Diplomarbeit zum Thema

Sicherheit in lokalen Netzen am Beispiel einer netzwerkweiten Verwaltung von UNIX-Systemdateien

von

Thomas Omerzu
Am Bertholdshof 2
4600 Dortmund 1

Matr.-Nr. 270666351716 / 0029348

Universität Dortmund
Fachbereich Informatik

24. Juli 1990

An dieser Stelle möchte ich allen danken, die mich bei der Realisierung dieser Arbeit unterstützt haben, insbesondere Peter Wittler und Jürgen Spies für wertvolle Anregungen und der Geschäftsleitung der Firma Quantum GmbH für die Bereitstellung aller notwendigen Resourcen.

Dieses Dokument wurde erstellt auf einer Sun 3/50 mit Hilfe des Textformatierers *troff*. Die beiden Grafiken wurden mit dem FrameMaker gezeichnet. Gedruckt wurde die Arbeit auf einem Apple-LaserWriter II.

Inhaltsübersicht

1. Einleitung 4
2. Aufgabenstellung 5
2.1. Funktionalität 5
2.2. Sicherheitsanforderungen 6
2.3. Konsistenz 7
2.4. Flexibilität 7
3. Existierende Lösungsansätze 9
3.1. Authentifizierung in Computernetzen 9
3.1.1. Einleitung 9
3.1.2. Grundlagen 10
3.1.3. Interaktive Verbindungen 11
3.1.4. Authentifizierungsserver 13
3.1.5. Unidirektionale Verbindungen 14
3.1.6. Digitale Unterschrift 15
3.1.7. Kommentar der Autoren 16
3.2. Sicherheitsservices und Sicherheitsmechanismen 17
3.2.1. Sicherheitsservices 17
3.2.2. Sicherheitsmechanismen 18
3.2.3. Verschlüsselungsalgorithmen 19
3.2.4. Key Management 22
3.3. Yellow Pages 23
3.3.1. Namenskonventionen 23
3.3.2. Funktionsweise 23
3.3.3. Vor- und Nachteile 23
3.3.4. Sicherheitsaspekte 24
3.4. RPC vs. Secure RPC 26
3.4.1. Funktionsprinzip 26
3.4.2. Funktionsweise 27
3.4.3. Authentifizierung unter RPC 27
3.4.4. Ergänzungen im Secure RPC 28
3.5. Kerberos 30
3.5.1. Intention 30
3.5.2. Realisierung 30
3.5.3. Ablauf der Authentifizierung 31
3.5.4. Kerberos-Datenbank 33
3.6. Andere Ansätze 35
3.6.1. RFC-1004: A Distributed-Protocol Authentication Scheme 35

3.6.2. RFC-1040: Privacy Enhancement for Internet Electronic Mail 36
3.7. Zusammenfassung und Vergleich 38
4. Ein Modell 41
4.1. Grobstruktur 41
4.2. Anforderungen an das Kommunikationsprotokoll 43
4.3. Datenmodell 44
4.4. Transfermodell 44
4.5. Paketaufbau und Sicherungsstrategie 46
4.6. Schlüsselhierarchie 47
4.7. Kommunikationsablauf 47
5. Realisierung 51
5.1. Grobspezifikation 51
5.2. Implementierung 53
5.2.1. Implementierungsentscheidungen 53
5.2.2. Konfigurationssprachen 57
5.2.3. Paketaufbau und Sicherungsstrategie 61
5.2.4. Kommunikationsablauf 62
5.2.5. Datenbankformat 63
5.2.6. Dateneditor auf dem Master 64
5.2.7. Konversionsprogramme 64
5.2.8. Systemeinbindung 65
5.3. Anwendungsbeispiel 66
5.3.1. Installation 66
5.3.2. Konfiguration 66
5.3.3. Test 68
5.3.4. Betrieb 69
5.4. Erfahrungen, Bewertung und Ausblick 70
5.4.1. Stärken 70
5.4.2. Implementationsbedingte Schwächen 72
5.4.3. Konzeptionelle und prinzipielle Schwächen 74
5.4.4. Möglichkeiten 74
6. Anhang 75
6.1. Literatur 75
6.2. Glossar 78

1. Einleitung

Heutzutage ist die Ansicht weit verbreitet, daß sich Rechnersicherheit und Netzwerkanschluß gegenseitig ausschließen. Das Vorhandensein eines solchen Anschlusses wird oftmals gleichgesetzt mit der Existenz von illegalen Zugriffen unberechtigter Personen auf den Rechner.

Tatsächlich ist der eigene Rechner gefährdet, und zwar insbesondere dann, wenn er berechtigten Anwendern über das Netz sicherheitskritische Zugriffe erlaubt. Darunter fallen u. a. Probleme wie die Fernwartung von Systemdateien, in denen Informationen wie Zugangsberechtigungen und ähnliche Daten geführt werden.

Immer häufiger werden von Anwendern *mehrere* (gleiche oder verschiedene) Rechner verwendet, um so die Vorteile einer heterogenen Rechnerstruktur zu nutzen. Damit kann ein großer Arbeitsaufwand für die Systemadministration verbunden sein. Auf jedem Rechner müssen die speziellen Systemeigenheiten beachtet werden und für jede administrative Tätigkeit ist ein Login auf dem entsprechenden Rechner nötig. Dies gilt für die Administration aller Systemdateien.

Da aber gleichzeitig auch immer mehr lokale Netze zum Einsatz kommen, bietet es sich an, wenigstens Teile der Systemadministration global von einem Zentralrechner aus durchzuführen. Ein solches Vorgehen ist aber nur vertretbar, wenn entsprechende Sicherheitsvorkehrungen getroffen werden.

In dieser Arbeit soll daher zunächst geklärt werden, welche Sicherheitsprobleme bei Verwendung eines Netzwerkanschlusses auftreten können und wie man diese Probleme vermeiden oder umgehen kann. Dazu werden einige existierende Netzwerksoftwarepakete bzw. die darin enthaltenen Sicherheitsvorkehrungen genauer untersucht.

Danach wird ein Modell entwickelt, welches eine netzglobale Systemadministration unterstützt, dabei aber die bekannten Sicherheitsprobleme geeignet umgeht bzw. abfängt. Dazu wird zunächst überlegt, welche der bekannten Sicherheitsrisiken für dieses Problem relevant sind und welche Lösungsansätze geeignet sind, um hier zum Einsatz gebracht zu werden.

Abschließend wird dann der praktische Einsatz dieser Überlegungen getestet werden, indem das zuvor entwickelte Modell für eine netzwerkweite Administration von Systemdateien auf einem TCP/IP-basierten, lokalen Netz von verschiedenen UNIX-Rechnern implementiert wird.

Ergebnis ist dann ein Programmsystem, welches eine komfortable Verwaltung einer Gruppe von Rechnern im lokalen Netz ermöglicht und dabei – richtig eingesetzt – die Sicherheit dieser Rechner sogar soweit erhöht, daß sie diejenige von Rechnern ohne Netzanschluß übertrifft.

2. Aufgabenstellung

Viele UNIX-Anwender betreiben ein kleines bis mittleres lokales Netz. Wenn die Anzahl der Rechner im Netz wächst, zeigt sich, daß der Administrationsaufwand überproportional stark ansteigt. Eine Verringerung des Aufwandes erreicht man nur, indem Teile der Verwaltung zentral ausgeführt werden. Dabei kann man z. B. Systemkonfigurationsdateien, die auf allen Rechnern gleich sind, zentral bearbeiten und dann über das Netz auf die anderen Rechner kopieren. Lösungen dieser Art führen aber zu Lücken in der Sicherheit der Rechner, die eine solche Systemdatei vom Netz entgegennehmen. Dies gilt um so mehr, je 'komfortabler' die zentrale Verwaltung gestaltet ist, da dann i. a. auf spezielle Zugangskontrollen verzichtet wird. Benutzer mit entsprechenden Kenntnissen sind dann in der Lage, sich unberechtigten Zugang zu solchen Rechnern zu verschaffen.

Mit den Standardwerkzeugen unter UNIX ist dieses Problem nicht oder nur unter Inkaufnahme von großer Unbequemlichkeit zu lösen. In dieser Arbeit wird daher ein System entwickelt, das eine komfortable Rechneradministration über ein Netzwerk erlaubt, *ohne* dabei die Sicherheit der verwalteten Rechner zu gefährden.

Im folgenden soll nun in einem ersten Ansatz möglichst detailliert definiert werden, welche Aufgaben ein System zur netzglobalen Rechneradministration unter besonderer Berücksichtigung des Sicherheitsaspektes zu bewältigen hat. Verschiedene Teile bleiben dabei zunächst noch relativ abstrakt. Diese Abstraktionen werden später (in Kap. 4) beim Entwurf eines Modells zur Realisierung des Verwaltungssystems durch konkrete Strukturen ersetzt.

Die verschiedenen Teilaspekte *Funktionalität*, *Sicherheit*, *Konsistenz* und *Flexibilität* des Systems werden nachfolgend in jeweils unterschiedlichen Kapiteln betrachtet.

2.1. Funktionalität

Es soll ein System entwickelt werden, welches dem Administrator einer Gruppe von Rechnern die Möglichkeit bietet, Verwaltungsaufgaben, die normalerweise lokal auf den einzelnen Rechnern ausgeführt werden müssen, unter Verwendung des lokalen Netzes zentral zu erledigen. Typischerweise handelt es sich bei solchen Verwaltungsaufgaben um die Wartung von Systemdateien.

Ziel ist es, den Arbeitsaufwand für den Systemadministrator zu minimieren.

Zum einen ergibt sich eine Arbeitszeitersparnis schon alleine durch den Wegfall der Zugangsprozedur *'Login'* auf die einzelnen Rechner.

Zum anderen steht in Form des globalen Verwaltungssystems eine für alle Rechner einheitliche Oberfläche für die Administration zur Verfügung, so daß eine Umstellung auf die lokalen Besonderheiten – wie z. B. andere physikalische Lokalisierung der Systemdateien oder unterschiedlicher physikalischer Aufbau – nicht nötig ist.

Des weiteren werden sich in einem solchen Rechnernetz viele Überschneidungen im Inhalt der Systemdateien der verschiedenen Rechner ergeben, d. h. einzelne Abschnitte des Inhaltes der Dateien sind auf verschiedenen Rechnern identisch. Bei entsprechender Implementierung brauchen diese Daten nur einmal im globalen Verwaltungssystem eingegeben werden und können dann auf alle Rechner verteilt werden.

Schließlich soll es auch möglich sein, daß das Verwaltungssystem automatisch Änderungen von lokalen Systemdateien der verwalteten Rechner entweder rückgängig macht – falls eine solche Änderung nicht zulässig war – oder aber diese Änderungen an einen oder mehrere der anderen Rechner weitergibt – falls diese Informationen für diese Rechner relevant sind. Durch diesen Mechanismus ergibt sich eine weitere Reduzierung des Arbeitsaufwandes für den Administrator:

- Die Durchführung illegaler lokaler Änderungen kann effektiv unterbunden werden, so daß für den Administrator die – u. U. aufwendige – Korrektur entfällt.
- Legal von Anwendern durchgeführte, lokale Änderungen müssen nicht manuell vom Administrator auf die anderen Rechner übertragen werden.

Auf den einzelnen Rechnern braucht nur einmalig ein Server-Programm installiert werden, das die speziellen Eigenschaften des jeweiligen lokalen Systems berücksichtigt. Dieses Programm bildet die Brücke zur globalen Verwaltung und übernimmt den rechnerlokalen Part des Systems.

2.2. Sicherheitsanforderungen

Die Realisierung einer netzwerkweiten, zentralen Administration von Systemdateien wirft erhebliche Sicherheitsprobleme auf.

Bei allen Übertragungen zwischen den Rechnern muß extreme Sicherheit gewährleistet sein, um so einen Mißbrauch möglichst auszuschließen.

Übliche Netzwerksoftware unter UNIX verwendet zur Identifikation der einzelnen Teilnehmer nur einfache Merkmale, z. B. die User-Id, den Rechnernamen oder die Internet-Adresse. Bei einem großen Netz reichen solche Merkmale aber nicht aus: wenn auf irgendeinem Rechner einer der Benutzer Superuser-Rechte erlangen kann – was bei der Verwendung von Workstations, die nur einzelnen Benutzern unterstehen, durchaus die Regel ist – kann dieser Benutzer alle diese Merkmale ohne weiteres ändern und sich so im Netz als beliebiger Benutzer und/oder Rechner ausgeben. Daher sind andere Authentifizierungsverfahren nötig, z. B. nach dem Modell von Needham und Schröder (vgl. Kap. 3.1 und [16]).

Bei der Kommunikation zwischen den Rechnern sind also die folgenden Punkte zu beachten:

- *Authentifizierung der Rechner*
  Bei jeder Übertragung zwischen zwei Rechnern muß gewährleistet sein, daß diese Daten auch wirklich von dem entsprechenden Rechner stammen und nicht von einem anderen Rechner im Netz.
- *Authentifizierung der Prozesse*
  Es muß sichergestellt sein, daß die Informationen von den entsprechenden Server-Programmen stammen und nicht von einem anderen Prozeß auf den Rechnern.
- *Verschlüsselung sicherheitsrelevanter Daten*
  Sicherheitsrelevante Daten (z. B. Paßwörter) dürfen nur verschlüsselt über das Netz gesendet werden, um bei eventuellen Abhörversuchen keine wichtigen Daten preiszugeben.
- *Begrenzung der Authentifizierung*
  Eine einmal vorgenommene Authentifizierung darf nur eine begrenzte Zeit gültig sein, um das Sicherheitsrisiko möglichst gering zu halten. Andererseits darf dem Anwender dadurch die Arbeit nicht unnötig erschwert werden.

Es muß ein geeignetes Protokoll entwickelt werden, welches den beteiligten Parteien die eindeutige Identifikation des jeweiligen Kommunikationspartners unter Berücksichtigung der oben genannten Punkte erlaubt.

2.3. Konsistenz

Es muß unter allen Umständen dafür gesorgt werden, daß die verwalteten Systemdateien auch in kritischen Fällen ihre Konsistenz behalten, d. h. temporär auftretende Fehler im System dürfen nicht zu Fehlern in den Systemdateien der Rechner führen. Andernfalls wäre der ordnungsgemäße Betrieb der Rechner in Frage gestellt.

Folgende Situtionen sind relevant:

- *Ausfall eines Rechners*
  Falls ein einzelner Rechner ausfällt, darf dadurch der lokale Betrieb der anderen Rechner nicht beeinträchtigt werden. Datenübertragungen zwischen dem ausgefallen Rechner und den übrigen Rechnern müssen nach dem Wiederaufsetzen der ausgefallenen Maschine automatisch nachgeholt werden.
- *Ausfall des Verwaltungssystems*
  Der lokale Betrieb der einzelnen Rechner muß bei Ausfall des Verwaltungssystems unbeeinträchtigt bleiben. Das Verwaltungssystem muß beim Wiederaufsetzen eventuell versäumte Datentransfers nachholen.
- *Ausfall des Netzes*
  Für den Ausfall des gesamten Netzes gelten analog die Forderungen für den Ausfall eines einzelnen Rechners.
- *Fehlerhafte Systemdateien*
  Werden auf einem Rechner Fehler in eine Systemdatei eingebracht, so dürfen sich diese Fehler auf gar keinen Fall auf die anderen Rechner fortpflanzen. Vielmehr sollte das Verwaltungssystem nach Möglichkeit solche Fehler automatisch korrigieren oder mindestens eine Warnmeldung an den Systemadministrator absetzen.
- *Fehlerhafte Konfiguration*
  Werden einzelne Rechner mit einer fehlerhaften Konfiguration des Server-Programms gestartet, so darf dies möglichst nicht zu einem fehlerhaften Betrieb des Server-Programmes führen, da sonst anderere Rechner im System gefährdet werden könnten. Vielmehr sollte in solchen Fällen die Ausführung abgebrochen und der Systemadministrator benachrichtigt werden.
- *Konfigurationsmodifikation*
  Die Flexibilität des Systems (vgl. Kap. 2.4) erlaubt eine weitreichende Konfiguration des Programmsystems. Eine nachträgliche Modifikation einzelner Konfigurationen darf jedoch nicht die Konsistenz des Gesamtsystems gefährden.

Ein anderer Aspekt ist die zeitliche Verzögerung bei der Übertragung. Durch die auf diese Weise eventuell auftretenden temporären Inkonsistenzen bzw. temporäre Differenzen im Datenbestand der verschiedenen Rechner darf es nicht zu Datenverlusten oder Programmfehlern kommen. Solche Probleme können z. B. auftreten, falls gleiche Systemdateien auf verschiedenen Rechnern gleichzeitig modifiziert werden.

2.4. Flexibilität

Ein solches System ist nur dann sinnvoll einsetzbar, wenn es leicht auf alle Rechner, die in die Verwaltung einbezogen werden sollen, übertragen werden kann. Dabei muß das Programmsystem möglichst leicht an die lokalen Gegebenheiten der einzelnen Rechner angepaßt werden können. So ist zum Beispiel denkbar, daß gleiche Systemdateien auf unterschiedlichen Systemen völlig anders strukturiert sind, also beispielsweise eine unterschiedliche Anordnung der Felder vorhanden ist,

eventuell zusätzliche Felder existieren oder aber eine Verteilung der Felder auf mehrere physikalische Dateien vorliegt.

Ein anderer Aspekt ist die Vielzahl der in Betracht kommenden Systemdateien. Prinzipiell sollte das System in der Lage sein, jede beliebige Systemdatei global zu verwalten, um so die Nutzbarkeit zu erhöhen.

Weiterhin ist wesentlich, daß es möglich sein muß, für jeden Rechner individuell zu entscheiden, welche der Systemdateien in die globale Verwaltung einbezogen werden. Ferner muß möglichst detailliert angegeben werden können, wie die globale Verteilung der Daten vorgenommen wird, d. h. welche auf den Rechnern lokal vorgenommenen Änderungen der global verwalteten Dateien an welche anderen Rechner weitergegeben werden.

Alles in allem zeigt sich, daß das System so gestaltet werden muß, daß auch *nach* einer Installation die Konfiguration an möglichst vielen Stellen möglichst leicht änderbar ist. So kann eine Verwendbarkeit mit möglichst vielen verschiedenen Typen von Systemdateien und Rechnern gewährleistet werden.

Zusammenfassend muß sich die Flexibilität des Systems erstrecken auf die Angabe spezifischer Methoden für:

- jede verwaltete Systemdatei
- jeden verwalteten Maschinentyp
- die Datenverteilung

Dabei muß jedoch gewährleistet werden, daß Konfigurationsänderungen auf einzelnen Maschinen nicht die Funktionalität des Gesamtsystems beeinträchtigen (vgl. Kap. 2.3).

3. Existierende Lösungsansätze

Hier sollen einige existierende Lösungsansätze genauer beschrieben werden. Dabei beschäftigen sich die beiden ersten Kapitel mit den theoretischen Ansätzen zur Lösung der Sicherheitsprobleme bei der Kommunikation über ein offenes Netzwerk. Dazu zunächst ein Abschnitt, der sich mit den Grundlagen der Authentifikation beschäftigt. Anschließend folgt dann ein Kapitel, welches einen allgemeinen Überblick über die verschiedenen Sicherheitsaspekte einer Netzwerkkommunikation gibt.

In weiteren Kapiteln werden dann verschiedene Programmpakete vorgestellt, die unterschiedliche Aspekte des in dieser Arbeit zu lösenden Problems aufgreifen.

Schließlich sollen dann in einem letzten Kapitel die Ergebnisse der unterschiedlichen Lösungsansätze vergleichend zusammengefaßt werden. Gleichzeitig wird dabei versucht, die Relevanz der von den verschiedenen Autoren behandelten Teilprobleme für die vorliegende Arbeit abzuschätzen und zu prüfen, inwieweit die vorgestellten Lösungswege sich auch zur Lösung der hier vorliegenden Aufgabe eignen.

Alle Autoren haben zur Beschreibung der Kommunikationsprotokolle eigene Notationen verwendet. In dieser Zusammenfassung wurden zwecks einheitlicherer Gestaltung die Darstellungen einander angepaßt, wodurch sich aber teilweise gravierende Unterschiede zu den Originalen ergeben.

Die hier verwendete Notation wird jeweils beim ersten Auftreten erläutert. Darüberhinaus existiert eine zusammenfassende Beschreibung im Glossar (Kap. 6.2) unter dem Stichwort *Notationen*.

3.1. Authentifizierung in Computernetzen

Viele der heute verwendeten Authentifizierungssysteme basieren auf relativ alten Methoden. Eine Übersicht zu diesem Thema gaben R. M. Needham und M. D. Schröder bereits im Jahre 1978 in ihrem Artikel *'Using Encryption for Authentication in Large Networks of Computers'* [16]. Wegen der Wichtigkeit ihrer Arbeit sollen ihre Ergebnisse nachfolgend zusammengefaßt werden.

3.1.1. Einleitung

Needham und Schröder verstehen unter dem Begriff Authentifizierung – im Kontext einer sicheren Kommunikation zwischen Rechnern – die gegenseitige Verifikation der Identität der kommunizierenden Parteien.

Da es in einem großen Rechnernetz möglicherweise kein zentrales System gibt, welches Beschreibungen der Authorisierung aller angeschlossenen Rechner und Benutzer enthält, schlagen Needham und Schröder verschiedene Protokolle zur dezentralen Authentifizierung in einem solchen Netzwerk vor. Dabei stellen sie nur minimale Anforderungen an die Verläßlichkeit des Netzes. Insbesondere setzen sie sich mit folgenden Punkten auseinander:

- Aufbau einer authentifizierten interaktiven Kommunikation zwischen zwei Parteien auf verschiedenen Maschinen.
- Authentifizierte unidirektionale Kommunikation – wie z. B. eine elektronische Post – bei der kein Handshake zwischen Sender und Empfänger möglich ist, weil nicht sichergestellt werden kann, daß beide Parteien gleichzeitig verfügbar sind.
- Signierte Kommunikation, bei der die Herkunft und die Integrität des Inhalts einer Nachricht gegenüber einer dritten Partei nachgewiesen werden kann.

Needham und Schröder machen diverse Annahmen als Basis für ihre Authentifizierungsmechanismen. Dies sind im einzelnen:

- Sichere Kommunikation in einem physikalisch unsicheren Netz basiert auf der Verschlüsselung des zwischen den beiden Kommunikationspartnern übertragenen Datenmaterials. Voraussetzung ist also die Existenz von effizienten, sicheren Ver- und Entschlüsselungsmechanismen auf beiden an der Kommunikation beteiligten Maschinen. Dabei kommen sowohl konventionelle† als auch Public-Key-Verfahren in Frage.
- Um ein wirklich sicheres Übertragungsprotokoll konzeptionieren zu können, wird der Worst-Case betrachtet: Man nimmt an, ein Gegner könne die Kommunikation in jeder Weise beeinflussen, also insbesondere
  - Teile der Nachrichten verändern oder kopieren
  - Bereits gesendete Nachrichten nochmals senden *('Replay')*
  - Falsche Nachrichten absenden
- Jeder einzelne Rechner für sich bietet eine sichere Umgebung, wie z. B. ein Personal-Computer oder ein sicheres Mehrbenutzersystem. Die kommunizierenden Parteien wünschen eine sichere Kommunikation. Needham und Schröder betrachten ausdrücklich nicht die Probleme, die sich ergeben, wenn die ausschließliche Verwendung von sicherer Kommunikation forciert werden soll oder wenn bestimmte Verbindungen unterbunden werden sollen.

Needham und Schröder weisen darauf hin, daß es ihnen nicht um eine allumfassende Sicherheit geht, sondern daß sie lediglich adäquate Beispiellösungen für die von ihnen definierten Probleme geben wollen. Weitergehende Aspekte, wie z. B. die Verhinderung einer Datenflußanalyse im Netzwerk (vgl. Kap. 3.2.1) bleiben unberücksichtigt.

3.1.2. Grundlagen

Bei Verwendung von Verschlüsselungsalgorithmen ist die Existenz geheimer Schlüssel wesentlich. Dabei ist es unerheblich, welche Art von Verschlüsselungsverfahren Verwendung findet, ob konventionell – beide Parteien teilen einen gemeinsamen geheimen Schlüssel, der auf der einen Seite zur Verschlüsselung, auf der anderen Seite zur Entschlüsselung verwendet wird – oder Public-Key – zur Ver- und Entschlüsselung werden unterschiedliche Schlüssel verwendet, wobei jedem Teilnehmer zwei solche Schlüssel zugeordnet sind, von denen einer öffentlich bekannt, der andere aber geheim ist.

In beiden Fällen muß ein Schlüsselverteilungssystem *'Authentifizierungsserver'* existieren. Da ein solcher Server zwangsläufig namensorientiert arbeitet, muß eine eindeutige Benennung aller Entitäten im Netz möglich sein. Dies übernehmen entsprechende Namensauthoritäten, von denen es mehrere im Netz geben kann. Die Benennung der teilnehmenden Parteien geschieht dann in der Form *'Namensauthorität.EinfacherName'*.

Die von Needham und Schröder vorgestellten Beispielalgorithmen erfordern die Möglichkeit eine Verschlüsselung durchzuführen, ohne die Ausgabe über das Netz zu senden. Daher ist es wichtig, daß der Verschlüsselungsmechanismus nicht direkt im Netzwerkinterface realisiert ist.

---

† Needham und Schröder bezeichnen mit konventioneller Verschlüsselung dasjenige Verfahren, welches an anderer Stelle *Private-Key-Verschlüsselung* oder *symmetrische* Verschlüsselung genannt wird. Näheres in Kapitel 3.2.2 f.

3.1.3. Interaktive Verbindungen

Interaktive Verbindungen zeichnen sich dadurch aus, daß beide Kommunikationspartner gleichzeitig verfügbar sind. Dadurch ist eine Authentifikation im Dialog möglich.

3.1.3.1. Protokoll 1: konventionelles Verfahren

Bei Verwendung eines konventionellen Verschlüsselungsverfahrens besitzt jede Partei je einen geheimen Schlüssel, der sonst nur dem Authentifizierungsserver bekannt ist. Um nun eine authentifizierte Verbindung zwischen zwei Parteien A und B herzustellen, muß der Initiator, also z. B. A eine Nachricht generieren, die folgende Eigenschaften erfüllt:

- Die Nachricht ist nur für B nutzbar, d. h. nur B darf ihren Inhalt verwenden können, um sich gegenüber A zu identifizieren.
- B muß erkennen können, daß die Nachricht nur von A stammen kann.

Eine mögliche Lösung des Problems ist nachstehendes Protokoll, welches zunächst davon ausgeht, daß A und B vom gleichen Authentifizierungsserver S bedient werden. Folgende Protokollschritte sind nötig:

A möchte eine Nachricht an B senden. Dazu schickt er in Schritt (1.1) eine Anfrage an den Authentifizierungsserver S. Die Anfrage enthält außerdem noch einen einmaligen Identifikator I_{A1}. Einmalig bedeutet hier, daß A in allen früheren Nachrichten des gleichen Typs immer andere Identifikatoren verwendet hat.

(1.1) $A \rightarrow S$: A, B, I_{A1}

In Schritt (1.2) entnimmt der Authentifizierungsserver S einer internen Tabelle die geheimen Schlüssel K_A und K_B von A und B. Außerdem bestimmt er einen zufälligen, möglichst noch nie vorher benutzten Konversationsschlüssel K_C. Das gesamte zweite Paket wird mit K_A verschlüsselt, dadurch kann nur A die Entschlüsselung vornehmen und den Konversationsschlüssel K_C lesen und für die weitere Verwendung speichern.

(1.2) $S \rightarrow A$: $K_A(I_{A1}, B, K_C, K_B(K_C, A))$

A überprüft, ob im entschlüsselten Paket die korrekten Werte B und I_{A1} enthalten sind. Nur die Überprüfung *beider* Merkmale kann garantieren, daß es sich bei dem empfangenen Paket um eine Antwort auf die gestellte Anfrage handelt. Würde der Name B ausgelassen, so könnte ein Gegner in Schritt (1.1) den Namen B gegen einen Namen X austauschen, bevor diese Nachricht den Authentifizierungsserver erreicht. Dies würde dazu führen, daß A später unwissentlich nicht mit B sondern mit X kommuniziert. Würde der Identifikator I_{A1} ausgelassen, so könnte ein Gegner die Antwort von S ersetzten durch eine bereits früher gesendete (von S an A bezüglich B) und so A dazu zwingen einen bereits früher verwendeten Konversationsschlüssel nochmals zu verwenden.

In Schritt (1.3) sendet A den durch K_B verschlüsselten Teil des Pakets (1.2) an B. Nur B kann dieses Paket entschlüsseln und den Konversationsschlüssel K_C lesen, den bereits A gespeichert hat. B kennt auch die Identität des Absenders, da dieser durch S authentifiziert wurde.

(1.3) $A \rightarrow B$: $K_B(K_C, A)$

Zu diesem Zeitpunkt ist sowohl für A als auch für B klar, daß jede durch K_C verschlüsselte Nachricht nur von B bzw. A stammen kann, da die einzigen Nachrichten, in denen K_C enthalten war nur mittels K_A bzw. K_B entschlüsselt werden konnten. A weiß weiterhin, daß K_C noch nie vorher verwendet wurde. Hier liegt B schlechter. Falls B sich nicht alle bisher in der Kommunikation mit A verwendenten Konversationsschlüssel K_C gemerkt hat, könnte es sich bei Paket (1.3) auch um ein Replay eines früheren Paketes handeln. Um dies auszuschließen, sendet B in Schritt (1.4) einen einmaligen Identifikator I_B, verschlüsselt mit K_C an A.

(1.4) B→A: $K_C(I_B)$

A antwortet in Schritt (1.5) mit einem zu Paket (1.4) verwandten Identifikator, z. B. I_B–1. Nachdem dieser Austausch zufriedenstellend abgeschlossen wurde, ist das gegenseitige Vertrauen genügend groß um mit der eigentlichen Übertragung, verschlüsselt durch K_C, zu beginnen.

(1.5) A→B: $K_C(I_B-1)$

Protokoll 1 enthält fünf Schritte. Diese Anzahl kann auf drei verringert werden, wenn A sich für regelmäßige Kommunikationspartner Authentifikatoren der Form B: K_C, $K_B(K_C, A)$ merkt, die aus Schritt (1.2) abgeleitet wurden, so daß dann die Schritte (1.1) und (1.2) entfallen können. Bei Verwendung dieser Art von Caching muß jedoch das Protokoll modifiziert werden, indem die Schritte (1.3) und (1.4) ersetzt werden durch:

(1.3´) A→B: $K_B(K_C, A)$, $K_C(I_{A2})$

(1.4´) B→A: $K_C(I_{A2}-1, I_B)$

Durch das Caching wird K_C immer wieder verwendet, daher ist diese Änderung ist nötig, um einen Replay-Schutz zu garantieren.

3.1.3.2. Protokoll 2: Public-Key-Verfahren

Im folgenden bezeichnet PK_A den öffentlichen, SK_A den geheimen Schlüssel von A. Es wird angenommen, daß sowohl A als auch B den öffentlichen Schlüssel PK_S des Authentifizierungs-servers S kennen.

Der Protokollablauf: Im Schritt (2.1) erfragt A beim Authentifizierungsserver S den öffentlichen Schlüssel von B und erhält die Antwort im Schritt (2.2).

