Entwurf eines einfachen Client-Server-Systems zur Verteilung von Schlüsseldaten asymmetrischer Kryptosysteme

Diplomarbeit von

Holger Kenn

angefertigt nach einem Thema von Herrn Prof. Dr. Johannes Buchmann am Fachbereich Informatik der Universität des Saarlandes, Saarbrücken

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Saarbrücken, im Dezember 1996

Mein Dank gilt in erster Linie Prof. Dr. J. Buchmann und seiner Mitarbeiterin Dr. Ingrid Biehl sowie seinen Mitarbeitern Dr. Bernd Meyer und Dr. Christoph Thiel für das Thema und die Unterstützung. Ich danke auch Jochen Schwarz, der mich mit vielen interessanten Anregungen während der Durchführung dieser Arbeit unterstützt hat.

Inhaltsverzeichnis

Einleitung

Mit der zunehmenden Verbreitung von globalen Computernetzen kommt der Public-Key-Kryptographie eine immer größere Bedeutung zu. In derartigen Netzen gibt es in der Regel keinen einzelnen Betreiber. Daher gibt es auch keine zentralen Stellen, die die Sicherheit der Daten, die durch diese Netze fließen, garantieren können. Die einzelnen Benutzer sind nicht in der Lage, aus dem Netz stammende Daten auf Authentizität zu prüfen oder sie gegen Ausspähung zu sichern.

Einen Ausweg aus dieser Situation bietet die Public-Key-Kryptographie. Zum einen kann damit die Ausspähung von Daten verhindert werden, die nur für einen Benutzer bestimmt sind, aber dennoch über das öffentliche Netz übertragen werden müssen. Zum anderen kann auf dem Wege der digitalen Signatur die Authentizität von Daten festgestellt werden. Dies ist entscheidend für die kommerzielle Nutzung von öffentlichen Netzen, z.B. für Einkäufe oder Bankgeschäfte.

Ein Kernproblem bei der Verwendung derartiger Systeme ist jedoch, die für Kommunikation oder Signaturprüfung notwendigen öffentlichen Schlüssel der daran beteiligten Personen zu beschaffen und die Authentizität dieser Schlüssel festzustellen. Dies kann entweder zwischen allen Benutzern „persönlich“ geschehen, also im direkten Austausch, oder über ein im Netz verfügbares System, daß diese Daten verteilt. Der persönliche Austausch ist nur bei kleineren Benutzergruppen, deren Mitglieder sich treffen können, sinnvoll. Aber gerade in weltumspannenden Netzen wie dem Internet ist ein Vorteil die Erreichbarkeit von Personen, die man nicht einfach persönlich aufsuchen kann. Also ist ein leistungsfähiges und sicheres System zur Verteilung von öffentlichen Schlüsseln notwendig.

Um die Schlüssel zu verteilen, gibt es mehrere Ansätze: Zum einen kann man das Verteilungssystem so sichern, daß es nur authentische Informationen enthält, indem man es in einem Verzeichnis zum Beispiel in Form eines Buches (Telefonbuch) oder eines nicht überschreibbaren Datenträgers (CD-Rom) unabhängig vom Netz vertreibt. Bei zunehmender Größe der Systeme ist es schwierig, die Informationen im Verzeichnis aktuell zu halten. Ein weiterer Ansatz besteht darin, die Information im Verzeichnis selbst so zu gestalten, daß ein gefälschter Eintrag von den Benutzern als solcher identifiziert werden kann. Diese Identifizierung von Fälschungen kann man mit Hilfe von digitalen Signaturen erreichen.

Ein weiterer wichtiger Punkt ist die einfache und schnelle Zugänglichkeit der vom Verzeichnis gespeicherten Informationen, denn ohne diese Eigenschaften wird ein solches System nicht von seinen Benutzern akzeptiert und trägt damit nicht zur Sicherheit bei. Ein Schlüsselverzeichnis muß für alle Benutzer eines Netzes zugänglich sein (und damit auch für alle von den Benutzern verwendeten Computern, Betriebssystemen, Programmen etc.). Das System sollte einen Zugriff mit einfachen Mitteln erlauben, z.B. ohne spezielle Zungangssoftware. Gleichzeitig sollte der Zugriff aber auch in bestehende Programme einfügbar sein, um auch ungeübten Benutzern die Verwendung des Systems zu ermöglichen.

Viele bisher existierende Verzeichnisse erfüllen einige der Anforderungen nur unzureichend. Gerade die einfache Zugangsmöglichkeit ist bei den bekannten Verzeichnissen oft nicht vorhanden.

Das Ziel dieser Arbeit ist, ein Verteilungssystem theoretisch zu entwerfen, das die hier aufgeführten Anforderungen erfüllt und die Implementierung dieses Systems detailiert zu beschreiben. Weiterhin werden Vergleiche zwischen bereits vorhandenen Schlüsselverzeichnissen und dem hier vorgestellten System durchgeführt.

Diese Arbeit ist im Rahmen des Projekts `LiSA` entstanden. `LiSA` (Library for Secure Applications) ist eine `C++`-Bibliothek, die die einfache Einbindung von kryptographischen Algorithmen in Anwendungsprogramme erlaubt. Innerhalb der Bibliothek werden die Schlüsseldaten des Programmbenutzers verwaltet. Führt das Programm eine Public-Key-Verschlüsselung durch, so werden die dazu benötigten öffentlichen Schlüssel automatisch vom Schlüsselverzeichnis angefordert, ebenso wie Signaturschlüssel für Schlüsselzertifikate. Mit Hilfe des Verzeichnisses kann die interne Schlüsselverwaltung von `LiSA` also alle nötigen Daten für die Verschlüsselung automatisch beschaffen, ohne daß dies vom Programmbenutzer bemerkt wird. Auch der Programmierer kann Verschlüsselung einfach in seine Programme integrieren, ohne sich Gedanken um Schlüsselverwaltung, Schlüsselgenerierung oder Zertifikatsprüfung zu machen.

Obwohl das in dieser Arbeit vorgestellte Schlüsselverzeichnis für die Zusammenarbeit mit `LiSA` entwickelt wurde, kann es unabhängig davon eingesetzt werden, denn der interne Aufbau des Verzeichnisses wurde so weit wie möglich offen gestaltet und ist von den abgespeicherten Daten unabhängig. Dadurch kann das Verzeichnis auch Schlüssel von anderen Programmen oder Verfahren abspeichern und verteilen.

Das erste Kapitel enthält einige Vorbemerkungen zum Thema. Zunächst werden einige kryptographische und technische Grundbegriffe eingeführt und die Aufgaben eines Schlüsselverzeichnisses erläutert. Dann werden Grundlagen der Netzwerkprogrammierung dargestellt und die Funktion von Zertifikaten zur Authentisierung von öffentlichen Schlüsseln erklärt.

Im zweiten Kapitel werden bekannte Schlüsselverzeichnisse vorgestellt. Ihr Aufbau wird untersucht und ihre Vor- und Nachteile aufgezeigt. Diese werden beim Entwurf des in dieser Arbeit vorgestellten Schlüsselverzeichnisses berücksichtigt.

Das dritte Kapitel beinhaltet den eigentlichen Entwurf des vorgestellten Verzeichnisses. Hier werden verschiedene Ansätze für die Struktur des Verzeichnisses diskutiert. Von einem einfachen Verzeichnis ausgehend, werden verschiedene Erweiterungen vorgestellt, die die Verwaltung und die Geschwindigkeit des Verzeichnisses verbessern.

Im vierten Kapitel werden die Implementierungen von Verzeichnis-Clients und -Servern detailliert beschrieben. Die einzelnen Architekturen aus dem dritten Kapitel werden hier in ihrer Umsetzung in Serverprogramme vorgestellt und die bei der Implementierung aufgetretenen Probleme und deren Lösung beschrieben.

Das fünfte Kapitel beinhaltet eine Diskussion der Sicherheit des vorgestellten Verzeichnisses. Hier werden einzelne Angriffsmöglichkeiten dargestellt und gezeigt, wie das Verzeichnis diese Angriffe abwehrt.

Im Anhang findet sich neben dem Literaturverzeichnis ein Programmbeispiel sowie ein Teil der UNIX-Programmdokumentation des Servers.

Kapitel 1

Vorbemerkungen

In diesem Kapitel werden zunächst einige zugrundegelegten Verfahren und Bezeichnungen eingeführt, die in den weiteren Kapiteln Verwendung finden. Darunter sind der Begriff „Verzeichnis“ und dessen Aufgaben. Weiterhin werden Grundlagen der Computervernetzung am Beispiel des Internet-Protokolls TCP/IP dargestellt und die Funktion von Schlüsselzertifikaten erläutert.

1.1 Theorie der Public-Key-Kryptographie

Die hier verwendeten Definitionen und Notationen stammen von den Erfindern der Publik-Key-Kryptographie, W. Diffie und M.E. Hellman [DH76].

Ein *kryptographisches System* ist eine parametrisierte Familie von invertierbaren Funktionen $\mathcal{S}_K, K \in \mathcal{K}, S \in \mathcal{S}$

$$S_K : \mathcal{P} \rightarrow \mathcal{C} \tag{1.1}$$

die einem Element einer Menge $\mathcal{P}$ von *Plaintext-Nachrichten* ein Element aus einer Menge $\mathcal{C}$ von *Ciphertext-Nachrichten* zuordnen. Der Parameter K heißt *Key (Schlüssel)*. Er wird aus einer endlichen Menge $\mathcal{K}$, dem*Keyspace* gewählt.

Das Ziel eines kryptographischen Systems ist, Ver- und Entschlüsselungsfunktionen zu realisieren, die möglichst effizient auswertbar sind, während das Brechen der Verschlüsselung, also die Kryptoanalyse, möglichst schwierig sein soll.

Ein *Public-Key-Kryptosystem* ist ein Paar von parametrisierten Familien von effizient berechenbaren Funktionen $\mathcal{E}_K$ und $\mathcal{D}_K, K \in \mathcal{K}$ mit $E_K \in \mathcal{E}_K$ und $D_K \in \mathcal{D}_K$,

$$E_K : \mathcal{P} \rightarrow \mathcal{C} \tag{1.2}$$

$$D_K : \mathcal{C} \rightarrow \mathcal{P}, \tag{1.3}$$

so daß

- für alle $K \in \mathcal{K}$ und $P \in \mathcal{P}$ gilt: $D_K(E_K(P)) = P$.

- für alle $K \in \mathcal{K}$ die Funktionen E_K und D_K einfach zu berechnen sind.
- es für fast alle $K \in \mathcal{K}$ schwierig ist, D_K zu berechnen, wenn nur E_K bekannt ist.
- zu einem gegebenen $K \in \mathcal{K}$ die Funktionen D_K und E_K einfach zu finden sind.

Zur Nachrichtenübertragung werden diese Funktionen folgendermaßen eingesetzt:

Ein Benutzer A gibt seine Funktion E_K oder einen sie berechnenden Algorithmus öffentlich bekannt. Wenn jetzt ein Benutzer B dem Benutzer A eine Nachricht $P \in \mathcal{P}$ senden will, so berechnet er $E_K(P)$ und veröffentlicht das Ergebnis. Da A im Besitz von D_K ist, kann er $D_K(E_K(P))$ bilden und erhält P.

Ein anderer Benutzer C kann aus E_K allein nicht D_K berechnen. D_K wird aber benötigt, um P zurückzugewinnen. Dadurch kann nur der richtige Adressat A die Nachricht P lesen.

Die Funktion E_K heißt *öffentlicher Schlüssel* oder *public key*. Die Funktion D_K heißt *privater Schlüssel* oder *private key*.

Um die Authentizität von Nachrichten zu beweisen, kommen *(Public-Key-) Signaturverfahren* zum Einsatz.

Ein *(Public-Key-) Signatursystem* ist ein Paar von parametrisierten Familien von durch Algorithmen berechenbaren Funktionen $\mathcal{S}_K$ und $\mathcal{V}_K, K \in \mathcal{K}$ mit $S_K \in \mathcal{S}_K$ und $V_K \in \mathcal{V}_K$,

$$S_K : \mathcal{P} \to \mathcal{C} \tag{1.4}$$

$$V_K : \mathcal{C} \times \mathcal{P} \to \{0,1\}, \tag{1.5}$$

so daß

- für alle $K \in \mathcal{K}$, $P \in \mathcal{P}$ und $C \in \mathcal{C}$ gilt:

$$V_K(C,P) = \begin{cases} 1 & \text{falls } C = S_K(P) \\ 0 & \text{sonst} \end{cases} \tag{1.6}$$

- für alle $K \in \mathcal{K}$ die Funktionen S_K und V_K einfach zu berechnen sind.
- es für alle $K \in \mathcal{K}$ schwierig ist, S_K zu berechnen, wenn nur V_K bekannt ist.
- zu einem gegebenen $K \in \mathcal{K}$ die Funktionen S_K und V_K einfach zu finden sind.

Diese beiden Funktionen können nun dazu dienen, die Authentizität von Nachrichten zu beweisen, und zwar in Form einer digitalen Signatur:

Will ein Benutzer A, daß andere Benutzer die Authentizität seiner Nachrichten überprüfen können, so gibt er seine Funktion V_K oder eine sie berechnenden Algorithmus bekannt.

Sendet nun A eine Nachricht $P \in \mathcal{P}$, so berechnet er zusätzlich $S = S_K(P)$ und hängt S an die Nachricht an.

Wenn nun ein anderer Benutzer die Authentizität der Nachricht P prüfen will, so prüft er, ob $V(S, P) = 1$ ist, denn dann ist die Nachricht authentisch.

Da für die Erzeugung der digitalen Signatur K oder S_K bekannt sein müssen, kann nur A der Urheber der Nachricht sein, denn nur er kennt K und S_K.

Die Funktion V_K heißt *öffentlicher Signaturschlüssel*. Die Funktion S_K heißt *privater Signaturschlüssel*.

Um einer digitalen Signatur vertrauen zu können, muß der Prüfer sicher sein, daß der öffentliche Signaturschlüssel tatsächlich von A stammt. Das heißt, er muß ihn also auf einem vertrauenswürdigen Weg erhalten haben. Beispielsweise kann der Signaturschlüssel selbst von einer Person signiert sein, die vertrauenswürdig ist.

1.2 Verzeichnisse und ihre Aufgaben

Ein *Verzeichnis* ist eine Tabelle, in der Einträge so verwaltet werden, daß zu einem bestimmten Schlüsselwert der Tabelleneintrag gefunden werden kann, in dem der Schlüsselwert abgelegt ist. Beispielsweise ist ein Telefonbuch ein Verzeichnis in diesem Sinn. Entscheidend ist auch, daß in einem Verzeichnis viel öfter nach Einträgen gesucht wird als Einträge geändert, hinzugefügt oder gelöscht werden. Das Verzeichnis kann auf unterschiedliche Weise implementiert werden, beispielsweise kann das Telefonbuch auf einer CD-Rom gespeichert werden Einträge oder bei der Telefonauskunft erfragt werden.

Ein *öffentliches Verzeichnis* ist ein Verzeichnis, dessen Einträge jedem Benutzer zur Verfügung gestellt werden, ohne das dabei die Identität des Benutzers eine Rolle spielt. Dadurch kann jeder Benutzer auf alle Einträge lesend zugreifen. Dies gilt allerdings nicht für schreibenden Zugriff.

Eine *Schlüsselinformation* ist eine standardisierte Repräsentation eines öffentlichen Schlüssels. Da der Schlüssel selbst eine Funktion E_K oder V_K ist, steht hier die Bezeichnung für einen allgemein verwendeten Algorithmus (z.B. RSA) und die für diesen Algorithmus verwendeten Parameter (z.B. n= 24012244832394572398495526919884324377, e=11), mit denen dann die Funkltionen berechnet werde können. Zusätzlich enthält diese Information eventuell vorhandene Zertifikate.

Ein *Zertifikat* ist eine in der Schlüsselinformation enthaltene Information, die die Authentizität der Schlüsselinformation bestätigen soll (siehe Abschnitt 1.3).

Innerhalb dieser Arbeit werden Verzeichnisse für die Speicherung von Schlüsselinformationen verwendet.

Ein öffentliches Verzeichnis stellt folgende Funktionen zur Verfügung:

1. Jeder Benutzer kann eine *Anfrage* an das Verzeichnis stellen. Dabei übergibt er einen *Schlüsselwert* (zum Beispiel den Namen eines anderen Benutzers). Als Antwort erhält er entweder
   - die gewünschte Schlüsselinformation,
   - die Auskunft, daß eine derartige Information nicht vorhanden ist, oder
   - eine Fehlermeldung.
2. Es besteht die Möglichkeit, Einträge einzufügen, zu löschen oder zu ändern. Dies kann entweder über eine im Verzeichnis integrierte Funktion geschehen oder durch eine Änderung an der Datenbasis, auf die das Verzeichnis zugreift.

Folgende Anforderungen werden an ein öffentliches Schlüsselverzeichnis gestellt:

Korrektheit: Enthält das Verzeichnis die gesuchte Information, so muß sie gefunden werden. Enthält das Verzeichnis keine solche Information, so muß dies erkannt werden und dem anfragenden Benutzer mitgeteilt werden. Beides sollte möglichst schnell geschehen und das Verzeichnis möglichst wenig belasten.

Konsistenz: Verwendet das Verzeichnis dynamische Daten, d.h. speichert es die gleichen Daten an verschiedenen Stellen im Verzeichnis, so muß die Konsistenz dieser Daten gewährleistet werden, d.h. die dynamischen Daten dürfen nicht zu Fehlern in der Anfragebearbeitung führen, falls sie nicht mehr mit den Daten der ursprünglichen Quelle übereinstimmen. Wenn dynamische Daten als „falsch“ erkannt werden, müssen sie also automatisch aktualisiert werden. Dadurch kann es natürlich zu Verzögerungen in der Bearbeitung von Anfragen kommen.

Authentizität: Weiterhin soll das Verzeichnis „schwer“ zu manipulieren sein, d.h. der Endanwender soll fehlerhafte Daten als solche erkennen können. Näheres zu diesem Problem findet sich im Abschnitt 1.3.

1.3 Schlüsselzertifikate und Trusted Authorities

Die Schlüsselinformationen, die ein Benutzer aus einem öffentlichen Verzeichnis entnimmt, können möglicherweise durch die Einwirkung Dritter verfälscht werden. Selbst wenn das Verzeichnis selbst gesichert ist, kann ein Angreifer den Zugriff eines Benutzers auf ein von ihm kontrolliertes Verzeichnis umlenken, indem er beispielsweise die Kommunikationsverbindungen des Benutzers manipuliert. Selbst in diesem Fall soll der Benutzer überprüfen können, ob die erhaltene Information authentisch ist.

Um diese Prüfung durchzuführen, verwendet man kryptographische Techniken. Zunächst einigen sich die Benutzer des Verzeichnisses auf eine Stelle, die die öffentlichen Schlüssel aller Teilnehmer signiert. Diese Stelle nennt man die *Trusted Authorithy*, kurz *TA*. Sie erhält die öffentlichen Schlüssel jedes Teilnehmers auf einem sicheren Weg, zum Beispiel durch persönliche Übergabe. Dann signiert die Trusted Authority den Schlüssel, indem sie ihm ein Zertifikat hinzufügt. Das Zertifikat entsteht, indem die Trusted Authority die aktuelle Zeit T, die Benutzer-ID und den öffentlichen Schlüssel E_{K_A} oder V_{K_A} des Benutzers A mit ihrem privaten Signaturschlüssel $S_{K_{TA}}$ verschlüsselt. Die Zeit T besteht aus Datum, Uhrzeit und Zeitzone zum Zeitpunkt der Erstellung des Zertifikats. Mit Hilfe dieser Informationen kann die Verwendung von ungültig gewordenen Zertifikaten verhindert werden.

