens einen wahlfreien Zugriff auf die Daten zu haben.

Ein weiteres ernsthaftes Problem, das hier nur am Rande zu erwähnen wäre, ist das Problem des Löschens der Daten. Eine Dokumentmethode kann beispielsweise für eine kurze Zeit sensible Daten im Klartext auf der Festplatte speichern und sie wieder löschen. Das

[4] Der Betreiber eines solchen Servers kann auch die Methodenzuordnung des Dokuments manipulieren. Er könnte beispielsweise Dokumentmethoden eintragen, die das Paßwort des Anwenders diesem Server zuschicken (eine Art Trojanisches Pferd). Falls ein Benutzer seine Dokumente von einem nicht-vertrauenswürdigen Server-Verbund bezieht, muß er unbedingt die Integrität und die Authentizität des Dokuments nachträglich prüfen. Ein Dokument ist auf einem solchen INDIGO-Server nur dann gegen eine Manipulation geschützt, wenn seine einzelnen Dokumentbereiche von seinem Autor signiert wurden.

dabei entstehende Problem ist ein typisches Sicherheitsproblem, das bei den meisten Betriebssystemen auftritt. Nach dem Löschen dieser Klartextdatei sind die Daten zwar aus dem Dateisystem verschwunden, aber es ist dennoch möglich, die physikalisch gespeicherten Daten wiederzugewinnen. Dies ist möglich, weil nur die Verbindung zwischen dem inzwischen vernichteten Dateinamen und den ursprünglichen Datenblöcken auf dem Speichermedium durch das Löschen der Datei aufgelöst worden ist. Die wirkliche Vernichtung der vertraulichen Daten wird nur dadurch erreicht, daß man die Datei in ihrer gesamten Länge vor dem Löschen mit anderen Daten überschreibt (und das möglichst mehrmals). Dieses Problem wurde von den verschiedenen Protokollen bereits erkannt. Sie bieten die entsprechenden Mechanismen, die dieses *file wiping* automatisch durchführen. PGP sieht beispielsweise für diesen Zweck eine Option „-w“ (steht für wipe) vor.

Anhang C

Software-Komponenten

C.1 Quelltexte der Implementation

Die Quellecodes des modifizierten INDIGO-Servers, der Metadokumente, der Clients und der weiteren implementierten Komponenten befinden sich auf der beiliegenden CD-ROM. Die aktuellen Programmpakete können auch von der Homepage des Programms geladen werden. Die entsprechende Adresse lautet:

http://www.tm.informatik.uni-frankfurt.de/~razi/IndigoSec/

Die vom INDIGO-Server benötigten Dateien befinden sich auf der CD-ROM im Verzeichnis „`server/`“ und die der Packages im Verzeichnis „`packages/java/`“. Eine Übersicht über die Klassen des INDIGO-Servers und der Packages stellt das UML-Klassendiagramm C.1 auf Seite 147 dar.

Das Verzeichnis „`documents`“ beinhaltet alle Metadokumente und ihre Dokumentmethoden; „`clients/`“ beinhaltet hingegen die Dateien der unterschiedlichen Clients. Außerdem existiert das Verzeichnis „`certificates/`“, das alle von den Clients und dem Server benötigten Keystores und Truststores beherbergt.

All diese Pfadangaben beziehen sich relativ auf das Verzeichnis „`jdl/`“ auf der CD-ROM. Zum Kompilieren der benötigten Dateien wird in diesem `jdl`-Verzeichnis der Befehl

```
make -f Makefile
```

aufgerufen. Zum Kompilieren dieser Dateien wird zusätzlich zum JDK auch das Programm *GNU Make* benötigt. Die Homepage dieses Programms erreicht man über die folgende Adresse: http://www.gnu.org/software/make/

C.2 Aufruf der Clients

SSLClient

Der Client „SSLClient“ kann beispielsweise wie folgt aufgerufen werden:

```
java -Dindigo.trustStore="/home/file.truststore" \
     -Dindigo.trustStorePassword="test" \
     -Dindigo.trustStoreType="JKS" \
     -Dindigo.trustManagerFactoryAlgorithm="SunX509" \
     -Dindigo.httpsProtocols="SSL" \
     SSLClient "141.22.23.24" "7333" \
     "/diglib/invoke/202?Present&141.2.3.4&7333" "SuperUserName:Password"
```