(2.1) A→S: A, B

(2.2) S→A: $SK_S(PK_B, B)$

Man beachte, daß die Nachricht in Schritt (2.2) nicht verschlüsselt wird, um sie geheim zu halten (sie enthält ohnehin nur einen öffentlichen Schlüssel), sondern um ihre Integrität zu sichern.

In Schritt (2.3) erhält B eine Nachricht – die nur B mit seinem geheimen Schlüssel SK_B decodieren kann –, in der jemand, der vorgibt A zu sein, eine Verbindung aufbauen möchte und daher einen einmaligen Identifikator I_A sendet.

(2.3) A→B: $PK_B(I_A, A)$

In den Schritten (2.4) und (2.5) erhält B – analog zu den Schritten (2.1) und (2.2) – von S den öffentlichen Schlüssel für A.

(2.4) B→S: B, A

(2.5) S→B: $SK_S(PK_A, A)$

Anschließend ist in den Schritten (2.6) und (2.7) noch ein doppelter Handshake nötig, um die Zeitintegrität der Konversation zu garantieren, d. h. zu gewährleisten, daß es sich bei keinem Teil der Identitikationsphase um einen Replay handelte.

(2.6) B→A: $PK_A(I_A, I_B)$

(2.7) A→B: $PK_B(I_B)$

Protokoll 2 enthält sieben Schritte im Vergleich zu fünf Schritten in Protokoll 1. Bei Einführung von lokalen Caches für öffentliche Schlüssel können jedoch vier Schritte entfallen, nämlich (2.1), (2.2), (2.4) und (2.5), so daß nur drei Schritte übrig bleiben, genau wie in Protokoll 1 nach Einführung von Caches.

Zu beachten ist auch, daß es in Protokoll 2 in der nachfolgenden Übertragung nicht ausreicht, wenn A seine Nachrichten an B mit PK_B verschlüsselt. Da PK_B allgemein bekannt ist, könnte jeder Gegner Daten in den Datenstrom einfügen. Daher ist entweder eine doppelte Verschlüsselung nötig, wie z. B. $PK_B(SK_A(...))$ oder den Nachrichten müssen Nummern mitgegeben werden, um eine Serialisierung zu ermöglichen. Diese Numerierung könnte z. B. mit I_A oder I_B initialisiert werden.

3.1.4. Authentifizierungsserver

Bisher wurde angenommen, daß A und B vom selben Authentifizierungsserver bedient werden. Diese Einschränkung ist jedoch nicht nötig.

Unter Protokoll 1 werde A vom Authentifizierungsserver S, B jedoch vom Authentifizierungsserver T bedient. Ziel einer Authentifizierung ist weiterhin die Erzeugung eines Authentifikators $K_B(K_C, A)$. An der Authentifizierung ist sowohl S als auch T beteiligt, denn nur S kann mit K_A und nur T kann Verschlüsselungen mit K_B durchführen. Ein Austausch der Schlüssel K_A oder K_B zwischen S und T kommt nicht in Frage, da die Integrität eines einzelnen Authentifizierungsservers die Sicherheit der anderen Server im Netz nicht beeinträchtigen darf.

Unter der Annahme, daß S und T bereits eine gegenseitige, sichere Verbindung – z. B. nach Protokoll 1 – aufgenommen haben, kann Protokoll 1 ergänzt werden um die Schritte

(1.1.1) $S \rightarrow T$: K_C, B, I_{A1}

(1.1.2) $T \rightarrow S$: $K_B(K_C, A)$, I_{A1}, A

Bei Verwendung von Public-Keys kann Protokoll 2 ohne Veränderung benutzt werden, wenn sich A in Schritt (2.1) direkt an T und B sich in Schritt (2.4) direkt an S wendet. Voraussetzung dazu ist allerdings, daß A den zu B gehörigen Authentifizierungsserver kennt.

Bei beiden Protokollen wird daher im Fall von mehreren Authentifizierungsservern angenommenen, daß immer volle Namen der Form *'Namensauthorität.EinfacherName'* Verwendung finden.

Auch beim Einsatz mehrerer Authentifizierungsserver kann für beide Protokolltypen die Anzahl der Protokollschritte auf drei reduziert werden.

Prinzipiell unterscheiden sich die beiden Verschlüsselungsmethoden erheblich, wenn die Implementierung von Authentifizierungsservern betrachtet wird:

- Im konventionellen Fall müssen im Authentifizierungsserver Daten der Form A: K_A geheim gehalten werden, z. B. durch Verschlüsselung mit K_S. Bei jeder Anfrage nach (1.2) müssen sichere Transaktionen durchgeführt werden, da die geheimen Schlüssel von A und B benötigt werden.
- Im Public-Key-Fall sind keine sicheren Transaktionen nötig, wenn Daten in der Form A: $SK_S(PK_A, A)$ gespeichert werden. Werden dagegen die öffentlichen Schlüssel direkt gespeichert, so ist bei jedem Zugriff nach (2.2) eine Verschlüsselung mit SK_S erforderlich.

Im allgemeinen werden sich jedoch die Implementierungen in beiden Fällen nicht wesentlich unterscheiden, da auch im Public-Key-Fall ist es nötig, die Integrität der Datenbank zu sichern.

Man beachte auch, daß in beiden Fällen die Server zustandslos arbeiten können – es sind bei keiner Übertragung Informationen über vorherige Transaktionen nötig. Dadurch ist der Authentifizierungsserver vor Fehlern in der Übertragung wie Verlust oder Duplizieren von Blöcken sicher und das Protokoll bleibt einfach. Einzige Ausnahme bildet Schritt (1.1.1), wo besondere Maßnahmen nötig sind.

3.1.5. Unidirektionale Verbindungen

In System für elektronische Post kann eine Authentifizierung nicht auf einer Interaktion zwischen Sender und Empfänger beruhen. Eine Nachricht wird einem Transportmechanismus übergeben und vielleicht erst ausgeliefert, wenn der Absender nicht mehr zur Verfügung steht. Daher ist ein anderes Verfahren nötig, wobei idealerweise die Sicherung unabhängig vom jeweils verwendeten Transportsystem ist.

3.1.5.1. Mit konventionellen Algorithmen

Man betrachte folgenden Block aus einem vorherigen Protokoll:

(1.3) $A \rightarrow B$: $K_B(K_C, A)$

Würde dieser Block einer durch K_C verschlüsselten Nachricht vorangestellt, so wäre die Nachricht selbstauthentifizierend, sowohl für den Absender als auch für den Empfänger, obwohl B keinen Beitrag dazu geleistet hat.

Es wird angenommen, daß die einzelnen Teilblöcke einer Nachricht auf geeignete Weise serialisiert sind. Dann verbleibt das Problem der Zeitintegrität einer Nachricht, d. h. die Absicherung gegen Replay. Dies Problem ist hier aufgrund der auftretenden Verzögerung besonders schwierig.

Eine adäquate Lösung scheint der Einsatz von Zeitmarken *('Time-stamps').* Die Verwendung solcher Zeitmarken wurde bisher vermieden, und zwar auf Grund der Probleme der Zeitsynchronisation in einem großen Netzwerk. Die vorgestellte Lösung hängt jedoch nicht von der Existenz einer Universalzeit im Netz ab.

Jede Nachricht beinhaltet eine Zeitmarke, die den Absendezeitpunkt angibt. Die Auflösung der Zeitangabe muß fein genug sein, damit nicht zwei Nachrichten aus der gleichen Quelle A die gleiche Zeitmarke tragen. Jeder Empfänger B speichert für jeden Absender Wertepaare der Form {Quelle, Zeitmarke}. Mit B ist ein Zeitinterwall T assoziiert. T charakterisiert die Obergrenze der Zeitabweichungen der Uhren im Netzwerk zzgl. der Differenzen zwischen Absenden und Empfangen einer Nachricht. Eine eingegangene Nachricht wird verworfen, wenn entweder {Quelle, Zeitmarke} bereits registriert wurde oder wenn seine Zeitmarke die aktuelle Zeit um mehr als T unterschreitet. Das Nachrichtenregister bleibt klein, da alle Einträge, die älter als T sind, entfernt werden können.

3.1.5.2. Mit Public-Key-Algorithmen

Die Methoden zur Absicherung der Zeitintegrität sind in diesem Fall die gleichen wie oben beschrieben.

Zur Authentifizierung gibt es zwei verschiedene Ansätze. Im ersten wird der Nachricht ein Kopf vorangestellt mit folgendem Inhalt:

$A \rightarrow B$: $PK_B(A, I_A, SK_A(B))$

Dabei bezeichnet A den Absender und $SK_A(B)$ erlaubt dem Empfänger B die Authentifizierung des Absenders nach (2.4) und (2.5). I_A ist wieder ein einmaliger Identifikator, der dazu verwendet wird, den Nachrichtenkopf mit dem Nachrichtenrumpf. Diese Verknüpfung geschieht bei der Verwedung von konventioneller Verschlüsselung implizit dadurch, daß der Nachrichtenrumpf mit K_C verschlüsselt wird. Hier dagegen wird der Rumpf mit PK_B codiert und beeinhaltet neben I_A noch eine Zeitmarke wie oben.

Alternativ kann auch eine digitale Unterschrift zur Authentifizierung verwendet werden. Dies wird im nächsten Abschnitt diskutiert.

3.1.6. Digitale Unterschrift

Alle obigen Protokolle dienen zur gegenseitigen Authentifizierung zweier Kommunikationspartner. Gelegentlich ist es jedoch nötig, einer dritten Partei nachzuweisen, daß eine Nachricht von einem bestimmten Absender stammt und daß keine Änderung nachträglich angebracht wurde. Insbesondere gilt dies Verfahren auch als Nachweis gegenüber dem Absender selbst (‘non-repudiation’, vgl. Kap. 3.2.1). Bei geschriebenen Dokumenten dient dazu die Unterschrift. Nachfolgend soll ein elektronisches Äquivalent vorgestellt werden.

3.1.6.1. Protokoll 3: Unterschrift mit konventioneller Verschlüsselung

Eine Möglichkeit zur Realisierung eines solchen Verfahrens basiert auf der Festlegung einer arithmetischen Funktion über dem zu signierenden Text. Diese Funktion sollte die Eigenschaft besitzen, daß es bei gegebenem Text, gegebener Funktion und gegebenem Funktionswert nahezu unmöglich ist, einen zweiten Text zu finden, der den gleichen Funktionswert liefert. Außerdem sollte der Funktionswert vom Datenvolumen her deutlich kleiner sein als der betrachtete Text.

Wenn A einen Text verschicken möchte, berechnet er zunächst den dazugehörigen Funktionswert F_A. Dann fordert er einen Unterschriftsblock (‘signature block’) vom Authentifizierungsserver S:

(3.1) $A \rightarrow S$: $A, K_A(F_A)$

welcher auch vom Server zur Verfügung gestellt wird:

(3.2) $S \rightarrow A$: $K_S(A, F_A)$

Das Paket (3.2) ist mit dem Schlüssel des Authentifizierungsservers verschlüsselt und daher nur für S lesbar. Man beachte, daß A die Korrektheit dieser Antwort nicht verifizieren kann. Aber falls die Antwort geändert wurde, wird eine spätere Überprüfung durch B scheitern und die Nachricht von B verworfen werden.

A sendet nun den Unterschriftenblock an B, gefolgt von dem zu unterzeichnenden Text. Nach dessen Erhalt entschlüsselt B ggf. den Text und berechnet den zugehörigen Funktionswert F_B. Dann sendet B den Unterschriftenblock an den Authentifizierungsserver, um ihn entschlüsseln zu lassen:

(3.3) $B \rightarrow S$: $B, K_S(A, F_A)$

Der Server entschlüsselt den Block und gibt dessen Inhalt an B zurück:

(3.4) $S \rightarrow B$: $K_B(A, F_A)$

Falls der zurückgegebene Wert F_A mit F_B übereinstimmt, dann ist die in (3.4) genannte Partei der Absender des unterzeichneten Textes. Andernfalls könnte das bedeuten, daß einer der Schritte (3.1) - (3.3) oder die Verbindung zwischen Text und Unterschriftenblock beeinflußt wurden.

Frührere Feststellung von Beeinflussungen ist möglich bei Verwendung von einmaligen Identifikatoren in den Transaktionen (3.1)/(3.2) und (3.3)/(3.4).

Falls B den Text als Beweismittel behalten möchte, muß er lediglich den Text mit dem zugehörigen Unterschriftenblock aufbewahren.

Eine Ausdehung dieses Protokolls auf den Fall von mehreren Authentifizierungsservern ist einfach möglich.

3.1.6.2. Unterschrift mit Public-Key-Verschlüsselung

Auch mit einem Public-Key-Verfahren kann ein Text – mittels einer Funktion wie oben beschrieben – signiert werden. Hier steht allerdings noch eine elegantere Methode zur Verfügung, die zuerst von Diffie und Hellman (vgl. Kap. 3.2.3) beschrieben wurde.

Zunächst bestimmt A den öffentlichen Schlüssel von B, entweder von einem Server oder aus einem Cache. Der Text wird dann doppelt verschlüsselt:

$$A \rightarrow B: \quad PK_B(SK_A(\ldots))$$

B kann die erste Entschlüsselung durchführen auf Grund seiner Kenntnis von SK_B, die zweite nach Erfragung von PK_A vom Server oder dem lokalen Cache. Es ist die Existenz eines Nachrichtenkopfes mit dem Namen des Absenders nötig, damit PK_A korrekt bestimmt werden kann. B kann die Nachricht nicht ändern, da er SK_A nicht kennt.

Man beachte, daß dieser Text zur nachträglichen Nachweis nur verwendet werden kann, wenn A sein Schlüsselpaar nicht ändert. In einem solchen Fall müßte der Authentifizierungsserver eine Liste von alten öffentlichen Schlüsseln führen, zusammen mit dem Zeitpunkt der jeweiligen Änderung. Hier liegt ein Vorteil des Unterschriftenprotokolls für konventionelle Verschlüsselung: der Authentifizierungsserver braucht nur seine *eigenen* alten Schlüssel zu speichern um eine korrekte Funktionsweise des späteren Verifikation von Texten zu gewährleisten.

3.1.7. Kommentar der Autoren

Protokolle mit konventioneller Verschlüsselung ähneln denen mit Public-Key-Verschlüsselung stark. Bei beiden Systemen ist Caching sehr wichtig. Viele Tricks, die bei der konventionellen Verschlüsselung verwendet werden, sind auch bei Public-Key-Verschlüsselung möglich, aber nicht immer nötig. Die innere Sicherheit für einen Public-Key-Authentifizierungsserver ist leichter zu realisieren, dies wird aber zum Teil zunichte gemacht durch die Notwendigkeit, alte öffentliche Schlüssel ebenfalls zu speichern.

Schließlich sind alle Protokolle wie die hier entwickelten sehr stark subtilen Fehlern ausgesetzt, die im normalen Betrieb wahrscheinlich nicht entdeckt werden. Daher besteht ein großer Bedarf an Techniken zur Verifikation der Korrektheit dieser Protokolle.

3.2. Sicherheitsservices und Sicherheitsmechanismen

Die ISO hat im Jahre 1988 ihr *OSI-Reference-Model* [5], allgemein bekannt als *7-Schichten-Kommunikationsmodell,* um eine Sicherheitsarchitektur erweitert [6].

Das OSI-Referenzmodell ist bekanntlich in sieben Schichten strukturiert, von der physikalischen Ebene am unteren Ende bis zur Anwendungsschicht ganz oben. Die unteren drei Schichten, die Bitübertragungsschicht *'physical layer',* die Sicherungsschicht *'data link layer'* und die Vermittlungsschicht *'network layer'* formen die Kommunikationsstruktur. Die höheren Schichten, die Transportschicht *'transport layer',* die Kommunikationssteuerschicht *'session layer',* die Darstellungsschicht *'presentation layer'* und die Verarbeitungsschicht *'application layer'* wenden die Kommunikationsfähigkeiten der Vermittlungsschicht an. Diese oberen Schichten können betrachtet werden, als ob sie end-to-end durch das Kommunikationssystem arbeiten.

Im Prinzip können Sicherheitsfeatures in jeder Ebene der OSI-Struktur eingebaut werden. Die Sicherheitsarchitektur definiert jedoch zunächst eine Reihe von Sicherheitsservices und spezifiziert dann für jeden Service, welche Ebenen ihn zur Verfügung stellen können.

Die OSI-Sicherheitsarchitektur macht eine klare Unterscheidung zwischen Sicherheits*services* und Sicherheits*mechanismen.* So ist z. B. Verschlüsselung ein Mechanismus, der von diversen Services benutzt werden kann. Der Hauptaugenmerk des OSI-Dokumentes liegt auf den Services, da die Mechanismen abhängiger Teil der unterschiedlichen Standards sind, die festlegen, wie die Services in den unterschiedlichen Ebenen zur Verfügung gestellt werden können.

3.2.1. Sicherheitsservices

Nachfolgend wird eine kurze Übersicht über die in der Sicherheitsarchitektur zum ISO-OSI-Referenzmodell [6] definierten Sicherheitsservices gegeben. Die Abschnitte sind jeweils mit den englischen Originalbezeichungen überschrieben, denen eine entsprechende deutsche Übersetzung gegenübergestellt wird. Die Zusammenfassung ist angelehnt an diejenige von Davies und Price [3].

- *Data Confidentiality – Vertraulichkeit der Daten*
  Die Informationen sind unauthorisierten Individuen, Entitäten oder Prozessen unzugänglich. Die Vertraulichkeit kann sich entweder auf die gesamte Nachricht oder auf einzelne Felder der Nachricht beziehen *('selective field confidentiality').*
  Eine spezielle Form der Vertraulichkeit ist die *Traffic Flow Confidentiality.* Dabei wird das Netz geschützt gegen Analyse des Datenflusses, insbesondere gegen Rückschlüsse, die aufgrund des Datenflusses über die versendeten Informationen gezogen werden können. Dieser Service ist i. a. relativ schwierig zu implementieren und verlangt häufig den Einsatz verschiedener Mechanismen gleichzeitig, z. B. Verschlüsselung im Physical Layer zusammen mit *'Traffic Padding'*, der Auffüllung von Lücken zur Verwirrung einer Datenflußbeobachtung durch Gegner.
- *Authentication – Authentifikation*
  Das ISO-OSI-Modell bezieht den Begriff Authentifikation auf die Gewißheit, daß die erhaltenen Daten tatsächlich vom angenommenen Absender stammen. Im Gegensatz zu Definitonen an anderen Stellen ist also nicht die Integrität der Daten gemeint.
- *Data Integrity – Datenintegrität*
  Datenintegrität bezeichnet die Eigenschaft der Daten, nicht in unauthorisierter Weise abgeändert oder zerstört worden zu sein.
- *Non-Repudiation – Nicht-Zurückweisbarkeit*
  Repudiation meint, daß eine der beiden an einer Kommunikation beteiligten Parteien bestreitet, an dieser Kommunikation ganz oder teilweise beteiligt gewesen zu sein. Non-Repudiation ist in zwei Ausprägungen möglich:

- *Non-Repudiation with proof of origin*
  Der Empfänger wird ausgestattet mit Daten, die ihn davor bewahren, daß der Absender versucht die Generierung der Nachricht oder ihren Inhalt zu bestreiten. Dies kann erreicht werden durch verschiedene Mechanismen, z. B. eine *'digitale Unterschrift'* oder durch *'Notarization',* also eine Beglaubigung seitens eines Dritten.
- *Non-Repudiation with proof of delivery*
  Ein ähnlicher Service, der dem Absender garantiert, daß der Empfänger den Empfang oder den Inhalt einer empfangenen Nachricht nicht bestreiten kann. In der Praxis bedeutet dies, daß der Empfänger eine unterschriebene Empfangsquittung zur Verfügung stellt.

- *Access Control – Zugriffskontrolle*
  Zugriffskontrolle kann ganz unterschiedlich eingesetzt werden, entweder nahe der Datenquelle, oder an der Datensenke oder aber in einem dazwischenliegenden Bereich.
  Die Zugriffskontrolle kann benutzt werden, um das Netzwerk gegen unbefugte (übermäßige) Nutzung durch Gegner zu schützen, aber im allgemeinen wird sie eher verwendet um Services von unauthorisierter Inanspruchnahme abzuschirmen. Daher wird dieser Service meistens im oder oberhalb des Application Layer angesiedelt.

Im allgemeinen Gebrauch sind die Services Authentifikation und Datenintegrität untrennbar verbunden: meistens basieren sie auf den gleichen Mechanismen und werden auch gleichzeitig benötigt.

Die dabei verwendeten Mechanismen stützen sich i. a. auf kryptographische Prozesse ab, unter Verwendung eines geheimen Schlüssels, den sich die kommunizierenden Parteien teilen. In vielen Fällen erwirkt der Besitz des Schlüssels die Authentizität der Datenquelle und erlaubt gleichzeitig die Überprüfung der Datenintegrität.

Was die gleichzeitige Notwendigkeit beider Services betrifft, so kann man sagen, daß es i. a. nutzlos ist zu wissen, daß Daten zwar aus der angeblichen Quelle stammen, wenn gleichzeitig aber nicht klar ist, ob die Daten nicht vielleicht unauthorisiert verändert worden sind. Andererseits ist es ebenso zwecklos überprüfen zu können, daß Daten nicht unauthorisiert geändert wurden, wenn sie gar nicht vom angeblichen Absender, sondern von einem Gegner stammen.

3.2.2. Sicherheitsmechanismen

Die Sicherheitsmechanismen, die zur Realisierung der oben genannten Sicherheitsservices verwendet werden, umfassen

- *Verschlüsselung*
  Einige verschiedene Verschlüsselungsalgorithmen und ihre Anwendung werden im nachfolgenden Kapitel vorgestellt. Es wird immer (wie auch schon im Kapitel 3.1) unterschieden zwischen
  - *symmetrischen Algorithmen*
    auch Private-Key-Verschlüsselung oder konventionelle Verschlüsselung genannt; zur Ver- und Entschlüsselung wird derselbe Schlüssel verwendet.
  - *asymmetrischen Algorithmen*
    Es existeren zwei verschiedene Schlüssel, die bezüglich der Ver- und Entschlüsselung invers sind. Die Kenntnis eines der beiden Schlüssel erleichert nicht die Herleitung des anderen.
    Ein besonderer Anwendungsfall eines asymmetrischen Algorithmus' wird *Public-Key-Verschlüsselung* genannt. Einer der beiden Schlüssel (der 'private') wird geheim gehalten, der andere (der 'öffentliche') ist allgemein bekannt. Mittels dieser Methode können auf elegante Weise z. B. digitale Unterschriften realisiert werden. Vorteilhaft bei dieser

Anwendung ist, daß bei der Erneuerung von Schlüsseln keine geheimen Informationen weitergegeben müssen. Andererseits hat dieses Verfahren größere Schwierigkeiten bei der Authentifizierung: bei einem symmetrischen Verfahren identifiziert die Kenntnis des Schlüssels die beiden Kommunikationspartner gegenseitig, ein Public-Key-Verfahren garantiert nur eine ‘Ein-Weg-Identifikation’.

- *Digitale Unterschrift*
  siehe Kapitel 3.1.6
- *Integritätsmechanismen*
  vergleiche Kapitel 3.1.3 ff.
- *Authentifizierungsmechanismen*
  vergleiche Kapitel 3.1.3 ff.
- *Traffic Padding*
  Lücken im Datenfluß auf dem Netz werden durch sinnlose, zufällige Daten gefüllt. Dadurch wird Gegnern die Entschlüsselung der auf dem Netz gesendeten Daten erheblich erschwert. Auch Rückschlüsse von der Frequenz der Netzwerkbenutzung auf die gesendeten Daten werden damit unterbunden.
- *Routing Control*
  Routing Control bezeichnet die Fähigkeit des Anwenders oder einer Soft- oder Hardware, selbst bestimmen zu können, auf welchem von ggf. mehreren möglichen Wegen – z. B. über welche zwischengeschalteten Rechner – gesendete Daten ihren Empfänger erreichen sollen.
- *Notarization*
  Eine – vertrauenswürdige – dritte Partei übernimmt die Beglaubigung von gesendeten Daten.

Darüber hinaus gibt es noch Mechanismen, die in allen Ebenen des Kommuniktionsmodells angewendet werden müssen, sogenannte *pervasive mechanisms.* Zu diesen Mechanismen zählen

- *Trusted Functionality*
  Alle Teile des Systems, die sicherheitsrelevante Funktionen ausführen, müssen geschützt und vertrauenswürdig sein.
- *Event Detection and Handling*
  Alle Versuche in das System einzubrechen oder Daten zu manipulieren müssen entdeckt und entsprechend behandelt werden.
- *Security Audit Trail*
  Eine Aufzeichnung aller Geschehnisse wird zur Verfügung gestellt, so daß im Problemfall die Sicherheit des Netzwerkes überprüft und entsprechende Maßnahmen ergriffen werden können.

3.2.3. Verschlüsselungsalgorithmen

Kernpunkt vieler Authentifikations- und Integritätsprüfungen ist die Verwendung eines Verschlüsselungsverfahrens. Gerade das Thema Verschlüsselungmethoden und -algorithmen ist jedoch sehr umfangreich; eine Einführung oder gar ein vollständiger Überblick kann daher im Rahmen dieser Arbeit nicht gegeben werden. Es sei dazu auf umfangreiche Literaturbestände verwiesen; z. B. geben Davies und Price in [3] einen umfangreichen Überblick über Grundlagen und weitergehende Methoden.

Hier sollen nur einige wenige für diese Arbeit in Frage kommende Verschlüsselungsalgorithmen knapp vorgestellt werden, so daß später dann eine Auswahl getroffen und begründet werden kann. Dennoch sei für Details auf die entsprechende Literatur (z. B. [14], [15], [3], [9]) verwiesen.

Übliche Verschlüsselungsalgorithmen sind

- *DES*
  Der *Data Encryption Standard,* ein vom amerikanischen Department of Defense eingeführter Standard-Verschlüsselungsalgorithmus; patentiert im Jahre 1975, zum nationalen Standard erhoben 1976 und veröffentlicht 1977 [7]. Er wurde später auch als ANSI X3.92-1981 vom American National Standards Institute adoptiert, dort ist er unter dem Namen *DEA-1 (Data Encryption Algorithm)* bekannt.
  Der DES ist ein Blockverschlüsselungsalgorithmus, die Blockgröße für Ein- und Ausgabetext beträgt 64 Bit. Der Schlüssel hat ebenfalls eine Größe von 64 Bit, wovon jedoch 8 Bits Paritätsfunktion (für ungerade Parität je Byte) haben.
  Der Algorithmus setzt sich aus vielen einfacheren Verschlüsselungsoperationen zusammen, darunter verschiedene Arten von Permutationen sowie Substitutionen und EXOR-Verknüpfungen. Genaue Beschreibungen des Ablaufes finden sich z. B. in [14] und [3].
  Der DES-Algorithmus wird durch ein Flag von Codierung auf Decodierung umgeschaltet, für beide Operationen wird der gleiche Schlüssel verwendet. Somit ist der DES also ein symmetrisches Verschlüsselungsverfahren.
  Der Algorithmus hat viele interessante Eigenschaften, die in der einschlägigen Literatur breit diskutiert werden. Hier ist nur wesentlich, daß der DES 'gut' verschlüsselt in dem Sinne, daß sich bei Änderung eines einzelnen Bits im Eingabetext ungefähr die Hälfte aller Ausgabebits ändern. Gleiches gilt bei Änderung eines Bits im Schlüssel.
  Der DES ist zum Standverschlüsselungsalgorithmus unter UNIX geworden. Die Bibliotheksfunktion *'encrypt'* zusammen mit der Funktion *'setkey'* stellt eine Implementierung des DES dar. Aufgrund der Exportbeschränkungen für Verschlüsselungsalgorithmen durch die amerikanische Regierung gibt es außerhalb der Vereinigten Staaten immer noch viele UNIX-Rechner, die den DES offiziell nicht unterstützen, obwohl er auch auf diesen Maschinen sogut wie immer in einigen Systemroutinen verwendet wird.
  Der DES ist mittlerweile auch als hochintegrierte Schaltung erhältlich. Diverse Systeme unterstützen solche Bausteine. Dadurch sind die Verschlüsselungen wesentlich schneller durchführbar (Software ≈ 100 ms; Hardware ≈ 5 µs).

- *RSA*
  Der RSA-Algorithmus, benannt nach den Autoren Rivest, Shamir und Adleman, wurde im Jahre 1978 publiziert [19] und stellte eine der ersten veröffentlichten Implementierungen eines asymmetrischen Verschlüsselungsverfahrens dar. Bereits im Jahre 1976 hatten Diffie und Hellman [4] gezeigt, daß ein solches Verfahren möglich ist, hatten aber keine Implementierung angeben können.
  Aus einem Startschlüssel werden mittels zweier Einwegfunktionen† die beiden Schlüssel berechnet, der Startschlüssel danach verworfen. Nun kann mittels eines Verschlüsselungsalgorithmus' mit dem einen Schlüssel codiert und dann mittels eines Entschlüsselungsalgorithmus' und dem zweiten Schlüssel wieder decodiert werden.
  Im allgemeinen ist die Konstruktion von sicheren Algorithmen zur Schlüsselgenerierung und Ver- bzw. Entschlüsselung für symmetrische Verschlüsselungsverfahren sehr schwierig. Die RSA-Methode ist eine der besten zur Zeit verfügbaren.
  Rivest, Shamir und Adleman verwenden zur Codierung die Exponentialfunktion:
  $$y = x^{PK} \quad \text{und} \quad x = y^{SK} \qquad (\text{modulo } m)$$
  wobei PK der zur Codierung verwendete öffentliche Schlüssel, SK der zur Decodierung verwendete geheime Schlüssel, x der Klartext und y der Codetext ist.

---

† *Einwegfunktion* bedeutet in diesem Zusammenhang, daß die die Berechnung der Umkehrfunktion zu aufwendig ist, um real durchgeführt werden zu können.

Für m kann keine Primzahl verwendet werden, da dann PK und SK durch einfache Berechnungen auseinander abgeleitet werden können. Die neue Idee in der RSA-Methode ist die Verwendung einer Nicht-Primzahl für m.
Sowohl für die Codierung als auch die Decodierung muß m bekannt sein. Bei der Schlüsselgenerierung wird m gebildet als Produkt zweier extrem großer Primzahlen p und q. Die Sicherheit des Systems beruht auf dem Problem der Faktorisierbarkeit von m.
Das Verfahren selbst fußt auf der Erkenntnis, daß nach Fermat's Theorem gilt:

$$x^{\lambda+1} = x \qquad (\text{modulo } m)$$

wobei λ die Wiederholungsperiode für die Exponentialfunktion modulo m ist.
Für die Codierungsfunktionen muß nun gelten:

$$x = y^{SK} = (x^{PK})^{SK} = x^{PK*SK} \qquad (\text{modulo } m)$$

Dies ist erfüllt für

$$PK*SK = 1 \qquad (\text{modulo } \lambda)$$

Für den allgemeinen Fall ist λ schwierig zu berechnen, jedoch gilt für unsere Annahme:

$$\lambda = kgV(p-1, q-1)$$

Dies kann leicht berechnet werden.
Der RSA-Algorithmus ist ein Blockverschlüsselungsalgorithmus. Problematisch sind die Berechnungszeiten bei der Ver- und Enschlüsselung: eine übliche Blockgröße beim RSA ist 512 Bit. Zur Verschlüsselung diese Blockes werden jedoch zwischen 512 und 1024 Multiplikationen benötigt. Bei Softwareimplementierungen führt dies selbst auf Mainframes zu einer Berechnungszeit im Sekundenbereich für die Codierung eines einzelnen Blockes.