Die Unterscheidung zwischen dem öffentlichen Schlüssel E_K und dem öffentlichen Signaturschlüssel V_K eines Benutzers ist nicht nötig, da beide öffentlichen Schlüssel auf die gleiche Weise zertifiziert werden können. Daher ist im folgenden nur noch vom öffentlichen Schlüssel E_K die Rede. Die gemachten Aussagen treffen aber in gleicher Weise für den öffentlichen Signaturschlüssel V_K zu. Bei den Trusted Authorities werden ausschließlich die Signaturschlüssel $S_{K_{TA}}$ und $V_{K_{TA}}$ verwendet.

Das Zertifikat für Benutzer A ergibt sich aus:

$$C_A = S_{K_{TA}}[T, A, E_{K_A}] \tag{1.7}$$

Dabei bedeuten die eckigen Klammern [und] die Konkatenation der in der Klammer angegebenen Parameter.

Um dieses Zertifikat überprüfen zu können, muß neben dem öffentlichen Schlüssel von A der Zeitpunkt der Zertifikatserstellung und die Benutzer-ID von A bekannt sein. Diese werden dazu mit dem Zertifikat zusammen verteilt.

In der kryptographischen Bibliothek `LiSA` wird hier ein an die Norm *X.509* angelehntes Zertifikatsformat verwendet:

```
AU:[...Benutzer-ID...]
AK:[...Schluesselinformation...]
AZ:
version                      :integer
Zertifikats-ID               :bytestring
Aussteller                   :string (TA-ID)
Gueltigkeit-von              :datestring (Datum)
Gueltigkeit-bis              :datestring (Datum)
Ueberpruefungsszeitraum      :datestring (Zeitraum)
LiSA-Signaturblock
```

Der `Lisa-Signaturblock` ist ein intern definiertes Format, in dem beliebige Daten signiert werden können. Mit Hilfe des Signaturblocks und des öffentlichen Signaturschlüssels der ausstellenden Trusted Authority kann diese Signatur dann von jedem anderen Benutzer überprüft werden.

Die gesamten Daten von der Benutzer-ID bis zum Überprüfungszeitraum werden bei der Zertifikatserstellung signiert.

Der Empfänger überprüft die Gültigkeit der Schlüsselinformation, indem er

$$V_{K_{TA}}\Big(C_A, [T, A, E_{K_A}]\Big) \tag{1.8}$$

berechnet.

Zudem besteht die Möglichkeit, mehrere Trusted Authorities einzusetzen. In diesem Fall wird der öffentliche Schlüssel eines Benutzers nur von der für ihn zuständigen Trusted Authority signiert. Die einzelnen Trusted Authorities signieren ihre öffentlichen Signaturschlüssel gegenseitig. Dadurch kann man zwischen zwei beliebigen Benutzern, die jeweils nur dem öffentlichen Signaturschlüssel ihrer Trusted Authority vertrauen, eine Kette von Zertifikaten aufbauen, so daß beide Benutzer den ausgetauschten öffentlichen Schlüsseln vertrauen können.

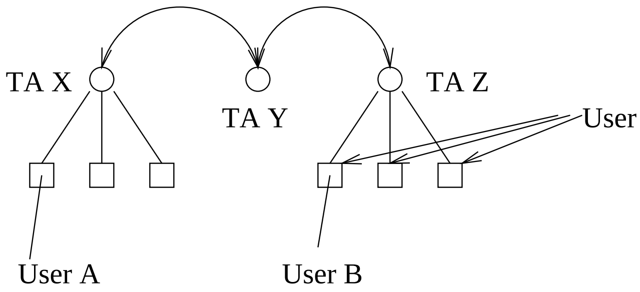

Abbildung 1.1: Mehrere Trusted Authorities

In diesem Beispiel ist Benutzer A von der TA X zertifiziert, während Benutzer B von TA Z zertifiziert ist. Diese beiden Trusted Authorities haben sich gegenseitig nicht zertifiziert, haben aber beide mit einer dritten TA Y Signaturschlüssel gegenseitig zertifiziert. Also gibt es

- die öffentlichen Schlüssel E_{K_A} und E_{K_B} der Benutzer A und B
- die Zertifikate dieser Schlüssel $C_{A_X} = S_{K_X}[T, A, K_{E_A}]$ und $C_{B_Z} = S_{K_Z}[T, B, K_{E_B}]$
- die Zertifikate der Signaturschlüssel
  $C_{Y_X} = S_{K_X}[T, Y, V_Y]$
  $C_{X_Y} = S_{K_Y}[T, X, V_X]$

$C_{Y_Z} = S_{K_Z}[T, Y, V_Y]$
$C_{Z_Y} = S_{K_Y}[T, Z, V_Z]$

Entnimmt nun A den öffentlichen Schlüssel von B aus dem öffentlichen Verzeichnis, so kann er das Schlüsselzertifikat zunächst nicht prüfen, da der Schlüssel von der TA Z zertifiziert wurde, während er nur den öffentlichen Signaturschlüssel seiner eigenen TA X kennt und ihm vertraut. Die TA X hat allerdings mit der TA Y gegenseitig Zertifikate getauscht. Damit kann A das von seiner TA ausgestellte Zertifikat für den öffentlichen Signaturschlüssel der TA Y prüfen.

$$V_{K_X}\Big(C_{Y_X}, [T, Y, V_{K_Y}]\Big) \tag{1.9}$$

Damit kann er auch V_{K_Y} vertrauen. Nun kann er auf die gleiche Weise das Zertifikat prüfen, das die TAs Y und Z ausgetauscht haben.

$$V_{K_Y}\Big(C_{Z_Y}, [T, Z, V_{K_Z}]\Big) \tag{1.10}$$

Nachdem er nun V_{K_Z} vertrauen kann, kann er auch das Schlüsselzertifikat an Bs Schlüssel C_{B_Z} prüfen.

$$V_{K_Z}\Big(C_{B_Z}, [T, B, E_{K_B}]\Big) \tag{1.11}$$

Trusted Authorities können hierarchisch angeordnet werden, wie in Abbildung 1.2 gezeigt wird. Eine Zertifizierung läuft in diesem Fall im Baum eine Strecke nach oben und dann zum Ziel wieder nach unten.

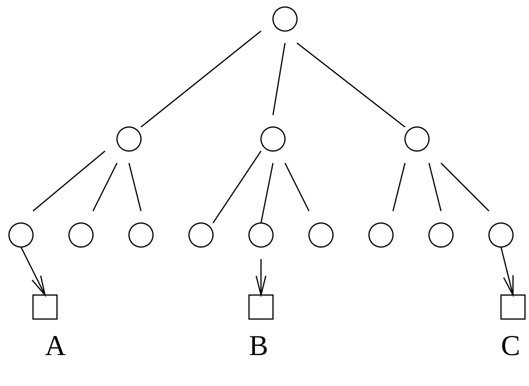

Abbildung 1.2: Hierarchische Anordnung von Trusted Authorities

Wenn Benutzer A den Schlüssel von Benutzer B erhält, so prüft er das Zertifikat, indem er eine Zertifikatskette zwischen der TA des entfernten Benutzers und seiner TA bildet und diese überprüft (Abbildung 1.3).

Können alle Zertifikate auf diesem Weg überprüft werden, so kann A darauf vertrauen, daß der so erhaltene Schlüssel wirklich von B stammt. Um den Weg

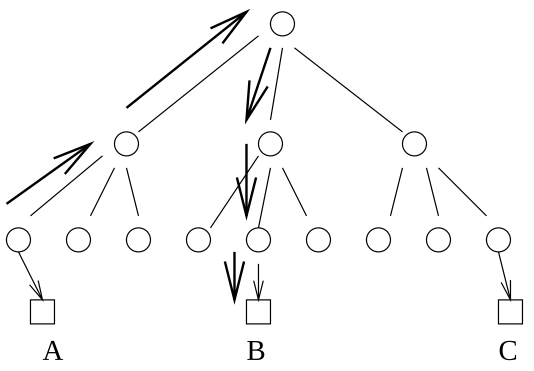

Abbildung 1.3: Zertifizierungskette zwischen A und B

dieser Zertifikatskette zu finden, müssen in den Zertifikaten zusätzliche Informationen enthalten sein, die den Aussteller des Zertifikats identifizieren. Entweder stehen diese getrennt in einem eigenen Feld oder ergeben sich aus der Benutzer-ID (und sind damit in ihr enthalten).

Die Signaturschlüssel von Benutzern und Trusted Authorities müssen so gekennzeichnet sein, daß kein Benutzerschlüssel mit dem einer Trusted Authority verwechselt werden kann, denn dnn bestünde die Möglichkeit, daß ein Benutzer ein Schlüsselzertifikat für einen anderen Benutzer erzeugt.

Die Implementierung dieser Zertifikatsprüfung ist dem Anwenderprogramm überlassen. Innerhalb des Schlüsselverzeichnisses kann wie gesagt die Authentizität der Daten nicht garrantiert werden. Daher muß die Zertifikatsprüfung erst nach der Entnahme der Schlüsselinformationen vom Benutzerprozeß durchgeführt werden, und das von einem Programm, dem der Benutzer vertraut. Beispielsweise kann eine derartige Zertifikatsprüfung in `LiSA` einfach implementiert werden, da die dafür nötigen Funktionen in der Bibliothek bereits vorhanden sind.

1.4 Netzwerk-Programmierung

Bei der Implementierung von Verzeichnissen ist die Netwerk-Programmierung ein wichtiger Punkt, der für die Verwendbarkeit entscheidend ist. Daher muß bei der Programmierung der Netzwerk-Schnittstellen Rücksicht auf mehrere Punkte genommen werden:

- Die verwendeten Schnittstellen müssen für möglichst viele unterschiedliche Betriebssysteme und Computertypen verfügbar sein.
- Übertragungen müssen gegen Übertragungsfehler gesichert sein.
- Das System sollte auch bei momentaner Nicht-Verfügbarkeit einzelner Rechner einen weiteren Betrieb eines möglichst großen Teils des Systems

zulassen.

- Auch die Server-Rechner selbst können beeinträchtigt werden, beispielsweise durch Hardware-Fehler oder Stromausfall. Daher sollte das Schlüsselverzeichnis möglichst viele seiner Daten so sicher wie möglich speichern, beispielsweise auf Festplatten und nicht nur im Hauptspeicher.
- Natürlich sollten über die Netzwerkverbindungen nicht mehr Informationen als unbedingt nötig übertragen, damit die Wartezeiten für den Benutzer möglichst kurz sind und die Belastung des Netzwerks möglichst gering ist.

Das heute am weitesten verbreitete Computer-Netzwerk ist das Internet. Daher soll es an dieser Stelle als Beispiel für den internen Aufbau von Computernetzen verwendet werden. Das gemeinsame Netzwerk-Protokoll im Internet ist das im Auftrag des amerikanischen Verteidigungsministerium entwickelte `TCP/IP-Protokoll` (Transmission Control Protocol/Internet Protocol) [LCPM85]. Dieses Protokoll ist in den sogenannten `RFC`s (Request for Comments) spezifiziert. `TCP/IP` ist ein Netzwerk-Protokoll, das sowohl für die Fernvernetzung als auch für die lokale Vernetzung verwendet werden kann. `TCP/IP`-Netzwerke können mit fast beliebigen Übertragungsmedien von Telefonleitungen und seriellen Schnittstellen bis zu Glasfaserstrecken und Satellitenverbindungen aufgebaut werden. Das `TCP/IP`-Protokoll ist heute für fast alle Betriebssysteme verfügbar. Einen Überblick über die einzelnen Teil-Protokolle bietet RFC 1011 [RFC87c].

Mit Hilfe von `TCP/IP` können unterschiedliche Übertragungsverfahren realisiert werden. Hier folgen einige Beispiele:

- `TCP/IP`-Vernetzung mit User-Datagrammen (`UDP` [RFC80])
  Bei dieser Schnittstelle werden Pakete vom Anwendungsprogramm zusammengestellt und über das Netzwerk an den Zielrechner gesendet. Ob das Paket ankommt oder ob die einzelnen Pakete in derselben Reihenfolge ankommen, in der sie abgeschickt wurden, wird nicht überprüft. Um diese Probleme hat sich das Anwendungsprogramm zu kümmern. `UDP` ist ein Basisbestandteil des `TCP/IP`-Systems und damit sehr weit verbreitet. `UDP` wird z.B. vom Internet Name Service `DNS` (Domain Name Service) [RFC87a] [RFC87b] verwendet, der zu Internet-Namen wie `www.microsoft.com` die zugehörigen Netzwerkadressen (`207.68.137.34`, `207.68.137.35`, `207.68.137.36`, `207.68.137.40` ,...) findet.
- `TCP/IP`-Vernetzung mit `TCP-Streams` [RFC81]
  Hier wird ein Kanal zwischen den beiden kommunizierenden Rechnern geschaltet. Dieser ist weitgehend gegen Übertragungsfehler gesichert. Paketreihenfolge und Paketverlust werden bereits vor der Schnittstelle zum Anwendungsprogramm ausgeglichen. `TCP-Streams` sind ebenfalls ein Basisbestandteil des `TCP/IP`-Netzwerk-Systems. Die Programme `ftp` [RFC85] und `telnet` [RFC83] verwenden `TCP-Streams`, um ihre Daten zu übertragen.

- Sun-RPC [RFC88]
  Dabei handelt es sich um ein ursprünglich von Sun Microsystems Inc. eingeführtes Protokoll, das bestimmte Programmteile (entfernte Prozeduren oder Remote Procedures) auf anderen Rechnern ausführen kann. Innerhalb einer Programmiersprache (wie zum Beispiel C) erscheint der Aufruf der entfernten Prozedur dann wie ein gewöhnlicher Funktionsaufruf. Ein Präprozessor bearbeitet nun den Quelltext des Programms und ersetzt den lokalen Funktionsaufruf durch einen entsprechenden Aufruf der entfernten Prozedur. Dabei werden die Argumente der Funktion in netzwerkweit standardisierte Datentypen (sogenannte XDR External Data Representation [RFC87d]) umgesetzt und übertragen. Im Zielrechner wird über ein spezielles Programm, den *Portmapper*, das richtige Zielprogramm ausfindig gemacht und dort die Funktion zur Ausführung gebracht. Die Ergebnisse dieses Funktionsaufrufs werden dann wieder umgewandelt und an den ursprünglichen Rechner zurückgeschickt. Dort stehen sie dann als Rückgabewerte der Funktion zur Verfügung. Sun-RPC ist gegen Übertragungsfehler und Paketvertauschungen gesichert, allerdings müssen die entfernten Prozeduren zustandslos sein und keine Seiteneffekte haben, da es auf Grund von Paketverlusten vorkommen kann, daß eine entfernte Prozedur mit denselben Daten mehrfach aufgerufen wird. Wurde der Zustand der Prozedur schon beim ersten Aufruf geändert und ist das Ergebnis dieses Aufrufs verlorengegangen, so wird die Prozedur im neuen Zustand erneut aufgerufen. Auch Seiteneffekte treten dann zweimal auf. Gemeinsame globale Daten zwischen Hauptprogramm und entfernter Prozedur können ebenfalls nicht benutzt werden, da diese nicht automatisch übertragen werden können. Sun-RPC benutzt in der Regel TCP/IP-Netzwerke zur Kommunikation, wird aber nicht auf allen Plattformen unterstützt, die auch TCP/IP unterstützen. Da für den Aufruf einer Remote Procedure für die verwendete Programmiersprache auch ein entsprechender Präprozessor vorhanden sein muß, der die lokale Funktion gegen den Aufruf der *remote procedure* ersetzt, ist Sun-RPC auch nicht für alle Programmiersprachen verfügbar. Insbesondere einfache Script-Programmiersprachen können mit Sun-RPC nicht zusammenarbeiten. Auf Sun-RPC basiert zum Beispiel das Netzwerk-Datei-System NFS[RFC89], das bei Unix-Rechnern dazu dient, gemeinsame Dateisysteme, wie zum Beispiel Home-Verzeichnisse, an viele Rechner zu verteilen.

Damit das in dieser Arbeit entwickelte Schlüsselverzeichnis von möglichst vielen unterschiedlichen Rechnersystemen genutzt werden kann, wurden TCP-Streams zur Kommunikation benutzt. Als einheitliche Schnittstelle zu den TCP-Streams wurde das ursprünglich mit dem BSD-Unix-Betriebssystem entstandene BSD-Socket-Interface [CS92] [Rag93] eingesetzt. Es ist auf fast allen Plattformen vorhanden, die TCP/IP unterstützen, so zum Beispiel in Microsoft Windows, in IBM OS/2, in allen UNIX-Betriebssystemen und im Apple-Macintosh-Betriebssystem.

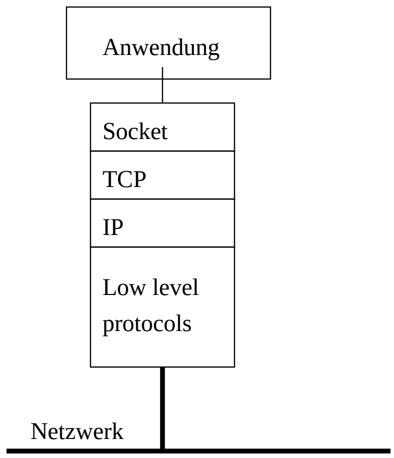

Abbildung 1.4: Schema der Socket-Schnittstelle

Die `BSD-Socket`-Schnittstelle simuliert eine Datei, in die das Anwendungsprogramm wie gewöhnlich schreiben kann und aus der es lesen kann. Nur werden die Daten dabei nicht auf die Festplatte geschrieben, sondern über das Netzwerk verschickt. Der Kommunikationspartner und sein Anwendungsprogramm haben ebenfalls eine solche virtuelle Datei. Schreibt nun das erste Programm seine Daten in die virtuelle Datei, kann das zweite Programm sie lesen und umgekehrt. Die Daten sind während der Übertragung gepuffert, das heißt, das Zielprogramm muß nicht unbedingt im selben Augenblick einen Lesezugriff machen, in dem das Ursprungsprogramm schreibt, sondern kann die Daten auch später abholen. Sie werden im jeweiligen Zielrechner zwischengespeichert.

Kapitel 2

Bekannte Schlüsselverzeichnisse

In diesem Kapitel werden bekannte Implementierungen von Schlüsselverzeichnissen vorgestellt und untersucht. Die Liste der aufgeführten Systeme erhebt keinen Anspruch auf Vollständigkeit, stellt allerdings einen guten Querschnitt durch die heute verwendeten Schlüsselverzeichnisse dar. Allerdings wurden hier keine Secret-Key-Verteilungssysteme wie z.B. Kerberos berücksichtigt.