Die Konfigurationsvariablen dieses Clients, die ihm über die „-D“-Optionen übergeben werden, haben die gleiche Bedeutung wie die Konfigurationsvariablen des INDIGO-Servers, die im Abschnitt 6.1.2 beschrieben wurden. In diesem Beispiel schickt der Client nach seinem Aufruf einem Server, der auf dem Socket „`141.22.23.24:7333`“ auf die Anfragen wartet, folgende Anfrage:

```
GET /diglib/invoke/202?Present&141.2.3.4&7333 HTTP/1.1
Authorization: Basic U3VwZXJVc2VyTmFtZTpQYXNzd29yZA==
```

Die Phrase „`SuperUserName:Password`“ ist die Superuser-spezifische Information. Falls diese Phrase beim Aufruf des Client benutzt wird, konvertiert der Client intern diese in das Base64-Format und schickt das Ergebnis im Authorization-Header an den Server.

SSLClientWithClientAuth

Der Aufruf des Clients „SSLClientWithClientAuth“ kann hingegen mit den entsprechenden Optionen wie folgt erfolgen:

```
java -Dindigo.trustStore="/home/file.truststore" \
     -Dindigo.trustStorePassword="test" \
     -Dindigo.trustStoreType="JKS" \
     -Dindigo.trustManagerFactoryAlgorithm="SunX509" \
     -Dindigo.keyStore="/home/file.keystore" \
     -Dindigo.keyStorePassword="test" \
     -Dindigo.keyStorePrivateKeyPassword="PKtest" \
     -Dindigo.keyStoreType="JKS" \
     -Dindigo.keyManagerFactoryAlgorithm="SunX509" \
     -Dindigo.httpsProtocols="SSL" \
     SSLClient "141.22.23.24" "7333" \
     "/diglib/invoke/202?Present&141.2.3.4&7333" "SuperUserName:Password"
```

Wie dieses Beispiel zeigt, muß man diesem Client über die entsprechenden Optionen (`-Dindigo.keyStore`, `-Dindigo.keyStorePassword`, usw.) zusätzlich noch Informationen über einen Keystore übergeben. Mit Hilfe dieses Keystores bzw. darin enthaltener Schlüssel kann der Client sich eindeutig identifizieren.

Literaturverzeichnis

[AES] Joan Daemen, Vincent Rijmen:
AES Proposal: Rijndael. AES Round 1 Technical Evaluation CD-1: Documentation, National Institute of Standards and Technology, August 1998.
http://www.nist.gov/aes.

[AgentTCL] Robert S. Gray:
Agent Tcl: A flexible and secure mobile-agent system. Dartmouth College, Hanover, New Hampshire, Juni 1997.

[Auc96] D. Aucsmith:
Tamper Resistant Software: An Implementation. 1^{st} Information Hiding Workshop, Springer Lecture Notes in Computer Science, vol. 1174, pp. 317–333, 1996.

[CFS93] Matthew Blaze:
A Cryptographic File System for UNIX. Pro. 1st ACM Conferende on Computer and Communications Security, Fairfax, VA, November 1993.

[Chap96] C. Brent Chapman, Elizabeth D. Zwicky:
Einrichtung von Internet Firewalls O´Reilly / Internation Thomson Verlag, 1996.

[DES] *Data Encryption Standard*, Federal Information Processing Standard (FIPS), Publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., Januar 1977.

[DH76] Whitfield Diffie, Martin Hellman:
New Directions in Cryptography. IEEE Transactions on Information Theory, November 1976.

[DivX] DivXNetworks:
DivX compression technology. a new format for digital video:
http://www.divx.com.

[Dre99] G. Drew:
Using SET for Secure Electronic Commerce. Upper Saddle River, NJ: Prentice Hall, 1999.

[Dro99] Oswald Drobnik:
Verteilte Systeme und Telematik. Scriptum zur Vorlesung an der J.W. Goethe-Universität Frankfurt am Main, Fachbereich Informatik, 1999.