- *MAA – MAC*
  Bei einem *Message Authenticator Code* handelt es sich um einen speziellen Authentifikator. Der Begriff wird normalerweise synonym zum *Message Authenticator Algorithm* verwendet. Der MAA ist ein im ISO 8731-2 definierter Standardalgorithmus zur Berechnung einer speziellen 'Prüfsumme' über einen Textblock.

(Block-)Verschlüsselungsalgorithmen lassen sich in unterschiedlichen Betriebsarten einsetzen. So wurden z. B. für den DES von der amerikanischen Regierung [15] und auch von der ISO [7] eine Reihe von Standardbetriebsmodi definiert.

Die wichtigsten Betriebsarten sind

- *ECB – Electronic Codebook*
  Bei dieser Verschlüsselungsmethode wird jeder Textblock einzeln und unabhängig vom Kontext verschlüsselt. Problematisch ist dabei, daß gleiche Quelltextstücke auch gleich verschlüsselt werden, so daß im Codetext die Struktur des Quelltextes u. U. erkennbar ist. Diese Methode nennt sich Electronic Codebook, weil – wie in einem Wörterbuch – jedem Eingabewort genau ein Ausgabewort zugeordnet wird.
- *CBC – Cipher Block Chaining*
  Hier wird die Ausgabe einer Verschlüsselung verwendet, um den nachfolgenden Eingabeblock vor der Verschlüsselung zu modifizieren (i. a. durch EXOR-Verknüpfung). Dadurch ist ein verschlüsselter Block abhängig von allen vorangegangenen Blöcken der Übertragung. Für den ersten Block einer Übertragung wird eine spezielle Initialisierung vorgesehen.
  Ein Fehler im Klartext wird durch das Verfahren exakt weitergegeben. Dagegen führt ein Fehler im Codetext nach der Entschlüsselung zu einem einzelnen, völlig zerstörten Block, gefolgt von einem Block mit den gleichen Bitfehlern wie im beschädigten Codetextblock. Nachfolgende Blöcke bleiben unangetastet.
- *CFB – Cipher Feedback*
  Eine Methode zur Verwendung eines Blockverschlüsselungsalgorithmus' für kleinere Dateneinheiten (z. B. Bytes), die sofort (wie z. B. auf einer Terminalleitung) übertragen werden müssen. Dabei erfolgt ähnlich wie beim CBC eine Verknüpfung mit einem Teil des Outputs für die

vorangegangenen Verschlüsselungen, um so den ECB-Effekt zu vermeiden.
Werden z. B. 64-Bit-Blöcke und 8-Bit-Bytes verwendet, so führt ein Fehler im Codetext zu einem Byte mit den gleichen Bitfehlern wie im Codetext, gefolgt von 8 zerstörten Bytes.

3.2.4. Key Management

Die Sicherheit jeglicher Verschlüsselung – also auch diejenige von auf Verschlüsselung aufbauender Integritäts- und Authentifikationsüberprüfungen – steht und fällt mit der Geheimhaltung des verwendeten Schlüssels. Daher ist für alle derartigen Systeme die Schlüsselverwaltung ein wesentlicher Aspekt.

Eine der größten Schwierigkeiten dabei ist die Auswahl eines Schlüssels, der dann auch noch beiden Enden des Kommunikationspfades verfügbar gemacht werden muß.

Ein anderes Problem bereitet die Speicherung von Schlüsseln: Man selbst muß die Schlüssel lesen können, anderen sollen sie unzugänglich sein.

Das Transportproblem vereinfacht sich bei Verwendung von Public-Key-Verfahren: nur der *öffentliche* Schlüssel muß dem Kommunikationspartner übermittelt werden.

Soll ein geheimer Schlüssel über das Netz transportiert werden, so muß er mit einem anderen Schlüssel codiert werden. Dies impliziert i. a. eine Schlüsselhierarchie: ein Masterkey wird nur zur Codierung von Session Keys verwendet; dadurch ist die Gefahr einer Entschlüsselung gering. Session Keys werden für jede Sitzung oder jede Übertragung neu generiert und dann zur Verschlüsselung der eigentlichen Daten verwendet. Wenn die Session Keys voneinander völlig unabhängig sind, ist es vergleichsweise ungefährlich, falls ein einzelner Session Key entdeckt wird: die Daten aller anderen Sitzungen bleiben geheim.

Eine Schlüsselhierarchie berücksichtigt also, daß ein Schlüssel um so gefährdeter ist (d. h. leichter entdeckt werden kann), je öfter er verwendet wird, bzw. je mehr Datenvolumen mit ihm verschlüsselt wird. Dem Problem wird damit entgegnet, daß diejenigen Schlüssel, die zur Verschlüsselung der meisten Daten verwendet werden auch am häufigsten gewechselt werden. Zur Realisierung dieses Austauschs kann dann der Schlüssel der nächsthöheren Stufe bei der Codierung verwendet werden.

Prinzipiell kann eine solche Hierarchie beliebig erweitert werden. Mittlerweile wurden auch von verschiedenen Institutionen Standards für solche Schlüsselhierarchien definiert (vgl. [3], S. 133 ff).

Wie groß eine solche Hierarchie sein muß, d. h. wie viele Stufen sie umfassen muß und wie oft die Schlüssel der einzelnen Stufen gewechselt werden müssen um eine ausreichende Absicherung zu erreichen, kann nur schwer beurteilt werden.

3.3. Yellow Pages

Beim Yellow Pages Service (abgekürzt YP) handelt es sich um einen von der Firma Sun Microsystems entwickelten Netzwerkservice (unter UNIX). Er soll Daten, die auf einem Zentralrechner gehalten werden, einer Gruppe von vernetzten Rechnern zugänglich machen oder – anders betrachtet – Daten von verschiedenen Rechnern zentral verwalten.

3.3.1. Namenskonventionen

Unter dem Yellow Pages Service heißt jede verwaltete Tabelle† von Daten *Map.* Eine solche Map wird spezifiziert durch einen Namen, genannt *Mapname.* Jede Map beinhaltet eine Liste von Paaren aus Schlüssel und Wert. Eine Gruppe von Maps heißt *Domain* und wird durch ihren Gruppennamen *'Domainname'* spezifiziert. Jeder Rechner des Netzes ist genau einem Domain zugeordnet. Zu jeder Map gibt es im Domain genau einen *Master Server* und eine beliebige Zahl von *Slave Servern.* Alle anderen Rechner im Domain sind bezüglich dieser Map dann *Clients.*

3.3.2. Funktionsweise

Der Master Server verwaltet die von ihm geführten Maps in einer Datenbank. Änderungen an der Datenbank dürfen nur auf dem Master Server vorgenommen werden. Die Datenbank wird vom Master Server komplett an die Slave Server propagiert. Die Server verhalten sich ansonsten passiv; sie warten auf eine Anfrage durch einen Client. Clients können beim Server den einem bestimmten Schlüssel zugeordneten Wert abfragen oder aber sich alle Schlüssel/Werte-Paare in der Datenbank aufzählen lassen. In einem stabilen Zustand ist es gleichgültig, durch welchen Server eine Anfrage beantwortet wird, da dann die zurückgegebene Antwort immer die gleiche ist. Die Clients schalten automatisch auf einen anderen Server um, falls der bisher verwendete Server nicht erreichbar oder überlastet ist. Prinzipiell ist es möglich, daß in einem Domain für verschiedene Maps auch verschiedene Master Server existieren, davon rät Sun Microsystems jedoch dringends ab (vgl. [21], S. 35ff).

Üblicherweise werden die Yellow Pages eingesetzt als Ersatz für einige UNIX-Systemdateien, z. B. `/etc/hosts` (Liste der Rechner im Netz mit zugehöriger Rechneradresse), `/etc/passwd` (Liste der Benutzer auf dem Rechner mit zugehörigen Daten) und `/etc/group` (Liste der Benutzergruppen eines Rechners mit zugehörigen Benutzern). Es ist auch möglich, eigene Datenbanken mittels des Yellow Pages Service zu verteilen.

3.3.3. Vor- und Nachteile

Das Yellow Pages Konzept bietet diverse Vorteile:

- *Hohe Verfügbarkeit*
  Da es im Domain mehrere Server für eine Map geben kann, ist die Verfügbarkeit der Daten sowie die Zugriffsgeschwindigkeit i. a. sehr hoch.
- *Flexibilität*
  Prinzipiell kann jeder Datenbestand bei entsprechender Anpassung mittels YP verteilt werden.

† mit *Tabelle* ist hier eine Relation im Sinne einer relationalen Datenbank gemeint

Das Yellow Pages Konzept hat aber auch verschiedene Nachteile:

- *Netzwerkzugriff verlangsamt Operationen*
  Während ohne Verwendung des Yellow Pages Service alle Informationen aus lokalen Dateien gewonnen werden konnten, sind nun sehr häufig Netzwerkzugriffe auf die Server nötig. Dadurch kann es zu Geschwindigkeitseinbußen kommen.
- *Ausfall der Server*
  Falls alle Server ausfallen (evtl. durch Unterbrechung der Netzverbindung), können viele Operationen nicht mehr ausgeführt werden. So können sich z. B. Benutzer, die nicht in der lokalen Benutzerliste eingetragen sind, sondern die normalerweise durch Informationen aus dem Yellow Pages Service zum Rechnerzugang berechtigt werden, sich nicht mehr auf den Rechner einloggen.
- *Änderungen nur auf Master*
  Änderungen an den durch Yellow Pages verwalteten Daten sind nur auf dem Master Server durchführbar. Lokal oder auf den Slave Servern durchgeführte Änderungen werden ohne Warnung ignoriert bzw. überschrieben und können sogar den Update-Algorithmus des Yellow Pages Systems empfindlich stören.
- *Aufteilung nicht möglich*
  Jeder Rechner ist fest einem Domain zugeordnet. Dadurch ist es nicht möglich, daß ein Rechner Informationen bezüglich einer Map M_1 aus dem Domain D_1 und gleichzeitig Informationen bezüglich einer anderen Map M_2 aus einem zweiten Domain D_2 bezieht.
- *Unterschiedliche Inhalte bei verschiedenen Servern*
  Die Existenz mehrerer Server (ein Master und ggf. mehrere Slave Server) für eine Map erhöht zwar Verfügbarkeit und Zugriffsgeschwindigkeit, birgt aber gleichzeitig die Gefahr von Inkonsistenzen. Falls der Master Server nach einer Änderung der Datenbank aus irgendwelchen Gründen nicht in der Lage ist, diese Daten unverzüglich an alle Slave Server weiterzugeben, existieren auf den Slave Servern überholte Daten, die trotzdem noch an Clients weitergegeben werden.
- *Einbindung*
  Alle Programme, die sich auf die Yellow Pages abstützen, müssen sich dessen 'bewußt' sein, d. h. bei Zugriffen auf den Datenbestand müssen die entsprechenden Zugriffsfunktionen für die Yellow Pages verwendet werden. Eine Umstellung einer bestehenden Software auf Yellow Pages ist nur möglich durch Modifikation der entsprechenden Zugriffsroutinen im Quellcode und Recompilation. Dies impliziert, daß die Umstellung von fest installierter oder erworbener Software ohne Quellcode für den Anwender unmöglich ist.

3.3.4. Sicherheitsaspekte

Die Yellow Pages unterscheiden zwischen 'lokalen' und 'globalen' Dateien. Bei 'globalen' Dateien werden grundsätzlich (sofern die Yellow Pages aktiviert sind) nur die YP-Routinen verwendet um an entsprechende Daten zu gelangen. Lokal vorhandene Dateien mit entsprechenden Daten werden ignoriert. Anders ist die Vorgehensweise bei 'lokalen' Dateien. Hier wird zunächst immer die lokal vorhandene Datei gelesen. Nur wenn diese Datei eine mit '+' beginnende Zeile enthält, werden an der Stelle Informationen aus den Yellow Pages gelesen, andernfalls bleibt der Yellow Pages Service unbenutzt.

Bei 'lokalen' Dateien liegt die Sicherheit des Systems also neben dem Server auch noch in der Hand des Clients. Dagegen vertrauen die Clients bei 'globalen' Dateien ganz auf die Korrektheit der vom Server gelieferten Daten. Im Sun-UNIX werden `/etc/passwd` und `/etc/group` als 'lokale'

Dateien behandelt, `/etc/hosts` und alle anderen YP-Maps werden als 'globale' Dateien betrachtet.

Die Yellow Pages Routinen stützen sich auf RPC-Routinen (vgl. Kap. 3.4 sowie [22], [23], [24] und [25]). Damit hängt die Sicherheit des Systems einzig und allein von der Sicherheit der verwendeten RPC-Routinen ab – wenn man von Sicherheitslücken absieht, die durch fehlerhafte Konfigurationen seitens des Benutzers entstehen können.

3.4. RPC vs. Secure RPC

RPC – Remote Procedure Call – ist ein Softwarepaket der Firma Sun Microsystems. Das Paket umfaßt eine Funktionsbibliothek mit verschiedenen Netzwerkfunktionen sowie einige Server-Programme.

Secure RPC stellt eine verbesserte Version des ursprünglichen RPC-Systems dar und enthält einen Authentifizierungsmechanismus, der die Sicherheit im Netzwerk verbessern soll.

Das RPC wurde in erster Linie für das Sun-UNIX entwickelt, soll aber so allgemein gehalten sein, daß auch eine Verwendung mit anderen UNIX-Versionen und nicht-UNIX-Systemen möglich ist. Da sich alle Sun-Netzwerkapplikationen – also z. B. NFS und YP – auf die RPC-Funktionen abstützen, ist eine Umstellung der gesamten Netzwerksoftware von RPC auf Secure RPC durch Binden mit der neuen Library möglich.

3.4.1. Funktionsprinzip

Das Sun RPC implementiert ein Modell zum Aufruf von Prozeduren auf Remote-Maschinen, welches ganz ähnlich dem Modell zum Aufruf lokaler Prozeduren ist.

Im lokalen Fall schreibt der Aufrufer einer Prozedur zunächst die Parameter an eine vordefinierte Stelle (z. B. ein Registerfenster). Dann wird die Kontrolle an die Prozedur übergeben. Schließlich erhält der Aufrufer die Programmkontrolle zurück. Zu diesem Zeitpunkt werden dann die Ergebnisse der Prozedurabarbeitung einer vordefinierten Stelle entnommen und dann mit der Ausführung fortgefahren.

Das RPC-Modell verhält sich hier ganz ähnlich: ein Kontrollfluß verbindet logisch zwei Prozesse, einen Caller und einen Server. Der Caller sendet eine Anfrage an den Server und wartet dann auf eine Antwort. Die Anfrage umfaßt den Prozedur'namen' und die Prozedurparameter, die Antwort die Prozedurergebnisse. Wurde eine Antwort erhalten, extrahiert der Caller die Ergebnisse aus dem Antwortpaket und fährt mit der Programmausführung fort. Auf der Server-Seite schläft ein Prozeß, bis ihn ein Aufruf erreicht. Trifft ein Anfrage ein, bestimmt der Prozeß die auszuführende Prozedur, extrahiert die Eingabeparameter, berechnet die Ergebnisse, sendet eine Antwort und erwartet den nächsten Aufruf.

Die RPC-Funktionsbibliothek stellt eine Reihe von Funktionen zur Verfügung, die es dem Programmierer ermöglichen, auf einfache Weise eine Kommunikation über das Netz zu implementieren. Ohne RPC müssen solche Operationen mittels 'Sockets' realisiert werden, welche aber nur vergleichsweise primitive Zugriffsoperationen bieten und daher u. U. relativ aufwendig zu programmieren sind. Das RPC bietet drei Gruppen von Funktionen mit unterschiedlichen Abstraktionslevels.

Auf dem obersten Level werden Funktionen zur Verfügung gestellt, die den Netzwerkzugriff vollständig verbergen. So existiert z. B. die Funktion `rnusers`, die als als Ergebnis die Anzahl von Benutzern auf einer Remote-Maschine zurückliefert. Dabei ist der Netzzugriff, der genau in der weiter unten beschriebenen Weise stattfindet, für den Anwender völlig transparent; die Funktionen verhalten sich äußerlich genau so, als ob sie lokal ausgeführt worden wären.
Zu den Funktionen dieses Levels gibt es im RPC-Paket auch bereits Server-Programme, die die entsprechenden Anfragen beantworten.

Die Funktionen des mittleren Levels stellen immer noch ein vergleichsweise einfach zu verwendendes Interface für die Nutzung des RPC-Services dar. Die beiden Basisfunktionen dieses Levels sind `registerrpc` und `callrpc`. Mittels `registerrpc` wird ein neuer RPC-Service beim System angemeldet; mit `callrpc` wird eine RPC-Funktion ausgeführt. Diese Funktionen erlauben die Programmierung eigener RPC-Service-Routinen.

Auf dem untersten Level stellt die RPC-Bibliothek eine Vielzahl von Funktionen zur Verfügung, die es dem Programmierer erlauben, das Standardverhalten der RPC-Funktionen des mittleren Levels zu modifizieren. Sun Microsystems empfiehlt jedoch, die Funktionen des untersten Levels nach Möglichkeit zu meiden (vgl. [25], S. 3), da hier ein detailliertes Wissen über die RPC-Mechanismen nötig ist.

3.4.2. Funktionsweise

Jede vom RPC-Service ausgeführte Prozedur wird durch eine Reihe von Nummern gekennzeichnet. Dabei handelt es sich um:

- *Programmnummer*
  Diese Kennzahl identifiziert das auf der Remote-Maschine zu verwendende Server-Programm. Für Standardprogramme gibt es weltweit einheitliche Nummern, deren Vergabe von Sun Microsystems verwaltet wird.
- *Versionsnummer*
  Mit dieser Nummer wird die Version des Programms angegeben. Auch sie wird zur Auswahl des auf der Remote-Maschine zu verwendenden Programms herangezogen. Dadurch ist es möglich, neue Versionen von Standard-Programmen zu erstellen, ohne daß eine neue Programmnummer beantragt werden muß. Da die Versionsnummer dem jeweiligen Server-Programm übergeben wird, kann auch ein Server-Programm mehrere Versionen bedienen.
- *Prozedurnummer*
  Die Prozedurnummer wird dem Server-Programm übergeben. Mittels der Prozedurnummer kann – falls das Server-Programm mehrere Funktionen anbietet – eine bestimmte Funktion selektiert werden.

Intern läuft die Kommunikation mit Remote-Maschinen – wie unter UNIX üblich – über Sockets. Diese Sockets werden durch Portnummern benannt. Damit nicht für jeden möglichen RPC-Service auf jeder Maschine ein Port reseviert werden muß, erfolgt die Adreßbindung dynamisch.

Will nun ein Client Kontakt mit einem RPC-Server auf einer Remote-Maschine aufnehmen, so ist ihm zunächst aufgrund der dynamischen Adreßbindung nicht bekannt, welche Portnummer er ansprechen muß. Die Abbildung zwischen UNIX-Portnummern und RPC-Programm-/Versionsnummern übernimmt daher ein spezieller RPC-Prozeß namens `portmap`, der als einziger RPC-Server eine reservierte Portnummer besitzt.

Sobald ein Server-Prozeß gestartet wurde, läßt er alle Programm-/Versionsnummern-Paare, die er zu bedienen beabsichtigt, zusammen mit seiner eigenen aktuellen Portnummer, mittels `registerrpc` beim Portmapper registrieren.

Bei einer Inanspruchnahme eines RPC-Services wird zunächst der Portmapper kontaktiert. Dieser liefert einen Fehler, falls der angeforderte Service nicht registriert wurde, andernfalls die Portnummer, unter der der Service dann angesprochen werden kann.

3.4.3. Authentifizierung unter RPC

Bereits in der ursprünglichen RPC-Version ist ein Authentifizierungsmechanismus vorgesehen. Die Datenstruktur, die dem Server vom Caller übergeben wird, sieht Felder vor, welche die Handhabung eines variabel großen Authentifizierungs-Datenbereiches ermöglichen. Somit ist die Art der Authentifikation prinzipiell frei programmierbar. Allerdings werden standardmäßig nur zwei Typen von Authentifikatoren unterstützt:

- AUTH_NULL
  Das Authentifikatorfeld ist leer, es wird keine Authentifikation durchgeführt. Dies ist der Standard unter RPC.
- AUTH_UNIX
  Das Authentifikatorfeld zeigt auf eine Datenstruktur `authunix_parms`, in der UNIX-Uid, UNIX-Gid's und Rechnername des Callers, sowie ein Zeitstempel enthalten sind.

Problem der Authentifikation ist der fehlende Verifikator: der Server hat keine Möglichkeit zu überprüfen, ob die Angaben des Clients korrekt sind. Prinzipiell kann der Client in den Authentifikator hineinschreiben, was er will, ohne daß der Server es bemerken kann (vgl. [25], S. 49).

3.4.4. Ergänzungen im Secure RPC

Im wesentlichen unterscheidet sich das Secure RPC vom ursprünglichen RPC durch die Unterstützung zweier zusätzlicher Authentifizierungs-Typen:

- AUTH_DES
  Der Authentifikator beinhaltet – im Gegensatz zur UNIX-Authentifikation – neben den Daten des Clients auch einen Verifikator. Hauptbestandteil des Verifikators ist ein mittels Public-Key-Verfahren verschlüsselter Zeitstempel.
- AUTH_SHORT
  Diese Authentifikation stellt eine verkürzte Version der Typen AUTH_UNIX und AUTH_DES dar: ist einmal eine Authentifikation erfolgreich abgelaufen, so besteht die Möglichkeit, daß der Server eine Datenstruktur vom Typ AUTH_SHORT zurückliefert. Diese kann dann in weiteren Zugriffen zur verkürzten Authentifikation verwendet werden. Der Server kann die Gültigkeit dieser Kurz-Authentifizierung willkürlich festlegen und jederzeit zurückziehen.

Basis des Authentifikationstyps AUTH_DES im Secure RPC sind DES-Verschlüsselung und Public-Key-Verschlüsselung.

Die öffentlich lesbare Systemdatei `/etc/publickey` enthält eine Liste von Benutzernamen, denen jeweils ein öffentlicher Schlüssel und eine codierte Version des entsprechenden privaten Schlüssels zugeordnet ist. Der Benutzername ist hier der sogenannte `netname`, ein willkürlicher, für jeden Netzbenutzer eindeutiger Name; damit wird gleichzeitig das Problem der UNIX-Namesgebung† gelöst. Bei einem Login wird mit dem eingegebenen Paßwort der geheime Schlüssel decodiert und an einen Daemon-Prozeß namens `keyserv` übergeben. Dieser speichert den privaten Schlüssel in der Datei `/etc/keystore`, die nicht öffentlich zugänglich ist. Zweck dieser Aktion ist es, den Authentifizierungsmechanismus für den Benutzer transparent zu machen; der Benutzer braucht nicht bei jeder einer Authentifizierung bedürfenden Aktion sein geheimes Paßwort einzugeben. Es ist vorgesehen, daß die Datei `/etc/publickey` mittels YP exportiert wird.

Während unter der früheren RPC-Version die Server bei der Authentifizierung auf die Richtigkeit der Angabe der Client-Rechner vertrauen mußten, ist im *Secure RPC* eine Verifikation möglich.

Der RPC-Client A generiert zufällig einen DES-Schlüssel, den *Conversation Key* K_C. Dem Server B werden zu Beginn der Anfrage folgende Informationen übermittelt:

$$A \rightarrow B: \quad A,\ PK_B(SK_A(K_C)),\ K_C(t_A),\ K_C(\Delta t),\ K_C(\Delta t+1)$$

† Unter UNIX wird der Benutzername in eine User-ID umgesetzt und die eigentliche Benutzeridentifikation dann mittels dieses numerischen Wertes durchgeführt. User-IDs sind jedoch rechnerlokal, gleiche Benutzer haben auf verschiedenen Maschinen durchaus unterschiedliche User-IDs.

Dabei ist t_A die aktuelle Uhrzeit und Δt ein Zeitfenster. Mittels t_A kann der Server überprüfen, ob die Anfrage gültig ist. Voraussetzung dafür ist jedoch die Zeitsynchronisation der beiden Rechner. Nachfolgende Anfragen werden vom Server nur beantwortet, falls der neue Timestamp im Vergleich zu t_A größer geworden ist. Dies bietet Schutz vor Replays. Das Zeitfenster spezifiziert einen Zeitraum, innerhalb dessen K_C gültig sein soll. Die zusätzliche Übermittlung von $\Delta t+1$ bietet einen Schutz gegen gefälschte Daten. Der Server antwortet mit

$$B \rightarrow A: \quad K_C(t_A - 1), ID$$

Dabei ist ID ein *Nickname,* realisiert als Integer-Index in eine interne Tabelle des Servers. Der Client kann die Identität des Servers überprüfen, da nur dieser durch Entschlüsselung von t_A die korrekte Antwort liefern kann.

Alle weiteren Anfragen (innerhalb des spezifizierten Zeitfensters) können dann mit der Authentifizierungsmethode `AUTH_SHORT` gehandhabt werden. Der Client authentifiziert sich dabei in verkürzter Form mittels seines 'Nicknames':

$$A \rightarrow B: \quad ID, K_C(t_{An})$$

t_{An} ist die jeweils aktuelle Uhrzeit. Der Server antwortet jeweils mit

$$B \rightarrow A: \quad K_C(t_{An} - 1)$$

3.5. Kerberos

Kerberos ist ein Authentifizierungssystem für offene Netzwerke. Dieses System wurde am Massachusetts Institute of Technology im Rahmen des Projekts *Athena* entwickelt und zur *USENIX* im Winter 1988 in Dallas erstmalig vorgestellt ([20], [2] und [17]).

3.5.1. Intention

Am Massachusetts Institute of Technology besteht ein extrem großes Rechnernetz: über 750 Computer sind in 30 Subnetzen miteinander verbunden und bedienen mehr als 5000 aktive Benutzer. Sowohl Single-User-Maschinen als auch Multi-User-Systeme sind dabei vertreten. Auf einigen Rechnern laufen jedoch Service-Programme, die allen Netzbenutzern zur Verfügung stehen. Da aber die meisten der kleineren Maschinen unter voller Kontrolle ihrer Benutzer stehen – sie also ohne weiteres Superuser-Rechte bekommen – kann den einzelnen Rechnern bei der Identifizierung der Benutzer nicht vertraut werden. Gleiches gilt für das Netz als solches: die Datenübertragung ist nicht sicher, da einzelne Benutzer den gesamten Verkehr abhören oder Mitteilungen mit falschen Adreßangaben absetzen können.

Um dennoch eine Identifikation der Benutzer möglich zu machen, wurde Kerberos entwickelt. Wesentlich dabei war die Angemessenheit: die Authentifizierung soll für den Benutzer transparent ablaufen und ihm – bei maximaler Sicherheit – möglichst wenig zusätzlichen Aufwand bereiten.

3.5.2. Realisierung

Kerberos basiert auf dem Verschlüsselungsprotokoll nach Needham und Schröder (vgl. [16] und Kap. 3.1). Ein Serverprogramm, genannt *Kerberos-Server,* läuft auf einem sicheren Rechner. Kerberos kennt für jeden verwalteten Benutzer und jeden verwalteten Server ein geheimes Paßwort. Da lediglich Kerberos diese geheimen Schlüssel kennt, kann er Mitteilungen erzeugen, die einen Teilnehmer gegenüber einem Dritten authentifizieren.

Außerdem generiert Kerberos temporäre geheime Schlüssel, sogenannte *Session Keys,* die nur an je zwei Teilnehmer vergeben werden und so zur Verschlüsselung der Übermittlungen zwischen diesen beiden Teilnehmern dienen können.

Kerberos bietet drei verschiedene Sicherheitsstufen:

- *authentifizierte Übermittlung*
  Nur beim Aufbau der Verbindung wird eine Authentifizierung der teilnehmenden Parteien durchgeführt. Alle weiteren Übermittlungen werden ohne weitere Kontrolle als gültig angesehen.
- *sichere Übermittlung*
  Jede einzelne Übermittlung wird authentifiziert.
- *private Übermittlung*
  Jede einzelne Übermittlung wird nicht nur authentifiziert, sondern zusätzlich noch verschlüsselt.

Es ist unter Kerberos möglich, auf weiteren sicheren Rechnern im Netz Read-Only-Kopien der Kerberos-Datenbank zu halten, so daß auch diese Rechner die Authentifizierung durchführen können.

Jeder Teilnehmer, d. h. Benutzer oder Service, wird durch einen eindeutigen Namen identifiziert und zwar nach dem Schema *name.instance@realm.* Dabei ist *name* der Name des Teilnehmers, also z. B. `root` oder `rlogin`. *instance* dient zur Unterscheidung von Varianten des Hauptnamens und spiegelt im allgemeinen den Rechnernamen wieder, also z. B. *rlogin.quando* für den `rlogin`-

Server auf der Maschine 'quando'. *realm* ist der Name einer Verwaltungseinheit, die einen eigenen Kerberos-Service verwendet.

3.5.3. Ablauf der Authentifizierung

Die Authentifizierung gliedert sich in drei Phasen:

- *Anforderung des Starttickets*
  Zu Beginn einer Sitzung identifiziert sich der Benutzer gegenüber dem Kerberos-Server und erhält danach ein Startticket, welches ihn zur Anforderung von Servicetickets berechtigt.
- *Anforderung von Servicetickets*
  Mittels des Starttickets fordert der Benutzer vom Kerberos-Server Servicetickets für die Services, die er in Anspruch nehmen möchte.
- *Übermittlung der Servicetickets*
  Der Benutzer weist sich mittels des Servicetickets beim Service als nutzungsberechtigt aus.

3.5.3.1. Identifikatoren

Im *Kerberos-Modell* werden zwei Arten von Identifikatoren verwendet: *Tickets* und *Authentifikatoren.* Beide Typen basieren auf einer Private-Key-Verschlüsselung.

Ein Ticket wird verwendet um die Identität der Person, der das Ticket ursprünglich ausgehändigt wurde, sicher vom Authentifizierungs-Server zum Anwendungs-Service zu übermitteln. Außerdem enthält das Ticket Informationen, die dazu verwendet werden können, sicherstellen, daß diejenige Person, die das Ticket verwendet, dieselbe ist, für die das Ticket ausgestellt wurde.

Der Authentifikator enthält zusätzliche Informationen, die bei einem Vergleich mit den im Ticket enthaltenen Daten garantieren, daß der Rechner, der das Ticket verwendet, derselbe ist, für den das Ticket ausgestellt wurde.

Ein Ticket ist gültig für einen einzelnen Service S und einen einzelnen Benutzer A. Es kann – innerhalb seiner festgelegten Gültigkeitsdauer – mehrfach verwendet werden und hat folgenden Aufbau:

$$T_{A,S} = K_S(A, S, Adr_A, t, \Delta t, K_{A,S})$$

Dabei bezeichnet Adr_A die Rechner-Internet-Adresse des Benutzers A, t ist ein Zeitstempel, Δt die Lebensspanne des Tickets und $K_{A,S}$ ein zufällig gewählter Session Key für die Kommunikation zwischen A und S.