2.1 Domain Name System Security Extension

Die von der DNS Security Working Group als zukünftige RFC vorgeschlagene Protokollerweiterung `DNS-SE` [DNS95] des Internet Domain Name Services hat zunächst zum Ziel, das bisher vollkommen ungesicherte DNS-Protokoll über die Verwendung von digitalen Signaturen zu sichern. DNS dekodiert Internetnamen (wie zum Beispiel `crypt8.cs.uni-sb.de`) zu IP-Adressen (wie `134.96.243.8`). Viele sicherheitsrelevante Internetprotokolle verlassen sich auf die Korrektheit dieser Informationen. Zum Beispiel kann beim Protokoll `rlogin` [RFC91] die Paßworteingabe vermieden werden, wenn der Benutzer einen entsprechenden Eintrag in der Datei `.rhosts` in seinem Homeverzeichnis des angesprochenen Rechners hat. Diese Datei ist in der Regel folgendermaßen aufgebaut:

```
rechner1 user1
rechner2 user1
```

Damit wird dem Benutzer `user1` auf den Rechnern `rechner1` und `rechner2` eine Benutzung des lokalen Rechners ohne Eingabe des Paßwortes ermöglicht. Der lokale Rechner löst die Namen `rechner1` und `rechner2` mit Hilfe von DNS zu IP-Adressen auf. Kann ein Angreifer nun DNS manipulieren, so kann er für `rechner1` eine andere IP-Adresse übermitteln, zum Beispiel die eines Rechners, auf dem er selbst einen Benutzer `user1` angelegt hat. Dadurch kann er ohne Paßwort in den lokalen Rechner gelangen. Die Security Extension signiert nun derartige Informationen. Damit kann der lokale Rechner feststellen, ob die von DNS erhaltene IP-Adresse authentisch ist. Dazu wurde DNS auch dahingehend erweitert, daß auch öffentliche signierte Schlüssel in DNS abgespeichert werden können. DNS-SE verwendet eine erweiterte Syntax des Konfigurationsfiles des DNS-Servers.

```
zb.mpi-sb.mpg.de.              IN      NS      ns.mpi-sb.mpg.de.
                               IN      NS      ns.cs.uni-sb.de.
                               IN      NS      ns.uni-sb.de.
                               IN      NS      ns.nic.de.
                               IN      A       139.19.1.1
                               IN      MX      50      mail.mpi-sb.mpg.de.
ns.mpi-sb.mpg.de.              IN      A       139.19.1.1
localhost                      IN      A       127.0.0.1
loopback                       IN      CNAME   localhost
*.zb.mpi-sb.mpg.de.            IN      MX      50      mail.mpi-sb.mpg.de.
                               IN      WKS     139.19.1.1      TCP     smtp
nurse-1                        IN      A       139.19.1.240
                               IN      MX      50      mail.mpi-sb.mpg.de.
nurse                          IN      CNAME   nurse-1
timehost                       IN      CNAME   nurse-1
```

Die einzelnen Einträge haben bei DNS folgende Bedeutung:

- NS-Einträge sind Verweise auf weitere DNS-Server
- A-Einträge geben die Adressen von Rechnern an
- CNAME-Einträge geben zusätzliche Alternativnamen für Rechner an
- MX-Einträge geben für die Rechner zuständige E-Mail-Gateways an.

Für DNS-SE kommen nun folgende Typen von Einträgen hinzu:

- KEY-Einträge geben Schlüsseldaten an.
- SIG-Einträge sind Signaturinformationen für Einträge, also Zertifikate.

```
nurse-1                        IN      A       139.19.1.240
                               IN      MX      50      mail.mpi-sb.mpg.de.
                               IN      KEY     KEY-INFORMATION
                               IN      SIG     SIGNATURE DATA
```

DNS-SE erlaubt nun zusätzlich Einträge für Benutzer:

```
kenn                           IN      KEY     USER-KEY-INFORMATION
                               IN      SIG     SIGNATURE DATA
```

Hierbei wird die Schlüsselinformation so gekennzeichnet, daß sie nicht mit der Schlüsselinformation eines Rechners verwechselt werden kann.

Die Vorteile von DNS-SE sind:

- Einfacher Einsatz: Durch Ersetzen der bisherigen DNS-Server durch solche, die das DNS-SE-Protokoll beherrschen.

- Hohe Geschwindigkeit: Das DNS-Protokoll selbst unterstützt das Caching von Informationen und ist für hohe Lasten ausgelegt. Eine Anfrage dauert typischerweise weniger als eine Sekunde, selbst bei transkontinentalen Verbindungen. DNS-SE erbt diese Fähigkeiten.

Die Nachteile sind:

- Die Schlüsselverteilung ist an DNS-Einträge gebunden. Benutzer, die nicht zu einer DNS-Domain gehören, sind schwer in das System einzuordnen.
- Keine Kontrolle über in Caches enthaltene Schlüsselinformationen. Beispielsweise kann ein Schlüssel auf dem ursprünglichen DNS-Server geändert sein, steht allerdings in der alten Fassung noch in den Caches anderer Server. DNS besitzt hier nur sehr unpräzise Einstellungen für die Verweildauer von Informationen in Caches.
- Abhängigkeit von DNS-System. Eine andere Anordnung der Schlüssel (zum Beispiel organisatorisch oder geographisch) wird nicht unterstützt.

2.2 Finger Protocol

Eine weiteres Protokoll geht auf P. Zimmermann zurück, den „Erfinder“ von `pgp`, eines sehr verbreiteten Programms zur Verschlüsselung von E-Mail. Hier wird das Finger-Protokoll benutzt. Dazu hängt jeder Benutzer an sein `.project`-File seinen öffentlichen Schlüssel an. Bei der Finger-Abfrage wird die Schlüsselinformation mit übertragen. Die Vorteile des Finger-Protokolls sind:

- Sehr einfacher Einsatz auf UNIX-Systemen: Hier gehört der Befehl `finger` und der zugehörige `finger`-Dämon zum Lieferumfang.
- Geringer Konfigurationsaufwand: Der Benutzer kann seinen Schlüssel selbst verwalten.

Die Nachteile sind:

- Das System ist sehr schwer außerhalb der Internet-Welt zu verwenden, da ein Online-Zugang unabdingbar ist. Allerhöchstend kann es noch in firmeninternen Netzen (sogenannten „Intranets“) verwendet werden.
- Benutzer ohne Unix-Zugang können ebenfalls nur schwer eingebunden werden, da die Schlüsselspeicherung in einem Unix-Homeverzeichnis erfolgt. Andere Betriebssystemarchitekturen (zum Beispiel Windows NT) werden dadurch nicht unterstützt.
- Da das Protokoll keine Caching-Mechanismen unterstützt, kann die Schlüsselverteilung zu großer Netzbelastung führen und unter Umständen sehr lange dauern oder gar versagen.

- PGP unterstützt selbst keine hierarchische Zertifizierung durch Trusted Authorities, sondern nur persönlich ausgetauschte Zertifikate, das sogenannte „Web of Trust". Daher ist die Überprüfung der Zertifikate der so erhaltenen Schlüsselinformationen ein ungelöstes Problem.

2.3 ITU X.500 Standard Directory

Innerhalb der ISO/OSI-Netzwerk-Protokolle gibt es wie im Internet ein System zum Austausch von E-Mails, X.400. Zu diesem System gibt es ein elektronisches Verzeichnis, das in der ISO-Norm X.500 spezifiziert ist. Dieses Verzeichnis kann natürlich auch zur Speicherung von Schlüsselinformatinen und Schlüsselzertifikaten verwendet werden.

Dazu gibt es eine eigene Standardisierung, die *ISO-Norm X.509* [ISO93b].

X.509 standardisiert dabei:

- die im Directory enthaltene Information,
- wie mit Hilfe dieser Informationen etwas authentisiert werden kann,
- auf welche Weise Informationen in das Verzeichnis gelangen und wie sie dort abgerufen werden und
- wie die Informationen zu verwenden sind in Bezug auf die Erstellung von Zertifikatsketten (siehe auch Abschnitt 1.3).

Weitere Teilbereiche der Norm sind zum Beispiel die Norm X.518 [ISO93e], die ein über mehrere Rechner verteiltes Verzeichnis beschreibt.

X.509 spezifiziert unter dem Begriff „strong authentication" die Verwendung von Public-Key-Kryptographie zur Erzeugung von signierten Zertifikaten und standardisiert auch die Formate dieser Zertifikate. Daher wird in vielen kryptographischen Industrieprodukten auf die Norm X.509 als Zertifikatsformat Bezug genommen.

Die X.509-Zertifikate werden von *Certificate Authorities*, kurz *CA*s ausgestellt, der X.509-Bezeichnung für Trusted Authorities.

Ein X.509-Zertifikat ist folgendermaßen aufgebaut:

```
Certificate            ::=SIGNED{SEQUENCE{
        version                  [0]     Version DEFAULT v1,
        serialNumber                     CertificateSerialNumber,
        signature                        AlgorithmIdentifier,
        issuer                           Name,
        validity                         Validity,
        subject                          Name,
        subjectPublicKeyInfo             SubjectPublicKeyInfo,
        issuerUniqueIdentifier   [1]     IMPLICIT UniqueIdentifier
```

```
                                                      OPTIONAL,
        subjectUniqueIdentifier [2]      IMPLICIT UniqueIdentifier
                                                      OPTIONAL }}
```

Dabei sind:

```
Version                 ::=     INTEGER{v1(0),v2(1)}

SerialNumer             ::=     INTEGER

AlgorithmIdentifier     ::=     SEQUENCE{
        algorithm       ALGORITHM.&id({SupportedAlgorithms})
        parameters      ALGORITHM.&Type({SupportedAlgorithms}
                                          {@algorithm})
                                OPTIONAL}

Validity                ::= SEQUENCE{
        notBefore       UTCTime,
        notAfter        UTCTime}

SubjectPublicKeyInfo    ::=     SEQUENCE{
        algorithm               AlgorithmIdentifier,
        subjectPublicKey        BIT STRING}
```

Diese Angaben folgen der Syntax für `ASN.1`-Datentypen, wie sie auch im mbox-Standard verwendet werden [ISO93b]. `ASN.1` bildet eine einfache Möglichkeit, systemübergreifend Datenstrukturen zu definieren. SEQUENCE bildet dabei eine Art Record-Struktur nach. Einfache Datentypen sind zum Beispiel INTEGER oder BIT STRING. Einige komplexe Datentypen sind vordefiniert, wie zum Beispiel UTCTime für eine weltweit eindeutige Zeit- und Datumsangabe.

Als Beispiel für einen Public-Key-Algorithmus ist im Anhang zum Standard RSA genannt, das mit einer Schlüssellänge von 512 Bit (Stand 1993) verwendet werden soll.

Im Standard X.509 sind einige Vorschläge gemacht, wie die Benutzer und Certificate Authorities untereinander Zertifikate austauschen können, um Zertifikatsketten zu erzeugen. Die Norm nennt hier folgende Fälle:

- Wenn zwei Benutzer Zertifikate von der gleichen CA besitzen, ist die Zertifikatskette trivial.
- Sind die CAs hierarchisch angeordnet, so merkt sich jeder User den Pfad zwischen der Top-Level-CA und sich selbst und speichert alle für diesen Weg notwendigen Zertifikate und Signaturschlüssel. Dann einigen sich die Benutzer auf einen gemeinsamen Schnittpunkt ihrer Zertifizierungspfade zur Top-Level-CA und können sich so zertifizieren.

- Zusätzlich können CAs *cross-zertifizieren* unabhängig von der Hierarchie zusätzlich Zertifikate tauschen.
- Als letzte Möglichkeit können die Benutzer sich auch unabhängig von CAs vertrauen, wenn sie persönlich und sicher Schlüssel ausgetauscht haben.

Zur Schlüsselgenerierung schlägt die Norm wieder mehrere Verfahren vor. Entweder generieren die Benutzer die Schlüssel selbst oder lassen sie durch eine dritte Partei generieren, die dann den privaten Schlüssel an den User übergibt und danach selbst löscht. Als Spezialfall kann die dritte Partei auch eine Certificate Authority sein.

Als Speicher für den privaten Schlüssel schlägt die Norm physikalisch gesicherte Speicherkarten (SmartCards) vor, die neben dem Private-Key auch den gesichert übertragenen Public-Key der Certificate Authority enthalten.

Um das Zertifikatssystem sicher und konsistent zu halten, schreibt der Standard zudem vor, daß jede Certificate Authority

- sich über die Identität des Users ausreichend informiert, bevor sie seinen Schlüssel signiert und
- keine zwei User mit dem selben Namen zur Zertifizierung zuläßt.

Darüber hinaus legt die Norm fest, daß alle Zertifikate nur eine endliche Gültigkeitsdauer haben sollen und die CAs dafür zu sorgen haben, daß für alle legitimen Benutzer zu jeder Zeit mindestens ein gültiges Zertifikat existiert.

Zudem führt jede CA eine Liste mit eigenen zurückgezogenen Zertifikaten sowie eine Liste mit allen von anderen CAs zurückgezogenen Zertifikaten, sofern diese bekannt sind. Diese sogenannten `CRL`s und `ARL`s werden ebenfalls signiert und über den Verzeichnisdienst den Benutzern zur Verfügung gestellt. `CRL`s (Certificate Revocation Lists) beziehen sich dabei auf Benutzerzertifikate und `ARL`s (Authority Revocation Lists) beziehen sich auf die Signaturschlüssel der anderen CAs.

2.4 SDSI – Simple Distributed Security Infrastructure

Das von R. Rivest und B. Lamport vorgeschlagene System *SDSI* [RL96] stellt eine Sicherheitsinfrastruktur dar, die nicht nur Schlüssel, sondern auch beliebige andere Daten zertifiziert und verteilt. Dazu gehören beispielsweise auch die Gruppenzugehörigkeit von Benutzern oder die Zugriffsrechte auf bestimmte Resourcen. Zudem gibt es automatische Server, die Zertifikate ausstellen oder bestätigen können. Dazu geht SDSI von der ständigen Erreichbarkeit dieser Server aus.

Die zentralen Elemente in SDSI sind öffentliche Schlüssel, genannt *principals*. Innerhalb von SDSI sind keine anderen handelnden Personen (wie z.B. Benutzer, Prozesse, Maschinen) definiert. In der Regel wird jedoch ein *principal* für solch eine handelnde Person stehen.

SDSI hat keine globale Hierarchie und keine speziellen Trusted Authorities. Jeder *principal* kann die Funktion einer Trusted Authority haben. Es gibt keine global eindeutigen Namen, jeder *principal* hat seinen eigenen *namespace*. Das bedeutet, ein *principal* von A kann beim *principal* von B unter „Alice" bekannt sein, während er beim *principal* von C als „A. Smith" gespeichert ist. Um diese *namespaces* zu verbinden, gibt es die Möglichkeit, auf den *namespace* von andern *principals* zu verweisen. Entweder in der Syntax

```
( ref: bob alice)
```

oder abgekürzt (und durch ein Benutzerinterface in die erste Form gewandelt) als

```
bob's alice
```

Verweise sind auch über mehrere *principals* möglich,

```
bob's alice's mother
```

wird dann intern zu

```
( ref: bob alice mother ).
```

Die Aufnahme der anderen *principals* in den eigenen *namespace* geschieht nicht automatisch, sondern manuell. Bei der Aufnahme in den *namespace* werden die Schlüssel dann auch persönlich ausgetauscht. Der Verweis auf den *namespace* eines anderen *principals* führt nicht mehr zum direkten Schlüsselaustausch, denn der andere *principal* dient dann als Trusted Authority für den Inhalt seines *namespaces*. Jedem *principal* ist ein online erreichbarer Server zugeordnet. Dieser Server stellt dann auf Anfrage Zertifikate über den Inhalt seines *namespaces* aus, d.h. er signiert Schlüssel und lokalen Namen eines *principals*. Dadurch hat der Server eines *principals* auch die Kontrolle über dessen private Schlüssel.

Zusätzlich gibt es noch einige globale *principals*, die im *namespace* aller Benutzer stehen und dort nicht umdefiniert werden können, wie zum Beispiel in

```
VeriSign!!'s MicroSoft's CEO
DNS!!'s edu's mit's lcs's theory's rivest
```

Auf diese Weise können existierende Strukturen wie DNS oder globale Trusted Authorities wie VeriSign (http://www.verisign.com) angebunden werden. VeriSign ist eine kommerzielle Trusted Authority, die beispielsweise für die Betreiber kommerzieller Web-Server Zertifikate ausstellt. Mit Hilfe dieser Zertifikate und Web-Browser-Programmen wie Netscape soll die Authentizität von Internet-Angeboten und die sichere Übertragung von sensitiven Daten über das Internet ermöglicht werden.

`VeriSign!!` und `DNS!!` sind dabei diese speziellen *principals*. DNS-Namen können als Spezialfall auch in ihrer gewöhnlichen Form `kenn@cs.uni-sb.de` angegeben werden.

Diese haben die selbe Bedeutung wie `DNS!!'s de's uni-sb's cs's kenn` und werden dann intern als `( ref: DNS!! de uni-sb cs kenn )` dargestellt.

Gruppen sind bei SDSI Mengen von *principals*. Gruppen können entweder durch Aufzählung `( Group: Tom Bill Tim)` oder durch algebraische Ausdrücke `( Group: ( AND: faculty staff ) )` bezüglich anderer Gruppen angegeben werden. Auf die Gruppen anderer *principals* kann wieder auf die übliche Weise zugegriffen werden. Beispielsweise kann eine Organisation `acm` eine Gruppe `members` definieren, auf die dann mit `acm's members` zugegriffen werden kann. Der Server des *principals* `acm` kann nun ein Zertifikat ausstellen, das für einen anderen *principal* dessen Mitgliedschaft in `acm's members` bestätigt.

Über die Zugehörigkeit zu Gruppen kann nun auch der Zugang zu bestimmten Informationen oder die automatische Generierung von Zertifikaten abhängig gemacht werden.

Jeder Server kann zudem sogenannte *signed objects* speichern. Damit stellt der *principal* dieses Servers ein Zertifikat für diese Objekte zur Verfügung. Dabei kann es sich um beliebige elektronisch gespeicherte Daten handeln, z.B. Bilder, Texte, Audiodaten, die dadurch vom Server des *principals* automatisch signiert werden können.

Zudem bietet SDSI noch viele weitere Funktionen, deren genaue Betrachtung hier nicht stattfinden kann. Diese sind unter anderem:

- Vertreter-Definitionen
- Mehrere Funktionen für einzelne Personen, z.B. als Vorsitzender eines Vereines oder als Geschäftsführer einer Firma.
- Gruppen-Signaturen, also Dokumente, die von einer bestimmten Anzahl von Gruppen-Mitgliedern signiert werden müssen.

Offensichtliche Vorteile von SDSI sind die Unabhängigkeit von zentralen Trusted Authorities, obwohl diese durchaus eingebunden werden können, sowie das Fehlen einer globalen Hierarchie.

Allerdings wird dem einzelnen Benutzer von SDSI ein nicht unerheblicher Verwaltungsaufwand zugemutet, denn er muß mit einigen lokalen Benutzern und einigen globalen Trusted Authorities persönlich Schlüsselzertifikate austauschen sowie seinen eigenen Zertifikats-Server betreiben. Zudem fehlen in SDSI bisher alle Maßnahmen, die zu einer Geschwindigkeitssteigerung beitragen könnten. Beispielsweise kann es in den Servern der globalen Trusted Authorities wie `!!DNS` zu Engpässen kommen, da jeder Benutzer der globalen Trusted Authorities Zugriffe über diesen Server durchführen muß.

Eine Gefahrenquelle in SDSI ist das automatische Ausstellen von Zertifikaten. Damit das möglich ist, muß jeder Server über die privaten Schlüssel seines *principals* verfügen. Dadurch können diese Schlüssel möglicherweise durch Angriffe auf diesen Server kompromittiert werden. Mehr noch, auch alle Trusted Authorities müssen einen solchen Server betreiben, der direkten Zugriff auf die geheimen Schlüssel der Trusted Authority hat. Eine X.509-CA hat beispielsweise die Möglichkeit, den privaten Signaturschlüssel in einer Chipkarte zu speichern, die nur für den eigentlichen Signaturvorgang aus einem Tresor entnommen wird. Dagegen muß der SDSI-Server 24 Stunden online sein, um neue Zertifikate ausstellen zu können, sobald ein Benutzer diese benötigt.