[DU93] *Duden „Informatik“:* ein Sachlexikon für Studium und Praxis. Mannheim; Leipzig; Wien; Zürich: Dudenverlag, 1993.

[Eck98] Bruce Eckel:
Thinking in Java. 2. edition, Revision 3. Prentice-Hall, Oktober 1999:
http://www.BruceEckel.com.

[ElGa85] Taher ElGamal:
A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. Advances in Cryptology: Proceedings of CRYPTO 84, Springer Verlag, 1985.

[Fla00] David Flanagan:
JAVA in a Nutshell. Deutsche Ausgabe für Java 1.2 und 1.3: O´Reilly Verlag, 2000.

[Gri94] Rüdiger Grimm:
Sicherheit für offene Kommunikation: verbindliche Telekooperation. Mannheim; Leipzig; Wien; Zürich: BI-Wissenschaftsverlag, 1994.

[Gri98] Rüdiger Grimm:
Sicherheit und Datenschutz in der Informationstechnik Scriptum zur Vorlesung an der J.W. Goethe-Universität Frankfurt am Main, Fachbereich Informatik, SS. 1998.

[Groe96] M. Grötschel, J. Lügger:
Neue Produkte für die digitale Bibliothek: die Rolle der Wissenschaft, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Technical Report, TR 96-05, März 1996.

[HMAC] H. Krawczyk, M. Bellare, R. Canetti:
HMAC: Keyed-Hashing for Message Authentication. Request for Comments 2104, Februar 1997:
http://www.ietf.org/rfc/rfc2104.txt.

[HTTP] T. Berners-Lee, R. Fielding, H. Frystyk:
Hypertext Transfer Protocol HTTP/1.0. Network Working Group, Request for Comments 1945, 1996:
http://www.ietf.org/rfc/rfc1945.txt.

[HTTP1.1] R. Fielding, J. Gettys, J. Mogul, DEC, H. Frystyk, T. Berners-Lee:
Hypertext Transfer Protocol HTTP/1.1. Network Working Group, Request for Comments 2068, Januar 1997:
http://www.ietf.org/rfc/rfc2068.txt.

[ITSK89] BSI:
IT-Sicherheitskriterien: Kriterien für die Bewertung der Sicherheit von Systemen der Informationstechnik (IT). Bundesanzeiger Nr. 99a vom 1.6.1989. BSI, Bundesamt für Sicherheit in der Informationstechnik. Bundesanzeiger, Köln 1989.

[JSSE] Sun Microsystems:
JavaTM Secure Socket Extension (JSSE). Reference Guide for the JavaTM 2 SDK, Standard Edition, v 1.4
http://java.sun.com/products/jsse/.

[Ker95] Heinrich Kersten:
Sicherheit in der Informationstechnik: Einführung in Probleme, Konzepte und Lösungen. 2. Auflage - München ; Wien : Oldenburg, 1995.

[Lipp00] Peter Lipp, Dieter Bratko, Johannes Farmer, Wolfgang Platzer und Andreas Sterbenz:
Sicherheit und Kryptographie in Java. Einführung, Anwendung und Lösungen. Addison-Wesley Verlag, 2000.

[MD5] Ron Rivest:
The MD5 Message-Digest Algorithm. Network Working Group, Request for Comments 1321, April 1992:
http://www.ietf.org/rfc/rfc1321.txt.

[Moe98] Christian Mönch, Oswald Drobnik:
Integrating New document Types into Digital Libraries. Frankfurt/Germany; In Proceedings of the IEEE Forum on Research and Technology Advances in Digital Libraries, IEEE ADL 1998.

[Moe00] Christian Mönch:
INDIGO – An Approach to Infrastructures for Digital Libraries. Research and Advanced Technology for Digital Libraries. Springer Verlag, 4th European Conference, ECDL 2000, Lisbon, Portugal, September 18-20, 2000 Proceedings.

[Moe01] Christian Mönch:
Eine verteilte Infrastruktur für typ- und diensterweiterbare orthogonale Digitale Bibliothek, Dissertationsarbeit. J.W. Goethe-Universität in Frankfurt am Main, Fachbereich Biologie und Informatik, Mai 2001.