Da das Ticket dem Verwender (mit dem geheimen Schlüssel K_S des Servers) verschlüsselt ausgehändigt wird, kann der Authentifizierungsserver sicher sein, daß der Benutzer das Ticket nicht (sinnvoll) verändern kann.

Ein Authentifikator kann – im Gegensatz zum Ticket – nur einmal verwendet werden. Dies stellt jedoch kein Problem dar, weil der Benutzer selbst Authentifikatoren generieren kann. Ein Authentifikator hat folgenden Aufbau:

$$A_{A,S} = K_{A,S}(A, Adr_A, t_A)$$

t_A ist dabei die aktuelle (Rechner-)Zeit zum Zeitpunkt der Generierung. Der Authentifikator wird mit dem Session Key $K_{A,S}$ verschlüsselt.

3.5.3.2. Das Startticket

Ein Benutzer A kann nach dem Login lediglich durch sein privates Kennwort bzw. durch den daraus abgeleiteten Schlüssel K_A identifiziert werden. Bei der Identifizierung gegenüber dem Authentifizierungsserver muß zwar sichergestellt werden, daß nur ein Benutzer, der das korrekte Kennwort besitzt, die Authentifikation korrekt absolvieren kann, jedoch muß auch die Gefahr der Kompromittierung durch andere Netzbenutzer minimiert werden (d. h. das Kennwort darf *nicht* über das Netz übertragen werden).

Daher übermittelt der Rechner beim Login eines Benutzers den Benutzernamen an den Kerberos-Server:

$$A \rightarrow KS: \quad A$$

Der Kerberos-Service antwortet – sofern er den genannten Benutzer kennt – mit einem Startticket und einem Session Key

$$KS \rightarrow A: \quad K_A(T_{A,TGS}),\ K_A(K_{A,TGS})$$

Das Startticket ist ein spezielles Serviceticket für den *Ticket-Granting Service (TGS).* Sowohl das Startticket als auch der Session Key sind mit dem Schlüssel K_A des Benutzers A codiert. Das Startticket darüberhinaus auch noch mit dem geheimen Schlüssel K_{TGS} des TGS (vgl. Aufbau des Starttickets).

Nur bei Kenntnis des richtigen Schlüssels K_A – der nur aus dem korrekten Kennwort für den Benutzer A abgeleitet werden kann – können Startticket und Session Key entschlüsselt und verwendet werden.

3.5.3.3. Service-Anforderung

Unter der Annahme, daß ein Benutzer A bereits das entsprechende Serviceticket besitzt, kann er einen Service S seiner Wahl in Anspruch nehmen.

Dazu generiert der Benutzer einen Authentifikator $A_{A,S}$ und verschlüsselt ihn mit dem Session Key $K_{A,S}$, den er von Kerberos für diese Verbindung erhalten hat. Dann übermittelt er den Authentifikator zusammen mit dem zugehörigen Serviceticket dem Server S:

$$A \rightarrow S: \quad K_{A,S}(A_{A,S}),\ K_S(T_{A,S})$$

Der Server S kann dann das Serviceticket, welches von Kerberos mit K_S verschlüsselt wurde, decodieren. Im Ticket ist eine Kopie des Session Keys $K_{A,S}$ enthalten, den S nun zur Decodierung des Authentifikators verwendet. Nur wenn die Daten aus dem Authentifikator mit denen aus dem Ticket übereinstimmen, gilt die Anforderung als authentifiziert und wird vom Server bearbeitet.

Übereinstimmung der Daten heißt in diesem Fall, daß sowohl A als auch Adr_A in Ticket und Authentifikator identisch sind und daß gilt: $t \leq t_A \leq t_A + \Delta t$.

Eine Annahme, die das Kerberos-Modell trifft, ist die Synchronisation der verschiedenen Rechneruhren im Netz bis auf wenige Minuten. Unter dieser Annahme ist es möglich, Service-Anforderungen mit zu großer Differenz zwischen Timestamp im Authentifikator und Uhrzeit im Server zurückzuweisen. Gleiches gilt für aufeinanderfolgende Anforderungen mit gleichem Timestamp.

Das bisherige Verfahren sichert lediglich die Authentizität des Benutzers gegenüber dem Server. Das Kerberos-Modell bietet zusätzlich noch die Möglichkeit der Authentifizierung des Servers gegenüber dem Benutzer.

Dieses Verfahren nennt das Kerberos-Modell *'Mutual Authentification'.* Dabei inkrementiert der Server nach Empfang und Entschlüsselung des Authentifikators den darin enthaltenen Timetag und

sendet ihn – verschlüsselt durch den Session Key – an den Benutzer zurück.

$$S \rightarrow A: \quad K_{A,S}(t_A+1)$$

3.5.3.4. Anforderung von Servicetickets

Wie bereits erwähnt, ist ein Ticket lediglich für einen einzelnen Service gültig. Daher muß der Benutzer für jeden Service, den er in Anspruch nehmen möchte, ein separates Serviceticket besitzen.

Diese Servicetickets kann er mittels seines Starttickets vom TGS anfordern. Da es sich auch beim TGS lediglich um einen speziellen Service handelt, kann man ihn nach dem oben beschriebenen Verfahren benutzen, um die nötigen Tickets unter Verwendung von $T_{A,TGS}$ zu erhalten.

Der TGS generiert nach Erhalt und Überprüfung einer entsprechenden Anforderung – die entsprechende Berechtigung des Benutzers vorausgesetzt – einen Session Key für die Verbindung zwischen dem neuen Server und dem Benutzer und sendet ihn, zusammen mit dem entsprechenden Service-Ticket, an den Benutzer zurück.

3.5.4. Kerberos-Datenbank

Alle bisher beschriebenen Zugriffe auf die Datenbank haben sich auf das Lesen von Daten beschränkt. Daher konnten alle diese Zugriffe auch von Slave-Servern bedient werden, die auf Read-Only-Kopien die Datenbank arbeiten. Für verschiedene Zwecke sind auch Schreibzugriffe nötig:

- Benutzer müssen ihre in der Datenbank gespeicherten geheimen Kennwörter ändern können
- Administratoren müssen neue Benutzer in die Datenbank eintragen können
- Administratoren müssen Kennwörter für Benutzer ändern können

Diese Änderungen werden im Kerberos-Modell von einem speziellen Service verwaltet, dem *Kerberos Database Management Service (KDBM).* In der derzeitigen Kerberos-Implementationen sind Datenbank-Änderungen nur auf der Master-Kopie möglich, daher läuft der KDBM auch nur auf dem Kerberos-Zentralrechner.

Der KDBM unterscheidet sich von allen anderen gewöhnlichen Services dadurch, daß der TGS für ihn keine Tickets ausgibt. Tickets für den KDBM werden nur vom Kerberos-Server selbst ausgegeben. Hierdurch wird der Benutzer gezwungen, sein Paßwort einzugeben, jedesmal wenn er den KDBM benutzen will. So kann verhindert werden, daß, wenn ein Benutzer sein Terminal unbeaufsichtigt läßt, trotzdem sein Paßwort nicht von jedem Passanten ohne weiteres geändert werden kann.

Auf der Client-Seite existieren für die KDBM-Benutzung zwei Programme nämlich `kpasswd`, welches für die Änderung von Benutzerpaßwörtern zuständig ist und `kadmin`, mit dem Administratoren ihre Anfragen durchführen.

Auf der Kerberos-Maschine gibt es eine Liste von Zugriffsberechtigen. Änderungen an Paßwörtern sind nur zugelassen, wenn derjenige, der die Änderung durchführt und derjenige, dessen Paßwort geändert wird, identisch sind, oder der Ändernde in der Liste von Zugriffsberechtigten eingetragen ist.

Eintragungen in dieser Liste gehören normalerweise immer einer bestimmten Instanz an, z. B. *'admin'*. Dies hat den Vorteil, daß Administratoren zur Durchführung ihrer administrativen Tätigkeiten ein anderes Paßwort verwenden können als für ihren gewöhnlichen Login.

Der KDBM protokolliert alle Anfragen, erfolgreich oder nicht, in einer Datei.

Wie bereits erwähnt, kann jedes Kerberos-Realm außer der Kerberos-Master-Datenbank noch mehrere Read-Only-Kopien der Datenbank auf anderen Rechnern besitzen. Diese Kopien sind nicht notwendig, erhöhen aber die Verfügbarkeit der Authentifikation. Gleichzeitig tritt dadurch aber auch das Problem von Datenkonsistenz auf. Praktische Tests der Entwickler haben gezeigt, daß vergleichsweise primitive Methoden ausreichen, um diese Konsistenzprobleme handzuhaben.

Auf dem Master-Server läuft ein Programm namens `kprop`, welches stündlich die komplette Datenbank an die Slave-Server verschickt. Auf den Slave-Servern werden die Daten von einem Peer namens `kpropd` entgegengenommen. Um eine sichere Übertragung zu garantieren wird vom Master eine Prüfsumme über die Datenbank berechnet. Diese Prüfsumme wird vom Slave-Server erst überprüft, bevor er den neuen Datenbankinhalt übernimmt. In der Datenbank sind alle Paßwörter mit dem Kerberos-Master-Schlüssel codiert, den sowohl Master- als auch Slave-Server kennen. Auch die Prüfsumme wird während der Übertragung über das Netz mit dem Master-Schlüssel codiert. Dadurch sind alle Daten, die über das Netz geschickt werden für einen Gegner nutzlos. Durch die Codierung und die Prüfsumme ist eine Veränderung der Daten bei der Übertragung ausgeschlossen.

3.6. Andere Ansätze

Im folgenden Kapitel werden noch einige Vorschläge verschiedener Autoren kurz beschrieben, die sich mit der Authentifizierung in offenen Netzen beschäftigen.

3.6.1. RFC-1004: A Distributed-Protocol Authentication Scheme

Im RFC-1004 [13] stellt D. L. Mills von der University of Delaware ein Schema für die Zugriffskontrolle und Authentifizierung in Netzen mit verteiltem Protokoll vor. Sein Authentifizierungsschema basiert auf dem Modell von Needham und Schröder [16].

Das Schema nach Mills geht davon aus, daß zwischen zwei Benutzern, die zwar jeweils selbst in einer gesicherten Umgebung arbeiten, eine Verbindung über ein ungesichertes Netz hergestellt werden soll. Dabei dürfen die beiden Benutzer unterschiedlichen Sicherheits-Umgebungen angehören.

Ziel des Schemas ist nicht die Verschlüsselung von Nachrichten (außer privaten Schlüsseln), sondern für zugelassene Verbindungen Spoofing und Replays auszuschließen.

Basis des Schemas nach Mills ist ein unabhängiger Server-Rechner, genannt *'Cookie Jar'*, welcher für die Vergabe der Authentisierungsschlüssel zuständig ist. Jedem authorisierten Benutzer im Netz ist außerdem ein eindeutiger Name A zugeteilt. Der Cookie Jar besitzt für jeden authorisierten Benutzer eine Liste L_A von zugelassenen Kommunikationspartnern und einen privaten Schlüssel K_A. Die Listen L_A sind öffentlich lesbar. Sowohl K_A als auch L_A können nur vom Cookie Jar geändert werden.

Ablauf einer Kommunikation:
Zunächst sendet der Benutzer A folgendes Paket an den Cookie Jar S:

$$A \rightarrow S: \quad A, S, I_{A1}, F_{A1}, A, B$$

Dabei sind die beiden ersten Felder ein Mitteilungskopf, der bei allen Nachrichten immer die Adressen von Sender und Empfänger beeinhalten soll. I_{A1} ist die Mitteilungskennung, welche von A zufällig generiert wurde. F_{A1} ist eine Prüfsumme über die gesamte Mitteilung. Die beiden letzten Felder repräsentieren die Anforderung an den Cookie Jar: er soll eine Verbindung zwischen A und B authentisieren.

Der Cookie Jar durchsucht seine interne Liste L_A. Falls eine Verbindung zwischen A und B dort nicht erlaubt ist, weist er die Anforderung zurück. Andernfalls sendet er folgende Antwort:

$$S \rightarrow A: \quad S, A, K_A(I_{A1}), K_A(F_S), K_A(K_C), K_A(K_B(K_C))$$

Dabei sind die beiden ersten Felder wieder der Mitteilungskopf. In Feld 3 schickt der Cookie Jar die ID aus dem Anforderungspaket zurück, jedoch verschlüsselt durch den privaten Schlüssel K_A von A. So kann A sicher sein, daß die Antwort auch wirklich vom Cookie Jar stammt, da nur er den entsprechenden Schlüssel kennt. Auch alle weiteren Felder sind mit K_A verschlüsselt. Durch die Codierung der Prüfsumme F_S mit K_A ist sichergestellt, daß niemand Teile der Mitteilung ausgetauscht oder modifiziert hat. K_C ist ein vom Cookie Jar zufällig generierter Session Key. Dieser Schlüssel wird für die nachfolgende Sitzung zur Authentifizierung der einzelnen Nachrichtenübermittlungen zwischen A und B verwendet. Zum Verbindungsaufbau sendet A nun folgendes Paket an B:

$$A \rightarrow B: \quad A, B, K_C(I_{A2}), K_C(F_{A2}), K_B(K_C)$$

I_A ist wiederum eine zufällig gewählte Mitteilungskennung, welche als Startwert für die Numerierung der nachfolgenden Sendungen von A verwendet werden soll. Das fünfte Feld der Mitteilung enthält den Session Key, verschlüsselt mit dem privaten Schlüssel von B. Für B ist an dieser Stelle klar, daß die Verbindung vom Cookie Jar authorisiert wurde, denn nur er wäre in der Lage, K_C mit K_B zu verschlüsseln.

Rechner B antwortet folgendermaßen:

B→A: B, A, $K_C(I_B)$, $K_C(F_B)$

Dabei ist I_B eine zufällig gewählte Mitteilungskennung, welche B als Startwert für die Numerierung seiner nachfolgenden Sendungen verwenden wird.

Alle nachfolgenden Pakete sind dann analog aufgebaut. Die Sequenznummer I_{A2} bzw. I_B wird jeweils inkrementiert. Mitteilungen mit ungültigen Sequenznummern werden verworfen.

3.6.2. RFC-1040: Privacy Enhancement for Internet Electronic Mail

Bereits im Januar 1988 stellte J. Linn im RFC-1040 [10] Konzepte und Implementierungsvoschläge für die Verschlüsselung von elektronischer Post auf dem Internet vor. Mittlerweile ist mit dem RFC-1113 [11] eine überarbeitete Version erschienen, welche ergänzt wird durch Ausführungen in RFC-1114 [8] und RFC-1115 [12].

Im Internet wird elektronische Post u. U. durch eine Vielzahl von Rechnern weitergereicht, bis sie schließlich ihren Empfänger erreicht. Dabei ist – standardmäßig – weder gewährleistet, daß die Nachricht nicht von Unbefugten gelesen werden kann, noch, daß sie nicht unauthorisiert verändert werden kann. Auch hat der Empfänger keinerlei Möglichkeit festzustellen, ob die Nachricht verändert wurde oder ob sie überhaupt vom angeblichen Absender stammt. Diese Probleme versucht Linn mit seinen Konzepten zu lösen.

Das Paket von Linn ist kompatibel zum bestehenden Mailing-System. Die Sicherung erfolgt end-to-end, die Implementierung geschieht lediglich auf dem Application Layer (vgl. Kap. 3.2.1), die zwischengeschalteten Übertragungswege bleiben unbeeinflußt.

Die Mechanismen unterstützen prinzipiell eine breite Zahl von Schlüsselverteilungsmethoden, empfohlen wird aber das im zweiten Teil [8] dargestellte Public-Key-Verfahren.

Kent betrachtet nur gewisse Aspekte der Absicherung, insbesondere will er den Benutzer nicht einschränken, sondern unterstützen. Betrachtet werden

- *Schutz vor unberechtigem Zugriff*
- *Authentifikation des Absenders*
- *Integrität der Nachricht*
- *Non-Repudiation* (bei Verwendung eines Public-Key-Verfahrens)

Alle anderen Punkte (vgl. Kap. 3.2.1) bleiben unbetrachtet.

Der Absender einer Nachricht entscheidet, ob auf einer Nachricht Sicherungsverfahren angewendet werden sollen oder nicht. Nachrichten an Empfänger ohne entsprechende Ausstattung dürfen nicht gesichert werden. Bei einer Implementierung wird man dies z. B. durch eine automatische Anfrage an einen Server realisieren, um so den Benutzer des Systems möglichst wenig zu belasten.

Zur Nachrichtenverschlüsselung wird eine zweistufige Schlüsselhierarchie eingesetzt:

- *Datenschlüssel*
  Sie werden zur Verschlüsselung der eigentlichen Daten verwendet. Datenschlüssel werden für jede Nachricht neu generiert.
- *Austauschschlüssel*
  Diese werden zur Codierung der Datenschlüssel und der evtl. vorhandenen Prüfsummen eingesetzt. Jeder Nachricht wird eine so verschlüsselte Version des jeweils verwendeten Datenschlüssels vorangestellt. Bei dem Austauschschlüssel ist zu Unterscheiden zwischen

   - *symmetrischer Verschlüsselung*
     In diesem Fall teilen sich Absender und Empfänger den Austauschschlüssel.
   - *asymmetrischer Verschlüsselung*
     Hier wird zur Codierung der Datenschlüssel der öffentliche Schlüssel des Empfängers verwendet, zur Codierung der Prüfsumme jedoch der private Schlüssel des Absenders.

Zweck der Verwendung unterschiedlicher Schlüssel ist deren Schutz: da mit den Austauschschlüsseln nur sehr kleine Datenmengen verschlüsselt werden, ist die Gefahr einer kryptoanlytischen Entschlüsselung nur sehr gering.

Der Ablauf einer geschützen Nachrichtensendung sieht wie folgt aus:

- Ein Datenschlüssel wird generiert (es sei denn, die Nachricht soll nicht verschlüssel werden); zusätzlich wird eine Variante des Datenschlüssels erzeugt, die für bei der Berechnung einer verwendet wird
- Der Kopf der Nachricht wird um ein Feld *'X-Sender-ID:'* ergänzt, welches dem Empfänger einen Hinweis darauf gibt, welcher Austauschschlüssel verwendet wurde. Ein weiteres Feld *'X-Key-Info:'* beeinhaltet eine mit dem Austausschlüssel codierte Version des verwendeten Datenschlüssels. Bei Verwendung von symmetrischer Verschlüsselung beinhaltet dieses Feld auch die verschlüsselte berechnete Prüfsumme. Andernfalls ist eine mit dem privaten Schlüssel des Absenders codierte Version der Prüfsumme in einem Feld *'X-MIC-Info:'* enthalten. Zusätzliche Felder sind vorgesehen für die Übermittlung von Authentifikatoren. Alle Schlüsselfelder sind versehen mit einer Kennung, die dem Empfänger mitteilt, welcher Verschlüsselungsalgorithmus verwendet wurde.
- Eine vierphasige Transformation auf dem Nachrichtentext sorgt dafür, daß der verschlüsselte Text in einer Form vorliegt, der zum einen über die gewöhnlichen Mailing-Services verschickt werden kann und zum anderen auch von jedem beliebigen Rechnertyp wieder entschlüsselt werden kann. Die Schritte umfassen
   - Konvertierung vom lokalen in ein kanonisches Format
   - Auffüllung des Textes auf eine für die Verschlüsselung geeignete Blockgröße
   - Verschlüsselung
   - Transformation des verschlüsselten Textes in ein von allen Mailern akzeptiertes Alphabet

Zur Verschlüsselung wird standardmäßig der DES (vgl. Kap. 3.2.3) verwendet; bei der Datenverschlüsselung wird der CBC-Modus angewendet, zur Schlüsselcodierung der ECB-Modus. Da allen Schlüsselfeldern ein Identifikationsstring mitgegeben wird, der die verwendete Codierung benennt, z. B. "DES-CBC", ist die Wahl der Verschlüsselung prinzipiell flexibel; natürlich muß darauf geachtet werden, daß dem Empfänger die entsprechenden Algorithmen zur Verfügung stehen.

Zur Berechnung von Prüfsummen empfiehlt der Autor die RSA-Methode (vgl. Kap. 3.2.3). Zusätzlich wird auch ein MAC (vgl. Kap. 3.2.3) unterstützt, jedoch sollte er nach Empfehlung des Autors höchstens für die Berechnung von Integritätschecks, nach Möglichkeit nicht für die Berechnung von Authentifikatoren benutzt werden, da er dazu nicht sicher genug sei.

3.7. Zusammenfassung und Vergleich

In den verschiedenen Abschnitten des dritten Kapitels wurden ganz unterschiedliche Aspekte der Aufgabenstellung angesprochen. Allen gemeinsam ging es um eine Datenübertragung über ein Netz und den damit verbundenen Problemen.

Hier sollen die unterschiedlichen Lösungsansätze knapp zusammengefaßt und – soweit möglich – miteinander verglichen werden. Im Anschluß an die einzelnen Zusammenfassungen wird dann noch geprüft, inwieweit die von den verschiedenen Autoren vorgestellten Lösungswege auch bei der Lösung der hier vorliegenden Aufgabe geeignet sind, soweits dies hier bereits beurteilt werden kann.

Needham und Schröder (Kap. 3.1) betrachten die Authentifizierung von interaktiven und unidirektionalen Netzverbindungen, sowie die Authentifizierung und Integritätsüberprüfung von übertragenen Daten mittels digitaler Unterschrift. Die Konzepte sind in sich sehr allgemein gehalten. Klar ist zwar, daß zu allen Authentifizierungen Verschlüsselungen nötig sind, aber Needham und Schröder lassen alle Varianten von Verschlüsselungsalgorithmen zu.

Für die vorliegende Aufgabe scheinen die Konzepte weitgehend geeignet zu sein. Der Fall der unidirektionalen Verbindung ist nicht so interessant, da die Verbindungen im geplanten Verwaltungssystem interaktiv sind. Die digitale Unterschrift kann auch bei den vorliegenden Verbindungen zur Integritätsprüfung der übertragenen Daten verwendet werden.

Das Kapitel **Sicherheitsservices und Sicherheitsmechanismen** (Kap. 3.2) gab einen Überblick der in der Sicherheitsarchitektur zum ISO-OSI-Referenzmodell definierten Sicherheitsservices und der verschiedenen Methoden diese Sicherheitsmerkmale mittels verschiedener Sicherheitsmechanismen zu realisieren. Desweiteren wurde auf verschiedene Verschlüsselungsmethoden und die Relevanz einer geeigneten Schlüsselhierarchie eingegangen.

Von den betrachteten Sicherheitsservices scheinen *Vertraulichkeit von Daten*, *Authentifikation* und *Datenintegrität* für die Realisierung der vorliegenden Aufgabe wichtig, um die einzelnen in der Aufgabenstellung genannten Sicherheitsanforderungen (Kap. 2.2) erfüllen zu können. Der Service *Non-Repudiation* in seinen verschiedenen Ausprägungen wird nicht benötigt, er geht deutlich über die gestellten Anforderungen hinaus. Eine *Zugriffskontrolle* ist bei der vorliegenden Aufgabe implizit gegeben: der Verwaltungsservice steht jedem Benutzer zur Verfügung, aber nur genau in dem Rahmen, wie er durch die Systemkonfiguration gegeben wird. Dies gilt insbesondere dann, wenn das Verwaltungssystem nur Anfragen von Rechnern akzeptiert, die ihm in seiner Konfiguration bekannt gemacht wurden.

Zur Beurteilung der Eignung der verschiedenen *Verschlüsselungsalgorithmen* ist es nötig zu wissen, zu welchem Zweck und in welchem Umfang Verschlüsselungen durchgeführt werden müssen und welche Voraussetzungen für die Implementierung gegeben sind. Daher kann diesbezüglich erst bei der Implementierung eine Entscheidung getroffen werden.

Relevant ist auch die Existenz eines geeigneten *Key Managements*. Wie dies im Rahmen der vorliegenden Aufgabe geschehen soll, muß an geeigneter Stelle entschieden werden.

Die **Yellow Pages** (Kap. 3.3) betreffen einen anderen Aspekt dieser Arbeit: globale Verwaltung von Daten. Auf einem Rechner existiert ein für sein Domain zuständiger Master-Server, der global gehaltene Daten über ggf. zwischengeschaltete Slave-Server an alle Rechner des Domains verteilt.

Unter den Yellow Pages existiert auf den Client-Rechnern keine Kopie der Daten, bei jedem Zugriff wird mittels entsprechender Funktionen der aktuelle Datenbestand bei einem Server nachgefragt. Die Verwendung von Yellow Pages ist für die Lösung der vorliegenden Aufgabe ausgeschlossen:

- Die Verwendung der Yellow Pages kann nicht nachträglich in bestehende Software eingebaut werden (ohne Neuübersetzung, die normalerweise für Systemsoftware oder kommerzielle Software nicht vom Anwender durchführbar ist).
- Yellow Pages sind nicht auf allen Rechnern verfügbar.
- Damit die Yellow Pages den geforderten Sicherheitsansprüchen genügen, muß das Secure RPC verwendet werden. Dieses ist ebenfalls nicht auf allen Rechnern verfügbar und müßte erst portiert werden. Eine nachträgliche Umstellung eines bestehenden YP-Systems auf Secure RPC ist für den gewöhnlichen Anwender nicht möglich.
- Die feste Zuteilung von Rechnern zu einem bestimmten Domain ist für die vorliegende Aufgabe nicht geeignet.
- Unter den Yellow Pages ist eine Änderung der verwalteten Dateien nur auf dem Master-Server möglich. Eine solche Einschränkung ist inakzeptabel, da die lokal laufenden Systemprogramme normalerweise in der Lage sein müssen auch Änderungen an den entsprechenden Systemdateien durchzuführen.

Das **RPC**-System (Kap. 3.4) stellt eine Funktionssammlung zur einfachen Realisierung von Netzkommunikation zur Verfügung. In der neuesten Version besteht die Option für den Einsatz einer Authentifizierung der kommunizierenden Parteien.

Die Verwendung von RPC bzw. Secure RPC würde die Implementierung von sicheren Netzzugriffen stark vereinfachen. Problematisch dabei ist jedoch die Verfügbarkeit des Programmpaketes. Zwar hat Sun Microsystems sein RPC-Paket bereits auf sehr viele Rechner portiert und gibt auch die Quellen heraus, so daß viele Hersteller selbst Portierungen durchgeführt haben; andererseits kann dennoch die Existenz dieses Systems nicht vorausgesetzt werden und man müßte auf verschiedenen Maschinen das RPC-System zunächst portieren. Dies kann sich schwierig gestalten, da das RPC vergleichsweise hohe Anforderungen† an die Umgebung stellt, die keinesfalls von allen UNIX-Rechnern erfüllt werden. Gleichzeitig bringt die Verwendung des RPC-Systems einen großen Overhead mit sich. Daher muß von dieser komfortablen Lösung Abstand genommen werden.

Kerberos (Kap. 3.5) bietet ein spezielles Authentifizierungs- und Integritätsprüfungskonzept für die Netzwerkkommunikation. Dies Konzept ist umfassender als der Umfang des RPC, es besteht nicht nur die Möglichkeit jede Übertragung zu authentifizieren, sondern darüberhinaus wird auch nur eine Authentifizierung beim Verbindungsaufbau oder aber auch eine automatische Verschlüsselung der Übertragung angeboten. Kerberos stützt sich auf die Authentifizierungsprotokolle nach Needham und Schröder. Es wird ein Public-Key-System implementiert. Der Kerberos-Server verwaltet die öffentlichen Schlüssel aller Teilnehmer und erzeugt – unterstützt von einem weiteren Service – Service-Tickets, mittels derer sich ein Benutzer gegenüber den verschiedenen im Netz zur Verfügung stehenden Dienstleistungsservices ausweisen kann.

Das Kerberos-System ist für den Einsatz bei der vorliegenden Aufgabe zu allgemein; eine Authentifizierung kann effektiver erreicht werden, wenn – ebenfalls basierend auf den Protokollen nach Needham und Schröder – ein Kommunikationsprotokoll implementiert wird, welches genau auf die vorliegende Problemstellung zugeschnitten ist. Dennoch kann das Kerberos-System als Vorbild bei der Implementierung dienen. Es zeigt besonders deutlich, welche Möglichkeiten die Authentifizierungsprotokolle nach Needham und Schröder bieten und wie sie modifiziert werden können um so eine Anpassung an anwendungsspezifische Gegebenheiten zu erreichen.

---

† Das Secure RCP setzt z. B. die Existenz der *mp*-Bibliothek voraus, um ein Public-Key-Verfahren zu implementieren. In dieser Bibliothek sind arithmetische Integer-Operationen für Ganzzahlen beliebiger Genauigkeit enthalten.

Im **RFC-1004** (Kap. 3.6.1) wird eine weitere Anwendung der Authentifizierungsprotokolle nach Needham und Schröder vorgestellt. Im Gegensatz zu Kerberos ist diese Anwendung recht klein gehalten. Des weiteren wird hier jede Authentifizierung vom Authentifizierungsserver selbst durchgeführt; eine Ticket-Vergabe wie unter Kerberos existiert nicht. Unter Kerberos liegt die Entscheidung für die Berechtigung eines Benutzers ganz allein beim Service. Im RFC-1004 übernimmt einen Teil dieser Entscheidung bereits der Server, indem er nur für registrierte Verbindungspaare Authentifizierungen durchführt. Die Verschlüsselung der übertragenen Daten wird nicht betrachtet.

Für die Anwendung des RFC-1004 im Rahmen dieser Arbeit gelten im wesentlichen die zu Kerberos gemachten Anmerkungen: interessant als Anwendungsbeispiel, aber zu speziell um direkt weiterverwendet zu werden.

Im **RFC-1040** bzw. im Nachfolgeartikel RFC-1113 (Kap. 3.6.2) betrachtet der Autor die Absicherung von elektronischer Post. Dabei wird sowohl die Authentifizierung als auch die Integritätsprüfung berücksichtigt. Im Gegensatz zu den obigen Authentifizierungsmechanismen setzt der RFC-1040 nicht die Existenz eines Authentifizierungsservers voraus, unterstützt sie aber prinzipiell. Der Artikel beschäftigt sich ausführlich mit verschiedenen Methoden zur Integritätsprüfung und der Garantie von Vertraulichkeit.

Eigentlich scheint diese Lösung auf den ersten Blick besonders interessant, da er Alternativen zum Einsatz eines Authentifizierungsservers zeigt. Die vorgestellten Alternativmöglichkeiten sind jedoch nicht sehr ergiebig, sie reichen kaum über eine manuelle Verteilung von Schlüsseln hinaus. Interessant dagegen sind die unterschiedlichen vorgestellten Methoden zur Absicherung der Vertraulichkeit und Integrität von Übertragungen.