Kapitel 3

Entwurf

Nach der Analyse der bekannten Schlüsselverzeichnisse folgt nun der Entwurf des in dieser Arbeit vorgestellten Schlüsselverzeichnisses.

Dieses System ist stark an die Norm X.509 angelehnt, da X.509 eine sehr große Verbreitung als Standard für Zertifikate besitzt. Da es allerdings zur Zeit kein globales X.500-Verzeichnis gibt, mit dem Schlüsseldaten verteilt werden könnten, sind X.509-Schlüssel und -Zertifikate zur Zeit kaum zu verwenden. Im wesentlichen besteht der Unterschied zwischen dem hier vorgestellten Verzeichnis und X.500/X.509 in der Behandlung von zurückgezogenen Zertifikaten und dem verteilten Betrieb des Verzeichnisses.

Nach X.509 müssen Zertifikate bis zum Ablauf ihrer Gültigkeit in einer CRL geführt werden. Das in dieser Arbeit vorgestellten System verwendet stattdessen den Ansatz von SDSI, in dem Zertifikate in regelmäßigen Abständen von der ursprünglichen Quelle erneut geladen werden. Allerdings wird in diesem Fall nicht wie bei SDSI ein neues Zertifikat erzeugt, sondern das ursprünglich ausgestellte Zertifikat beibehalten. Dadurch erhält kein Teil des Verzeichnisses die Kontrolle über private Schlüssel, wie das bei SDSI der Fall ist, sondern die Trusted Authorities sind vollkommen unabhängig vom Verzeichnis. Dennoch kann durch Entfernung des Schlüssels oder Zertifikats aus dem ursprünglichen Server dessen Ungültigkeit an alle Benutzer mitgeteilt werden, und zwar einfach dadurch, daß bei der nächsten Überprüfung das Zertifikat oder der Schlüssel nicht mehr an der ursprünglichen Quelle vorliegt. Als zusätzliche Sicherheit haben die Schlüssel und Zertifikate nur begrenzte Gültigkeit.

Der verteilte Betrieb des Verzeichnisses wird dadurch erleichtert, das die Organisation der Schlüsseldaten in der Speicherung abgebildet werden kann. Beispielsweise kann jede am verteilten Verzeichnis teilnehmende Organisation für ihre Mitglieder einen eigenen Server betreiben und deren Daten in diesem Server unabhängig von anderen Teilen des Verzeichnisses verwalten. Dies ist möglich, da die Server hierarchisch nach beliebigen Kriterien angeordnet werden können, beispielsweise nach geographischen oder organisatorischen Hierarchien.

Im weiteren werden folgende Bezeichnungen verwendet:

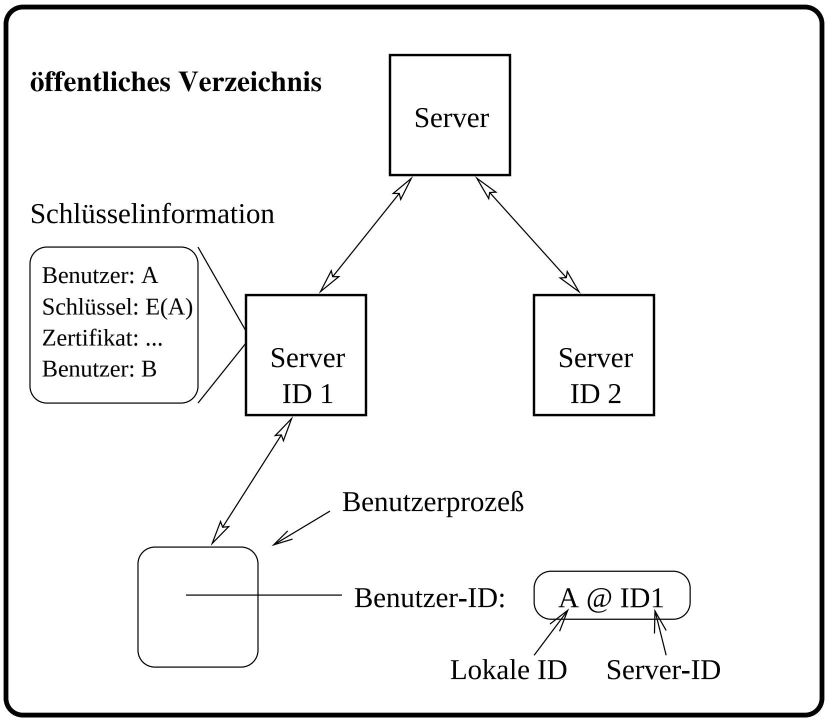

Abbildung 3.1: Schema des öffentlichen Verzeichnisses

- Das *öffentliche Verzeichnis* umfaßt alle an der Verteilung der öffentlichen Schlüssel beteiligten Systeme.

- Ein *Schlüsselserver* verwaltet die Schlüsselinformationen einer bestimmten Gruppe von Benutzern. Der Server ist für die Schlüsselinformation dieser Gruppe *zuständig*. Ein Schlüsselserver ist ein Teil des öffentlichen Verzeichnisses.

- Jedem Schlüsselserver ist eine innerhalb des öffentliche Verzeichnisses eindeutige Bezeichnung zugeordnet, die *Server-ID*. In der Regel handelt es sich dabei um eine Zeichenkette.

- Jedem Benutzer des öffentlichen Verzeichnisses ist *genau ein* Schlüsselserver zugeordnet, in dem seine Schlüsselinformation verwaltet wird. Jeder Benutzer hat innerhalb der Gruppe, für die sein Schlüsselserver zuständig ist, eine eindeutige Bezeichnung, die *lokale ID*.

- Daraus ergibt sich, daß das Tupel aus lokaler ID und Server-ID innerhalb des öffentlichen Verzeichnisses eindeutig ist. Diese Bezeichnung heißt *Benutzer-ID* (Benutzer-ID = (lokaleID, ServerID)).

- Ein *Benutzerprozeß* ist ein Programm, das zu einem Schlüsselserver eine Verbindung aufbaut. Er ist Teil des öffentlichen Verzeichnisses.

3.1 Ablauf einer Anfrage

An das Verzeichnis werden durch die Benutzerprozesse Anfragen gestellt. Dabei heißen Anfragen *gültig*, wenn das Verzeichnis einen Server enthält, dessen Server-ID mit der Server-ID der Anfrage übereinstimmt. Dieser Server kann dann die Anfrage endgültig bearbeiten und entweder die geforderten Informationen liefern oder mit einer Fehlermeldung reagieren.

Eine Anfrage kann daher zu drei Ergebnissen führen:

- Die Anfrage wird beantwortet.
- Die Anfrage ist ungültig. Sobald das erkannt worden ist, wird eine Meldung zurückgeliefert, die den Benutzerprozeß darüber informiert.
- Die Anfrage ist zwar gültig, aber der zuständige Server hat keine Informationen über den Benutzer in der Datenbank. Auch dann wird eine entsprechende Meldung zurückgeliefert.
- Die Anfrage kann aufgrund eines Fehlers nicht bearbeitet werden. In diesem Fall wird ein Felercode zurückgeliefert, der den Benutzerprozeß über die Ursache des Fehlers informiert.

Die Durchführung einer Anfrage erfolgt in mehreren Schritten:

Schritt 1: Der Benutzerprozeß baut eine Verbindung zu einem ihm bekannten Schlüsselserver auf

Schritt 2: Der Benutzerprozeß übergibt die Anfrage.

Schritt 3: Der Schlüsselserver bearbeitet die Anfrage. Dafür werden eventuell Informationen aus anderen Schlüsselservern benötigt. Deren Antworten werden dann wieder zum ursprünglichen Schlüsselserver zurückgesendet.

Schritt 4: Der Schlüsselserver liefert die Antwort der Anfrage. Sie besteht entweder aus der gewünschten Schlüsselinformation oder aus einer Fehlermeldung.

Schritt 5: Der Benutzerprozeß schließt die Verbindung.

Nach Schritt 4 können noch weitere Anfragen an den Schlüsselserver gestellt werden. Dazu wird das Protokoll bei Schritt 2 fortgesetzt. Dies kann dazu dienen, weitere Schlüsselinformationen anzufordern, die zur Überprüfung des Zertifikats der Schlüsselinformation benötigt werden (siehe Abschnitt 1.3).

Dieses Verfahren führt zu einer sehr einfachen Implementierung des Benutzerprozesses, die unabhängig von der Organisation des öffentlichen Verzeichnisses ist (siehe Abschnitt 3.2). Der Benutzerprozeß erhält die Antwort immer vom ersten Schlüsselserver. Der Rest des öffentlichen Verzeichnisses ist für den Benutzer damit unsichtbar.

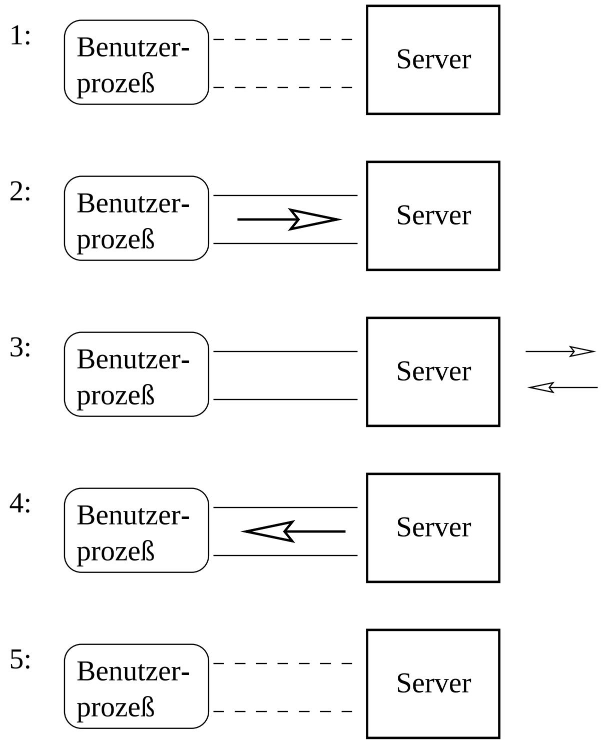

Abbildung 3.2: Schema des Protokollablaufs

3.2 Aufbau des Verzeichnisses

Die Schlüsselinformation eines Benutzers B ist dem für B zuständigen Schlüsselserver bekannt. Es gibt *genau einen* solchen Server für jeden Benutzer. Um die Schlüsselinformation zu erhalten, ist zunächst erforderlich, diesen für B zuständigen Schlüsselserver zu identifizieren. Dann wird zu diesem eine Verbindung aufgebaut und die Schlüsselinformation übertragen. Der für B zuständige Server ergibt sich aus der Server-ID in der Benutzer-ID des Benutzers B. Die Suche nach dem für eine Server-ID zuständigen Schlüsselserver wird ebenfalls von einem Schlüsselserver durchgeführt.

Im folgenden werden nun einige Organisationsformen des Verzeichnisses dargestellt und diese in Bezug auf die Forderungen aus Abschnitt 1.2 untersucht.

3.2.1 Einstufige Organisation des Verzeichnisses

Die einfachste Form der Verbindung zwischen den verschiedenen Schlüsselservern ist folgende:

Jeder Server verfügt über eine Programmfunktion, die zu einer gewünschten Benutzer-ID, in der jeweils eine Server-ID enthalten ist, die Netzwerkadresse des zuständigen Servers findet. Jeder Server verfügt dazu über eine Tabelle, in der zu jeder Server-ID die Adresse des zuständigen Schlüsselservers gespeichert ist. Diese Tabelle heißt *Routingtabelle*. Über den Tabelleneintrag wird der zuständige Server ermittelt und die Anfrage dorthin vermittelt.

Die Routingtabelle muß durch geeignete Maßnahmen immer auf dem neuesten Stand gehalten werden. In der Regel wird dies dadurch erreicht, daß beim Einrichten neuer Schlüsselserver eine Nachricht an die Verwalter aller schon vorhandenen Server gesendet wird, in der die Informationen über den neuen Server enthalten sind. Zur Sicherheitsrelevanz dieses Vorgehens siehe Kapitel 3.

Diese Tabelle wächst linear mit der Anzahl der Schlüsselserver. Der Speicherplatz der Gesamttabelle (in allen Teilen des öffentlichen Verzeichnisses) wächst dadurch allerdings quadratisch, denn in jedem der Server ist jeder Server einmal eingetragen. Zudem ist die Vereinbarung einer Vorgehensweise zur Verbreitung dieser Adressinformationen nötig, um Änderungen oder Neueintragungen in diesen Tabellen in allen Servern durchzuführen.

Ein Vorteil der einstufigen Organisation des Verzeichnisses ist allerdings die Geschwindigkeit, denn eine Anfrage läuft über maximal zwei Server:

- Kann der erste Server die Anfrage beantworten, so ist die Anfrage abgeschlossen.
- Kann der erste Server die Anfrage nicht beantworten, so bestimmt er mit Hilfe der Benutzer-ID die Adresse des zuständigen Servers und führt die Anfrage dort durch. Dieser Server kann die Anfrage dann beantworten.

Auch die Entscheidung über die Gültigkeit der Anfrage kann schon im ersten Server entschieden werden.

Offensichtlich ist dieses Verzeichnis in Bezug auf die Korrektheit optimal. Konsistenz ist trivialerweise erfüllt, da es keine dynamischen Daten gibt, die inkonsistent werden könnten. Dies wird aber durch den maximalen manuellen Verwaltungsaufwand erkauft.

3.2.2 Mehrstufige Verzeichnisse

Um den hohen Verwaltungsaufwand zu umgehen, kann man folgendermaßen vorgehen:

Zunächst sorgt man dafür, daß die Adressinformation ebenfalls verteilt gespeichert wird. Damit müssen im Fall einer Änderung nicht mehr alle Tabellen in allen Servern geändert werden, sondern nur noch die Tabellen, in denen diese Information wirklich gespeichert ist. Auch bei einer Neueintragung muß die entsprechende Information nur an wenigen Stellen eingetragen werden.

Dazu wird die Semantik der Routingtabelle folgendermaßen geändert: Ein Eintrag bedeutet nun, daß alle Anfragen für diese Server-ID *und alle Server-IDs, deren Suffix diese ID ist* an diesen Server weitergeleitet werden sollen. Zusätzlich führt man ein Symbol „*“ ein, das für den Leerstring ϵ steht, der Suffix von allen Server-IDs ist. Weiterhin führt man einen Zähler z ein, der für jede Anfrage die Anzahl der Weitervermittlungen zählt. Dabei sei z_{max} die maximale erlaubte Anzahl der Weitervermittlungen. Für $z \geq z_{max}$ wird die Anfrage nicht weiterverfolgt.

Es gibt genau einen Server, der für eine Gruppe von Benutzern die Schlüsselinformationen verwaltet. Dieser ist für jeden dieser Benutzer in seiner Benutzer-ID spezifiziert. Die Aufgabe der Einträge in der Routing-Tabelle ist es, eine Anfrage, die irgendwo im Verzeichnis nach der Schlüsselinformation eines Benutzers gestellt wird, zu diesem zuständigen Server zu leiten. Daher kann man das Verzeichnis dann als wurzelgerichteten Baum auffassen, in dessen Wurzel der für den Benutzer zuständige Server stehen muß. Wird eine Anfrage durch das Verzeichnis bearbeitet, so wird entlang der Kanten dieses Baumes nach der Antwort gesucht, bis die Wurzel erreicht ist. Um Endlosschleifen zu vermeiden, begrenzt der Zähler z die Anzahl der Weiterleitungen.

Nun stellt sich die Frage, ob ein gegebenes Verzeichnis (Server und Routing-Tabellen mit dieser Semantik) die Korrektheitsbedingung erfüllt. Um dies zu überprüfen, geht man folgendermaßen vor:

Man definiert das Verzeichnis zunächst als Graph. Dabei sind die einzelnen Server die Knoten und die in den Routing-Tabellen eingetragenen Verbindungen die Kanten.

Sei $G = (V, E)$ der gerichtete Graph des Verzeichnisses, V die Menge der Schlüsselserver, E die Kantenmenge des Graphen. Sei die Kante $e = (u, v) \in E$, wenn es einen Routingeintrag in der Tabelle von Server u gibt, der die Adresse von Server v angibt, der also Anfragen an Server v vermittelt. Sei I die Menge der möglichen Server-IDs, $I' \subset I$ die Menge der tatsächlich existierenden Server-IDs, also derjenigen, für die es einen zuständigen Server gibt. Für ein sinnvolles Verzeichnis ist $|I'| = |V|$, da jeder Server genau eine einzigartige Server-ID hat. Man beachte, daß I durchaus unendlich viele Elemente haben kann, zum Beispiel die Wörter beliebiger Länge über einem endlichen Alphabet, aber I' immer nur endlich viele Elemente enthält, denn das Verzeichnis selbst ist endlich. Sei $f : V \times I \to V \cup \{NIL\}$ eine Funktion, die die Routingtabelle in den Schlüsselservern darstellt. Der Wert $f(v, i)$ gibt an, an welchen Server der Server $v \in V$ die Anfrage mit Server-ID $i \in I$ weiterleitet. *NIL* bedeutet, daß die Anfrage als ungültig abgewiesen wird. Befindet sich in der Routingtabelle also kein passender Eintrag, so hat f den Wert *NIL*. Das bedeutet *nicht*, daß für alle $i \notin I'$ die Funktion $f(v, i)$ den Wert *NIL* hat, denn um diese Entscheidung treffen zu können, muß jeder Server die Menge I' kennen, was wieder auf das einstufige Verzeichnis aus Abschnitt 3.2.1 hinausläuft.

Sei nun $i \in I$ die Server-ID in einer Anfrage. Dann sei der Graph $G^{(i)}$ folgendermaßen definiert: $G^{(i)} = (V, E^{(i)})$. Die Kantenmenge $E^{(i)}$ ergibt sich dadurch, daß für jeden Knoten $v \in V$ das Routingziel $f(v, i)$ bestimmt wird: Sei $e = (u, v) \in E^{(i)}$, wenn es $u, v \in V$ gibt, so daß $f(u, i) = v$. $E^{(i)}$ hat also wesentlich weniger Elemente als E, da all jene Kanten gestrichen wurden, die mit der aktuellen Anfrage nichts zu tun haben.

Um nun die Korrektheit anhand dieser Mengen und Funktionen zu überprüfen, geht man folgendermaßen vor:

Die Anwendung der Funktion f auf die Knoten des Graphen G und eine der Server-IDs $i \in I'$, also eine gültige Anfrage, ergibt einen gerichteten Graphen $G^{(i)}$. Hat $G^{(i)}$ nun eine Wurzel und steht nun in der Wurzel dieses Graphen ein Server mit der Server-ID i, so ist Korrektheit für diesen Spezialfall erfüllt worden. Nun testet man diese Bedingung für jedes $i \in I'$. Dabei ist noch zu beachten, das z_{max} dabei genügend groß sein muß, zum Beispiel $z_{max} = |V|$

Die Korrektheit für ungültige Anfragen auf die selbe Weise zu untersuchen, ist nicht möglich,denn da die Menge $I - I'$ möglicherweise unendlich viele Elemente enthält, kann man nicht einfach für jedes $i \in I - I'$ testen. Mit Hilfe des Zählers z kann man jedoch folgendermaßen argumentieren: Für eine ungültige Anfrage gibt es zwei Möglichkeiten, verarbeitet zu werden. Entweder wird sie an einen Server weitervermittelt, der sie nicht mehr weitervermitteln kann. Damit wird sie als ungültig erkannt. Oder die Anfrage wird „im Kreis“ vermittelt wird, läuft also im Graph $G^{(i)}$ einen Zyklus entlang. Dann wird sie aber nach z_{max} Weiterleitungen durch den Zählerstand z als ungültig erkannt. Damit werden auch die ungültigen Anfragen auf jeden Fall korrekt bearbeitet.