[MP3] Scot Hacker:
MP3: The Definitive Guide. O´Reilly Verlag, März 2000.

[MSAT] Microsoft Corporation:
Microsoft Authenticode Technology.
http://www.microsoft.com/technet/security/prodtech/certauth/default.asp.

[Muss89] G. Musstopf:
Trojanische Pferde, Viren und Würmer. Fachgruppe Personal Computing der Gesellschaft für Informatik. Hamburg: perComp Verlag, 1989.

[Niel95] Jakob Nielsen:
Multimedia and Hypertext. The Internet and Beyond. Boston u.a., Academic Press 1995.

[Pfitz97] Birgit Pfitzmann:
Anonymous Fingerprinting. Advances in Cryptology - EUROCRYPT '97, S.88-102, Springer Verlag, 1997:
http://www-krypt.cs.uni-sb.de/.

[REMO] E. Amann:
Glossar der Basisbegriffe. REMO Arbeitskreis Terminologie und Grundlagen. REMO Projektpapier IABG.REMO.00078, 15.3.93, 8 Seiten. Veröffentlicht als: E. Amann, H. Atzmüller: IT-Sicherheit – was ist das? Datenschutz und Datensicherung (DuD) 6/92, 286-292. Braunschweig; Wiesbaden: Vieweg Verlag, 1992.

[RFC1421] J. Linn:
Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. Network Working Group, Request for Comments 1421, Februar 1993:
http://www.ietf.org/rfc/rfc1421.txt.

[RFC1883] S. Deering, R. Hinden:
Internet Protocol, Version 6 (IPv6), Specification. Network Working Group, Request for Comments 1883, Dezember 1995:
http://www.ietf.org/rfc/rfc1883.txt.

[RFC2045] N. Freed, N. Borenstein:
Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. Network Working Group, Request for Comments 2045, November 1996:
http://www.ietf.org/rfc/rfc2045.txt.

[RFC2069] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, L. Stewart:
An Extension to HTTP : Digest Access Authentication. Network Working Group, Request for Comments 2069, Januar 1997:
http://www.ietf.org/rfc/rfc2069.txt.

[RFC2246] T. Dierks, C. Allen:
The TLS Protocol Version 1.0. Network Working Group, Request for Comments 2246, Januar 1999:
http://www.ietf.org/rfc/rfc2246.txt.

[RFC2401] S. Kent, R. Atkinson:
Security Architecture for the Internet Protocol. Network Working Group, Request for Comments 2401, November 1998:
http://www.ietf.org/rfc/rfc2401.txt.

[RFC2617] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, L. Stewart:
HTTP Authentication: Basic and Digest Access Authentication. Network Working Group, Request for Comments 2617 (Obsoletes: 2069), Juni 1999:
http://www.ietf.org/rfc/rfc2617.txt.

[RFC2632] B. Ramsdell:
S/MIME Version 3 Certificate Handling. Network Working Group, Request for Comments 2632, Juni 1999:
http://www.ietf.org/rfc/rfc2632.txt.

[RFC2633] B. Ramsdell:
S/MIME Version 3 Message Specification. Network Working Group, Request for Comments 2633, Juni 1999:
http://www.ietf.org/rfc/rfc2633.txt.

[RFC2660] E. Rescorla, A. Schiffman:
The Secure HyperText Transfer Protocol. Network Working Group, Request for 2660, August 1999:
http://www.ietf.org/rfc/rfc2660.txt.

[RFC3275] D. Eastlake 3rd, J. Reagle, D. Solo:
(Extensible Markup Language) XML-Signature Syntax and Processing. Network Working Group, Request for Comments 3275, März 2002:
http://www.ietf.org/rfc/rfc3275.txt.

[RIPE160] H. Dobbertin, A. Bosselaers, B. Preneel:
RIPEMD-160: A Strengthened Version of RIPEMD. Fast Software Encryption, LNCS 1039, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71-82.