4. Ein Modell

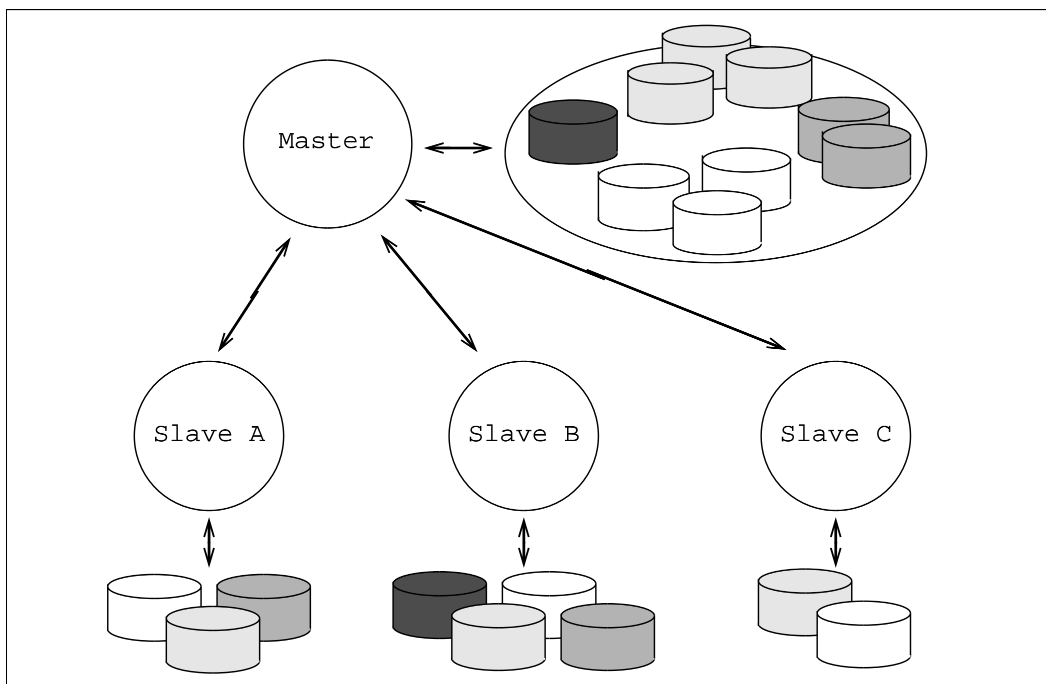
Ausgehend von den in Kapitel 3 vorgestellten existierenden Lösungsansätzen soll nun ein eigenes Modell entworfen werden. Dieses Modell muß zum einen die in der vorgegebenen Aufgabenstellung geforderten Eigenschaften besitzen, soll zum anderen aber nicht zu weit darüber hinaus gehen, um weder die spätere Implementierung zu erschweren, noch die Performance eines auf diesem Modell basierenden Programmsystems zu gefährden.

4.1. Grobstruktur

Ziel ist die Schaffung eines Systems, welches einem Administrator die Möglichkeit bietet, von zentraler Stelle aus Verwaltungsaufgaben auf verschiedenen Rechnern zu erledigen, ohne dabei gezwungen zu sein, sich auf den verschiedenen Maschinen einzuloggen und sich an die jeweils unterschiedlichen Arbeitsumgebungen anzupassen. Dies ist nur unter Inanspruchnahme eines Netzwerkes möglich, welches die zu verwaltenden Rechner vernetzt. Die Kommunikation der Rechner innerhalb dieses Verwaltungssystems muß geeignet abgesichert werden, um auszuschließen, daß die Verwaltungsmechanismen durch unberechtigte Dritte mißbraucht werden.

Die Durchführung der o. g. administrativen Tätigkeiten besteht i. a. in der Wartung von Konfigurationsdateien, so daß sich unser Modell reduzieren läßt auf die zentrale Verwaltung von Systemdateien über ein Netzwerk.

Eine solche zentrale Verwaltung legt es nahe, einen bestimmten Rechner als Zentralrechner auszuzeichnen, von dem aus die Administration der anderen, untergeordneten Rechner durchgeführt wird. In den Yellow Pages (Kap. 3.3), die sich ebenfalls mit der zentralen Verwaltung von Daten auseinandersetzen, gibt es einen Master-Server, der den globalen Datenbestand bereithält und auf Anfrage den Client-Rechnern übermittelt. Diese Lösung erlaubt zwar die globale Datenverwaltung, entzieht aber auf der anderen Seite den untergeordneten Rechnern die Möglichkeit der Modifikation der Systemdateien vollständig. Außerdem stellt eine solche Lösung einen tiefen Einschnitt in die Systemstruktur der Client-Rechner dar, denn diese besitzen dann *keine* lokalen Kopien der Systemdateien mehr, so daß jegliche Software, die sich auf die betreffenden, zentral verwalteten Daten abstützt, entsprechend modifiziert werden muß.



In unserem Modell (vgl. Bild oben) behalten daher die untergeordneten Rechner *'Slaves'* ihre lokalen Kopien der verwalteten Dateien. Es gibt einen ausgezeichneten Zentralrechner *'Master'*, der globale Kopien der Systemdateien besitzt. Diese zentrale Datenbank kann der Systemadministrator modifizieren. Die vorgenommenen Änderungen müssen dann aber *sofort* an alle betroffenen Slaves übertragen werden – im Gegensatz zu YP, wo die Clients die Daten bei Bedarf vom Master anfordern – weil die Slave-Rechner völlig unabhängig vom Master auf ihren lokalen Kopien arbeiten und andernfalls die globale Änderung nicht wahrnehmen würden.

Die Existenz von lokalen Kopien senkt gleichzeitig den Netzwerkverkehr, da lesende Zugriffe völlig unabhängig vom Master durchgeführt werden können. Auch bei schreibenden Zugriffen sind die Slaves nicht von der Reaktionsgeschwindigkeit des Masters abhängig. Es kann daher angenommen werden, daß die Kapazität des Masters zur Bewältigung des Datenvolumens ausreicht und auf den Einsatz von Hilfsservern (analog den Slave-Servern unter YP) verzichtet werden kann.

Für die Modifikation der globalen Kopie auf dem Master-Rechner kann ein spezielles Editor-Programm konzipiert werden, welches eine komfortable Datenänderung erlaubt.

Auch die lokale Modifikation der global verwalteten Systemdateien soll in unserem Modell möglich sein, denn nur so kann die volle Funktionalität der verschiedenen Systemprogramme garantiert werden. Dies bedeutet aber, daß die Slaves lokal an den Systemdateien vorgenommene Änderungen dem Master übertragen müssen, damit keine Inkonsistenzen im Datenbestand auftreten. Somit wird also der unidirektionale Datenfluß vom Master zu den Clients, wie er im YP-Modell gegeben ist, hier ersetzt durch bidirektionale Verbindungen.

Auf jedem der betroffenen Rechner muß ein Serverprogramm existieren – ein Masterserver oder ein Slaveserver – welches die geforderten Transfers einleitet bzw. entgegennimmt.

Unter Verwendung der Yellow Pages ist der Datenbestand in einer Systemdatei im ganzen Domain (d. h. in der Rechnergruppe) identisch. Dies erscheint unpraktisch; in vielen Systemdateien möchte man – neben den global einheitlichen Daten – auch ausschließlich lokal gültige Einträge haben. Der Master kann daher in unserem Modell zwar auch alle Daten einer Systemdatei von verschiedenen Rechnern zusammenfassen, muß sich aber für jeden Eintrag den Gültigkeitsbereich merken.

Dieses Konzept der Gültigkeitsbereiche läßt sich noch erweitern: man kann auch Rechnergruppen definieren, innerhalb derer eine bestimmte Systemdatei immer den gleichen Inhalt haben soll. Der Master muß dann nur alle Änderungen dieser Systemdatei, die entweder global durchgeführt oder aber lokal auf einem der Rechner der Gruppe vorgenommen wurden, an alle anderen Rechner der Gruppe weiterverteilen. Eine solche Gruppierungsmethodik ist wesentlich flexibler als das Domainkonzept in den Yellow Pages.

Master- und Slaveserver benötigen eine Konfiguration, in der alle für den Betrieb relevanten Informationen abgelegt werden können. Für den Masterserver sind dies z. B. die Namen der zu verwaltenden Slaves und deren Gruppierung, sowie die Namen der verschiedenen Systemdateien. In den Slaveserverkonfigurationen muß angegeben werden, welche Systemdateien global verwaltet werden sollen, da dies von Slave zu Slave variieren kann. Da auf den unterschiedlichen Slaves der lokale Aufbau einer Systemdatei von dem auf anderen Maschinen abweichen kann, muß dem Slaveserver auch das Format der lokalen Datei mitgeteilt werden, damit es an ein vom Masterserver definiertes globales Format angepaßt werden kann.

Da deutlich wird, daß in den Konfigurationen viele Angaben nötig sind, bietet es sich an jeweils eine Sprache zu definieren, die die entsprechenden Konstrukte zur Verfügung stellt. Die Syntax dieser Sprachen (eine für die Master-, eine für die Slavekonfiguration) wird bei der Implementierung in Kap. 5.2.2 festgelegt.

4.2. Anforderungen an das Kommunikationsprotokoll

Bei der bisherigen Modellierung blieben verschiedene Aspekte noch völlig unberücksichtigt, und zwar insbesondere der Ablauf der Kommunikation zwischen Master und Slaves sowie die geeignete Absicherung derselben. Dies geschieht nachfolgend durch Entwicklung eines geeigneten Kommunikationsprotokolls.

Es soll ein Kommunikationsprotokoll modelliert werden. Da dieses Protokoll zur Absicherung des Systems dienen soll, muß es u. a. die in Kap. 2.2 gestellten Sicherheitsanforderungen erfüllen, indem es die in Kap. 3.1 und 3.2 aufgezeigten Sicherheitsrisiken abdeckt. Ferner muß es sich in die in Kap. 4.1 bereits vordefinierte Umgebung einfügen:

- *Umgebung*
  Weiter oben wurde festgelegt, daß es einen ausgezeichneten, zentralen Masterrechner gibt und eine Vielzahl von untergeordneten Slaverechnern. Auf jedem der beteiligten Rechner übernimmt ein Serverprogramm die Datenübermittlung. In allen in Kap. 3 vorgestellten Authentifizierungsmodellen wird davon ausgegangen, daß es mindestens einen 'sicheren', d. h. vertrauenswürdigen Rechner gibt. Dies ist bei Needham und Schröder (Kap. 3.1) der Authentifizierungsserver, unter Kerberos (Kap. 3.5) der Kerberos-Server und der TGS. Dies scheint auch sinnvoll, andernfalls ist eine Absicherung der globalen Daten nahezu unmöglich. Es wird also angenommen, daß ein Masterrechner, der als sicher gilt, mit einer Vielzahl von ungesicherten Slave-Rechnern über ein ungesichertes Netz kommuniziert.
- *Authentifizierungsmechanismus*
  Da das System 'sicher' sein soll, ist die gegenseitige Authentifikaton von Master und Slave wesentlich. Der Master hat mittels der Slaves – soweit deren lokale Konfiguration es erlaubt – Kontrolle über die Systemdateien der Slave-Rechner. Der Master ist (per def.) auf einem sicheren Rechner installiert, daher ist eine Manipulation am Master selbst nicht zu befürchten. Andererseits könnten sich andere Rechner gegenüber einem Slave als Master ausgeben und so unberechtigt Datenänderungen auf dem Slave initiieren. Oder ein Rechner könnte sich beim Master als ein beliebiger, dem Master bekannten Slave ausgeben und dann dem Master irgendwelche Datenänderungen mitteilen, die dieser seinerseits unwissentlich an andere Slaves weiterverteilen würde.
  Es ist also nötig, den Nachrichten Authentifikatoren mitzugeben. Dazu könnte man z. B. Protokoll 1 oder 2 nach Needham und Schröder implementieren (Kap. 3.1.3). Diese Protokolle sehen die Existenz eines Authentifizierungsservers vor. Da in der vorliegenden Anwendung bereits ein sicherer Rechner (nämlich der Master) existiert, ist er für diese Aufgabe eigentlich prädestiniert. Andererseits sieht die bisherige Planung keine Kommunikation zwischen den Slaves vor, d. h. eine gegenseitige Authentifizierung zweier Slaves ist nicht nötig. Somit ist eine Vereinfachung der Protokolle möglich.
- *Integritätsprüfung*
  Bei der Datenübertragung zwischen Master und Slaves muß unbedingt die Datenintegrität sichergestellt werden. Der Empfänger einer Nachricht muß sich vergewissern können, daß die erhaltenen Daten genau so vom Absender übertragen wurden. Dies umfaßt sowohl inhaltliche wie auch zeitliche Integrität. Dabei müssen Übertragungsfehler und Manipulationsversuche ausgeschlossen werden. Möglich ist die Verwendung einer digitalen Unterschrift nach Needham und Schröder (Kap. 3.1.6), angepaßt an das noch zu definierende Verschlüsselungs- und Key-Management-Verfahren.
- *Vertraulichkeit*
  Das Protokoll soll eine – im Rahmen der gegebenen Möglichkeiten – optionale Verschlüsselung zur Verfügung stellen um vertrauliche Daten bei der Übertragung schützen zu können.

- *Paket- und Quittungsaustausch*
  Das Protokoll muß festlegen, welches Datenformat *('Paketaufbau')* die übermittelten Nachrichten haben, welche Partei zu welchem Zeitpunkt eine Kommunikation initiieren und welcher Art diese Kommunikation sein darf. Außerdem muß geklärt werden, mit welchen Quittungen auf erhaltene Daten reagiert wird.

  Sowohl Master als auch Slaves müssen regelmäßig die globalen bzw. lokalen Systemdateien auf lokal vorgenommene Änderungen überprüfen und ggf. einen Transfer einleiten. Die Gegenseite muß die Daten entgegennehmen und verarbeiten. Dieser Teil ist auf dem Master vielschichtig, da der Master je nach Konfiguration unterschiedlich auf die erhaltenen Daten reagieren muß.

4.3. Datenmodell

In der netzglobalen Datenbank auf dem Masterrechner sollen für jede Systemdatei die Anteile des Datenbestandes, die auf verschiedenen Rechnern gleich sind, zusammengefaßt werden. So ergibt sich für den Systemadministrator, der den globalen Datenbestand bearbeitet, eine Arbeitserleichterung und das Datenvolumen auf dem Master kann verringert werden.

Wenn Slaverechner ihren Datenbestand dem Master übermitteln, ist es sinnvoll, daß der Master erkennen kann, ob es sich um Daten*änderungen*, Daten*neuaufnahmen* oder Daten*löschungen* handelt. Nur so können kompliziertere Transfermodi realisiert werden (vgl. unten).

Um dies zu erkennen, müssen die Dateien in Datensätze aufgegliedert werden können, die anhand eines eindeutigen Schlüssels identifizierbar sind. Das bedeutet, daß die Datensätze in Felder aufgespalten werden müssen, von denen eines als Schlüsselfeld ausgezeichnet wird (oder mehrere im Falle eines zusammengesetzten Schlüssels).

In den Konfigurationssprachen müssen für die Aufteilung der Datensätze in Felder und für die Markierung der Schlüsselattribute geeignete Konstrukte zur Verfügung gestellt werden.

4.4. Transfermodell

Wenn ein Slave S eine Änderung an einer Systemdatei F feststellt, überträgt er die gesamte Datei an den Master M. S kann lediglich anhand des Modifikationsdatums von F feststellen, *daß* die Datei geändert wurde. *Welche* Änderungen das sind, kann er jedoch nicht erkennen.

Der Master muß im Vergleich mit der global gehaltenen Kopie G und den in seiner Konfiguration angegebenen Transferanweisungen überprüfen, welche Änderungen durchgeführt wurden und ob diese Änderungen zulässig waren.

Transferanweisungen sollen angeben, von welchem Rechner der Master Daten annimmt, unter welchen Bedingungen Daten angenommen werden und ob diese Daten an andere Rechner weitergegeben werden sollen. Transferanweisungen haben also prinzipiell die Form

$$F:\ Q \rightarrow_{X} Z$$

d. h. die Systemdatei F wird von einem Rechner aus der Rechnergruppe Q unter der Bedingung X angenommen und an alle Rechner der Gruppe Z weitergegeben. Dies Konstrukt muß in der Konfigurationssprache des Masters geeignet repräsentiert werden.

Genügen alle Änderungen bei einer Übertragung der Bedingung X, so werden sie in die globale Kopie G übernommen und ggf. zur Weitergabe an die Slaves Z vorgemerkt.

Andernfalls werden die unzulässigen Änderungen verworfen und nur die gültigen Modifikationen in die Globaldatei G übernommen. F wird auch für die Übertragung an den Absender $S \in Q$ vorgemerkt, um in der lokalen Kopie von F auf S die ungültigen Änderungen rückgängig zu machen.

Für die Bedingung X sind folgende Änderungsmodi denkbar:

- *Änderungen erlaubt*
  Dem Slave ist jede Art der Änderung erlaubt. Geänderte Datensätze werden übernommen.
- *keine Änderungen erlaubt – 'reject'*
  Dem Slave sind keine Modifikationen der Daten erlaubt. Jede Änderung wird zurückgewiesen.
- *einmalig Änderungen erlaubt – 'once'*
  Bei der ersten Übertragung der Datei durch einen Slave werden alle Daten akzeptiert. Bei jeder weiteren Übertragung werden alle Änderungen zurückgewiesen. Dies ist sinnvoll, wenn man dem Slave eigentlich keine Datenmodifikation erlauben will, beim Start des Programmsystems aber einmal der Datenbestand von allen Slaves gesammelt werden soll.
- *nur Hinzufügen – 'nochange'*
  Der Slave darf nur neue Datensätze hinzufügen. Bereits bestehende Datensätze dürfen nicht geändert werden.

In den Bedingungen X kann auch spezifiziert werden, *wie* die neuen Daten in die Globaldatei übernommen werden sollen:

- *Anfügen – 'append'*
  Die vom Slave gelieferten Datensätze werden an die Globaldatei angefügt. Falls neue Datensätze mit gleichem Schlüsselfeld wie bereits vorhandene Sätze übertragen werden, so ersetzen die neuen Daten die alten.
- *Ersetzen – 'replace'*
  Die übertragenen Daten ersetzen die vorhandenen komplett. Alle alten Datensätze, deren Schlüssel nicht im neuen Datenbestand enthalten sind, gehen verloren.

Die Entscheidung, ob Datenänderungen durch einen Slave zulässig sind oder nicht, ist kompliziert, da aus den Transferdefinitionen nicht unmittelbar erkennbar ist, welche Auswirkungen auf den globalen Datenbestand ihre Anwendung hat. Um dies festzustellen, wird aus G eine neue Datei G' erzeugt, und zwar wie folgt:

- alle Sätze aus G mit gleichen Schlüsseln wie in F müssen für alle Zielrechner Z, welche in einer für F und S gültigen Transferdefinition auftreten, ungültig gemacht werden
- alle Sätze aus F werden mit Gültigkeit für alle Z an G angefügt, bzw. die Gültigkeit vorhandener Sätze in G mit gleichen Schlüsseln wie in F auf alle Z ausgedehnt

Die auf diese Weise entstandene neue Globaldatei G' wird mit G verglichen. Dabei sind folgende Ergebnisse möglich:

- *keine Änderungen sind aufgetreten*
  G' wird verworfen
- *es sind nur neue Sätze angefügt worden*
  Wenn eine Transferdefinition für F und S des Typs *'reject'* existiert, wird G' verworfen und ein Transfer von F nach S vorbereitet, um die Änderungen an F auf S rückgängig zu machen. Gleiches gilt für einen *'once'*-Transfer, bei dem es sich nicht um die erste Übertragung von F durch S handelt.
- *es sind Sätze geändert worden*
  Falls für F und S eine *'nochange'*-Transferdefinition existiert, werden alle *Änderungen* an Sätzen in G' (also nicht die Neuaufnahmen) wieder rückgängig gemacht, danach wird G' zu G gemacht. Falls eine *'reject'*-Transferdefinition für F und S existiert, wird G' ganz verworfen (analog für *'once'*). In beiden Fällen wird ein Transfer von F nach S eingeleitet, um die ungültigen Änderungen auf S rückgängig zu machen. Existiert keine Einschränkung in den Transferanweisungen, so wird G' zu G.

4.5. Paketaufbau und Sicherungsstrategie

Master- und Slaveserver müssen sich Nachrichten unterschiedlichen Inhaltes zusenden, z. B.

- Nachrichten, die eine (initiale) Authentifizierung garantieren
- Nachrichten, die den Inhalt einer Systemdatei beinhalten
- Quittungen, die mitteilen, daß eine Nachricht akzeptiert wurde oder warum eine Nachricht nicht akzeptiert wurde

Alle diese Nachrichten müssen eine geeignete Kennzeichnung besitzen, die es dem Empänger ermöglicht zu erkennen, welcher Art diese Nachricht ist.

Unter Berücksichtigung der Tatsache, daß das System auf einem paketorientierten Netzwerk (mit vorgegebener maximaler Paketlänge) verwendet werden soll, bietet es sich an, auch die versendeten Nachrichten in Pakete aufzuteilen. Die Absicherung der Kommunikation kann dann gleich auf der Paketebene erfolgen, indem jedem Paket entsprechende Kontrollfelder mitgegeben werden.

Insgesamt ergeben sich folgende Paketteile:

- *Absender*
  Es muß in der Nachricht ein Feld geben, anhand dessen der Absender der Nachricht erkannt werden kann (Name oder Adresse).
- *Authentifikator*
  Dieses Feld ermöglicht die Authentifizierung des Absenders und kann gleichzeitig zur Sequentialisierung eingesetzt werden.
  In vielen Modellen† wird zur Authentifizierung auf die Systemzeit zurückgegriffen. Unter UNIX ist ein Zugriff auf die Systemzeit mittels der Funktion `time` möglich. Sie liefert die Anzahl der seit dem 1. Januar 1970 vergangenen Sekunden. Wollte man diesen Wert jeweils für die eindeutige Markierung der gesendeten Pakete verwenden, so könnte pro Sekunde nur ein einziges Paket erzeugt werden. Dies ist nicht akzeptabel. Eine andere Lösung ist die Verwendung der Systemzeit nur beim Programmstart. In diesem ersten Paket kann dann willkürlich eine Sequenznummer festgelegt werden. Diese Nummer wird bei jeder Sendung inkrementiert, der Empfänger verwirft alle Pakete mit kleineren oder gleichen Sequenznummern. Damit ist gleichzeitig das Replay-Problem gelöst.
  Wenn Systemzeit bzw. Sequenznummer jeweils codiert übertragen werden, kann der Absender anhand der korrekten Dekodierung identifiziert werden.
- *Prüfsumme*
  Dieses Feld soll den Integritätstest garantieren. Mittels einer noch zu definierenden Funktion wird über das gesamte Paket eine Prüfsumme berechnet. Der so erhaltene Wert wird codiert und dem Paket hinzugefügt. Durch die Codierung der Prüfzahl sind unentdeckte Änderungen des Paketes durch Dritte ausgeschlossen.
- *Pakettyp*
  Der Pakettyp zeigt an, um welche Art von Paket es sich handelt. Hier kann auch vermerkt werden, ob der Datenblock des Paketes verschlüsselt wurde.
- *Paketlänge*
  Hier wird die tatsächliche Länge des Datenblocks angegeben. Dadurch ist es möglich, die Anzahl der Bytes beim Datentransfer zu verringern, da z. B. Quittungspakete nicht die volle maximale Paketlänge benötigen.

---

† so z. B. für unidirektionale Verbindungen mit konventionellen Algorithmen bei Needham und Schröder (Kap. 3.1.5.1), im Secure RPC (Kap. 3.4.4) und für Kerberos-Tickets (Kap,. 3.5.3.1)

- *Daten*
  Dieses Feld beinhaltet die eigentlichen Daten, deren Art durch den Pakettyp bestimmt wird.

4.6. Schlüsselhierarchie

Um eine adäquate Absicherung des Systems durchführen zu können, muß eine Schlüsselhierarchie modelliert werden (vgl. Kap. 3.2.4). Eine solche Hierachie sollte mehrstufig sein, um ein ausreichendes Maß an Sicherheit zu gewährleisten. Hier soll eine dreistufiges Modell verwendet werden. Die Zahl drei wurde exemplarisch gewählt, um ein gewisses Maß an Mehrstufigkeit zu erreichen und so bereits deren Vorteile nutzen zu können. Ob es sinnvoll ist, diese Hierarchie noch zu erweitern, soll später noch diskutiert werden.

Jedes Programm (Master- oder Slaveserver) besitzt einen 'eingebrannten' A-Schlüssel. Dieser Schlüssel ist nicht änderbar. Jeder Slave besitzt einen B-Schlüssel, der auch dem Master bekannt ist. Die B-Schlüssel werden bei der lokalen Speicherung durch Codierung mit dem programmeigenen A-Schlüssel geschützt. Vor dem ersten Programmstart müssen die B-Schlüssel manuell eingegeben werden, anders ist eine sichere Identifizierung nicht möglich. Bei jedem weiteren Programmstart ändert der Master nach der Kontaktaufnahme und Authentifizierung in Absprache mit dem Slave den gemeinsamen B-Schlüssel.

Der Mechanismus des eingebrannten A-Schlüssels und des durch diesen Schlüssel codierten B-Schlüssels, garantiert die in Kap. 2.2 geforderte Identifikation von Rechnern und Prozessen, sofern der B-Schlüssel bei der lokalen Speicherung gegen öffentlichen Zugriff geschützt werden.

Die Änderung des B-Schlüssels bei jedem Programmstart realisiert die in Kap. 2.2 geforderte Begrenzung der Authentifizierung.

Für jede Übertragung einer Datei muß eine Authentifizierung und eine Integritätssicherung vorgenommen werden. Zur Codierung von Authentifikator und Integritätsprüfsumme wird der B-Schlüssel verwendet.

Bei der Übertragung von Dateien mit vertraulichen Daten oder von Schlüsseln ist zusätzlich eine Verschlüsselung nötig. Um die B-Schlüssel nicht zu sehr zu gefährden, scheint es für die Codierung von Dateien sinnvoll, bei jeder Übertragung einen C-Schlüssel zu verwenden, der jeweils neu generiert wird.

Hiermit kann die in Kap. 2.2 geforderte Codierung sicherheitsrelevanter Daten gewährleistet werden. Die regelmäßige Änderung der C-Schlüssel unterstützt noch die geforderte Begrenzung von Authentifizierungen.

4.7. Kommunikationsablauf

Unter Verwendung des bereits definierten Paketaufbaus (Kap. 4.5) und der Schlüsselhierarchie (Kap. 4.6) wird nun entsprechend der Grobstruktur (Kap. 4.1) ein Kommunikationsablauf entworfen. Er regelt den Paket- und Quittungsaustausch zwischen Master und Slaves bei der initialen Authentifizierung, dem Vergleich der Dateiformate und der Übermittlung von Systemdateien.

Nach dem Programmstart sendet der Master der Reihe nach an alle seine Slaves, die ihm aus seiner Konfiguration bekannt sind, eine Initialisierungsaufforderung.

Das Initialisierungspaket ist vom Typ `DATA_INIT`, der Authentifikator ist die mit dem B-Schlüssel K des Slaves S codierte Uhrzeit T und das Datenfeld enthält eine ebenfalls codierte, willkürlich gewählte Sequenznummer I_1. I_1 soll vom Slave als Startwert für die Numerierung seiner Datenpakete an den Master verwendet werden.

M→S: K(*Check*, T), `DATA_INIT`, *Länge*, K(I_1)

Als Antwort auf `DATA_INIT` erwartet der Master ein Paket vom Typ `DATA_AUTH`.

S→M: K(*Check*, I_1+1), `DATA_AUTH`, *Länge*, K(I_2), K(K_n)

Dabei ist I_2 eine vom Slave willkürlich gewählte Sequenznummer, die vom Master zur Numerierung seiner Datenpakete an den Slave verwendet werden soll. K_n ist ein vom Slave willkürlich gewählter neuer B-Schlüssel, der von nun an verwendet werden soll. So wird erreicht, daß bei jedem Start des Masters neue B-Schlüssel aktiviert werden. In allen weiteren Paketen ist der Aufbau† wie folgt:

M→S: K_n(*Check*, I_2+*n*), *Typ*, *Länge*, *Daten*

bzw.

S→M: K_n(*Check*, I_1+*n*), *Typ*, *Länge*, *Daten*

Dabei in *n* die jeweils aktuelle Paketnummer.

Alle beim Master eintreffenden Antworten werden registriert und bearbeitet. Falls die Authentifizierung oder die Integritätsprüfung für ein empfangenes Paket fehlschlägt, wird das Paket verworfen und mit einem Paket `DATA_AUTHERR` oder `DATA_INTGERR` quittiert.

Für jeden Rechner werden Statusvariablen mitgeführt. In seiner Hauptprogrammschleife testet der Master regelmäßig die Statusvariablen aller Slaves sowie den Status der Globaldateien. Die gesamte Schleife wird von vier Zeitkonstanten 't_1', ..., 't_4' gesteuert, die in der Masterkonfiguration definiert werden.

Der Wert t_1 gibt eine Verzögerung in der Hauptschleife an und dient zur Senkung der durch den Master verursachten Systemlast. Es sollte nicht zu hoch gewählt werden, damit der Master bei hoher Belastung schnell genug reagiert.

Nach einer Zeitdauer von t_2 werden jeweils die Globaldateien auf Änderungen überprüft.

Aktionen werden immer nur dann eingeleitet, wenn die letzte Kommunikation mit dem jeweiligen Slave länger als t_3 zurückliegt. (Ausnahme: Status `HO_OK`). Ist diese Zeitdauer verstrichen, wird den Statuswerten entsprechend eine adäquate Aktion eingeleitet. Statuswerte können sein:

- `HO_OK`
  Bei diesem Slave war die letzte Übertragung erfolgreich. Ist seit der letzten Übertragung eine Zeit > t_4 verstrichen, so wird, falls nötig ein Dateitransfer eingeleitet. Andernfalls wird dem Slave, sofern die letzte Übertragung länger als t_3 zurückliegt, ein `DATA_PING` gesendet und der Status auf `HO_WAIT` gesetzt. Das Paket `DATA_PING` ist ein Statusrequest und soll überprüfen, ob der Slave noch *'online'* ist.
- `HO_WAIT`
  Von diesem Slave wird eigentlich eine Antwort auf `DATA_PING` erwartet. Da mittlerweile eine Zeit > $2*t_3$ seit dem letzten Empfang einer Nachricht verstrichen ist, so wird angenommen, daß der Slave *'down'* ist. Der Status wird auf `HO_DOWN` gesetzt.
- `HO_DOWN`
  Von diesem Slave wurde seit langer Zeit (> $3*t_3$) keine Antwort mehr erhalten. Ihm wird ein `DATA_INIT` als Initialisierungsaufforderung gesendet.
- `HO_UNKNOWN`
  Zu diesem Slave bestand noch keine Verbindung. Ihm wird ein `DATA_INIT` gesendet.
- `HO_REJECT`
  Bei diesem Slave sind schwerwiegende Fehler aufgetreten (z. B. wurde die initiale

† In allen weiter unten aufgeführten Paketbeschreibungen werden der Übersichtlichkeit halber nur der Pakettyp und das Datenfeld angegeben und K_n(*Check*, I_x+*n*) (= Prüfsumme und Authentifikator) sowie das Feld *Länge* weggelassen.

Authentifizierung zurückgewiesen). Er wird nicht mehr bedient.