Hier folgen nun einige einfache Beispiele für korrekte Verzeichnisse.

Zyklische Verzeichnisse Alle Server können auf einem einzelnen Zyklus angeordnet werden. Jeder Server hat dann nur einen einzigen Routing-Tabellen-Eintrag, der auf den jeweils nächsten Server im Zyklus zeigt.

Domain	Server
*	my.next.server

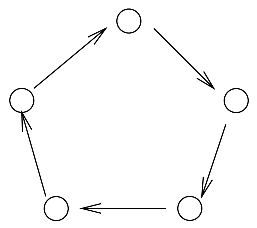

Abbildung 3.3: Zyklisches Verzeichnis

Dieses Verzeichnis erfüllt damit offensichtlich die Korrektheit. Aber eine Anfrage kann im schlechtesten Fall alle anderen Server durchlaufen, bevor sie den zuständigen Server erreicht. Alle ungültigen Anfragen durchlaufen sogar immer jeden Server. Der Verwaltungsaufwand ist sehr gering. Wenn ein Server hinzugefügt werden soll, so sucht man zwei benachbarte Knoten und „hängt“ den neuen Server „dazwischen“. Wird ein Server entfernt, so muß man nur seine beiden „Nachbarn“ informieren. Fällt allerdings ein Server aus, so ist das gesamte System beeinträchtigt.

Das „Vater-Kind“-Verzeichnis Eine andere Möglichkeit besteht darin, zwei Arten von Servern einzuführen, viele „Kinder“ und einen „Vater“. Die Kinder erhalten wieder eine einzeilige Routingtabelle:

Domain	Server
*	vater.knoten.server

Der Vater nun hat eine Routing-Tabelle, die für jede existierende Server-ID die Anfrage an das zuständige „Kind“ weiterleitet.

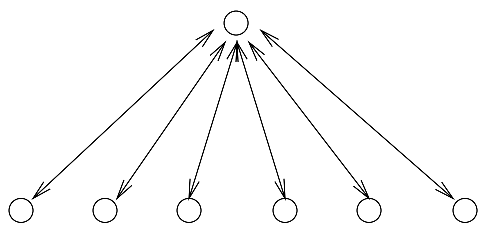

Abbildung 3.4: „Vater-Kind“-Verzeichnis

Domain	Server
id1	kind1.knoten.server
id2	kind2.knoten.server
id3	kind3.knoten.server
id4	kind4.knoten.server
id5	kind5.knoten.server
...	

Auch dieses System erfüllt offensichtlich die Korrektheit. Nur läuft jede Anfrage, die nicht lokal beantwortet werden kann, über den „Vater“. Die Rechnerlast in diesem Knoten wird also erheblich sein und einen „Flaschenhals“ für die Geschwindigkeit des Gesamtsystems darstellen. Die Verwaltung ist hier ebenfalls sehr einfach, denn Änderungen werden nur in der Routingtabelle des „Vaters“ und in der des betroffenen Servers durchgeführt. Wenn ein „Kind“ ausfällt, so ist das System nur minimal betroffen, jedoch ist es bei Ausfall des „Vaters“ vollständig defekt.

Natürlich kann man auf dieser Basis auch das einstufige Verzeichnis aus Abschnitt 3.2.1 implementieren. Dies maximiert allerdings wie gesagt den Verwaltungsaufwand. Allerdings erreicht es optimale Geschwindigkeit und Ausfallsicherheit.

Hierarchische Verzeichnisse Die beiden einfachen Architekturen haben also Nachteile. Es ist also notwendig, eine Architektur zu finden, die sowohl schnell und ausfallsicher, als auch einfach zu verwalten ist.

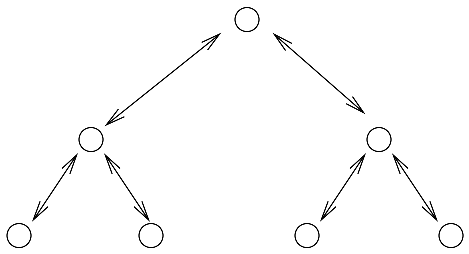

Abbildung 3.5: Ein hierarchisches öffentliches Verzeichnis

Um dies zu erreichen, führt man auf den Servern eine Hierarchie ein. Erhält ein Server innerhalb der Hierarchie eine Anfrage, die nicht für ihn bestimmt ist, so muß er nur entscheiden, ob der zuständige Server sich „unterhalb“ von ihm in der Hierarchie befindet oder nicht. Befindet sich der zuständige Server „unterhalb“, so übergibt der Server die Anfrage dem „richtigen“ seiner Söhne. Befindet sich der zuständige Server „sonstwo“, so übergibt der Server die Anfrage seinem Vater.

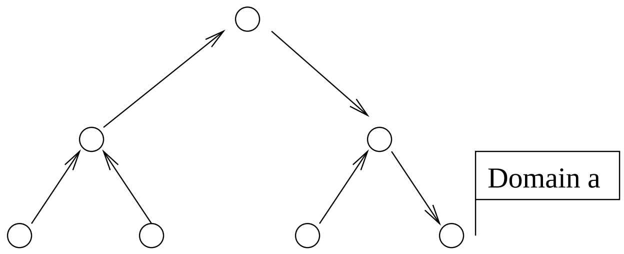

Abbildung 3.6: Das Verzeichnis mit Anfragewegen für die Server-ID **a**

Um zwischen „unterhalb“ und „sonstwo“ zu unterscheiden, vergleicht der Server die Server-ID in der Anfrage $I\!D_A$ mit seiner eigenen ID $I\!D_E$. Ist $I\!D_E$ ein Suffix von $I\!D_A$, so übergibt der Server die Anfrage an denjenigen seiner Söhne, dessen ID ein Suffix von (oder gleich) $I\!D_A$ ist (Suffix-Regel). Findet er keinen solchen Sohn, so ist die Anfrage ungültig. Ist $I\!D_E$ kein Suffix von $I\!D_A$, so übergibt der Server die Anfrage an seinen Vater. Gibt es keinen Vater, so ist die Anfrage ebenfalls ungültig.

Um „passende“ Server-IDs zu erzeugen, ordnet man die Server-IDs mit Hilfe einer Domain-Hierarchie an:

Seien A und B zwei Server-IDs. Ist nun A ein Suffix von B, d.h. gibt es eine Zeichenkette X mit $B = XA$, dann gehört B zur Domain A.

Beispiel: „uni-sb.de“ ist in der Domain „de“

Nun kann man den Verzeichnisgraphen einfach induktiv generieren. Zunächst führt man eine Domain ϵ ein. Da dieser Leerstring Suffix von jeder Zeichenkette ist, liegen damit alle Domains in dieser ϵ-Domain. Diese Domain und die Server-ID ϵ wird einem Server zugeordnet. Um das System nicht unnötig zu komplizieren, verwendet man ihn nur als Routing-Server, d.h. es gibt keine User, für die er zuständig ist. (Induktionsanfang)

Im Induktionsschritt fügt man an die im letzten Schritt generierten Knoten weitere Knoten an, und zwar an jeden Knoten nur weitere Knoten, die auch in seiner Domain liegen. Die dazu gehörigen Server sind nun für die User mit dieser so entstandenen Server-IDs zuständig.

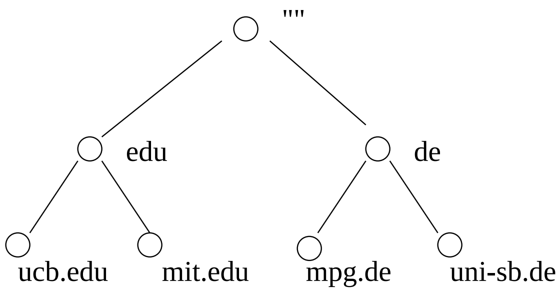

Abbildung 3.7: Domain-hierarchisches öffentliches Verzeichnis

Die Routingeinträge für die einzelnen Server ergeben sich daraus wie folgt:

Der ϵ-Server bekommt für jedes seiner Kinder einen Eintrag in der Routingtabelle:

Domain	Server
first	first.domain.server
second	second.domain.server
...	

Ein „Blatt“ im Hierarchie-Baum bekommt nur einen Eintrag zum übergeordneten Server:

Domain	Server
*	father.domain.server

Ein „innerer Knoten“ hat eine Kombination aus beiden Listen. Zunächst enthält er die Einträge für seine „Kinder“, dann den Eintrag zum Vater:

Domain	Server
first	first.domain.server
second	second.domain.server
. . .	
*	epsilon.server

3.2.3 Verzeichnisse mit externer Hierarchie

Wenn bereits eine hierarchische Anordung von Anwendungsdaten gegeben ist, kann man das Schlüsselverzeichnis an diese Hierarchie „anhängen".

Dies verdeutlicht man am einfachsten an folgendem Beispiel:

> *Anwendung*: Versand elektronischer Post über das Internet (SMTP)
> *Externe Hierarchie*: Domain Name Service (DNS oder BIND)
> *Aufgabe*: User A (a@mpi-sb.mpg.de) will an User B (b@cs.uni-sb.de) eine verschlüsselte Nachricht schicken und benötigt dazu die Schlüsselinformation des Benutzers B. Die externe Hierarchie benutzt nun folgenden Algorithmus, um die Schlüsselinformation zu erhalten:
>
> - Identifiziere die Server-ID (hier cs.uni-sb.de)
> - Gibt es einen Server mit Adresse keyserver.cs.uni-sb.de ?
> - Wenn ja: Erfrage Schlüsselinformation
> - Wenn nein: Entferne passendes Präfix von Server-ID und versuche es erneut.
>
> Existiert nun zum Beispiel ein Server mit Adresse keyserver.uni-sb.de, so findet der Algorithmus diesen im zweiten Schritt.

Durch dieses Verfahren wird die innere Struktur des Verzeichnisses an die der externen Hierarchie angeglichen. Damit kann das Verzeichnis sehr schnell und einfach implementiert werden. Jedoch kann das Verzeichnis niemals über die bestehende externe Hierarchie hinaus erweitert werden. Gleichzeitig können Sicherheitsprobleme aus dieser externen Hierarchie entstehen, die innerhalb des Verzeichnisses dann nicht kontrolliert werden können, da die Daten der externen Hierarchie immer als „wahr" angenommen werden. Kann ein Angreifer die externe Hierarchie manipulieren, so kann er damit möglicherweise das Verzeichnis kompromittieren (siehe auch 5.1.2).

3.2.4 Dynamisches Routing

Wenn eine Anfrage durchgeführt wird, erhält der erste Server bei der Bearbeitung der Anfrage nach und nach immer neue Routinginformationen von den einzelnen Servern, die er bei der Suche nach dem zuständigen Server befragt. Durch eine einfache Protokollerweiterung können diese Server zusätzlich ihre eigene Server-ID mitliefern. Wird dann in Zukunft wieder ein Schlüssel mit dieser Server-ID angefordert, so kann der Server direkt den zuständigen Server befragen, ohne andere Server vorher nach dessen Adresse fragen zu müssen. Dadurch können bei zukünftigen Anfragen möglicherweise Teile der Hierarchie übersprungen werden (Abbildungen 3.8 und 3.9).

Die hier verwendete Technik stammt ursprünglich aus der Theorie der parallelen Algorithmen, wo sie als *Pointer Jumping* bezeichnet wird [JáJ92]. Pointer

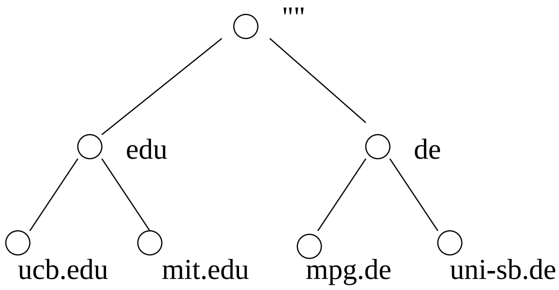

Abbildung 3.8: Anfangszustand des öffentlichen Verzeichnisses

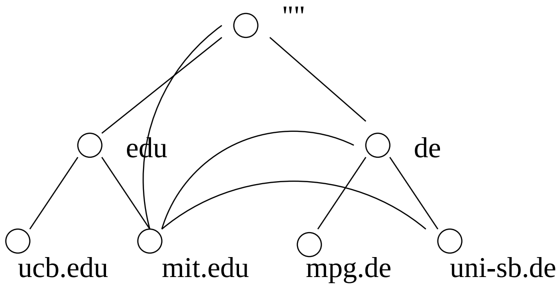

Abbildung 3.9: Zustand, nachdem eine Anfrage durchgeführt wurde

Jumping dient zum Auffinden von Wurzeln in Wäldern. Da aber in der Wurzel eines Anfragebaumes (siehe 3.2.2) immer der Knoten steht, der die Anfrage beantworten kann, verkürzt der Algorithmus die für die nächste Anfrage an dieselbe Server-ID notwendigen Serverzugriffe auf zwei Schritte. Zudem können auch Zugriffe verkürzt werden, die in eine „ähnliche" Richtung gehen (Abbildung 3.10).

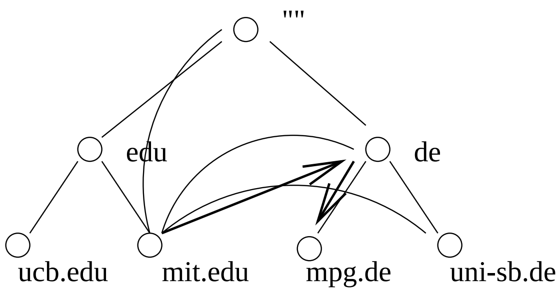

Abbildung 3.10: Eine weitere Anfrage in eine „ähnliche" Richtung

Das Abspeichern derartiger zusätzlicher Routingdaten führt dazu, daß jeder Server nach und nach intern eine Liste von Server-IDs und zuständigen Servern aufbaut. Dies führt zu der gleichen Situation wie in Abschnitt 3.2.1, in der alle

Server über eine vollständige Liste aller anderen Server verfügen.

Ändern sich diese Routinginformationen, nachdem sie in die Routingtabellen eingetragen wurden, so führen die Anfragen, die diese Einträge verwenden zwangsläufig zu fehlgeleiteten Anfragen. Wird eine solche Fehlleitung festgestellt, indem der angesprochene Server nicht reagiert oder keine Informationen über die gewünschte Server-ID zur Verfügung hat, so muß der Listeneintrag wieder entfernt werden. Die Anfrage wird dann ohne den falschen Listeneintrag erneut durchgeführt.

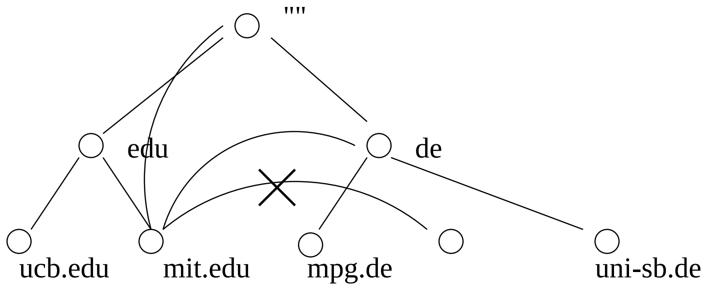

Abbildung 3.11: Korrektur eines falschen Routing-Eintrags

Damit aber nicht die ursprünglichen, sondern nur die dynamisch erzeugten Routingeinträge entfernt werden, erhält jeder Routingeintrag eine Kennzeichnung, die angibt, ob der Eintrag dynamisch erzeugt wurde oder zur ursprünglichen Routinginformation gehört, also statisch ist. Dieser Fall könnte eintreten, wenn ein statisch eingetragener Server durch einen Netzwerkfehler oder Überlastung zeitweise nicht erreichbar ist oder in diesem Server die Schlüsseldatenbank zeitweise nicht lesbar ist, weil gerade Änderungen vorgenommen werden.

Beide Arten von Einträgen werden persistent (zum Beispiel auf Plattenspeicher) gespeichert, um ein erneutes Aufbauen dieser Tabelle beim Neustart eines Servers zu verhindern.

Alle hier vorgestellten Verzeichnisstypen wurden implementiert. Dabei entstand ein Server-Programm, das durch Konfigurationsdateien auf jede der vorgestellten Verzeichnisarchitekturen eingestellt werden kann (siehe Kapitel 4).

3.3 Konsistenz von Schlüsseln und Zertifikaten

Gerade bei Schlüsselinformationen ist es wichtig, immer aktuelle (und damit hoffentlich nicht kompromittierte oder gefälschte) Informationen zu verwenden. Andere Schlüsselverzeichnisse wie z.B. die Norm *X.509* [ISO93b], die in Abschnitt 2.3 beschrieben ist, verwenden hierfür `CRLs`, *Certificate Revocation Lists*. In diesen Listen werden ungültig gewordene Informationen aufgelistet. Will ein Benutzer nun eine Information aus dem Verzeichnis verwenden, muß er zunächst prüfen, ob diese Information in der `CRL` als ungültig aufgelistet ist. Diese `CRLs` werden (aufgrund ihrer Sicherheitsrelevanz) in der Regel von einer übergeordneten Stelle elektronisch signiert. Dadurch muß der Benutzer auch zunächst das Zertifikat dieser Liste prüfen.

SDSI [RL96] (siehe Abschnitt 2.4) benutzt ein anderes System, das auf der wiederholten Überprüfung von Zertifikaten und Schlüsseln basiert.

In dieser Arbeit wurde der Ansatz aus SDSI gewählt, bei dem sich die Verwendung von CRLs erübrigt: Jede Schlüssel- und Zertifikatsinformation enthält eine Angabe über einen Überprüfungszeitraum. Nach Ablauf dieses Zeitraums gilt die Information als fragwürdig und wird über das Verteilungssystem neu angefordert. Das bedeutet, daß eine Information längstenfalls über die Dauer des Überprüfungszeitraums trotz Ungültigkeit verwendet werden kann. Der Urheber der Information muß nun nur dafür sorgen, daß im zuständigen Server die Information gelöscht bzw. aktualisiert wird. Für wichtige Informationen kann der Überprüfungszeitraum so kurz gesetzt werden, daß sie praktisch bei jeder Verwendung überprüft werden. Im Gegensatz zu SDSI werden jedoch die Zertifikate bei der Überprüfung nicht neu erzeugt, um dem Server keine Kontrolle über die privaten Signaturschlüssel der Trusted Authority zu geben, die im Falle eines Angriffs auf den Server in die Hände des Angreifers fallen könnten.

Kapitel 4

Implementierung

Bei der Realisierung der in den Abschnitten 3.2.1, 3.2.2, 3.2.3 und 3.2.4 beschriebenen Algorithmen wurde Wert auf einfache und stabile Implementierung gelegt. Variablen sind in der Regel statisch und Kommunikation zwischen den einzelnen Serverprozessen läuft über das gemeinsame Filesystem des Servercomputers ab. Fast alle verwendeten Tabellen stehen in eigenen Dateien, auf die über den Datenbankkern `gdbm` [FSF] zugegriffen wird. Zur weiteren Beschleunigung der Serverprozesse können hier (je nach verwendetem Betriebssystem) auch schnellere Methoden wie *shared memory*, also von mehreren Prozessen gemeinsam benutzte Speicherbereiche, oder *remote procedures* (siehe Abschnitt 1.4) eingesetzt werden. Diese würden aber die Portierbarkeit einschränken und sind daher hier nicht verwendet worden.