[Roe97] Klaus Röhrle:
Konzeption, Implementierung und Analyse von Verwürfelungsmechanismen für Quellcode, Diploma Thesis Nr. 1541, Faculty of Informatics, University of Stuttgart, Germany, 1997.

[RSA] Ron Rivest, Adi Shamir, Len Adleman:
A Method for Obtaining Digital Signatures and Public Key Cryptosystems, Communications of the ACM, Februar 1978.

[Schn96] Bruce Schneier:
Angewandte Kryptographie, Addison-Wesley, Bonn, 1. Auflage 1996/1., korrigierter Nachdruck 1997.

[Schw01] Jörg Schwenk:
Sicherheitsstandards im Internet. Scriptum zur Vorlesung an der J.W. Goethe-Universität Frankfurt am Main, Fachbereich Biologie und Informatik, SS. 2001.

[SFS] James Hughes, Chris Feist, Steve Hawkinson, Jeff Perrault, Matthew O Keefe, David Corcoran:
Secure File System (SFS). Storage Technology Corp., University of Minnesota, Purdue University; Februar 2000:
http://www.cs.auckland.ac.nz/~pgut001/sfs/.

[SHA] FIPS 180-1:
Secure hash standard. NIST, US Department of Commerce, Washington D.C., April 1995.

[SMIME] S/MIME Working Group:
Secure MIME (S/MIME).
http://www.ietf.org/html.charters/smime-charter.html und
http://www.imc.org/ietf-smime/.

[SSH] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen:
SSH Protocol Architecture. Network Working Group, Internet-Draft; August 2002:
http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-12.txt.

[SSL] Alan O. Freier, Philip Karlton, Paul C. Kocher:
The SSL Protocol Version 3.0, Internet-Draft; März 1996:
http://www.netscape.com/eng/ssl3/draft302.txt.

[Sta95] William Stallings:
Sicherheit in Netzwerk und Internet. München; London; Mexico City; Singapur; Sydney; Toronto: Prentice-Hall, 1995.

[Sta00] William Stallings:
Sicherheit im Internet Anwendungen und Standards. Addison-Wesley Verlag, 2000.

[Stei88] J. Steiner, C. Neuman, J. Schiller:
Kerberos: An Authentication Service for Open Networked Systems. Proceedings of the Winter 1988 USENIX Conference, Februar 1988.

[TCFS] *Transparent Cryptographic File System (TCFS)*: University of Salerno. April 1997:
http://www.globenet.it/ermmau/tcfs/index.html.

[URL] T. Berners-Lee, R.T. Fielding und H. Frystyk Nielsen:
Uniform Resource Locators (URL). RFC 1738. 1994:
http://www.ietf.org/rfc/rfc1738.txt.

[Witten87] I. H. Witten:
Computer (In)Security: Infiltrating Open Systems. ABACUS Vol. 4, No.4. Springer Verlag. New York; 1987.

[Wobst98] Reinhard Wobst:
Abenteuer Kryptologie: Methoden, Risiken und Nutzen der Datenverschlüsselung. Bonn; Reading, Massachusetts: Addison-Wesley-Longman, 1998.

[X500] Comité Consultatif International Télégraphique et Téléphonique (CCITT):
CCITT Recommendation X.500, 1988 (seit 1993 ISO/IEC Standard 9594)

[X509] Comité Consultatif International Télégraphique et Téléphonique (CCITT):
CCITT Recommendation X.509, The Directory - Authentication Framework, 1988 (Rev. 1993 und 1997).

[XML] W3 CONSORTIUM:
Extensible Markup Language (XML).
http://www.w3.org/XML/.

[XML1.0] W3 CONSORTIUM:
Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, Oktober 2000:
http://www.w3.org/TR/2000/REC-xml-20001006.

[XMLSig] W3 CONSORTIUM:
XML-Signature Syntax and Processing.
http://www.w3.org/TR/xmldsig-core/.

[Zimm96] Philip Zimmermann:
Pretty Good Privacy – Public Key Encryption for the Masses. PGP User's Guide. Vol. I: Essential Topics, Vol. II: Special Topics. PGP Version 2.6.3, Jan. 1996.

Abbildung C.1 UML-Klassendiagramm des INDIGO-Servers und der Packages