- `HO_TRANS`
  Dieser Slave überträgt gerade eine Datei. Es werden keine Aktionen eingeleitet.
- `HO_ERROR`
  Bei diesem Slave sind Fehler aufgetreten. Er wird mit `DATA_INIT` zur Initialisierung aufgefordert.
- `HO_RECV`
  Dieser Slave bekommt vom Master eine Datei übertragen. Es werden keine Aktionen eingeleitet.
- `HO_AUTHERR`, `HO_INTGERR`
  Bei diesem Slave sind in wesentlichen Übermittlungsphasen (z. B. bei der initalen Authentifizierung) Authentifizierungs- bzw. Integritätsprüfungsfehler aufgetreten. Er wird nicht mehr bedient.

Wenn der Master Übertragungen an einen Slave vornimmt, verwendet er dazu das weiter unten beschriebene Protokoll.

Wird auf einem Slave ein Serverprogramm gestartet, so wartet der Slave auf eine Meldung des Masters. Von anderen Rechnern empfangene Meldungen werden verworfen und mit `DATA_REJECT` quittiert. Trifft ein Paket `DATA_INIT` vom Master ein, so wird dieses akzeptiert, sofern die darin enthaltene Zeit T höchstens um einen bestimmten Betrag Δt von der lokalen Zeit abweicht und außerdem die Prüfsumme korrekt ist. Nicht akzeptierte Pakete werden je nach Fehlerursache mit `DATA_AUTHERR` bzw. `DATA_INTGERR` quittiert. Bei Erhalt eines korrekten `DATA_INIT`-Paketes wird die oben beschriebene Authentifizierungssequenz eingeleitet.

Nach Abschluß der initialen Authentifizierung müssen sich Master und Slave über das Format der einzelnen Systemdateien abstimmen, da es Unterschiede zwischen dem globalen und dem lokalen Format geben kann (vgl. Kap. 4.1). Es wird angenommen, daß der Master dem Slave das globale Format mitteilt und der Slave – nach entsprechender Prüfung – entsprechende Konversionsroutinen bereitstellt. Alle Dateitransfers erfolgen dann im globalen Format. Der Paketaufbau für die Formatübermittlung wird bestimmt durch die Spezifikationsmöglichkeiten in den Konfigurationsdateien und kann daher erst in der Implementierungsphase (Kap. 5.2.4) festgelegt werden.

Nach dem Abgleich der Formate beginnt der Slave mit der Übertragung der Dateiinhalte. Die Dateiübertragung geschieht mit folgender Sequenz:

(1)	S→M:	`DATA_FSTART`, *Dateiname*
(2)	M→S:	`DATA_ACCEPT`
(3)	S→M:	`DATA_FCONT`, *Daten...*
(4)	M→S:	`DATA_ACCEPT`
		...
(5)	S→M:	`DATA_FEND`, *m*
(6)	M→S:	`DATA_ACCEPT`

Das obige Protokoll beschreibt eine fehlerfreie Übertragung. Im Schritt (2) antwortet der Master mit `DATA_FUNKN`, falls ihm der Dateiname unbekannt ist. Im Schritt (6) antwortet der Master mit `DATA_ERROR`, falls die Anzahl der erhaltenen Daten-Blöcke nicht mit der Anzahl *m* der vom Slave gesendeten Blöcke übereinstimmt.

Statt `DATA_FSTART`, `DATA_FCONT` und `DATA_FEND` können auch die Token `DATA_fSTART`, `DATA_fCONT` und `DATA_fEND` verwendet werden. Damit wird angezeigt, daß der Datenblock codiert wurde. Im Datenblock `DATA_fSTART` ist dann zusätzlich der für die Dateicodierung verwendete C-Schlüssel enthalten.

Der Master nimmt alle übertragenen Dateien entgegen und entnimmt den Transferdefinitionen, was mit den Daten geschehen soll. Falls für die Übertragung keine Transferdefiniton existiert, antwortet der Master in Schritt (6) mit `DATA_REJECT`. Andernfalls werden die Daten weiterverarbeitet (vgl. Kap. 4.4). Wenn sich aufgrund der Transferanweisungen neue Übertragungen ergeben, markiert der Master die Dateien entsprechend.

Der Slave wechselt nach der initialen Übertragung in eine Schleife. Diese Schleife wird (ähnlich wie die des Masters) von drei Zeitkonstanten bestimmt.

Wie beim Master, so gibt auch beim Slave t_1 eine Verzögerung in der Hauptschleife an.

In Abständen von t_2 werden die Systemdateien auf Änderungen überprüft und ggf. eine Übertragung zum Master eingeleitet.

Der Wert t_3 spezifiziert einen Timeout für die Kommunikation mit dem Master. Wurde nach dieser Zeit auf eine Anfrage beim Master keine Antwort erhalten, so wird angenommen, daß der Master *down* ist; der Slave führt einen Restart durch.

5. Realisierung

In den nachfolgenden Kapiteln soll ein Programmsystem realisiert werden, welches die in Kapitel 2 geforderten Eigenschaften besitzt. Als Grundlage für die Realisierung soll dabei das Modell dienen, das in Kapitel 4 entwickelt wurde, wobei bisher ungenutzte Erfahrungen aus Kapitel 3 berücksichtigt werden sollen.

5.1. Grobspezifikation

In Kapitel 2 wurde schon ausführlich die Aufgabenstellung dargelegt, dennoch soll hier nochmals ein Leistungsumfang spezifiziert werden. Die Überlegungen in Kapitel 3 und 4 haben bei verschiedenen Aspekten der Aufgabenstellung zu einer anderen Sichtweise geführt, die sich in vielen Punkten zwar im wesentlichen auf implementierungstechnische Details auswirkt, teilweise aber auch die Aufgabenstellung an sich in einem anderen Licht erscheinen läßt. Bei anderen Punkten hat sich aufgrund der vorgenommenen Überlegungen bereits eine starke Konkretisierung ergeben, die die Art einer späteren Implementierung weitgehend festlegt.

Es soll ein sicheres System für eine Administration von UNIX-Systemdateien über ein Netzwerk implementiert werden. Dieses System soll flexibel und leicht portierbar sein. Daher ist bei allen Implementierungsentscheidungen unter den angemessenen Konzepten ein möglichst *einfaches* zu wählen um die Portierbarkeit zu erleichtern. Ferner muß eine nachträgliche Konfiguration einfach möglich sein, so daß eine Anpassung an unterschiedlichste Umgebungen leicht durchgeführt werden kann.

Auf jedem der verwalteten Rechnern im Netz wird ein Slave-Programm installiert, das lokale Änderungen der betrachteten Systemdateien beobachtet und diese einem Zentralrechner mitteilt.

Auf dem Zentralrechner läuft ein Master-Programm. Dieses sammelt die von den Slave-Rechnern übertragenen Systemdateien in einer globalen Kopie. Es ist dem Administrator möglich, diese globale Kopie mit einem Editor zu bearbeiten. Das Master-Programm entscheidet anhand seiner Konfiguration über Gültigkeit der von den Slave-Rechnern gemeldeten Daten und deren Relevanz für andere Rechner. Gemäß dieser Entscheidung leitet der Master die Daten an andere Rechner weiter oder macht die Änderung durch Übertragung der alten Daten an den Slave wieder rückgängig. Ferner überwacht der Master die globalen Kopien auf Änderungen, die vom Administrator manuell angebracht wurden und leitet ggf. Übertragungen an die betroffenen Slave-Rechner ein.

Die Slaves nehmen alle vom Master gelieferten Daten entgegen und übertragen sie in ihre lokalen Dateien.

Sowohl Master- als auch Slave-Programme sollen möglichst flexibel konfigurierbar sein, damit ein großes Anwendungsspektrum erhalten bleibt. Auf dem Master müssen folgende Angaben spezifiziert werden können:

- die Namen der Rechner, die verwaltet werden sollen
- die Namen der Systemdateien, die verwaltet werden sollen
  Dies müssen nicht die UNIX-Dateinamen sein, gemeint ist vielmehr ein ‘logischer’ Name.
- die Datensatzstruktur der Systemdateien
- Behandlung der Daten auf dem Masterrechner, z. B. Schlüsselfelder der Systemdateien
- Datenverteilungsschemata.
  In diesen Schemata wird jeweils für eine Systemdatei und je Rechner(gruppe) angeben, wie die

gemeldeten Änderungen an der Systemdatei behandelt werden sollen. Mögliche Varianten wurden bereits ausführlich im Transfermodell (Kap. 4.4) behandelt.

- physikalischer Name der Datenbank
- Sicherheitsklassifizierung der Daten in den Systemdateien (näheres in Kap. 5.2.1)

Auch die Slave-Rechner sind zu konfigurieren.

Es scheint nicht sinnvoll mehrere alternative Rechner für einen einzelnen Slave als Master-Rechner zuzulassen. Falls der Master-Rechner für längere Zeit ausfällt, ist es leicht möglich, die Konfigurationsdateien auf den Slaves zu ändern. Ferner wird so ein Wettbewerb mehrerer Master vermieden, der zu Inkonsistenzen führen könnte. Es ist daher sinnvoll den Namen des Master-Rechners in der Konfiguration anzugeben.

In der Konfiguration der Slaves ist also mindestens die Angabe der folgenden Daten nötig:

- der Name des Master-Rechners
- die logischen Namen der betrachteten Systemdateien
  Darüberhinaus wird zu jeder Systemdatei zusätzlich angegeben
  - das lokale Format
  - der physikalische Name
  - Konvertierungsfunktionen für die Konvertierung zwischen globalem und lokalem Format
  - Sicherheitsklassifizierung der Daten (näheres in Kap. 5.2.1)

5.2. Implementierung

Die Realisierung des Systems geht – trotz der bereits im Modell (Kap. 4) und der Implementierungsspezifikation (Kap. 5.1) getroffenen Entscheidungen – einher mit der Klärung einer großen Zahl von bisher offenen Implementierungsfragen.

5.2.1. Implementierungsentscheidungen

- *Rechnerumgebung*
  Die Rechnerumgebung ist im wesentlichen bereits durch die Aufgabenstellung vorgegeben: die Anwendung soll auf UNIX-Systemen in einem TCP/IP-basierten Netz laufen, sie soll leicht portierbar sein. Um das Programmsystem zu testen sind mindestens zwei Rechner nötig. Es bietet sich daher an, die Implementierung auf zwei möglichst verschiedenen Rechnern gleichzeitig durchzuführen. Dadurch wird sofort eine Programmflexibilität erzwungen. Bei späteren Tests können dann weitere Portierungen auf andere Rechner durchgeführt werden und ggf. die Flexibilität zusätzlich ausgebaut werden. Als Kanditaten für möglichst verschiedene Rechner bieten sich in diesem Hause an
  - eine Sun Sparc-Station mit SunOS 4.0 (ein BSD 4.3-Kernel, Big Endian)
  - eine DEC-5400 mit Ultrix V3.1C-0, Rev. 42 (Kernel BSD-like, Little Endian)

  Für weitere Implementierungen in der Testphase stehen noch diverse andere Systeme zur Verfügung. Dabei bieten sich an
  - eine Nixdorf Targon/31 mit System V.4.0.10 (ein System V R3-Kernel, Big Endian)
  - eine Nixdorf Targon/35 mit TOS 3.2-06
    (PyramidOS, Betrieb wahlweise BSD- oder System V-like, Big Endian)
  - eine Siemens MX500 mit DYNIX-2.0 (Kernel BSD-like, Little Endian)

  Darüberhinaus existieren diverse andere Rechner mit mehr oder weniger ähnlichen Betriebssystemversionen, die für ausgedehnte Tests in Betracht kommen.
- *Programmiersprache*
  Aus Portabilitätsgründen ist nur die Sprache C möglich; nur sie ist mit Sicherheit auf allen UNIX-Rechnern verfügbar.
- *Netzwerkzugriff*
  Die Verwendung von RPC bzw. Secure RPC für die Implementierung der Netzwerkkommunikation wurde weiter oben (Kap. 3.7) bereits ausgeschlossen.
  Für die Realisierung der Netzwerkkommunikation müssen daher Sockets und die zugehörigen Systemroutinen verwendet werden. Nun sind Sockets ursprünglich ein BSD-Feature und in alten AT&T-Kernels nicht vorhanden. Allerdings ist in diesen alten Kernels normalerweise auch kein Netzwerkanschluß vorgesehen. Die Netzwerksoftware in neueren AT&T-Systemen ist dagegen fast immer stark an die BSD-Implementierungen angelehnt, so daß in diesen Fällen eine Portierung vergleichsweise leicht möglich sein sollte.
- *Server oder Service?*
  Prinzipiell kann die Implementierung als Service oder als Server erfolgen. (Beim Service wird die Anwendung nur dann gestartet, wenn sie wirklich benötigt wird; ein Server wird einmal gestartet und wartet dann, bis er angesprochen wird).
  Der Master wird sehr oft in Anspruch genommen und muß außerdem den Status der Slave-Rechner überwachen. Bei einer Realisierung als Service müßten alle Statusinformationen in Dateien festgehalten werden; dennoch würde fast ununterbrochen das Master-Programm laufen. Eine Implementierung als Service bringt also nur zusätzlichen Aufwand.

Die Slaves müssen regelmäßig – also z. B. im Minutenabstand – die Veränderungen der zu überwachenden Systemdateien überprüfen; die Zeitspanne sollte nicht wesentlich größer sein, da sonst Inkonsistenzen im Netz Probleme schaffen können. Bei einer solch hohen Programmfrequenz ist auch hier eine Implementierung als Server vorzuziehen.

- *Netzwerkprotokoll*
  Auf Sockets sind standardmäßig zur Übertragung auf Netzwerken die Protokolle TCP und UDP implementiert. UDP ist ein 'verbindungsloser' Paketservice, TCP ein 'verbindungsorientierter' Streamservice. Für die geplante Anwendung ist es nötig, daß der Master viele Verbindungen zu mehreren Slaves unterhalten muß; im Extremfall muß ein Kontakt zu allen Slaves gleichzeitig unterhalten werden. Andererseits treten lange Zeitperioden auf, in denen zwischen dem Master und einem Slave gar keine Kommunikation stattfindet. Es bietet sich daher die Verwendung des UDP-Protokolls an.

- *Datenbank*
  Auf dem Master-Rechner muß eine Kopie der einzelnen lokalen Systemdateien vorhanden sein, damit unerlaubt lokal vorgenommene Änderungen wieder rückgängig gemacht werden können. Außerdem ist auch nur auf diese Weise eine sinnvolle Bearbeitung der lokalen Dateien vom Master aus möglich. Dabei sollen (aus Platzgründen) gleiche Datensätze von verschiedenen Rechnern nur einmal gespeichert werden.
  Auf dem Master-Rechner existiert also eine Datenbank mit Tabellen (pro unterschiedlicher Systemdatei). Die Datensätze in den Tabellen entsprechen den Datensätzen in den einzelnen lokalen Systemdateien, ergänzt um ein Attribut, welches den Gültigkeitsbereich (welche Rechner?) des Satzes kennzeichnet.
  Es stellt sich die Frage, welches Datenbanksystem verwendet werden soll. Dieses System muß zum einen natürlich die Anforderungen der gestellten Aufgabe erfüllen können. Andererseits ist ein wesentlicher Gesichtspunkt wieder die Portabilität: das Datenbanksystem muß rechnerunabhängig und auf allen Maschinen verfügbar sein. In Frage käme auch die Verwendung eines datenbankunabhängigen Interfaces für die Datenbankzugriffe, wie z. B. *qdb* von Quantum [18]. Problematisch ist dabei jedoch, daß dieses Interface zwar datenbankunabhängig eine ganze Reihe von verschiedenen Datenbanksystemen unterstützt, dabei aber natürlich die Existenz eines solchen Systems vorausgesetzt wird. Für die vorliegende Aufgabe sollte eine solche Voraussetzung jedoch vermieden werden.
  Auf UNIX-Maschinen ist üblicherweise eine Funktionsbibliothek namens *dbm* vorhanden, die eine Reihe einfacher Datenbankoperationen erlaubt. Die Verwendung dieser Bibliothek stellt jedoch einen schlechten Kompromiß dar. Da diese Funktionen sehr universell gehalten sind, ist eine Anpassung an die vorliegende Aufgabe nötig. Systemdateien sind üblicherweise reihenfolgesensitiv. Demgegenüber wird in einer dbm-Datenbank die Reihenfolge durch eine Hash-Funktion (über den Schlüssel) vorgegeben. Eine Beibehaltung der Datensatzreihenfolge erfordert also einen zusätzlichen Aufwand.
  Übrig bleibt somit nur die Implementierung eines eigenen, einfachen 'Datenbanksystems'. Da im Rahmen dieser Arbeit kein großer Aufwand in diese Teilaufgabe investiert werden kann, bietet sich die Verwendung von ASCII-Dateien an: Je Datenbanktabelle eine Datei, in der Datei müssen die Datensätze geeignet getrennt werden.
  Diese Entscheidung folgt auch einer alten UNIX-Philosopie, wonach alle System- und Konfigurationsdateien nach Möglichkeit als reine ASCII-Textdateien implementiert werden, so daß eine Bearbeitung immer mit den gleichen, einfachen Werkzeugen möglich ist. Dieser Philosophie folgend, können dann dieselben Standard-UNIX-Werkzeuge auch im Rahmen dieser Arbeit zur Verarbeitung der globalen Files verwendet werden, so daß sich an vielen Stellen der Programmieraufwand deutlich verringert.

- *Datenformat*
  Es ist möglich, daß auf unterschiedlichen Maschinen gleiche Systemdateien verschiedene Formate aufweisen. Dies betrifft die Reihenfolge der Attribute und die Trennzeichen, aber auch ggf. vorhandene, zusätzliche Attribute. Ferner ist denkbar, daß manche Rechner eine 'logische' Systemdatei auf mehrere physikalische Dateien aufgeteilt haben.
  Es ist daher nötig, für jede Systemdatei ein netzwerkeinheitliches globales Format definieren zu können. Der Einfachheit halber wird die Konvertierung vom lokalen in das netzglobale Format den Slaves überlassen. Das Netzformat wird in der Konfigurationsdatei des Masters definiert und den Slaves auf Anfrage mitgeteilt.
- *Authentifikaton*
  Die Wichtigkeit der Authentifizierung wurde bereits in Kap. 4.2 ausführlich diskutiert und in Kap. 4.5ff ein angemessenes Konzept entwickelt, welches hier verwendet werden kann.
- *Integrität von Daten*
  In Kap. 4.2 wurde die Integritätssicherung berücksichtigt und muß nun geeignet realisiert werden. Wesentlich ist dabei die geeignete Verschlüsselung des Prüfsummenblocks, damit eine Manipulation ausgeschlossen werden kann.
- *Vertraulichkeit von Daten*
  Ein Teil der Informationen in den Systemdateien mag mehr oder weniger vertraulich sein. Je nach Vertraulichkeitsstufe sollte eine unverschlüsselte Speicherung oder Weitergabe vermieden werden.
  Klar ist, daß für die Verschlüsselung von Systemdateien nur ein symmetrisches Verschlüsselungsverfahren in Frage kommt. Es müssen u. U. vergleichsweise große Datenmengen ver- und entschlüsselt werden, daher ist ein asymmetrisches Verfahren mit Sicherheit zu rechenaufwendig; die einzelnen beteiligten Maschinen würden zu stark belastet.
  Es bietet sich die Verwendung des DES (Kap. 3.2.3) an. Dieser Algorithmus ist vergleichsweise schnell und sicher. Er ist auf vielen UNIX-Rechern in Funktionsbibliotheken enthalten, wird teilweise sogar hardwaremäßig unterstützt. Der Algorithmus ist auch im Quellcode verfügbar.
  Es soll dem Anwender möglich sein anzugeben, ob eine Systemdatei vertraulich sein soll oder nicht. Dabei stellt sich die Frage, in welcher Form solche Klassifizierungen sinnvoll sind:
  - *Speicherung auf dem Master*
    Der Master gilt als sicherer Rechner. Es scheint daher nicht sinnvoll, eine Verschlüsselung der Daten in der Master-Datenbank zu fordern. Ohnehin ist auf dem Master ein Editor-Programm vorgesehen, welches die globale Modifikation der Systemdateien erlauben soll. Selbst wenn in der Masterdatenbank die Daten verschlüsselt wären, könnten sie mit dem Editor-Programm entschlüsselt werden.
  - *Speicherung auf den Slaves*
    Die einzige auf den Slaves existierende Kopie der Systemdateien sind die Systemdateien selbst. Es ist klar, daß hier – außer den vom System vorgegebenen Funktionen – keinerlei zusätzliche Absicherungen möglich sind, ohne die Funktionalität zu gefährden. Lediglich zu beachten ist die Vertraulichkeit von ggf. auftretenden Temporärdaten.
  - *Netzübertragung*
    Am Ethernet kann aufgrund der Bus-Struktur jeder angeschlossene Rechner alle übertragenen Daten lesen. Es scheint daher sinnvoll, die Option der Verschlüsselung von Daten während der Netzübertragung anzubieten.
  - *Klassifizierung lokal oder global*
    Prinzipiell könnte eine Klassifizierung der Daten entweder in den Konfigurationsdateien der Slaves vorgenommen werden oder in der Konfiguration des Masters. Eigentlich sollte

jeder Slave selbst entscheiden können, ob er seine Daten vertraulich behandelt wissen möchte. Andererseits kann es nicht sinnvoll sein, daß Slave A seine Daten codiert an den Master sendet, und sofort danach der Master diese Daten uncodiert an einen Slave B propagiert, weil bei diesem diese Systemdatei als nicht vertraulich gilt.
Daher soll in allen Konfigurationsdateien eine Klassifizierung als vertraulich zugelassen werden., Wenn aber irgendein beteiligter Slave fordert, daß eine Systemdatei vertraulich behandelt werden soll, dann muß diese Vertraulichkeit netzweit gelten.

- *Key Management*
  Es muß geklärt werden, wie die in Kap. 4.6 entworfene Schlüsselhierarchie implementiert werden soll.
  Zunächst einmal könnte sowohl ein symmetrisches als auch ein asymmetrisches Verfahren für die Codierung der Authentifizierungsschlüssel verwendet werden. In beiden Fällen würden die Slaves jeweils ihren geheimen Schlüssel besitzen. Im symmetrischen Fall wäre dieser geheime Schlüssel auch dem Master bekannt, im asymmetrischen Fall besäße der Master alle öffentlichen Schlüssel, sowie einen eigenen geheimen Schlüssel und einen eigenen öffentlichen Schlüssel, den er auf Anfrage bekannt geben würde.
  Das größte Problem ist die Initialisierung des Systems: die Slaves müssen einen geheimen Schlüssel besitzen, der Master den entsprechenden Gegenpart. Eine Übertragung über das Netz kommt nicht in Frage, denn dazu müßte eine Authentifikation erfolgen, welche aufgrund der fehlenden Schlüssel nicht durchgeführt werden kann. Somit bleibt nur die Möglichkeit einer manuellen Vorverteilung der Schlüssel.
  Als nächstes stellt sich das Problem der Abspeicherung der Schlüssel. Auf dem Master ist dies kein Problem, der Server dort läuft in einer sicheren Umgebung. Problematisch sind die Slaves, da an sie keine zu hohen Sicherheitsanforderungen gestellt werden können. Daher ist es fraglich, wo der Schlüssel des Slaves abgelegt werden soll. Prinzipiell gibt es drei Möglichkeiten:

  - *im Programmcode*
  - *uncodiert in Datei*
  - *codiert in Datei, Hauptschlüssel im Programmcode*

  Keines der drei Verfahren ist optimal. Das Eincodieren von Schlüsseln in den Programmcode ist nicht gern gesehen, denn die Sicherheit einer Verschlüsselung sollte unter keinen Umständen lediglich auf der Geheimhaltung des Algorithmus' beruhen. Und wenn dieser Schlüssel dann – wie im ersten Verfahren – nicht einmal geändert werden kann, ist das besonders schlecht.

  Im zweiten Verfahren scheint die Lesbarkeit des Schlüssels (mindestens durch den Superuser) etwas unglücklich, obwohl nicht definitiv gesagt werden kann, ob dies von echter Bedeutung ist. Vorteilhaft ist die leichte Änderbarkeit des Schlüssels.

  Das dritte Verfahren scheint der beste Kompromiß: eine Entdeckung des eincodierten Schlüssels gefährdet die Sicherheit des Systems nicht unmittelbar; dadurch wird lediglich – wie im zweiten Verfahren – der Authentifizierungsschlüssel für den Entdecker des eingebrannten Schlüssels lesbar, sofern er Superuser-Rechte besitzt.
  Die Verwendung eines symmetrischen Verfahrens für den Authentifizierungsschlüssel erscheint (aufgrund der evtl. temporären Preisgabe des Schlüssels) etwas unglücklich, ein asymmetrisches Verfahren wäre deutlich besser.
  An dieser Stelle wird die Implementierung aber Probleme bereiten: ein asymmetrisches Verfahren ist nur hinreichend sicher, wenn wirklich *große* Zahlen bei der Codierung verwendet werden. Die Größe dieser Zahlen (bei RSA z. B. 512 Bit, vgl. Kap. 3.2.3) übersteigt jedoch die Ganzzahlarithmethik 'normaler' Rechner hoffnungslos. Viele UNIX-Rechner bieten für solche Anwendungen eine Bibliothek *mp,* mit arithmetischen Funktionen über Ganzzahlen von

willkürlicher Genauigkeit (im Rahmen der Speicherkapazität des Rechners). Leider ist diese Bibliothek aber auf vielen Systemen standardmäßig nicht vorhanden und müßte erst implementiert werden. Daher wird im Rahmen dieser Arbeit für eine erste Implementierung ein symmetrisches Verfahren (und zwar DES) eingesetzt. In einer späteren Erweiterung kann dann leicht auf RSA gewechselt werden.

- *Konfiguration*
  Sowohl Master- als auch Slave-Programm sind weitgehend frei konfigurierbar (vgl. Kap. 5.1), daher wurde bereits festgelegt, daß einfache Sprachen zur Konfiguration eingesetzt werden sollen. Auf allen UNIX-Rechnern stehen mit `lex` und `yacc` leistungsfähige Tools zu Entwicklung eines Parsers zur Verfügung. Es bietet sich daher an, diese Werkzeuge für die Programmierung einer Erkennung der Konfigurationssprachen einzusetzen. Das Master- bzw. Slave-Programm bekommt beim Start den Namen der jeweiligen Konfigurationsdatei übergeben und konfiguriert sich nach der Durchführung eines entsprechenden Parser-Laufs. Die Sprachen werden gemäß den im Modell gestellten Anforderungen im folgenden Kapitel geeignet definiert.

5.2.2. Konfigurationssprachen

Da auf dem Master zum Teil andere Definitionen nötig sind als auf den Slaves, müssen sich die Konfigurationsprachen zwangsläufig unterscheiden. Die syntaktischen Unterschiede sollten so gering wie möglich gehalten werden, um den Anwender den Umgang mit den Konfigurationen nicht unnötig zu erschweren.

Ein großer Teil derjenigen Daten, der in den jeweiligen Sprachen spezifizierbar sein muß, wurde bereits im Kapitel 4 vorgegeben. Zusätzliche Aspekte und weitergehende Detaillierung ergibt sich aus Kapitel 5.2.1. Nachfolgend sollen die beiden Sprachen definiert werden. Die Syntax wird in einer an die Backus-Naur-Form angelehnten – `yacc`-ähnlichen – Schreibweise† angegeben. Die jeweilige Semantik ist, sofern nicht offensichtlich, in einer anderen Schrifttype angegeben.

Zum besseren Verständnis der Bedeutung der einzelnen Konstrukte sei an dieser Stelle bereits auf das Beispiel verwiesen, welches in Kap. 5.3.2 vorgestellt wird.

5.2.2.1. Konfigurationssprache für den Master

Ein Masterskript ist die Konkatenation verschiedener Definitionsblöcke.

```
masterscript
: dbdef timedef hostdef groupdef filedef transdef
```

In der Datenbankdefinition wird der physikalische Name für das Datenbankdirectory angegeben. Bei fehlender Definition wird das aktuelle Directory beim Programmstart eingesetzt.

```
dbdef
: DATABASE Pathname ;
| ε
```

In der Definition des Timings werden Zeitkonstanten (in Sekunden) für die Regelung des Zeitverhaltens des Masters bei der Kommunikation mit den Slaves und für die Überprüfung lokaler Dateiänderungen definiert. Bei fehlender Definition verwendet der Master folgende Standardwerte

† Nichtterminale sind in Kleinbuchstaben angegeben, ':' und '|' sind Metazeichen, die Definition bzw. Alternative bedeuten. Wörter mit großem Anfangsbuchstaben sind 'atomare' Nichtterminale, die durch reguläre Lex-Ausdrücke beschrieben werden. Das Zeichen 'ε' kennzeichnet einen leeren Ausdruck. Alle anderen Zeichen sind Terminalzeichen.

(in Sekunden): $t_1=1$, $t_2=5$, $t_3=30$ und $t_4=5$. Die Bedeutung der einzelnen Konstanten ('t_1', ..., 't_4') wurde bei der Beschreibung des Kommunikationsablaufes in Kap. 4.7 erläutert.

```
timedef
: TIMING Number , Number , Number , Number ;
| ε
```

Im `hostdef`-Block werden die Namen der betrachteten Slave-Rechner angegeben. Alle aufgezählten Namen müssen in `/etc/hosts` eingetragen bzw. dem Rechner per YP, Nameserver o. ä. bekannt gemacht worden sein.

```
hostdef
: SLAVES namelist ;

namelist
: Name
| Name , namelist
```

Im Rechnergruppen-Definitionsblock können Rechner gruppiert werden. Alle in der Namensliste genannten Namen müssen *entweder* bereits definierte Gruppennamen *oder* definierte Rechnernamen sein.

```
groupdef
: GROUP Name = namelist ; groupdef
| ε
```

Die Beschreibung des Aufbaus der verwalteten Dateien erfolgt in der Filedefinition:

```
filedef
: FILE Name = attribs filetyp ; filedef
| FILE Name = attribs filetyp ;
```

In der Definition der Fileeigenschaften wird mit `SECURE` spezifiziert, daß diese Datei bei der Übertragung verschlüsselt werden soll. Fehlt die Angabe, so wird keine Verschlüsselung durchgeführt, sofern kein Slave eine Sicherung fordert.
`INITDEL` gibt an, daß die globale Datei beim Systemstart gelöscht werden soll.

```
filetyp
| SECURED filetyp
| INITDEL filetyp
| ε
```

Zu jeder Dateibeschreibung gehört eine Attributliste. Die vorkommenden Namen müssen eine Obermenge der in den Attributlisten der entsprechenden Slave-Definitionen vorkommenden Attribute sein. Das netzglobale Dateiformat wird durch die Reihenfolge der Attribute in der Master-Attributliste festgelegt.
Der Schrägstrich trennt die Schlüsselattribute von den ggf. vorhandenen Nichtschlüsselattributen. Die Schlüsselattribute dienen zur eindeutigen Kennzeichnung der Datensätze, zur Sortierung und werden verwendet, um festzustellen ob Datensätze geändert oder aber hinzugefügt wurden.

Bei den Schlüsselattributen kann im Definitionsteil `attrnum` angegeben werden, ob es sich um ein numerisches oder ein alphanumerisches Attribut handelt. Numerische Attribute werden durch ein dem Namen vorangestelltes `%`-Zeichen gekennzeichnet.