4.1 Das Client-Protokoll

Ein Client kann an einen Server die folgenden Befehle übermitteln:

- `get` <userid@domain>
  Damit wird eine Anforderung nach der Schlüsselinformation für den Benutzer mit der Userid <userid@domain> ausgelöst. Der Server antwortet mit einem Datenblock von folgender Struktur:
  - eine Zeile mit dem Schlüsselwort „`BEGIN`“
  - eine Zeile, in der die Benutzer-ID nochmals enthalten ist, ebenso wie der vollständige Name des Benutzers
  - die Schlüsselinformation als 7-Bit-Ascii-Text
  - eine Zeile mit dem Schlüsselwort „`END`“
  - eine Zeile, in der ein Rückgabewert zurückgeliefert wird. Sie hat die Form:
    `RET:` <Rückgabewert>
    Ein Rückgabewert von 0 gibt an, daß die Anfrage erfolgreich durchgeführt werden konnte.

- `dir`
  Hiermit gibt der Server eine Liste aus, in der alle Benutzer-IDs aufgeführt sind, für die lokal Schlüsselinformationen vorliegen. Der Server antwortet auch hier mit einem Datenblock mit folgendem Aufbau:
  - eine Zeile mit dem Schlüsselwort „`BEGIN`“
  - eine Zeile, in der die Benutzer-ID und der vollständige Name eines Benutzers aufgelistet ist. Solch eine Zeile wird für jeden Benutzer gesendet, für den eine Schlüsselinformation vorliegt.
  - eine Zeile mit dem Schlüsselwort „`END`“
  - eine Zeile, in der ein Rückgabewert zurückgeliefert wird. Sie hat die Form:
    `RET:` <Rückgabewert>
    Ein Rückgabewert von 0 gibt an, daß die Anfrage erfolgreich durchgeführt werden konnte.
- `help`
  Mit diesem Befehl wird ein kurzer Hilfetext ausgegeben, der die Versions-Nummer des Server-Programms sowie alle verfügbaren Befehle enthält.
- `quit`
  Dieser Befehl dient zum Abbruch der Verbindung mit dem Server.
- `sget` <userid@domain>
  Dieser Befehl ist für Anfragen, die von einem Server an andere Server gestellt werden. Eine Erklärung für die Notwendigkeit dieses zweiten Befehls befindet sich in Abschnitt 4.6.1. Er verhält sich genau wie der `get`-Befehl, wenn die Server-ID des angesprochenen Servers mit derjenigen der Anfrage übereinstimmt. Stimmen die IDs nicht überein, so antwortet der Server mit einem anderen Datenblock:
  - eine Zeile mit dem Schlüsselwort „`BEGIN`“
  - ein Routingeintrag, der auf den zuständigen Server zeigt, sofern ein solcher bekannt ist.
  - ein Routingeintrag, der die Server-ID des angesprochenen Servers und seine Adresse enthält. Dieser Eintrag dient zum Verkürzen von zukünftigen Anfragen.
  - eine Zeile mit dem Schlüsselwort „`END`“
  - eine Zeile, in der ein Rückgabewert zurückgeliefert wird. Sie hat die Form:
    `RET:` <Rückgabewert>
    Ein Rückgabewert von 1 gibt an, daß sich zwischen `START` und `END` ein Routingeintrag befindet.

Andere Returnwerte deuten auf Fehler bei der Bearbeitung der Anfrage hin. Dies sind insbesondere:

- ret=-1
  Der Server kann keinen Routingeintrag finden, der zur Domain der Anfrage paßt. Damit wird die Anfrage endgültig abgelehnt, da kein Server für die Server-ID der Anfrage existiert. Der Datenblock ist leer.

- ret=-2
  Der Server kann keinen Routingeintrag finden, der zur Domain der Anfrage paßt. Jedoch ist die Server-ID in der Anfrage nicht aus der Domain des Servers. Vermutlich liegt also ein Routingfehler vor, der möglicherweise auf einen falschen dynamischen Routingeintrag zurückgeht. Der Datenblock ist leer. Dies ist das Signal für den anfragenden Server, einen dynamischen Routingeintrag, der zu dieser Weiterleitung führte, zu entfernen.

- ret=-3
  Der Server ist zuständig, aber die User-ID ist unbekannt. Der Datenblock ist leer.

- ret=-4
  Der Server ist zuständig, aber es ist ein Fehler aufgetreten. Der Datenblock ist leer.

- ret=-5
  Der Server ist nicht zuständig, aber auf Grund eines Fehlers kann auch kein Routing-Eintrag übermittelt werden. Der Datenblock ist leer.

- ret=-6
  Ein Fehler ist aufgetreten, bevor die Zuständigkeit überprüft werden konnte.

- ret=-7
  Die Anfrage konnte syntaktisch nicht ausgewertet werden.

4.2 Zugriff mit dem `telnet`-Programm

Da das hier implementierte Protokoll ein sogenanntes *Simple*-Protokoll (wie beispielsweise auch das Simple Mail Transfer Protocol [Pos82]) ist, kann über eine einfache Socket-Schnittstelle auf den Server zugegriffen werden. Die einfachste Möglichkeit der Entnahme von Schlüsselinformationen ist das Programm `telnet`. Um so vorgehen zu können, muß die Adresse eines Servers bekannt sein. Ein Dialog mit dem Server könnte folgendermaßen aussehen:

```
kenn@mpii-vo:~ [509] telnet crypt11.cs.uni-sb.de 5000
Trying 134.96.243.11 ...
Connected to crypt11.cs.uni-sb.de.
Escape character is '^]'.
%pkeyd>help
PKEYD HELP:
 GET <username> transfers the public key of <username>
```

```
 DIR lists all known usernames
 HELP shows help
 QUIT exits
Version 1.0 (c) 1996 H. Kenn
%pkeyd>dir
START
<jschwarz@cs.uni-sb.de>
<bmeyer@cs.uni-sb.de>
<ingi@cs.uni-sb.de>
<kenn@cs.uni-sb.de>
<mini@cs.uni-sb.de>
<thieloph@cs.uni-sb.de>
END
RET: 0
%pkeyd>get <kenn@cs.uni-sb.de>
START
AU:Holger Kenn <kenn@cs.uni-sb.de>
AK:0082QywWxpu:0:5:042VFvrC..........
END
RET: 0
%pkeyd>quit
Connection closed by foreign host.
```

4.3 Client in LiSA

Wie bereits in der Einleitung erwähnt, wurde dieses Schlüsselverzeichnis zusammen mit der Kryptographie-Bibliothek LiSA entwickelt. Dabei wurde auch ein Client in die Schlüsselverwaltung der Kryptographie-Bibliothek integriert. Werden zur Verschlüsselung, zur Authentisierung oder zur Zertifikatsprüfung öffentliche Schlüssel benötigt, so wird automatisch eine Anfrage an das Verzeichnis durchgeführt. Dazu wird von jedem Benutzer von LiSA über das Konfigurationsfile `cryptrc` ein Server angegeben, an den diese Anfragen gestellt werden sollen. Die Anfragen selbst laufen dann automatisch ab. Kann eine Anfrage beantwortet werden, so wird der so erhaltene Schlüssel in die lokale Schlüsseldatenbank der Bibliothek übernommen. Kann kein Schlüssel gefunden werden, so wird dies an das Anwendungsprogramm zurückgemeldet, das dann den Benutzer informaiert. LiSA ist vollständig in [Sch96] beschrieben. Eine Übersicht über LiSA und das Schlüsselverzeichnis ist in [BKM+96] enthalten.

4.4 Client im WWW

Mit einfachen Mitteln kann auch von einem WWW-Clienten eine Anfrage an das öffentliche Verzeichnis gerichtet werden. Dazu installiert man auf einem WWW-Server ein spezielles Dokument, eine sogenannte HTML-Form.

```
<HTML><HEAD>
<TITLE> Schl&uumlsselsuche </TITLE></HEAD><BODY>
<H2> Bitte geben Sie die User-ID ein (ohne &lt und &gt) !</H2>
<FORM METHOD="GET" ACTION="/cgi-bin/pget">
<INPUT NAME="input">
<INPUT TYPE="submit" VALUE="OK"></FROM>
</BODY></HTML>
```

Diese wird im WWW-Client als Texteingabefeld angezeigt. Der Benutzer kann seine Anfrage nun in dieses Textfeld eintragen. Über eine standardisierte Schnittstelle (das Common Gateway Interface CGI) erzeugt der WWW-Serverprozeß daraus einen Programmaufruf, der auf dem Rechner, auf dem der WWW-Server läuft, ausgeführt wird. Das so gestartete Programm führt die Anfrage durch. Die aufbereitete Antwort wird dann wieder an den WWW-Clienten des Benutzers zurückgeliefert werden. Als Anfrageprogramm bieten sich hier Scriptsprachen wie `Perl 5` [Vro96] an. `Perl 5` erlaubt die direkte Verwendung des Socket-Interfaces und die einfache Umwandlung von Textdateien. Im Anhang befindet sich ein `Perl`-Script, das an dieser Stelle eingesetzt werden kann. Dieses Script wandelt die Eingabe aus der WWW-Seite in eine Server-Anfrage um und wandelt die Ausgabe des Servers in eine HTML-Seite um.

4.5 Das Ein-Server-Verzeichnis

Besteht das öffentliche Verzeichnis nur aus einem einzigen Server, so werden alle Schlüsselinformationen in diesem gespeichert. Der für diese Architektur entwickelte *einfache Server* verwendet mehrere Techniken, um die Zugriffsgeschwindigkeit auf die Datenbank zu erhöhen:

- Parallelität auf Prozeßebene: Der Server besteht aus einem Masterprozeß, der auf Anrufe von Clients wartet. Sobald eine Verbindung besteht, startet der Masterprozeß einen Childprozeß, der die weitere Behandlung der Anfrage übernimmt. Dadurch kann der Server auf unterschiedliche Lastanforderungen flexibel reagieren.

- Datenbankkern: Um Probleme zu umgehen, die beim gleichzeitigen Zugriff auf den Datenbestand des Servers entstehen, wurde ein einfacher Datenbankkern zur Verwaltung des Datenbestandes eingesetzt. Die Datenbank `gdbm` (Gnu DataBase Manager) [FSF] unterstützt gleichzeitigen lesenden Zugriff auf eine Datenbank, automatische Verriegelung bei Schreibzugriffen und effizienten Zugriff über eine Hashtable. Das Programmierinterface bietet einfache Such- und Zugriffsfunktionen.

- Serververhalten: Der Server ist auf die Zusammenarbeit mit UNIX-ähnlichen Betriebssystemen ausgelegt. Daher wird beim Start zunächst ein zweiter Prozeß gestartet, der eigentliche Master-Server-Prozeß. Der Startprozeß wird beendet. Dann schließt der Master-Server-Prozeß alle offenen

Dateien, um sich vom Steuerterminal abzunabeln. Dadurch wird erreicht, daß der Server auch unabhängig von Benutzerprozessen weiterläuft. Zudem legt der Server eine Lockdatei an, um einen erneuten Start eines Servers zu verhindern. In dieser wird die Prozeß-Id des laufenden Servers abgespeichert. Wird der Serverprozeß beendet, entfernt er zuvor diese Lockdatei wieder. Der Betrieb des Servers kann über eine Logdatei verfolgt werden, in der alle Aktionen des Servers dokumentiert werden.

- Portierbarkeit: Der Server verwendet nur wenige betriebssystemspezifische Funktionen. Diese sind so einfach, daß sie auf verschiedenen Betriebssystemen einfach implementiert werden können. Im wesentlichen sind dies die Befehle für das Server-Verhalten (Lockfiles, Prozeß-Start- und Prozeß-Kommunikations-Befehle). Alle betriebssystemspezifischen Funktionen befinden sich in eigenen Prozeduren, die gegebenenfalls auf die einzelnen Betriebssysteme angepaßt werden müssen. Die Funktionen, die eine Netzwerk-Socket-Verbindung aufbauen, sind in einem eigenen Source-File untergebracht, das auf die Socket-Implementierung des Betriebssystems angepaßt werden muß. Durch Änderung dieses Files können auch andere TCP-Implementierungen verwendet werden. Auch die Benutzerprozesse können dieses File verwenden, beispielsweise wurde so die Integration in `LiSA` realisiert. Wie bereits in der Einleitung zu diesem Kapitel erwähnt, werden keine weitergehenden Betriebssystemfunktionen wie zum Beispiel shared memory verwendet. Die Verwendung dieser Mechanismen würde zwar zur weiteren Geschwindigkeitssteigerung des Serverprozesses beitragen, führt aber auch zu aufwendigerem und schlechter portierbarem Programmcode, da diese Funktionen nicht in jedem Betriebssystem und wenn, dann nicht auf die gleiche Weise unterstützt werden. Daher wurden hier die einfacheren Mechanismen verwendet, wenn dadurch die Funktionalität nicht eingeschränkt wurde.

  Die Installationsroutine des Servers verwendet zudem das GNU autoconf System. Damit ist es möglich, daß bei der Übersetzung des Servers die Quelltexte automatisch an das verwendete Betriebssystem angepaßt werden. Zur Zeit werden auf diese Weise die Betriebssysteme Solaris (Sun SPARC), Linux (Intel) und Irix (SGI) unterstützt.

Die Programmstruktur dieses Servers ist in Abbildung 4.1 dargestellt.

Alle weiteren hier vorgestellten Serverprogramme beinhalten auch alle diese Funktionen. Wird ein Server, der auch nicht-lokale Anfragen bearbeiten kann, ohne Angaben über andere Server gestartet, so verhält er sich wie der einfache Server.

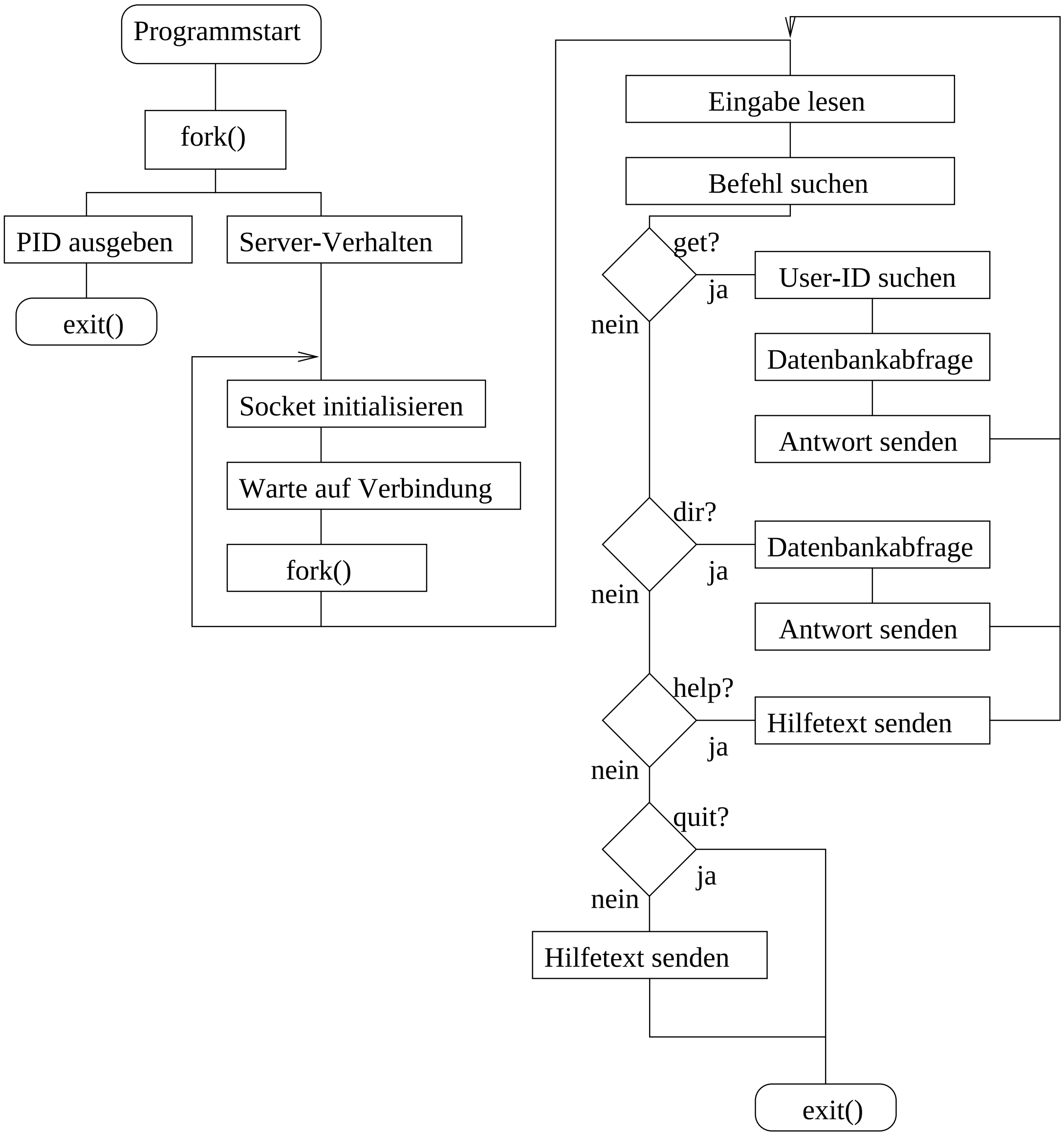

Abbildung 4.1: Die Programmstruktur des einfachen Servers

4.6 Das verteilte Verzeichnis

4.6.1 Nicht-lokale Anfragen

Wenn das öffentliche Verzeichnis mehr als einen Server umfaßt, so muß jede Anfrage den zuständigen Server erreichen. Dies ist auf zwei Arten möglich: Entweder nimmt ein Server die Anfrage entgegen und führt sie selbst beim zuständigen Server durch und übergibt dem ursprünglich anfragenden Client das Ergebnis der Anfrage, oder er übergibt dem Client nur die Adresse des zuständigen Servers und der Client muß die Anfrage dann dort erneut durchführen. Das erste Verfahren hat den Vorteil, daß es auf der Anfragerseite sehr einfach zu implementieren ist, denn die Antwort kommt immer vom ersten Server, unabhängig vom wirklichen Ursprung der Daten. Damit ist das „Innere" des Verzeichnisses vor dem Client verborgen. Dieses Vorgehen wurde im hier implementierten Verzeichnis für die Kommunikation zwischen dem Benutzerprozeß und dem ersten Server verwendet.

Der Nachteil des ersten Verfahrens ist möglicherweise die Erzeugung von langen Laufzeiten, wenn eine Anfrage über sehr viele Server läuft, bis sie den endgültig zuständigen erreicht, denn die Daten, die mit der Anfrage gefunden werden, müssen auf dem Rückweg wieder durch jeden einzelnen Server laufen. Zudem muß jeder Server für die Anfrage weiter Speicherplatz und Netzwerkverbindungen aufrecht erhalten, bis die Anfrage beantwortet ist.

Daher wurde für die Verbindung zwischen den einzelnen Servern des Verzeichnisses das zweite Verfahren implementiert. Um nicht mit den Benutzeranfragen, die nach dem ersten Verfahren abgearbeitet werden, zu kollidieren, muß diese Anfrage ein anderes Format haben. Hier benutzen die Clients den Befehl `get` während die Server den Befehl `sget` verwenden (siehe Abschnitt 4.1).