Das letzte Attribut darf ein zusammengesetztes Attribut sein und wird dann durch nachgestellte Klammern gekennzeichnet. Alternativ kann die Attributliste auch leer sein. Die Datei ist dann unstrukturiert und wird als einziger Datensatz behandelt.

```
attribs
: attrnum Name
| attrnum Name / attrlist
| attrnum Name , attrnum Name / attrlist
| ε
attrlist
: Name , attrlist
| Name
| Name [ ]
attrnum
: %
| ε
```

Der Transferdefinitionsblock spezifiziert wie mit erhaltenen Informationen verfahren werden soll. Jede Transferdefinition gilt jeweils für eine logische Systemdatei (mit dem Namen `Name`), und für Informationen, die von einem der Rechner aus der Quellgruppe (Gruppenname hinter `FROM`) stammen.
In der Variante `TO groupname` werden alle Änderungen für die in der Zielgruppe liegenden Rechner gültig.
Mit der Angabe `NOCHANGE` wird nur das Hinzufügen von Sätzen erlaubt, Änderungen bestehender Sätze werden zurürckgewiesen.
Die Angabe `REJECT` bedeutet, daß lokale Dateiänderungen nicht erlaubt sind; alle gemeldeten Änderungen werden rückgängig gemacht.
Der Wert `ONCE` bedeutet, daß nach dem Programmstart einmal eine Übertragung durch den Slave akzeptiert wird. Danach verhält sich `ONCE` wie `REJECT`.
Die Varianten ohne `TO groupname` verhalten sich wie `TO MASTER`. Alle Daten von Rechnern ohne Transferangabe für die jeweilige Datei werden ignoriert.
Mit der `transappend`-Definition wird angegeben, ob die neuen Datensätze die vorhandenen dieser Zielgruppe komplett ersetzen (`REPLACE`), oder ob sie einfach hinzugefügt werden und nur Sätze mit demselben Schlüssel einen vorhandenen Satz ersetzen (`APPEND`). Falls `APPENDFIRST` angegeben wird, führt der Master bei der ersten Übertragung eine `APPEND`, bei jeder weiteren ein `REPLACE` durch. `APPENDFIRST` ist nur sinnvoll, falls für die entsprechende Datei ein `INITDEL` angegeben wurde. Eine leere `transappend`-Definition bedeutet `REPLACE`. Bei `NOCHANGE` wird immer `APPEND` angenommen.

```
transdef
: TRANSDEF Name FROM groupname transtyp ; transdef
| TRANSDEF Name FROM groupname transtyp ;
transtyp
: TO groupname transappend
| transappend
| TO groupname NOCHANGE
| NOCHANGE
| TO groupname ONCE transappend
| ONCE transappend
| REJECT
```

```
transappend
: APPEND
| APPENDFIRST
| REPLACE
| ε
```

Als Gruppennamen in der Transferdefinition sind entweder einfache Namen (aus der Rechner- oder Gruppendefinition) zugelassen oder die symbolischen Namen `ALL` (alle Rechner) und `MASTER` (nur der Master).

```
groupname
: Name
| ALL
| MASTER
```

5.2.2.2. Konfigurationssprache für die Slaves

Auch das Slaveskript basiert auf der Verkettung von Definitionsblöcken, die denen des Masters ganz ähnlich sind.

```
slavescript
: dbdef timedef hostdef filedef
```

Der Datenbankdefinitionsteil `dbdef` hat die gleiche Syntax wie derjenige in der Masterkonfiguration.

In der Definition des Timings werden – ähnlich wie in der Masterkonfiguration – Angaben über das Zeitverhalten des Slaves gemacht. Slaves erhalten jedoch nur drei Zeitkonstanten. Fehlt dieser Definitionsblock, so gelten die Standardwerte (in Sekunden) $t_1 = 2$, $t_2 = 5$, $t_3 = 10$.

```
timedef
: TIMING Number , Number , Number ;
| ε
```

Im `hostdef`-Block wird der Rechnername des Masters angegeben. Für diesen Namen gelten die gleichen Bedingungen wie für die Slave-Namen in der Masterkonfiguration.

```
hostdef
: MASTER Name ;
```

In der Filedefinition des Slavescripts wird der physikalische Reihenfolge der Attribute in den Systemdateien spezifiziert. Die Attributliste muß eine Untermenge der in der Masterdefinition angegebenen Attributliste sein. Die vom Master definierten Schlüsselattribute müssen vorhanden sein.

```
filedef
: FILE Name = attribs filetyp ; filedef
| FILE Name = attribs filetyp ;
```

Die Definition der Attributlisten erfolgt im wesentlichen genau wie diejenige im Masterskript, nur die Kennzeichung der Schlüsselattribute unterbleibt.

```
attribs
: attrlist
| ε
```

```
attrlist
: Name , attrlist
| Name
| Name [ ]
```

In der Definition der Fileeigenschaften wird mit `PATH` wird der physikalische Dateiname spezifiziert; fehlt diese Angabe, so wird angenommen, daß der physikalische Name `~/`*logname* lautet, wobei ~ das Datenbankdirectory und *logname* der logische Dateiname ist.
Mit `CONV` wird der physikalische Dateiname des Datenkonvertierungsprogrammes angegeben. Hier lautet der Defaultwert ist `~/conv.`*logname*. Die Eigenschaften eines Konvertierungsprogrammes werden später noch definiert.
`SECURED` fordert eine codierte Übertragung dieser Datei. Eine fehlende Angabe bedeutet uncodierte Übertragung, es sei denn, der Master ordnet eine Verschlüsselung an.

```
filetyp
: PATH Pathname filetyp
| CONV Pathname filetyp
| SECURED       filetyp
| ε
```

Bei Pfadangaben in der Konfigurationsdatei darf ~ angegeben werden, wenn das Datenbankdirectory referenziert werden soll.

5.2.3. Paketaufbau und Sicherungsstrategie

Bereits geklärt wurde die Verwendung des DES für die Verschlüsselungen und Authentifizierungen. Im Modell wurde auch bereits eine Schlüsselhierarchie entworfen.

Der im Modell postulierte A-Schlüssel wird zur Compilationszeit in jedes Programm eingebrannt und ist somit – außer durch Recompilation – nicht änderbar. Wie bereits im Modell geplant, werden die B-Schlüssel mit den A-Schlüsseln codiert und jeweils in einer lokalen Datei abgelegt.

Zur Integritätssicherung wird eine Prüfsummenfunktion entworfen. Diese Prüfsummenfunktion kann dann vor der Übertragung eines Datenblockes auf denselben angewendet werden. Die so erhaltene Prüfzahl wird dann mit dem entsprechenden B-Schlüssel codiert und mit dem Block übertragen.

Das verwendete UDP-Protokoll garantiert weder, daß die abgesendeten Pakete in der korrekten Reihenfolge eintreffen, noch, daß die Pakete bei der Übertragung nicht dupliziert werden. Dieses Problem kann bei der Integritätssicherung gleich mit abgedeckt werden.

Unter UDP ist die maximale Blockgröße für eine Netzübertragung eingeschränkt. Üblicherweise liegt die Blockgröße bei $\geq$ 1 KByte. Damit unsere Übertragungen problemlos möglich sind, sollte man die Blockgröße also auf 1 KByte einschränken.

Der in Kapitel 4.5 entworfene Paketaufbau läßt sich unmittelbar in C abbilden als:

```
struct packet {
    long   pack_check;
    long   pack_auth;
    short  pack_type;
    short  pack_length;
    byte   pack_data[DATALEN];
}
```

Dabei ist `pack_check` die mit dem augenblicklich gültigen B-Schlüssel codierte Prüfzahl über das Datenpaket.

Das Feld pack_auth beinhaltet den Authentifikator. Die im Modell geplante Variante löst nicht nur das Replay-Problem, sondern auch das Problem der UDP-Paketduplizierung. Auch die korrekte Paketreihenfolge ist gewährleistet, da das Modell nach jeder Paketsendung einen Handshake durch Senden eines Quittungspaketes vorsieht. Diese Methode ist zwar nicht sehr effizient, reicht aber für den vorgesehenen Zweck aus. Verbesserungen, z. B. die Einführung eines Übertragungsfensters sind im Rahmen dieser Arbeit zu aufwendig in der Realisierung, lassen sich aber nachträglich leicht einbringen.

Das Feld pack_type beinhaltet den Pakettyp.

Das Feld pack_length gibt die tatsächliche Länge des Datenblocks an (nur der Bereich pack_data).

Das Feld pack_data beinhaltet die eigentlichen Daten. Die Art der Daten wird durch den Pakettyp bestimmt. Die Konstante DATALEN ist so gewählt, daß die maximale UDP-Paketgröße nicht überschritten wird. MAXLEN sollte außerdem ein Vielfaches von acht Bytes groß sein, damit ein Einsatz des DES zur Verschlüsselung des Datenblocks leicht möglich ist.

Im Modell war auch ein Paketfeld für den Absender vorgesehen. Dieses Feld braucht hier nicht explizit verwendet werden, da jedes UDP-Paket automatisch die Internet-Adresse des Absenders beeinhaltet. Diese Internet-Adresse kann mittels entsprechender Funktionen leicht in den zugehörigen Rechnernamen konvertiert werden. Zwar kann jeder Rechner seine Internet-Adresse beliebig verändern; dies ist hier aber unkritisch, da mittels des Authentifikators eine Verifikation möglich ist.

5.2.4. Kommunikationsablauf

Nachfolgend soll beschrieben werden, wie sich die Implementierung des Kommunikationsmodells verhält.

Beim Start des Masterprogramms wird die Konfigurationsdatei gelesen und auf Korrektheit überprüft. Danach wird getestet, ob für alle konfigurierten Slaves die B-Schlüssel bekannt sind. Falls ja, werden die B-Schlüssel gelesen, mit dem A-Schlüssel dekodiert und intern für die weitere Verwendung gespeichert. Falls nein, wird die Eingabe der fehlenden Schlüssel verlangt. Diese werden dann sowohl intern gespeichert, als auch (mit dem A-Schlüssel) codiert in den entsprechenden Schlüsseldateien abgelegt.

Anschließend wird in den Kommunikationsmodus gewechselt und – wie im Modell beschrieben – die Kommunikation mit den Slaves aufgenommen.

Zur Codierung der Authentifikatoren und Prüfsummen wird der B-Schlüssel des jeweiligen Slaves verwendet.

Wird auf einem Slave ein Serverprogramm gestartet, so wird ebenfalls zunächst die Konfigurationsdatei geparst. Bei fehlerfreier Konfiguration wird, genau wie auf dem Master, zunächst sichergestellt, daß ein B-Schlüssel existiert und danach in den Kommunikationsmodus gewechselt.

Nach Anschluß der initialen Authentifizierung fordert der Slave vom Master für jede der in der Slave-Konfiguration definierten Files das netzglobale Format an. Wie der Aufbau der dabei verwendeten Pakete ist, konnte im Modell (Kap. 4.7) noch nicht festgelegt werden, da noch Unklarheit über die Spezifikationsmöglichkeiten für die Dateiattribute bestand (nun definiert in Kap. 5.2.2).

Die Kommunikation läuft wie folgt ab:

S→M: DATA_FREQ, *Secured?*, *Dateiname*

In *Secured?* gibt der Slave an, ob er eine codierte Übertragung der Datei *Dateiname* wünscht. Falls dem Master die genannte Datei unbekannt ist, antwortet er mit DATA_FUNKN, andernfalls mit

S→M: DATA_FREQ, *Secured?*, *Attr1*, *Attr2*, ..., *Attrn*, *AttrnIsList*

Die Angabe *Secured?* gibt der Master an, ob die Datei codiert übertragen werden soll. Falls ein Slave eine codierte Übertragung vom Master fordert, markiert der Master diese Datei als netzglobal codiert zu übertragen. Falls der Master vom Slave eine codierte Übertragung fordert, markiert der Slave diese Datei entsprechend. *Attr1* bis *Attrn* ist die Liste der in der Masterkonfiguration spezifizierten Attribute für die Datei *Dateiname*. *AttrnIsList* gibt an, ob es sich beim letzten Attribut um ein Listenattribut handelt oder nicht.

Der Slave vergleicht die erhaltene Konfiguration mit der lokalen. Treten dabei Fehler auf (d. h. falls im lokalen Format Attribute spezifiziert sind, die im globalen Format nicht enthalten sind oder aber das Schlüsselattribut des Masters im lokalen Format fehlt) so quittiert der Slave mit DATA_REJECT, andernfalls mit DATA_ACCEPT. Im Vergleich der Attributlisten berechnet der Slave zwei Permutationsstrings, die später dem Konvertierungsprogramm übergeben werden. Diese Strings geben dem Konvertierungsprogramm an, wie die Attributfelder umsortiert werden müssen, um eine Konversion vom lokalen ins globale Format und umgekehrt zu erreichen.

Nachdem alle Dateibeschreibungen angefordert wurden, beginnt der Slave mit der Übertragung der Dateiinhalte. Mittels der entsprechenden Konvertierungsprogramme werden die Dateien vom lokalen ins globale Format übertragen.

5.2.5. Datenbankformat

In Abschnitt 5.2.1 wurde entschieden, daß eine 'ASCII-Datenbank' verwendet werden soll. Deren Aufbau und der Aufbau der Daten beim Filetransfer über das Netz soll hier angegeben werden.

Auf jedem Rechner existiert ein Datenbankdirectory (spezifiziert im Konfigurationsteil dbdef). Auf dem Master existieren darin mehrere Unterverzeichnisse, eines mit dem Namen global, die anderen mit den Namen der im Masterscript konfigurierten Rechner. Diese Directories werden beim Lesen des Masterscripts automatisch erzeugt. Im global-Verzeichnis befinden sich Files mit den im der Masterscript definierten Filenamen. Sie beinhalten die globalen Datenversionen. In den Directories mit Rechnernamen finden sich zeitweilig ähnliche Files. Hier werden vor einem Master-Slave-Transfer die lokalen Versionen generiert und bei einem Slave-Master-Transfer werden hier temporär die übertragenen Daten abgelegt, bevor sie weiterverarbeitet werden.

Im globalen Format haben die ASCII-Zeichen LF und FF besondere Bedeutung. LF ist das Attributtrennzeichen, FF das Datensatztrennzeichen. Der Aufbau also (Zeilenende=LF):

FF 00000000
Satz1-Attr1
Satz1-Attr2
. . .
Satz1-Attrn
FF 00000000
Satz2-Attr1
. . .
FF

Dabei steht 00000000 stellvertretend für eine hexadezimale Codierung des Bitstrings mit der Rechnergültigkeit. Diese Angabe ist nur in der globalen Datenbank enthalten, im globalen Transferformat fehlt sie.

5.2.6. Dateneditor auf dem Master

Auf dem Master wäre die Existenz eines speziellen Editors für die Bearbeitung der globalen Systemdateien wünschenswert.

Dieser Editor könnte die Modifikation der globalen Systemdateien dadurch wesentlich erleichtern, daß er die vom Masterserver vorgenommenen Codierungen in unmittelbar lesbarer Form darstellt (z. B. die Rechnernamen im Klartext, sowie Attributnamen in den einzelnen Datensätzen). Außerdem könnte der Editor auf Wunsch Datensätze ausblenden, so daß man nur die Version für einen einzelnen Rechner oder eine Rechnergruppe sieht.

Ein solcher Editor ist jedoch sehr aufwendig in der Programmierung und sprengt in jeder Weise – sowohl zeitlich als auch thematisch – den Rahmen dieser Arbeit. Da es sich bei der Datencodierung (vgl. Kap. 5.2.5) um ein reines ASCII-Format handelt, kann auf jeden Fall auch jeder Standard-Editor zur Bearbeitung der Systemdateien verwendet werden.

Es wird daher auf eine eigene Entwicklung zunächst verzichtet.

5.2.7. Konversionsprogramme

Auf den einzelnen Slaves sind Datenkonversionsprogramme vorgesehen, die das netzglobale Datenformat in das jeweilige rechnerlokale Format – und umgekehrt – konvertieren.

Die Datenkonvertierung muß sehr flexibel gehalten werden, da die Attributreihenfolge im globalen Format beliebig stark von derjenigen im lokalen Format abweichen kann und diese Reihenfolgen auch jederzeit beliebig umdefiniert werden können. Zudem ist prinzipiell ist für jeden Typ von Systemdatei ein eigenes Konvertierungsprogramm nötig, welches die speziellen Eigenschaften dieser Datei (z. B. Trennzeichen, ...) berücksichtigt.

Die Konvertierungsprogramme, deren Name im Slavescript in der Filedefinition unter `CONV` angegeben wird, werden vom Slaveserver bei Bedarf – mit entsprechenden Parametern versehen – aufgerufen, um eine entsprechende Konvertierung durchzuführen.

Der Aufruf sieht wie folgt aus:

conv direction input output permutation

Die einzelnen Teile bedeuten dabei:

- *conv*
  Der Programmname, der der `CONV`-Definition entnommen wird.
- *direction*
  Bestimmt die Konversionsrichtung; `NTOL` konvertiert vom Netz- ins Lokalformat, die Umkehrung erreicht man mit `LTON`.
- *input, output*
  Name der Eingabe- bzw. Ausgabedatei.
- *permutation*
  Ein Buchstabenstring, der die Feldreihenfolge in der Ausgabe angibt. Der String `CA@B` gibt an, daß in der Ausgabe folgende Reihenfolge der Eingabefelder verwendet wird: 3. Feld, 1. Feld, Leerfeld, 2. Feld.

5.2.8. Systemeinbindung

Das Socket-Konzept unter UNIX verlangt, daß dem Absender einer Nachricht die Portnummer des Empfängers bekannt sein muß. Dies bedeutet für unsere Anwendung, daß dem Master die Portnummern der Slaves bekannt sein müssen und den Slaves die Portnummer des Masters. Am einfachsten ist es, allen Slaves die gleiche Portnummer zu geben. Der Master kann nicht die gleiche Portnummer benutzen, damit auf dem Masterrechner neben dem Masterserver auch gleichzeitig ein Slaveserver laufen kann, der die lokalen Systemdateien des Masterrechners verwaltet.

Es müssen Portnummern gefunden werden, die netzweit unbelegt sind. Da die Nummernvergabe jedoch von Netz zu Netz unterschiedlich geregelt sein kann, sollen die Nummern nicht fest in die Programm hineincodiert werden. Stattdessen verwenden die Programme symbolische Namen und erfragen die zugehörigen Portnummern mittels der Funktion `getservbyname` vom System.

Die tatsächlich verwendeten Nummern müssen dann nur bei der Installation in der entsprechenden Konfigurationsdatei eingetragen werden. Diese Datei heißt meistens `/etc/services`. Der Masterserver trägt den symbolischen Namen `sysadm`, die Slaveserver den Namen `sysslv`. Ein Eintrag könnte somit wie folgt aussehen:

```
sysadm    27066/udp    # sysadm master
sysslv    27067/udp    # sysadm slave
```

Wesentlich bei den Einträgen ist somit neben den Namen, daß keine bereits vergebenden Portnummern benutzt und beide Programme als UDP-Services gekennzeichnet werden.

5.3. Anwendungsbeispiel

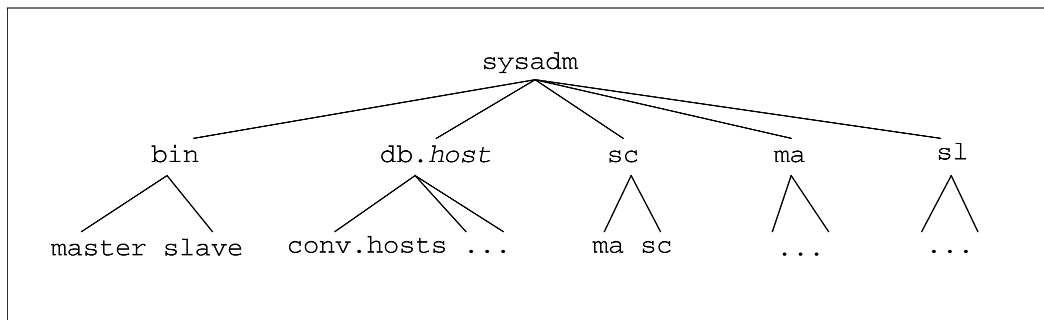
In diesem Kapitel soll die Installation und der Einsatz des Programmsystems an einem kleineren Anwendungsbeispiel dargestellt werden.

5.3.1. Installation

Damit das System leicht zu installieren ist, wurden die gesamten Quellen in zwei Dateien zusammengefaßt.

Die Datei `sysadm.tar` ist eine Archivdatei, die zweite Datei ist ein ausführbares Kommandoscript und heißt `sysadm.install`. Das Programmsystem erwartet, daß in der Umgebungsvariablen `HOST` der Rechnername steht. Die gesamte Installation kann dann durchgeführt werden durch Aufruf von `sysadm.install`. Dieser Aufruf entpackt das Quellarchiv, erzeugt die nötige Dateiverzeichnisstruktur, verteilt die Quellen auf die erzeugten Verzeichnisse und führt die nötigen Compilerläufe durch.

Es wird folgender Dateibaum erzeugt:



Das Directory `sysadm/bin` enthält das Master- und das Slaveprogramm. In `sysadm/sc` stehen Beispielscripten für Master und Slave. Das Verzeichnis `db.`*host* ist gedacht als Datenbankdirectory. Es enthält einige Beispielkonvertierungsprogramme. Die Verzeichnisse `ma` und `sl` beinhalten die Quellen von Master- und Slaveprogramm.

Anschließend müssen die Programme in das Rechnersystem eingebunden werden. Dazu sind Eintragungen in der Netzwerksoftwarekonfiguration des Rechners vorzunehmen. Dies wurde bereits in Kap. 5.2.8 beschrieben.

5.3.2. Konfiguration

Eine Entwurfsvorgabe bei der Entwicklung des Systems war die flexible Konfigurierbarkeit. Dementsprechend umfangreich fällt daher nun die Arbeit bei der Konfigurierung aus.

Als Grundlage für Master- und Slavescripten können die beiden Beispielscripten `sysadm/sc/ma` und `sysadm/sc/sl` verwendet werden, die dann an die lokalen Gegebenheiten angepaßt werden müssen. Anschließend ist noch die Erstellung der entsprechenden Datenkonvertierungsprogramme nötig. Dies kann z. B. durch eine Anpassung der bereits vorhandenen Beispielkonvertierungsprogramme (in `sysadm/db.`*host*) geschehen.

Das Konfigurationsscript des Masters könnte z. B. so aussehen:

```
/* ---- Datenbankdefinition ---- */

DATABASE "$(SYSADM)/db.master";


/* ---- Rechnerdefinition ---- */

SLAVES qu31, qualpha, qubeta, qugamma, quarc, qudec, quando;


/* ---- Gruppendefinitionen ---- */

GROUP suns = qualpha, qubeta, qugamma;


/* ---- Dateidefinitionen ---- */

FILE passwd = %uid, logname / gid,passwd,fullname,home,shell
     INITDEL SECURED;

FILE hosts  = internet / hostname , nickname[]
     INITDEL;

FILE aliases =
     INITDEL;


/* ---- Transferdefinitionen ---- */

TRANS hosts   FROM ALL  TO ALL APPENDFIRST;
TRANS passwd  FROM suns TO suns;
TRANS passwd  FROM qu31 REJECT;
TRANS aliases FROM ALL  ONCE;
```

In der Datenbankdefinition wurde angeben, in welchem Directory die Datenfiles des Masterservers abgelegt werden sollen. Dabei referenziert `$(SYSADM)` die Umgebungsvariable `SYSADM`, die bei einem Programmstart entsprechend gesetzt werden müßte.

Der Master will die Rechner mit den Namen *qu31*, *qualpha*, ... bedienen. Unter diesen Namen kann auch sein eigener Name sein.

In der Gruppendefinition werden drei Rechner aus Gruppe *suns* zusammengefaßt. Die Verwendung des Gruppennamens ist dann in der Transferdefinition möglich.

Es sollen die Dateien `passwd`, `hosts` und `aliases` verwaltet werden. Die Datei `passwd` umfaßt sieben Felder, die Felder `uid` und `logname` bilden zusammen einen eindeutigen Schlüssel. Das Feld `uid` ist dabei numerisch. Die Datei `aliases` ist ein unstrukturiertes ASCII-File. Alle drei Dateien werden beim Programmstart gelöscht. Eine verschlüsselte Übertragung wird bei der Datei `passwd` durchgeführt.

In der Transferdefinition wird angegeben, daß die Datei `hosts` von allen Rechnern akzeptiert wird. Gemeldete Änderungen werden an alle anderen Rechner weitergegeben. Bei der ersten Übertragung werden alle Sätze 'gesammelt' (`APPENDFIRST`).

Die Datei `passwd` wird von den Rechnern der Gruppe `suns` akzeptiert und dann an alle anderen Rechner der Gruppe weitergeschickt. Jede neue Sendung ersetzt die alte Kopie komplett. Achtung: Eine solche Konfiguration ist im allgemeinen nicht sinnvoll, da beim Programmstart mit unterschiedlichem Datenbestand auf den Rechnern zwangsläufig Daten verloren gehen. Dem Rechner *qu31* wird eine Änderung seiner `passwd`-Datei verboten.

Die zugehörige Konfiguration für den Slave *qualpha* könnte z. B. so aussehen:

```
/* ---- Datenbankdefinition ---- */

DATABASE "$(SYSADM)/db.$(HOST)";


/* ---- Hostdef ---- */

MASTER $(MASTERHOST);


/* ---- Filedefinitionen der lokalen Formate ---- */

FILE passwd = logname,passwd,uid,gid,fullname,home,shell
     PATH "~/passwd"  CONV "~/conv.passwd";

FILE hosts  = internet, hostname, nickname[]
     PATH "~/hosts"   CONV "~/conv.hosts";
```

Zunächst wurden in der Datenbankdefinition der Datenbankpfad und der Name des Masterrechners spezifiziert. Dabei wurden die drei Umgebungsvariablen `SYSADM`, `HOST` und `MASTERHOST` verwendet, die vor einem Programmstart entsprechend gesetzt werden müßten.

In der Filedefinition werden die lokalen Dateiformate für die Dateien `passwd` und `hosts` angegeben. Zusätzliche Angaben beinhalten die Dateinamen für die lokalen Systemdateien und die entsprechenden Konvertierungsprogramme. Im Beispiel sind sowohl die Konvertierungsprogramme als auch die Systemdateien alle im Datenbankdirectory enthalten (Pfadangabe "~").

5.3.3. Test

Wichtig beim realen Einsatz des Programmsystems sind ausführliche Tests der angegebenen Konfiguration. Aufgrund seiner Flexibilität kann die Konfiguration viele logische Fehler enthalten, die vom Programmsystem nicht erkannt werden können. Auch Fehler an selbsterstellten Datenkonversionsprogrammen können zur Zerstörung der Systemdateien führen. Es empfiehlt sich daher bei der Konfiguration zunächst auf Kopien der Systemdateien (wie im obigen Beispiel) zu testen, wie das System beim Programmstart und bei Änderungen der einzelnen Systemdateien auf den verschiedenen Slaves reagiert. Erst wenn alles wie gewünscht abläuft, sollte auf die 'echten' Systemdateien gewechselt werden.

Normalerweise laufen Master und Slave als Hintergrundprozesse und schreiben ihre Ausgaben in das Datenbankdirectory in eine Datei `master.log` bzw. `slave.log`. Dabei werden aber nur die wesentlichsten Aktionen protokolliert. Für Tests empfiehlt es sich, die Prozesse im Vordergrund laufen zu lassen und ggf. das 'Loglevel' heraufzusetzen, um noch mehr Informationen zu erhalten (vgl. Kap. 5.3.4).

5.3.4. Betrieb

Beide Programme wird man, wenn man sie im täglichen Betrieb verwenden möchte, nicht manuell starten, sondern dafür Sorge tragen, daß sie beim Systemstart automatisch aktiviert werden.

Der Master wird mit folgender Kommandozeile gestartet:

```
master [-e] [-k] [-i] [-l loglevel] script
```

Dabei fordert der optionale Parameter `-i`, einen interaktiven Programmbetrieb. Fehlt der Parameter, so läuft das Programm im Hintergrund und die Ausgaben werden im Datenbankdirectory in die Datei `master.log` geschrieben. Im Datenbankdirectory wird aber auf jeden Fall eine Datei `master.pid` angelegt, in der der Prozeß seine Prozeßnummer schreibt. Diese Datei wird beim Verlassen des Programms gelöscht und verhindert, daß auf einem Rechner zwei Masterprogramme gestartet werden.

Der gleichfalls optionale Parameter `-l`, gefolgt von einem numerischen Wert von 0 bis 3 spezifiziert das 'Loglevel'. Je höher das 'Loglevel', desto umfangreicher die Kontrollausgaben. Standardmäßig wird das Loglevel auf 1 gesetzt.

Mit dem Parameter `script` wird dem Master der Name der Konfigurationsdatei mitgeteilt, die er lesen soll.

Während des laufenden Betriebs läßt sich der Prozeß mittels der unter UNIX verfügbaren Signale manipulieren. Der Prozeß terminiert beim Empfang eines `QUIT`-Signals (im interaktiven Betrieb erzeugbar durch `Ctrl-\`). Eine Statusausgabe erreicht man durch ein `INT`-Signal (interaktiv je nach Terminal `Ctrl-C` oder `DEL`). Ferner läßt sich mit den Signalen `USR1` und `USR2` das 'Loglevel' in- bzw. dekrementieren. Alle anderen Signale führen zur unkontrollierten Termination des Programms. Zum Absenden von Signalen bietet sich die Verwendung der Datei mit der Prozeßnummer an; folgender Befehl läßt den Master eine Statusausgabe erzeugen:

```
kill -INT `cat master.pid`
```

Wenn einer der Parameter `-e` oder `-k` angegeben wird, wird der Master nicht als Server gestartet, sondern führt lediglich gewisse administrative Tätigkeiten durch und terminiert dann wieder. In diesen Betriebsarten ist die Angabe der übrigen optionalen Parameter nicht erlaubt.

Mit dem Parameter `-e` instruiert man den Master zu überprüfen, ob für alle in der Konfigurationsdatei eingetragenen Slaves auch Schlüssel vorhanden sind. Fehlende Schlüssel können eingegeben und vorhandene Schlüssel korrigiert werden.

Der Parameter `-k` instruiert den Master, einen bereits laufenden Masterserverprozeß zu terminieren.

Für das Slaveprogramm `slave` gilt alles oben Gesagte analog.

5.4. Erfahrungen, Bewertung und Ausblick

Das hier entwickelte System zur netzwerkweiten Verwaltung von Systemdateien in einem lokalen Netz von UNIX-Rechnern hat viele Stärken, jedoch sind an einigen Stellen auch Verbesserungen denkbar. Die Schwächen des Systems sind unterschiedlicher Natur. Zum einen gibt es Schwächen in der Implementierung, zum anderen Schwächen in der Konzeption.

5.4.1. Stärken

Die meisten der ursprünglich gestellten Anforderungen und Wünsche an das System konnten erfüllt werden. Bei diversen Punkten konnte sogar eine nahezu optimale Realisierung erreicht werden, die sich in Tests als ausgesprochene Stärken des Systems herausgestellt haben:

- *Funktionalität*
  Die in Kap. 2.1 geforderte Funktionalität des Systems konnte vollständig realisiert werden. Das Programmsystem erlaubt nach einmaliger Installation eine Remote-Administration aller konfigurierten Rechner und Systemdateien. Die Möglichkeiten zur Steuerung des Datenflusses zwischen Master und Slaves übertreffen bereits die ursprünglichen Forderungen und könnten ohne Probleme noch erweitert werden.
- *Sicherheitsanforderungen*
  Ein wesentlicher Punkt waren die in Kap. 2.2 gestellten Sicherheitsanforderungen an das System, d. h. die Absicherung der einzelnen Rechner gegen unberechtigten Zugriff über das Netz, die sich aus der Einrichtung der Remote-Administration ergeben. Teilprobleme sind dabei:
  - *Authentifizierung der Rechner und Prozesse*
    Mittels eines im Modell in Kap. 4 entwickelten Authentifizierungssystems kann sichergestellt werden, daß Master- und Slaverechner sich eindeutig identifizieren können. Der Codierungsschlüssel, der zur Authentifizierung zwischen Master- und Slaveserverprozeß verwendet wird, ist ausschließlich dem jeweiligen Prozeß bekannt. Da dieser Schlüssel nur codiert aufbewahrt wird und nicht selbst weitergegeben wird, ist es ausgeschlossen, daß ein dritter Prozeß diesen Schlüssel zu seiner eigenen Authentifizierung verwenden könnte. Das in Kap. 4 entwickelte Modell schließt ferner den Mißbrauch von Replays zur Authentifizierung aus.
  - *Verschlüsselung sicherheitsrelevanter Daten*
    Das Kommunikationsmodell aus Kap. 4 sieht vor, daß Systemdateien auf Wunsch ausschließlich verschlüsselt übertragen werden. Dabei garantiert das in der Implementierung verwendete DES-CBC-Verfahren mit neuen Schlüsseln für jede Dateicodierung für eine ausreichende Sicherheit.
  - *Begrenzung der Authentifizierung*
    Die verwendete Schlüsselhierarchie sorgt dafür, das Schlüssel – und damit verbunden die Authentifizierung – nur eine bestimmte Zeit gültig sind. Es bestünde zwar die Möglichkeit, zusätzlich zum Tausch beim Serverstart die B-Schlüssel auch noch in regelmäßigen Zeitabständen automatisch zu tauschen, um so die Sicherheit des Systems noch weiter zu erhöhen. Ob dies jedoch nötig ist oder nicht kann höchstens durch statistische Aussagen belegt werden (vgl. Needham und Schröder, [16]).
  - *Sind durch das System neue Sicherheitslücken entstanden?*
    Es ist natürlich nicht möglich zu zeigen, daß durch das Programmsystem keine neuen Sicherheitslücken geschaffen wurden. Andererseits ist es nun offensichtlich möglich, Sicherheitslücken, die bisher *zwangsläufig* geschaffen wurden, wenn eine oder mehrere Systemdateien zentral verwaltet werden sollten, zu umgehen.

- *Konsistenz*
  In Kap. 2.3 wurden eine Reihe von Konsistenzforderungen gestellt. Die meisten der dort geforderten Punkte wurden vollständig erfüllt. Einige wenige sind teilweise aus prizipiellen Gründen nicht vollständig realisierbar (vgl. Kap. 5.4.3).
  - *Ausfall eines Rechners*
    Es war gefordert worden, daß der Ausfall einzelner Rechner nicht den lokalen Betrieb der anderen Rechner im Netz gefährden darf. Dies ist vollständig gewährleistet dadurch, daß jeder Rechner unabängig von den anderen auf seinen lokalen Kopien arbeiten kann. Nach dem Wiederaufsetzen des ausgefallenen Serverprogrammes werden automatisch alle Datentransfers nachgeholt.
  - *Ausfall des Netzes*
    Für die Konsistenz im Falle eines Netzzusammenbruches gilt analog das zum Ausfall eines Rechners Gesagte.
  - *Fehlerhafte Systemdateien*
    Es war gefordert worden, daß sich Fehler in lokalen Systemdateien nicht auf andere Rechner fortpflanzen dürfen.
    Soweit sich diese Forderung auf die Syntax der Systemdateien bezieht, ist sie erfüllt. Die lokalen Konvertierungsprogramme, die das lokale Format in das netzglobale Format konvertieren, führen implizit eine Syntaxüberprüfung durch. Im Fehlerfalle unterbleibt ein Transfer und es wird eine entsprechende Fehlermeldung generiert.
    Über die Syntax hinausgehende Konsistenzprüfungen unterscheiden sich von Systemdatei zu Systemdatei sehr stark. Da aber die lokalen Konvertierungsprogramme leicht für jede einzelne Systemdatei modifiziert werden kann, ist eine Anpassung – auch für vergleichsweise komplizierte Überprüfungen – ohne Probleme möglich.
  - *Fehlerhafte Konfiguration*
    Eine Forderung in Kap. 2.3 lautete, daß fehlerhafte Konfigurationen von einzelnen Servern nicht zum fehlerhaften Betrieb des Servers führen dürfen. Für syntaktische Fehler ist diese Forderung vollständig erfüllt. Zusätzlich werden auch viele logische Fehler erkannt und abgefangen.
- *Flexibilität*
  Verschiedene Forderungen wurden in Kap. 2.4 unter dem Punkt Flexibilität zusammengefaßt. Die meisten dieser Forderungen konnten erfüllt werden:
  - *Portabilität*
    An vielen Stellen in der Entwicklung des Programmsystems wurde ein besonderes Augenmerk auf die Portabilität gerichtet, da ein solches System nur sinnvoll eingesetzt werden kann, wenn es leicht auf alle Rechner portierbar ist. Daß diese Bemühungen erfolgreich waren, zeigte sich deutlich in der Testphase: das Programmsystem konnte auf über ein Dutzend Rechner mit teilweise recht unterschiedlichen Hardware-/Softwarekonfigurationen nahezu problemlos portiert werden. Es kann also davon ausgegangen werden, daß auch weitere Portierungen leicht möglich sind. Somit wurde diese Anforderung voll erfüllt.
  - *Konfigurierbarkeit der Datenverteilung*
    Eine Forderung lautete, daß eine umfassende Konfigurierbarkeit für die Angabe, welche Daten von welchen Rechnern wohin transferiert werden sollen, vorhanden sein muß. Hier stellt das System eine sehr differerenzierte Spezifikation von Weitergabebedingungen zur Verfügung. Insbesondere macht die Möglichkeit der Rechnergruppierung das System flexibel nutzbar. Es setzt sich hier deutlich vom Yellow Pages Service (Kap. 3.3) ab, bei dem zwar auch eine Rechnergruppierung (Domain) vorgesehen ist, diese Gruppierung aber nur Einschränkungen mit sich bringt.

5.4.2. Implementationsbedingte Schwächen

Einige Schwächen des Systems ergeben sich aus der Form der Implementierung. Diese Schwächen könnten durch Einsatz eines größeren Zeitaufwandes i. a. ohne Schwierigkeiten behoben werden. Nachfolgend werden einige dieser Probleme aufgeführt und erläutert:

- *Dateispezifikation ist zu primitiv*
  Die Spezifikation der Dateistruktur, die in den Konfigurationsdateien von Master und Slave vorgenommen werden kann, schränkt die Anwendungsmöglichenkeiten des Systems ein. Es können nur Dateien mit völlig regelmäßigem Aufbau bedandelt werden. Dateien mit aufwendigerer Struktur werden nicht berücksichtigt. Dies Problem wurde teilweise dadurch abgemildert, daß Dateien auch als unstrukturiert gekennzeichnet werden können, allerdings gehen auf diese Weise einige Verwendungsmöglichkeiten verloren. Auch die Definitionsmöglichkeiten für Schlüsselattribute könnten umfangreicher sein.
- *Transferdefinition ist zu primitiv*
  In der bestehenden Version ist nur der Transfer vollständiger Datensätze vorgesehen. Für verschiedene Anwendungen wäre es jedoch sinnvoll, auch nur einzelne Attribute zu transferieren, während die übrigen Attribute auf den verschiedenen Rechnern unterschiedliche Werte behalten können. Auch die Angabe der Transferbedingungen könnte deutlich umfangreicher sein. So ist z. B. eine Einschränkung des Änderungsverbotes auf einzelne Attribute denkbar.
- *B-Schlüssel in regelmäßigen Abständen ändern*
  Die B-Schlüssel werden nur beim Systemstart geändert. Die Sicherheit des Systems könnte erhöht werden, indem die B-Schlüssel in regelmäßigen Abständen geändert würden.
- *Editor für globale Daten*
  Wie in Kap. 4.1 und 5.2.6 erläutert, sollte es einen Editor geben, der die Modifikation der globalen Daten auf dem Master erleichtert.
- *Synchronisation beim Editieren*
  Für die Bearbeitung der globalen und lokalen Systemdateien sollte eine Synchronisation mit dem jeweiligen Server (also Master resp. Slave) existieren, die verhindert, daß manuelle Änderungen an den Systemdateien durch Übertragungen vom Netz überschrieben werden bzw. diese überschreiben.
- *differenzierter Transfer*
  Eine differenziertere Auswahl der zu übertragenden Daten auf dem Master könnte die vom System verursachte Netzbelastung deutlich vermindern. Nach einer manuellen Änderung an einer globalen Systemdatei wird diese derzeit an alle partizipierenden Rechner übertragen, unabhängig davon ob diese Rechner von der Änderung betroffen sind oder nicht. Es wäre auch möglich, nur die jeweils geänderten Datensätze zu übertragen.
  Alle diese Ansätze würden zwar die Netzbelastung senken, dafür aber den Rechenaufwand auf Master und Slaves erheblich erhöhen. Anders liegt der Fall bei gesicherten Übertragungen. Hier würde – sofern keine Hardwareunterstützung zur Verschlüsselung vorhanden ist – der durch die Differenzierung entstandene Rechenaufwand wahrscheinlich leicht durch die Einsparungen bei der Verschlüsselung wettgemacht, sofern die verwalteten Systemdateien groß im Vergleich zu den üblicherweise durchgeführten Änderungen sind.
  Hier müßte also im Einzelfall geklärt werden, in welche Richtung optimiert werden soll und wie eine solche Optimierung zu erreichen ist.
- *Änderungsüberwachung aufwendig*
  Master und Slave testen in regelmäßigen Abständen die globalen bzw. lokalen Systemdateien auf Änderungen (durch Überprüfung des Modifikationsdatums der Datei). Dies ist entweder sehr aufwendig (wenn die Abstände der Tests kurz gehalten werden) oder (bei langen Zeitabständen) unbequem und möglicherweile sogar gefährlich wegen der auftretenden inkonsistenten

Datenbestände.
Auf dem Master ließe sich dieses Problem mit dem oben erwähnten Editor beseitigen. Wenn ausschließlich dieser eine Editor zur Modifikation der Systemdateien herangezogen wird, kann er Dateiänderungen dem Master melden und ihn zu einer Übertragung veranlassen.
Auf den Slaves ist dies Problem wohl eher konzeptioneller Natur: i. a. können Systemdateien von ganz unterschiedlichen Systemprogrammen oder dem Betriebssystem modifiziert werden, so daß hier keine andere Lösung greifbar scheint, die ohne größere Modifikationen an der Betriebssystemsoftware auskommt.

- *Weitergabe des 'Secure'-Attributs ungenügend gelöst*
  Wenn ein Slave beim Master einen gesicherten Transfer für eine bestimmte Systemdatei anfordert, sollte dies eigentlich allen Slaves mitgeteilt werden, die diese Systemdatei betrifft. In der bisherigen Implementierung ist dies nicht der Fall: das Secure-Attribut wird nur an diejenigen Slaves weitergegeben, die später das Dateiformat anfordern, alle anderen bleiben unberücksichtigt. Grund dafür ist, daß sich der Master bezüglich der Verteilung der Dateiformate bisher passiv verhält und diese Modifikation ein zusätzliches Protokoll mit entsprechender Synchronisation zu den anderen Übertragungen nötig wäre.
- *Init-Request*
  Bisher verhalten sich Slaves bei der Kontaktaufnahme zum Master passiv. Eine schnellerer Verbindungsaufbau wäre möglich, wenn sich die Slaves selbständig beim Master melden würden.
- *Konfigurationsänderungen werden nur ungenügend betrachtet*
  Konfigurationsänderungen des Masters in einem bereits aktiven System werden derzeit nur schlecht behandelt. Unproblematisch sind Änderungen an Dateidefinitionen von Dateien, die mit `INITDEL` markiert sind, sowie das Hinzufügen von Slaves.
  Probleme treten dagegen auf, wenn die Reihenfolge der Slaves in der Rechnerdefinition geändert wird oder eine Änderung der Dateidefinition auf dem Master durchgeführt wird und die betroffenden Globaldateien bereits existieren und nicht das Attribut `INITDEL` tragen. Die in diesen Dateien enthaltenden Daten sind dann unbrauchbar, weil die Zuordnung der Datensätze zu den Slaves sowie die Attributzugehörigkeit der Datensatzfelder nur aus der Konfigurationsdatei hätte entnommen werden können.
- *Ausführen von Update-Aktionen auf den Slaves*
  Es wäre schön, wenn auf dem Slave unter bestimmten Voraussetzungen automatisch zusätzliche Aktionen eingeleitet werden könnten. Manche Änderungen an Systemdateien, die möglicherweise vom Master angeordnet werden, sind nur sinnvoll, wenn noch weitere Schritte ausgeführt werden.
  So ist ein neuer Eintrag in der `passwd`-Datei nur sinnvoll, wenn auch ein Home-Directory für diesen neuen Benutzer angelegt wird, und eine Änderung der Konfigurationsdateien für diverse UNIX-Systemprogramme (z. B. `/usr/lib/aliases` für `sendmail` oder `/etc/servers` für `inetd`), werden erst aktiv, wenn dem jeweiligen Prozeß per Signal eine entsprechende Meldung gemacht wird.
- *Überprüfung der Konfigurationsdateien*
  Sowohl die syntaktische als auch die semantische Überprüfung der Konfigurationsdateien sind unzureichend gelöst. Der Parser bricht bei syntaktischen Fehlern normalerweise sofort mit einer wenig aussagekräftigen Fehlermeldung ab. Bei semantischen Fehlern ist das Verhalten i. a. besser.

5.4.3. Konzeptionelle und prinzipielle Schwächen

Einige Schwächen des Systems sind schon im Konzept enthalten und lassen sich nur schwer durch nachträgliche Modifikation beheben. Andere sind prinzipieller Natur; diese Probleme lassen sich ohne weiteres gar nicht beheben.

- *logische Fehler in den Konfigurationen*
  Ein großes Problem stellen die logischen Fehler dar. Sie können vom System nicht erkannt werden, bergen aber die Gefahr der Zerstörung von wesentlichen Systemdaten (vgl. Kap. 5.3.3). Dies scheint im wesentlichen ein konzeptionelles, wenn nicht sogar grundsätzliches Problem zu sein. Es wäre schwierig zu klären, in wieweit hier überhaupt eine Verbesserung möglich ist.
- *Sortierbarkeit der Systemdateien*
  Das Programmsystem nimmt an, daß Systemdateien nach gewissen Schlüsselattributen sortiert sind oder sortiert werden können; auf dieser Annahme beruht die Speicheroptimierung durch Zusammenfassung gleicher Datensätze von verschiedenen Rechnern.
  Es sind jedoch auch Systemdateien denkbar, deren Ordnung nicht durch ein einfaches Sortierkriterium angegeben werden kann. Solche Dateien könnten in der vorgegebenen Konzeption nur als ‘unstrukturiert’ spezifiziert werden, um so eine Umsortierung zu verhindern.
- *Rechnerbelastung und Performance*
  Durch die regelmäßig nötigen Tests der Änderungen von Systemdateien (vgl. oben), die ggf. nötigen Verschlüsselungen und die Dateiformatkonvertierungen werden die Rechner nennenswert belastet. Besonders auf leistungsschwacheren Rechnern kann dies zu Performance-Problemen führen. Das Konzept läßt hier jedoch keine Änderungen zu.

5.4.4. Möglichkeiten

Das vorgestellte System bietet noch viele Einsatzvarianten, die hier nicht angesprochen wurden oder deren Ausführbarkeit noch überprüft werden muß.

Eine solche Möglichkeit ist z. B. die Aufteilung der verwalteten Rechner auf zwei oder mehrere Master. Dies stellt noch eine Ausweitung des Gruppierugnskonzeptes dar und entspricht teilweise der Einführung mehrerer Domains bei Verwendung des Yellow Pages Service (Kap. 3.3).

Wenn sich die Geltungsbereiche der Master nicht überschneiden, ist die Realisierung problemlos. Im Gegensatz zum Yellow Pages *müssen* die Verwaltungsbereiche der verschiedenen Master aber nicht disjunkt sein.

Die einfachste Art der Überschneidung ist bereits ohne Änderung der bestehenden Programmversion realisierbar. Dazu müßte lediglich auf einem Master-Rechner A ein Slave-Server S_A laufen, welcher von einem zweiten Master-Server M_B auf einem anderen Rechner B bedient wird. Wenn der Slave-Server S_A für einen Austausch einer oder mehrerer der Globaldateien von M_A nach M_B sorgt, so ist eine paritielle Überlappung der Verwaltungsbereiche bereits gegeben.

Eine andere Art der Realisierung erfordert, daß die in der aktuellen Version des Programmsystems festgelegte, starre Bindung von Master und Slave an bestimmte Portnummern aufgehoben wird. Dann können auf einem Rechner zwei verschiedene Slave-Server laufen, die jeweils unterschiedliche Dateien an unterschiedliche Master-Server weitergeben.

6. Anhang

6.1. Literatur

Auf die nachfolgend genannte Literatur wurde bei der Erstellung dieser Arbeit zurückgegriffen. Auf viele der Quellen wurde auch bereits im Text gesondert verwiesen.

[1] AT&T (Hrsg.)
The UNIX System Users Manual
Engelwood Cliffs, 1986

[2] Bryant, Bill
Designing an Authentication System: a Dialogue in Four Scenes
Project Athena, Massachusetts Institute of Technology
Cambridge, Massachusetts, Februar 1988

[3] Davies, D. W.; Prince, W. L.
Security for Computer Networks
Wiley & Sons Ltd., Chichester, 2nd ed. 1989

[4] Diffie, W.; Hellman, M. E.
"New directions in cryptography"
IEEE Transactions on Information Theory, IT-22, No. 6, pp. 644 - 654, November 1976

[5] ISO (Hrsg.)
Information Processing – Open Systems Interconnection – Basic Reference Model
International Standard ISO 7498
International Organization for Standardization, 1984 (1977)

[6] ISO (Hrsg.)
Information Processing – OSI Reference Model – Part II: Security Architecture
International Standard ISO 7498/2
International Organization for Standardization, Genf 1988

[7] ISO (Hrsg.)
Information Processing – Modes of operation for a 64-bit block cipher algorithm
International Standard ISO 8372
International Organization for Standardization, Genf 1987

[8] Kent, S.; Linn, J.
Privacy Enhancement for Internet Electronic Mail:
Part II: Certificate-Based Key Management
DARPA Networking Group Report RFC-1114
IAB Privacy Task Force, August 1989

[9] Leiss, Ernst L.
Principles of Data Security
Plenum Press, New York 1982

[10] Linn, J.
Privacy Enhancement for Internet Electronic Mail:
Part I: Message Encipherment and Authentication Procedures
DARPA Networking Group Report RFC-1040
IAB Privacy Task Force, Januar 1988

[11] Linn, J.
Privacy Enhancement for Internet Electronic Mail:
Part I: Message Encipherment and Authentication Procedures
DARPA Networking Group Report RFC-1113
IAB Privacy Task Force, August 1989

[12] Linn, J.
Privacy Enhancement for Internet Electronic Mail:
Part III: Algorithms, Modes, and Identifiers
DARPA Networking Group Report RFC-1115
IAB Privacy Task Force, August 1989

[13] Mills, D. L.
A Distributed-Protocol Authentication Scheme
DARPA Networking Group Report RFC-1004
University of Delaware, April 1987

[14] National Bureau of Standards (Hrsg.)
"Data Encryption Standard"
Federal Information Processing Standards Publication 46
Januar 1977

[15] National Bureau of Standards (Hrsg.)
"DES Modes of Operation"
Federal Information Processing Standards Publication 81
US Department of Commerce, Dezember 1980

[16] Needham, R. M.; Schröder, M. D.
"Using Encryption for Authentication in Large Networks of Computers"
Communications of the ACM, Vol. 21, No. 12, pp. 993 - 999, Dezember 1978

[17] Neuman, B. Clifford; Steiner, Jennifer G.
Authentication of Unknown Entities on an Insecure Network of Untrusted Workstations
Project Athena, Massachusetts Institute of Technology
Cambridge, Massachusetts (o. J.)

[18] Quantum GmbH (Hrsg.)
qdb – Das Mega-Datenbankinterface
Dortmund 1988

[19] Rivest, R. L.; Shamir, A.; Adleman, L.
"A method of obtaining digital signatures and public key cryptosystems"
Communications of the ACM, Vol. 21, No. 2, pp. 120 - 126
Februar 1978

[20] Steiner, Jennifer G.; Neuman, B. Clifford; Schiller, Jeffrey
Kerberos: An Authentication Service for Open Network Systems
Project Athena, Massachusetts Institute of Technology
Cambridge, Massachusetts, März 1988

[21] Sun Microsystems Inc. (Hrsg.)
"Sun Network Services"
from "System Administration for the Sun Workstation", pp. 7 - 65
Mountain View, 1986

[22] Sun Microsystems Inc. (Hrsg.)
Networking on the Sun Workstation
Mountain View, 1986

[23] Sun Microsystems Inc. (Hrsg.)
"Secure Networking" from RPC 4.0 Source Distribution
Mountain View, August 1988

[24] Sun Microsystems Inc. (Hrsg.)
RPC: Remote Procedure Call Protocol Specification, Version 2
DARPA Networking Group Report RFC-1057
Juni 1988

[25] Sun Microsystems Inc. (Hrsg.)
Remote Procedure Call Programming Guide
Mountain View, Februar 1986

[26] Sun Microsystems Inc. (Hrsg.)
Commands Reference Manual
Mountain View, Februar 1986

[27] Sun Microsystems Inc. (Hrsg.)
Unix Interface Reference Manual
Mountain View, Februar 1986

6.2. Glossar

Big Endian
: bezeichnet eine spezielle Art der Byteorganisation eines Rechners: in Wort- und Langwortwerten steht das höchstwertigste Byte an der niedrigsten Adresse

BSD
: *Berkeley Software Distribution*, bezeichnet in diesem Kontext die verschiedenen Versionen der an der University of California in Berkeley erstellten Variante des → *UNIX*-Betriebssystems.

DEA
: → *DES*

DES
: Der *Data Encryption Standard*, ein vom amerikanischen Department of Defense eingeführter Standard-Verschlüsselungsalgorithmus; patentiert im Jahre 1975, zum nationalen Standard erhoben 1976 und veröffentlicht 1977 [7]. Er wurde später auch als ANSI X3.92-1981 vom American National Standards Institute adoptiert, dort ist er unter dem Namen *DEA-1 (Data Encryption Algorithm)* bekannt. → Kap. 3.2.3

Domain
: bezeichnet i. a. einen Geltungsbereich. Im Zusammenhang mit Kommunikation sind hier interessant → *Internet*-Domains, welche Rechner im Internet hierarchisch gruppieren um so eine Adressierung leicht möglich zu machen. Ein anderer Punkt ist der → *YP*-Domain, welcher eine Gruppe von Rechnern zusammenfaßt, die von einem gemeinsamen Master-Server bedient wird. → Kap. 3.3

Internet
: Ein Rechnerverbundnetz, welches sich ausgehend von den U.S.A. mittlerweile über nahezu die ganze Welt ausgedehnt hat. Viele der heutigen in → *LAN*s und → *WAN*s verwendeten Standardprotokolle wurden im Internet entwickelt und erprobt.

IP
: Das *Internet Protocol* ist ein Netzwerkprotokoll und deckt in Verbindung mit TCP oder UDP die Schichten 5 bis 7 des ISO-OSI-Kommunikationsmodells ab.

ISO
: Die *International Organization for Standardization*, eine internationale Normierungsorganisation.

Kernel
: bezeichnet den Basis des → *UNIX*-Betriebssystems. Der Kernel umfaßt eine Sammlung von Systemservices, die die Resourcen des Systems für die verschiedenen Applikationsprogramme verwalten.

LAN
: *Local Area Network*, lokales Netz; bezeichnet ein kleineres, hausinternes oder grundstückgebundenes Rechnernetz mit Leitungslängen bis ca. 1 km

Little Endian
: bezeichnet eine spezielle Art der Byteorganisation eines Rechners: in Wort- und Langwortwerten steht das niederwertigste Byte an der niedrigsten Adresse

NFS — Das *Network File System*, ein von der Firma Sun Microsystems entwickelter Netzwerkservice, der es ermöglicht, Filesysteme oder Teile von Filesystemen zu exportieren, so daß auf sie von einem oder mehreren Rechnern im Netz so zugegriffen werden kann, als ob sie lokal installiert wären.

Notationen — Nachfolgend eine kurze Zusammenfassung der Semantik der in den Kapiteln 3 und 4 verwendeten Notationen für Kommunikationsprotokolle:

A	Rechner mit Namen A (analog für B)
$A_{A,S}$	Ein Authentifikator für den Rechner A gegenüber dem Service S
Adr_A	Die Rechneradresse des Rechners A
$A \rightarrow B$: x	Rechner A sendet einen Datenblock mit Aufbau x an Rechner B
Δt	Eine Zeitspanne
F_A	Ein vom Rechner A berechneter Funktionswert (analog für B)
I_A	Identifikator des Rechners A, eine willkürlich gewählte Zahl, die zur Identifikation verwendet wird. (analog für B; auch mit Ziffernindex)
ID	Ein Identifikator, der statt des Rechnernamens zur Identifikation verwendet wird
K_A	Geheimer Schlüssel des Rechners A (analog für B, S und T)
$K_{A,S}$	Ein Konversationsschlüssel für die Rechner A und S
K_C	Ein Konversationsschlüssel
KS	Der Kerberos-Server, ein spezieller Authentifizierungsserver
K(x)	Paket x, codiert mit Schlüssel K
L_A	Eine Liste der für den Rechner A zugelassenen Kommunikationspartner
PK_A	Öffentlicher Schlüssel des Rechners A (analog für B, S und T)
S	Authentifizierungsserver oder Service mit Namen S (analog für T)
SK_A	Privater Schlüssel des Rechners A (analog für B, S und T)
t_A	Die aktuelle Uhrzeit des Rechners A (analog für B; auch mit Ziffernindex)
$T_{A,S}$	Ein Ticket für den Rechner A zur Benutzung des Service S
TGS	Der Ticket-Granting-Service, ein spezieller Service

OSI — *Open Systems Interconnection*, ein System von Standards der ISO, welches eine sinnvolle Kommunikation zwischen beliebigen, an einem Netzwerk angeschlossenen Rechnern ermöglichen soll.

Paketservice — bezeichnet einen paketorientierten Kommunikationsservice. Der Absender schickt Datenblöcke an den Empfänger, welche die Pakte in genau derselben Blockstruktur erhält. Paketservices werden auch *verbindungslos* genannt, im Gegensatz zu den verbindungsorientierten → *Streamservices*. Ein Beispiel für ein paketorientiertes Kommunikationsprotokoll ist das → *UDP*.

Replay — bezeichnet im Zusammenhang mit Netzwerksicherheit die Verwendung (d. h. die Neuaussendung) eines bereits zu einem frühren Zeitpunkt von einem Netzanwender versendeten Datenpaketes durch einen Dritten, um den ursprünglichen Empfänger zu verwirren und zu fehlerhaften Reaktionen zu veranlassen. → Kap. 3.1.1

RPC	Der *Remote Procedure Call* ist die von Sun Microsystems entwickelte Netzwerk-Variante eines Funktionsaufrufs. Im Gegensatz zum gewöhnlichen Funktionsaufruf wird die Funktion jedoch nicht auf der lokalen Maschine, sondern auf einem anderen Rechner im Netz ausgeführt. → Kap. 3.4
RSA	Ein nach den Autoren *Rivest*, *Shamir* und *Adleman* benanntes asymmetrisches Verschlüsselungsverfahren. → [19], → Kap. 3.2.3
Socket	ein Teil eines speziellen Kommunikationskanals unter → *UNIX*. In den meisten heutigen UNIX-Versionen existiert eine große Sammlung von Systemfunktionen, die die Nutzung von Sockets erlauben. Sockets sind universell verwendbar zur rechnerlokalen oder netzwerkweiten Kommunikation. Sockets sind beliebig programmierbar. Standardmäßig kann auf Sockets sowohl das → *UDP* als auch das → *TCP* verwendet werden, aber auch eigene Protokolle sind möglich.
Spoofing	bezeichnet den Versuch einer Person / eines Rechners / eines Programms sich durch Nachahmung des Verhaltens als jemand anderes auszugeben um so dessen (Zugriffs-)Rechte zuerkannt zu bekommen.
Streamservice	bezeichnet einen nicht-paketorientierten Kommunikationsservice. Der Absender schickt einen Datenstrom an den Empfänger, welcher ebenfalls einen Datenstrom empfängt. Eine eventuell vorhandene Blockstrukturierung im Sendestrom kann dabei verloren gehen. Paketservices werden auch *verbindungsorientiert* genannt, im Gegensatz zu den verbindungslosen → *Paketservices*. Ein Beispiel für ein verbindungsorientiertes Kommunikationsprotokoll ist das → *TCP*.
System V	Nachfolger des System III; beides sind Typbezeichnungen der AT&T-Versionen des → *UNIX*-Betriebssystems
TCP	Das *Internet Transmission Control Protocol*, ein streamorientiertes Netzwerkprotokoll, welches sich auf das → *Internet Protocol (IP)* abstützt.
UDP	Das *Internet User Datagram Protocol* ist ein paketorientiertes Protokoll für Netzwerkkommunikation, welches sich auf das → *Internet Protocol (IP)* abstützt.
UNIX	Warenzeichen der AT&T Bell Laboratories; Name für ein von AT&T entwickeltes multiuser- und multitasking-fähiges Betriebssystem. Im übertragenen Sinne auch verwendet für diverse Varianten des gleichen Betriebssystems von anderen Herstellern.
WAN	*Wide Area Network*, weiträumiges Netz; bezeichnet ein größeres, grundstückgrenzenüberschreitendes Rechnernetz mit Leitungslängen bis ca. 10 km
YP	Der *Yellow Pages Service*, ein von der Firma Sun Microsystems entwickelter Netzwerkservice, der dazu dient, Dateien von netzwerk- oder netzgruppenweitem Interesse von einem Server an alle Clients der entsprechenden Gruppe weiter zu verteilen. → Kap. 3.3

UNIVERSITÄT DORTMUND
Fachbereich Informatik

ERKLÄRUNG

Hiermit erkläre ich, daß ich die Diplomarbeit

Sicherheit in lokalen Netzen am Beispiel
einer netzwerkweiten Verwaltung von UNIX-Systemdateien

selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Dortmund, den 24.7.90, Thomas Omer[illegible]

(Unterschrift)