4.6.2 Der Server für das verteilte Verzeichnis

Gegenüber dem lokalen Server benötigt der Server für verteilte Verzeichnisse weitere Informationen, um seine Aufgabe zu erfüllen. Wenn eine Anfrage an den Server gerichtet wird, so muß der Server zunächst entscheiden, ob diese Anfrage innerhalb des lokalen Servers beantwortet werden kann (dann verhält er sich wie der lokale Server) oder ob die Anfrage an einen anderen Server zur Bearbeitung weitergegeben werden muß. Dies wird anhand der Server-ID der Anfrage entschieden. Stimmt diese mit der Server-ID des Servers überein, so ist der Server für diese Anfrage verantwortlich. Dann kann er die Anfrage vollständig durchführen und die Antwort zurückliefern. Ist die Server-ID dagegen verschieden, so muß er die Anfrage an einen anderen Server weiterreichen. Um den zuständigen Server oder einen Pfad zu ihm herauszufinden, verwendet der Server eine Liste von weiteren Servern, denen jeweils eine Server-ID (und damit ein Teil des öffentlichen Verzeichnisses) zugeordnet ist, die *Routingtabelle*. Zum Einlesen dieser Liste verwendet der Server eine Datei mit folgendem Format

```
#Server-ID              SERVER
cs.uni-sb.de            crypt8.cs.uni-sb.de
cscip.uni-sb.de         cip57.cscip.uni-sb.de
mpi-sb.mpg.de           mpi-sb.mpg.de
```

In diesem Fall wurde als Server-Bezeichnung die Bezeichnung der Internet-Domain gewählt. Dies ist aber nicht unbedingt nötig. Auch eine Konfiguration wie die folgende ist möglich:

```
#Server-ID              SERVER
sb-saar-germany         123.45.6.7
2-75-france             12.34.56.78
```

Eine eingegangene Anfrage wird also an den zuständigen Server weitergeleitet. Dort wird sie beantwortet und das Ergebnis an den ursprünglichen Server gesendet.

Mit diesem Server können also einstufige Verzeichnisse, wie sie in Abschnitt 3.2.1 beschrieben sind, aufgebaut werden. Jedem einzelnen Server müssen daher alle Server-IDs innerhalb des Verzeichnisses bekannt sein. Dies ist nur für kleine Verzeichnisse realisierbar.

Die Programmstruktur dieses Servers ist in Abbildung 4.2 dargestellt.

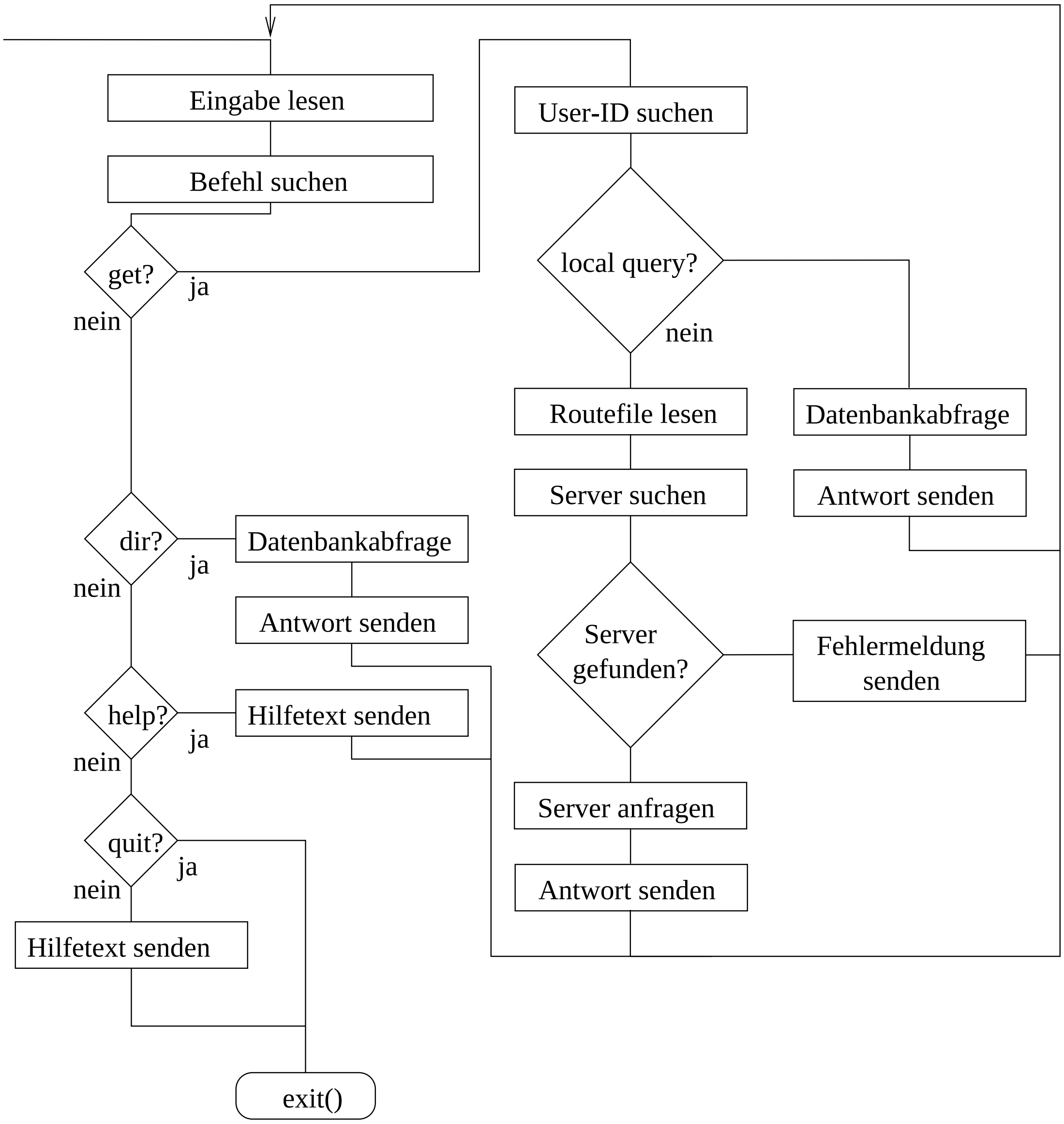

Abbildung 4.2: Die Programmstruktur des Servers für verteilte Verzeichnisse

4.6.3 Der Server für hierarchische Verzeichnisse

Um die Server-IDs in einer hierarchischen Struktur anordnen zu können, wie sie in Abschnitt 3.2.2 beschrieben ist, muß der einzelne Server seine Routingdaten anders interpretieren. Wurde beim einstufigen Verzeichnis eine genaue Übereinstimmung von Server-ID der Anfrage und Server-ID des Routingeintrags gefordert, so reicht nun aus, daß die Server-ID des Routingeintrags ein Suffix der Server-ID der Anfrage ist.

Beispielsweise kann ein Server eine Anfrage mit der Server-ID „cs.uni-sb.de“ erhalten.

Durchsucht der Server nun eine Routingtabelle von folgender Form von vorne nach hinten und wendet dabei die Suffix-Regel an, so wird bereits in der ersten Zeile ein passender Eintrag gefunden. Nur steht in der dritten Zeile der eigentlich richtige Eintrag.

```
de                       1.1.1.1
uni-sb.de                134.96.7.3
cs.uni-sb.de             134.96.1.2
```

Dagegen würde der Server bei folgender Routingtabelle den richtigen Eintrag finden.

```
cs.uni-sb.de             134.96.1.2
uni-sb.de                134.96.7.3
de                       1.1.1.1
```

Wichtig ist also, daß der Server nicht nur einen passenden Eintrag findet, sondern denjenigen, der am weitesten mit der Server-ID der Anfrage übereinstimmt. Um dieses Verhalten zu implementieren, sortiert der Server die Routingeinträge zunächst der Länge nach so, daß die längsten Einträge am Anfang stehen. Dann sucht der Server die Einträge von vorne nach hinten durch und benutzt den ersten Eintrag, dessen Server-ID ein Suffix der Anfrage ist. Damit ist das Verhalten des Servers unabhängig von der Reihenfolge der Einträge in der ursprünglich eingelesenen Routingtabelle.

Für eine Zeile der Form

```
*                        1.2.3.4
```

wird eine Sonderbehandlung implementiert. Sie wird vom Sortieralgorithmus als Server-ID der Länge Null angesehen und damit ganz an den Schluß der Routingtabelle sortiert. Wenn bis zu dieser Zeile kein passender Server gefunden wurde, wird die Anfrage *auf jeden Fall* an den hier angegebenen Server weitergeleitet. Innerhalb der Hierarchie wird solch eine Zeile dazu benutzt, den Vaterknoten anzugeben.

Von dem angesprochenen Server kann der Server entweder die gesuchte Schlüsselinformation erhalten, oder er bekommt einen Routingeintrag, dem er folgen soll. Dieser Routingeintrag vermittelt die Anfrage dann an den vermutlich zuständigen Server. Um diesen Server zu erreichen, fügt der Server zunächst die erhaltenen Routingeinträge temporär in seine lokale Routingtabelle ein und führt die Anfrage erneut durch. Da jetzt neue Einträge in der Routingtabelle stehen, führt die Anfrage jetzt zu einem neuen Server. Mit diesem neuen Server wird dieses Verfahren nun wiederholt, bis die gesuchte Schlüsselinformation gefunden wurde oder die maximale Anzahl der Weiterleitungen erreicht ist.

4.6.4 Dynamisches Routing

Die jeweils angesprochenen Server können zusätzliche Routinginformationen übertragen, damit in Zukunft der übertragende Server schneller gefunden werden kann. Dazu sendet jeder Server eine zusätzliche Zeile, in der seine eigene Server-ID und seine Adresse steht. Diese wird dann in die lokale Routingtabelle eingetragen und dabei vermerkt, daß es sich um einen dynamischen Eintrag handelt. Bei der Suche nach einem zuständigen Server in der Routingtabelle wird dieser Eintrag wie ein statischer Routingeintrag behandelt. Führt jedoch eine von diesem Routingeintrag verursachte Weiterleitung nicht zum gewünschten Ziel, so wird dieser Routingeintrag entfernt und die Anfrage unter Verwendung eines passenden anderen Routingeintrag noch einmal durchgeführt. Dies geschieht so lange, bis nur noch statische Routingeinträge an der Weiterleitung beteiligt sind. Führt auch diese Anfrage nicht zum Ziel, so wird die Anfrage abgewiesen.

Um die Routingtabelle in allen Serverprozessen auf dem gleichen Stand zu halten, wird sie nach Änderungen erneut sortiert, auf Platte geschrieben und vor der Bearbeitung einer Anfrage neu eingelesen.

In der Konfigurationsdatei des Servers kann festgelegt werden, ob der Server dynamisches Routing verwendet oder nicht. Ist das dynamische Routing eingeschaltet, so werden auch die im vorherigen Abschnitt beschriebenen temporären Routingeinträge mit in die globale Routingtabelle geschrieben.

4.6.5 Externe Hierarchie

Bei der Verwendung des Schlüsselverzeichnisses in Weitverkehrsnetzen wie dem Internet ergibt sich eine weitere Methode, den zuständigen Server für eine Anfrage zu finden. Wird (wie auch in den Beispielen vorher) als Server-ID-Bezeichnung die Bezeichnung aus dem Internet-Name-Service (Domain Name Service `DNS` [RFC87a]) verwendet, so kann man durch geschickte Vergabe von DNS-Einträgen und Server-IDs einen einfachen Algorithmus verwenden, um einen zuständigen Server zu ermitteln.

Beispiel: Dem Server einer Server-ID D wird der Server mit dem DNS-Eintrag `keyserver.D` zugewiesen. Alle Anfragen an die Server-ID D werden damit an diesen Server vermittelt. Gibt es diesen Server nicht (d.h. kann kein DNS-Eintrag für den Server gefunden werden), so entfernt der Server ein passendes Präfix von der Server-ID und versucht erneut, einen Server zu finden. Eine Anfragekette für die Anfrage „`user@cs.uni-sb.de`" könnte folgendermaßen aussehen:

```
keyserver.cs.uni-sb.de ?
keyserver.uni-sb.de ?
keyserver.de ?
```

Damit ist DNS als externe Hierarchie für die Schlüsselverteilung verwendet worden, wie in Abschnitt 3.2.3 dargestellt worden ist. Über eine Option in der Konfigurationsdatei kann die Verwendung der externen Hierarchie ein- oder ausgeschaltet werden.

Kapitel 5

Sicherheitsbetrachtung

Nachdem nun das Verzeichnis in Struktur und Implementation vorgestellt ist, wird in diesem Kapitel untersucht, wie es auf Manipulationen von außen reagiert. Ein Ziel dieser Manipulationen könnte zum Beispiel darin bestehen, das Verzeichnis so zu stören, daß die legitimen Benutzer es nicht mehr verwenden können und damit auf den Einsatz von Kryptographie verzichten. Ein anderes Ziel besteht darin, dem Verzeichnis und damit seinen Benutzern falsche Informationen unterzuschieben. In diesem Kapitel wird gezeigt, daß es nur zwei gefährdete Punkte im Verzeichnis gibt, wenn man auf eine externe Hierarchie verzichtet. Der erste Punkt sind die geheimen Signaturschlüssel der Trusted Authorities, der zweite Punkt die Stabilität der einzelnen Server gegen Überlastung oder Beeinflussung des Server-Rechners. Die Sicherung der geheimen Signaturschlüssel muß jedoch den Trusted Authorities überlassen werden.

5.1 Angriffe auf den Server

5.1.1 Denial of Service durch Überlastung

Denial-of-Service-Angriffe sind die Klasse von Angriffen, die darauf abzielen, einen angebotenen Dienst so zu stören, daß kein legitimer Benutzer ihn mehr ordnungsgemäß verwenden kann. Derartige Angriffe sind auch aus dem Bereich der Netzwerk- und Betriebssystemsicherheit bekannt [GWS94]. Eine Möglichkeit, die Übertragung von Schlüsseldaten zu verhindern, ist die Überlastung des öffentlichen Verzeichnisses mit falschen Anfragen. Damit ein derartiger Angriff Erfolg hat, müssen folgende Voraussetzungen erfüllt sein:

- Die generierten Anfragen müssen verschieden sein. Eine einmal durchgeführte Anfrage wird beim zweiten Mal durch die geschwindigkeitssteigernden Maßnahmen vom Server schneller ausgeführt und belastet das öffentliche Verzeichnis daher weniger (siehe Abschnitt 3.2.4).
- Die verwendeten Server-IDs mußsen existieren, sonst kann die Anfrage schnell abgelehnt werden.

- Der Angreifer muß viele Anfragen gleichzeitig durchführen, da der Server auch viele Anfragen gleichzeitig bearbeiten kann.

Da der Server aber zur Bearbeitung vieler paralleler Anfragen ausgelegt ist und für jede Anfrage ein eigener Server-Prozeß gestartet wird, muß auch der Angreifer viele parallele Anfragen starten. Zum Beispiel so:

```
int[MAX] sockets;
char*[MAX] request;
int i;
for (i=0;i<MAX;i++)
        {
         if (sockets[i]=connectTCP(Host,Service)) >=0)
                 write(sockets[i],request[i],strlen(request[i]));
        /* Die Antwort warten wir nicht ab */
        }
```

Allerdings gibt es betriebssysteminterne Begrenzungen für die Anzahl der gleichzeitig offenen Socketverbindungen pro Benutzer und pro Prozeß. Ist diese Begrenzung im Serversystem höher als im System des Angreifers, so kann dieser Angriff nicht erfolgreich sein.

Um eine sichere Überlastung zu erreichen, muß der Angreifer daher die genauen Parameter des Betriebssystems und der Netzwerksoftware des Serversystems kennen und selbst dann kann der Angriff nur mit einer gewissen Wahrscheinlichkeit gelingen, denn seine Zugriffe auf den Server konkurieren mit den Zugriffen der Benutzer und er kann nicht voraussehen, ob einer seiner Zugriffe oder ein Zugriff eines Benutzers zuerst abgelehnt wird.

Vermutlich ist es hier für den Angreifer erfolgversprechender, die Netzverbindung des Rechners, auf dem der Server läuft, physikalisch zu stören oder das Betriebssystem des Serverrechners so zu manipulieren, daß der Serverprozeß beendet wird.

5.1.2 Angriffe auf die externe Hierarchie

Manipuliert man die externe Hierarchie des öffentlichen Verzeichnisses, so führt eine Anfrage an das öffentliche Verzeichnis zur Ablehnung der Anfrage, da keine passende Server-Domain gefunden werden kann. Damit diese Manipulation allerdings erfolgreich sein kann, darf die Server-ID des angeforderten Schlüssels nicht als dynamischer Eintrag in der Routingtabelle des lokalen Servers enthalten sein, da dann die externe Hierarchie nicht mehr verwendet wird. Ein dynamischer Eintrag wird immer dann erzeugt, wenn vor dem Start des Angriffs bereits ein Zugriff auf den Server mit dieser Server-ID durchgeführt worden ist. In dem dynamischen Eintrag steht dann die Adresse des zuständigen Servers, die externe Hierarchie wird dann nicht mehr benutzt, um diese Adresse zu finden (siehe auch 2.1).

5.1.3 Ersetzen des Servers durch ein Trojanisches Pferd

Wenn der Angreifer Zugriff auf den Rechner hat, auf dem ein Server implementiert ist, kann er den dort laufenden Server entfernen und statt dessen einen eigenen Server starten, der die Arbeitsweise eines Servers imitiert. Dadurch können sowohl falsche Daten untergeschoben als auch Anfragen verhindert werden. Hat der Angreifer keinen Zugriff zum Server-Rechner, so kann er dennoch versuchen, die externe Hierarchie so zu manipulieren, daß der angesprochene Server alle Anfragen an einen bestimmten anderen Server weiterleitet. Dieser kann auf einem Rechner gestartet werden, auf den der Angreifer Zugriff hat. Dieser Server kann dann gezielt Anfragen abweisen oder falsche Schlüsselzertifikate übermitteln. Dies ist dann eine Art selektiver Denial-of-Service-Angriff.

Angreifer, die eine Denial-of-Service-Attacke durchführen, sind nur schwer aufzuhalten. Die Abwehr einer derartigen Attacke ist schwierig, wenn der Angreifer über detaillierte Informationen über den Aufbau des Verzeichnisses verfügt. Zudem liegt ein großer Teil der Angriffspunkte nicht im Bereich des Verzeichnisses selbst, sondern befindet sich an den verwendeten Server-Computern, deren Betriebssystemen und an Netzwerkverbindungen. Allerdings fallen erfolgreiche Denial-of-Service-Attacken zwangsläufig auf, da das betroffene System den Betrieb einstellt. Die Benutzer werden dann versuchen, den Urheber der Störung finden und dabei auf den Angreifer stoßen. Das hier entworfene Verzeichnis unterstützt diese Suche durch die Generierung von Zugriffslogdateien, in denen jeder Zugriff zusammen mit Zeit, Ursprung und Inhalt der Anfrage gespeichert wird. Betriebssysteme und Netzwerkverbindungen verfügen über ähnliche Mechanismen. Bei einem Denial-of-Service-Angriff hinterläßt der Täter also zwangsläufig Spuren, die später dazu dienen können, ihn ausfindig zu machen. Daher ist diese Art des Angriffs nur in Ausnahmefällen sinnvoll.

5.2 Angriffe auf die Trusted Authorities

Alle bisher vorgestellten Angriffe führen zu Fehlern bei Benutzern, denn entweder sind Schlüsseldaten nicht verfügbar oder Zertifikate können nicht überprüft werden. Diese Angriffe werden also schnell von den Benutzern als solche identifiziert und führen damit unmittelbar zu einer Suche nach dem Urheber dieser Fehler. Daher sind solche Denial-of-Service-Attacken nur kurzfristig sinvoll. Zudem hinterlassen sie in der Regel Spuren, beispielsweise in den Protokoll-Dateien der beteiligten Server.

Dagegen kann ein Angriff auf eine Trusted Authority bei geschickter Durchführung lange unbemerkt bleiben. Jedoch sind solche Angriffe nur dann möglich, wenn der Angreifer in den Besitz des privaten Signaturschlüssels kommt.

5.2.1 Angriffsversuch ohne den privaten Schlüssel

Führt der Benutzer des Schlüsselverzeichnisses seine Zertifikatsprüfung korrekt durch, so ist dieser Angriff nicht möglich. Dabei versucht der Anwender, wie in Abschnitt 1.3 beschrieben, einen Zertifizierungspfad zu einer Trusted Authority aufzubauen, der er vertraut (und deren öffentlichen Signaturschlüssel er auf einem sicheren Weg erhalten hat).

Wenn nun ein Benutzer C sich als Benutzer B ausgibt, so kann er durch Manipulation des Verzeichnisses zwar A einen falschen öffentlichen Schlüssel mitteilen, dessen Zertifikat ist allerdings fehlerhaft und A wird ihm nicht vertrauen, denn A kann keinen Zertifizierungspfad zu seiner TA aufbauen. Könnte C aber eine gültige Signatur erzeugen, so würde A dem Schlüssel vertrauen. Für diesen Angriff benötigt C aber den privaten Schlüssel der TA von B.

5.2.2 Kompromittierte Zertifizierungsschlüssel

Gelingt es einem Angreifer C, den privaten Signaturschlüssel seiner TA in Erfahrung zu bringen, so kann er sich zunächst als C' ausgeben (Abbildung 5.1).

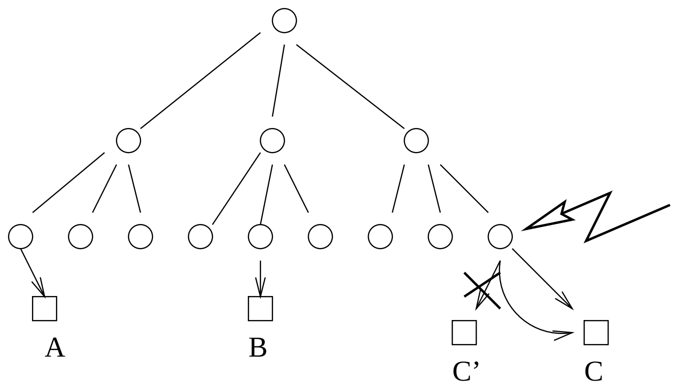

Abbildung 5.1: C kennt den Signaturschlüssel seiner TA

Damit kann sich C für jeden anderen Benutzer ausgeben, der ebenfalls von der angegriffenen Trusted Authority zertifiziert wurde, denn er kann einfach ein neues Schlüsselpaar generieren und dieses mit der „passenden“ Benutzer-ID zusammen mit dem TA-Schlüssel signieren. Jeder andere Benutzer hält dann den bekanntgegebenen öffentlichen Schlüssel für den des wirklichen Benutzers, denn das Zertifikat, das die Identität bestätigen soll, ist ja korrekt. In Wirklichkeit kennt C den zugehörigen privaten Schlüssel und kann alle Nachrichten lesen, die unter Verwendung des neuen Schlüssels an den ursprünglichen Benutzer gesendet worden sind. Ebenso kann er mit einem Signaturschlüsselpaar Nachrichten signieren, die er selbst verfaßt hat. Beim Empfänger der Nachrichten erscheint es jedoch so, als sei der ursprüngliche Benutzer der Absender.

Die Kommunikation zwischen zwei Benutzern kann C ebenfalls belauschen, indem er die sogenannte *man-in-the-middle*-Attacke durchführt. Dabei simuliert er A gegenüber B und B gegenüber A. Dazu generiert er zwei neue Benutzer A' und B'. Nun übermittelt er die Schlüsseldaten so, daß A den Schlüssel B' erhält, ihn aber für den Schlüssel von B hält und umgekehrt. Sendet nun A eine Nachricht an B (die dann mit dem Schlüssel von B' verschlüsselt worden ist), so fängt C diese Nachricht ab und dekodiert sie, da er den geheimen Schlüssel von B' kennt. Um seine Anwesenheit zu verbergen, verschlüsselt er nun die Nachricht mit Bs echtem Schlüssel und schickt sie an B weiter. Ebenso kann C die Verbindung von B nach A belauschen und eventuell verwendete digitale Signaturen fälschen.

Alle Zertifizierungsketten laufen in diesem Fall über die angegriffene Trusted Authority und werden als korrekt erkannt.

Damit sind alle Schlüssel, die ebenfalls von der TA von Benutzer C signiert wurden, unsicher. Genauso kann der Benutzer C sich gegenüber C' (und damit jedem Benutzer, dessen Schlüssel auch von seiner TA signiert wurde) als jeder Benutzer außerhalb dieses TA-Bereichs ausgeben, indem er den Rest des TA-Systems mit neu generierten Schlüsseln simuliert (Abbildung 5.2). Auch hier führen alle Zertifizierungswege über die angegriffene Trusted Authority.

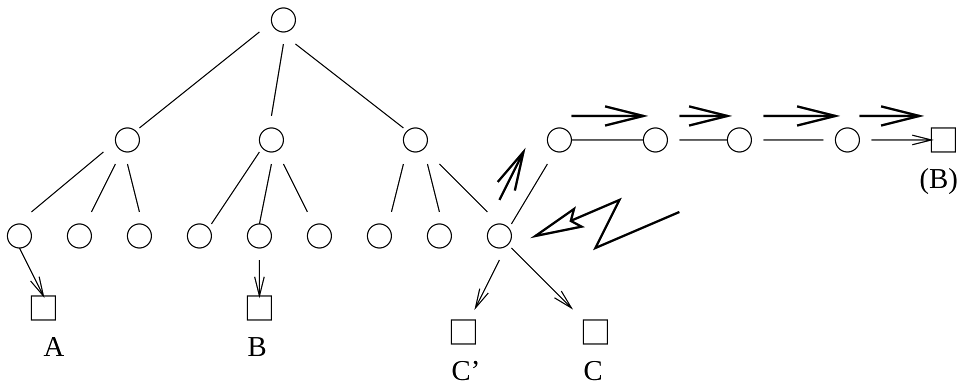

Abbildung 5.2: C simuliert alle anderen Benutzer

Die Auswirkungen gehen aber noch darüber hinaus. C kann andere Teile des Systems ebenfalls beeinflussen, indem er Schlüssel signiert, die nicht zum Bereich seiner TA gehören und in diesen Schlüsseln Informationen unterbringt, die dazu führen, daß die Zertifizierungskette zu der von ihm kontrollierten TA führt (Abbildung 5.3).

Dies kann C folgendermaßen erreichen: In der Benutzeridentifikation von B gibt es ein Feld, in dem die Zugehörigkeit von B zu seiner TA steht. In der Regel steht dort einfach die Identifikation der Trusted Authority die den Schlüssel signiert hat. Wenn nun C einen neuen Benutzer (B) generiert, trägt er in dieses Feld einfach die von ihm kompromittierte TA als zuständige TA für diesen Benutzer ein. Dadurch wird der Benutzer (B) von der kompromittierten TA „adoptiert“. Prüft nun A die Signatur, so wird auch diese Prüfung über die kompromittierte

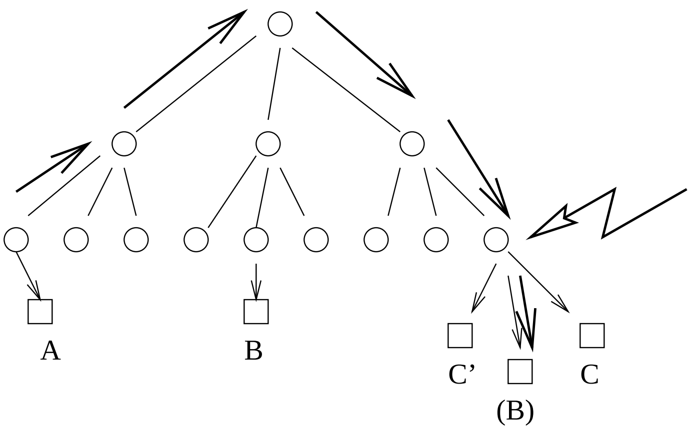

Abbildung 5.3: C simuliert B gegenüber A

Trusted Authority geleitet.

Um diesen Angriff zu vermeiden, muß die Benutzer-ID so beschaffen sein, daß sie durch Änderung der verwendeten Trusted Authority unbenutzbar wird. Insbesondere sollten Applikationshierarchien und Zertifikatshierarchien gleich sein. Ein Ansatz, der eine Hierarchie für die Applikation (Email nach SMTP) und eine andere zur Zertifizierung benutzt (Zertifikate nach X.509) ist durch diese Art von Angriffen gefährdet.

5.3 Schlußfolgerungen

Die Vermeidung von Denial-of-Service-Attacken ist nur sehr schwer durchzuführen, denn sie sind durch viele verschiedene Faktoren begünstigt, die nicht vom Schlüsselverzeichnis zu beeinflussen sind, beispielsweise die Stabilität von Netzverbindungen oder Betriebssystemen. Jedoch ist das Verzeichnis so entworfen, daß diese Angriffe leicht zu erkennen sind und Spuren hinterlassen, die Rückschlüsse auf die Täter zulassen.

Die Kompromittierung von privaten Signaturschlüsseln von Trusted Authorities ist dagegen ein ernstzunehmendes Problem, denn diese können nicht so einfach festgestellt werden. Allerdings gibt bietet das vorgestellte Verzeichnis Möglichkeiten, die Schäden auf ein Mindestmaß zu begrenzen. Insbesondere kann durch eine geschickte Auswahl der hierarchischen Anordnung der Benutzer-IDs eine Absicherung gegen den „Adoptions"-Angriff erreicht werden. Wenn die verwendeten Benutzer-IDs dem lokalen Benutzer zudem angezeigt werden und diese so einfach strukturiert sind, daß hier Manipulationen auffallen, ist dieser Angriff fast ausgeschlossen.

Anhang A

Quelltexte und Programmdokumentation

A.1 Unix Man-Page des Servers

```
pkeyd(1)                      1996                     pkeyd(1)



NAME
     pkeyd - a public key information server

SYNOPSIS
     pkeyd [ port ]

DESCRIPTION
     pkeyd is a TCP-Server which connects to the given port
     and waits for a connection. A connecting party can do sev-
     eral queries on the server's database. After the connection
     has been established, the server sends its prompt :

          %pkeyd>

     Now the connecting party can issue several commands:

COMMANDS
     get <username>
          transmits the public key stored in the record of user-
          name

     get <username@domain>
          Tries to connect to the appropriate server  for  domain
```

```
          and  executes  a  query for username, then returns the
          answer to the original connecting party.

     dir  lists all users with records in this server

     help shows a short description of the commands  and  version
          informations

     quit terminates the connection


OPTIONS
     There are no options at this time

OPERANDS
     There is only one possible operand

     [port]  The  port to connect to.  If no [port] is specified,
             the getservbyname(3N) system call is  used  to  find
             the  service  named  pkeyd.   This  behaviour can be
             changed in the source file config.h or in  the  con-
             figuration  file  pkeyd.conf which is located in the
             server's home directory specified in config.h.


CONFIGURATION
     The configuration file pkeyd.conf has the following format

          variable=value

     The following variables can be set

     routefile
          specifies the path to the route file of the server

     mydomain
          sets the own domain  of  the  server  to  auto-identify
          local   queries  which  are  entered  as  <user@domain>
          queries.

     keydb
          specifies the path to the gdbm database containing  the
          public key data records

     securedir
          specifies  the  path to the directory where the logfile
          and the lockfile are generated.
```

#

initiates a comment.

ROUTING

To find the appropriate server for the get <username@domain> queries, the server uses a route file named pkeyd.route in the current directory. Again, the name and location of this file can be changed in config.h or pkeyd.conf. It has the following format

domain server

Additionally, # can be used to initiate a comment.

LOGGING AND LOCKING

When starting, the server generates a lock file called pkeyd.lock in which it prints its process id and a logfile called pkeyd.log in which it logs connections and queries depending on the debug level specified in config.h.

FILES

config.h, pkeyd.conf, pkeyd.route, pkeyd.log, pkeyd.lock

SEE ALSO

LISA, dbconv dbprint

NOTES

No notes.

BUGS

No (known) bugs. Please report to kenn@cs.uni-sb.de

COPYRIGHT

(c) 1996 H. Kenn

A.2 PERL-Script für den WWW-Server

```
#! /usr/bin/perl
        ($who) = @ENV{"QUERY_STRING"};
        $who=~s/input=//;
        $port = 5000 ;
        $them = 'crypt11.cs.uni-sb.de' ;
        use Socket;
        $sockaddr = 'S n a4 x8';
        chop($hostname = `hostname`);
        ($name, $aliases, $proto) = getprotobyname('tcp');
        ($name, $aliases, $port) = getservbyname($port, 'tcp')
            unless $port =~ /^\d+$/;
        ($name, $aliases, $type, $len, $thisaddr) =
                        gethostbyname($hostname);
        ($name, $aliases, $type, $len, $thataddr) = gethostbyname($them);
        $this = pack($sockaddr, AF_INET, 0, $thisaddr);
        $that = pack($sockaddr, AF_INET, $port, $thataddr);
        socket(S, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
        bind(S, $this) || die "bind: $!";
        connect(S, $that) || die "connect: $!";
        print "Content-type: text/html\n";
        print "\n";
        print "<HTML><HEAD>\n";
        print "<TITLE> Schl&uumlsselausgabe </TITLE>";
        print "</HEAD><BODY>\n";
        print "<H2> Schl&uumlssel von &lt";
        print $who;
        print "&gt : </H2>\n";
        select(S); $| = 1; select(stdout);
        $LENGTH= 7;
        sysread S,$dummy,$LENGTH ;
        print S "GET <",$who,">\015\n";
            while (<S>) {
                if (/^START/) { print "<XMP>\n";}
                if (!(/^START/ || /^END/)) {
                        $line=$_;
                        chomp $line;
                        while (length $line >80) {
                        print substr $line,0,80;
                        print "\n";
                        $line = substr $line,80 ;
                        }
                        print $line;
                        print "\n";
                        }
                if (/^END/) { print "</XMP>";print S "QUIT\n";goto raushier;}
            }
        raushier:
        print "</BODY></HTML>\n";
        close(S);
```

Literaturverzeichnis

[BKM+96] I. Biehl, H. Kenn, B. Meyer, B. Müller, J. Schwarz und C. Thiel. LiSA - Eine C++-Bibliothek für kryptographische Verfahren. In *Digitale Signaturen*. Vieweg, 1996.

[CS92] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP*. Prentience Hall, 1992.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6), November 1976.

[DNS95] DNS Security Working Group. Domain Name System Security Extensions. Internet Draft, Oktober 1995.

[FSF] Free Software Foundation, Inc. *Gnu dbm library 1.7.3*.

[Gro95a] IPSEC Working Group. Internet Security Association and Key Management Protocol (ISAKMP). Internet Draft, November 1995.

[Gro95b] PKIX Working Group. Internet Public Key Infrastructure. Internet Draft, November 1995.

[GWS94] S. Garfinkel, D. Weise, and S. Strassmann. *The UNIX-HATERS Handbook*. IDG Books, 1994.

[ISO93a] ISO. *Information technology - Open Systems Interconnection The Directory: Abstract Service Definition Recommendation X.511 ISO/IEC 9594-3*, 1993.

[ISO93b] ISO. *Information technology - Open Systems Interconnection The Directory: Authentication Framework Recommendation X.509 ISO/IEC 9594-8*, 1993.

[ISO93c] ISO. *Information technology - Open Systems Interconnection The Directory: Models Recommendation X.501 ISO/IEC 9594-2*, 1993.

[ISO93d] ISO. *Information technology - Open Systems Interconnection The Directory: Overview of Concepts, Models, and Services Recommendation X.500 ISO/IEC 9594-1*, 1993.

[ISO93e] ISO. *Information technology - Open Systems Interconnection The Directory: Procedures for Distributed Operations Recommendation X.518 ISO/IEC 9594-4*, 1993.

[ISO93f] ISO. *Information technology - Open Systems Interconnection The Directory: Protocol Specifications Recommendation X.519 ISO/IEC 9594-5*, 1993.

[ISO93g] ISO. *Information technology - Open Systems Interconnection The Directory: Replication Recommendation X.525 ISO/IEC 9594-9*, 1993.

[ISO93h] ISO. *Information technology - Open Systems Interconnection The Directory: Selected Attribute Types Recommendation X.520 ISO/IEC 9594-6*, 1993.

[ISO93i] ISO. *Information technology - Open Systems Interconnection The Directory: Selected Object Types Recommendation X.500 ISO/IEC 9594-7*, 1993.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

[Kop95] H. Kopka. *LaTeXEinführung.* Addison-Wesley, 1995.

[KR88] B. W. Kerningham and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[LCPM85] B. Leiner, R. Cole, J. Postel, and D. Mills. The DARPA Protocol Suite. In *IEEE INFOCOM*, 1985.

[Pla92] P. J. Plauger. *The standard C library.* Prentice Hall, 1992.

[Pos82] J. B. Postel. Simple Mail Transfer Protocol. Request for Comments: 821, August 1982.

[Rag93] S. A. Rago. *UNIX System V Network Programming.* Addison-Wesley, 1993.

[RFC80] User Datagram Protocol. Request for Comments: 768, August 1980.

[RFC81] Transmission Control Protocol. Request for Comments: 793, September 1981.

[RFC83] Telnet Protocol Specification. Request for Comments: 854, Mai 1983.

[RFC85] File Transfer Protocol. Request for Comments: 959, Oktober 1985.

[RFC87a] Domain Names : Concepts and Facilities. Request for Comments: 1034, November 1987.

[RFC87b] Domain Names : Implementation and Specification. Request for Comments: 1035, November 1987.

[RFC87c] Official Internet Protocols. Request for Comments: 1011, November 1987.

[RFC87d] XDR: External Data Representation Standard. Request for Comments: 1014, Juni 1987.

[RFC88] Remote Procedure Call Protocol. Request for Comments: 1057, Juni 1988.

[RFC89] Network File System Protocol. Request for Comments: 1094, März 1989.

[RFC91] BSD Rlogin. Request for Comments: 1258, September 1991.

[RFC93a] Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. Request for Comments: 1421, Februar 1993.

[RFC93b] Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. Request for Comments: 1422, Februar 1993.

[RFC93c] Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. Request for Comments: 1423, Februar 1993.

[RFC93d] Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. Request for Comments: 1424, Februar 1993.

[RL96] R. Rivest and B. Lampson. SDSI - A Simple Distributed Security Infrastructure, Version 1.0. http://theory.lcs.mit.edu/~rivest/sdsi10.html, April 1996.

[Sch96] J. Schwarz. Entwurf und Implementierung einer objektorientierten Softwarebibliothek kryptographischer Algorithmen. Diplomarbeit, Universität des Saarlandes, 1996.

[Sta95] W. Stallings. *Network and Internetwork Security, Principles and Practice.* Prentice Hall, 1995.

[Ste92] W. R. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

[Vro96] J. Vromans. *Perl 5 Desktop Reference.* O' Reilly and Associates, Inc., 1996.