Universität Hamburg,
Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg

Studienarbeit

Authentisierung in Tycoon: Eine polymorphe Entwicklungsschnittstelle zu Kerberos

André Willomat

11. Juli 1996

Inhaltsverzeichnis

1. Einführung **4**

1.1 Sicherheit in offenen Netzen 4

1.1.1 Kryptographische Verfahren 6

1.1.2 Symmetrische und asymmetrische Verschlüsselung 7

1.1.3 Der Data Encryption Standard (DES) 9

1.1.4 Verschlüsselung nach Rivest, Shamir und Adleman (RSA) 9

1.1.5 Hash-Algorithmen 10

1.1.6 Blockverschlüsselung 11

1.1.7 Authentisierungsprotokolle 12

1.2 Das Kerberos System 14

1.2.1 Die Architektur von Kerberos 15

1.2.2 Authentisierungsprotokolle 16

1.3 Realisierung in Tycoon 19

1.3.1 Das Tycoon-System 20

1.3.2 Überblick: Authentisierung in Tycoon 21

2. Die Kommunikationsumgebung **23**

2.1 Sockets, Domänen und Protokolle 23

2.2 Kommunikation mit Sockets 24

2.3 Die Schnittstelle zur Kommunikationsumgebung 26

2.3.1 Adressierung im Internet 26

2.3.2 Die Netzwerkdatenbank 28

2.3.3 Die Schnittstelle zu Sockets 29

2.3.4 Nachrichten und Datenpakete 33

2.3.5 Beispielklient 35

3. Die Kerberos-Schnittstelle **37**

3.1 Bytearrays in der Schnittstelle zu C 37

3.2 Die DES Schnittstellen 39

3.3 Das Basisschnittstelle zu Kerberos 40

3.4 Verschlüsselung und Signierung 42

3.5 Kerberos Login und Ticket-Files 44

3.6 Authentisierung und Tickets 46

3.7 Client-/Server-Authentisierung 49
3.8 Beispiele der Kerberos-Bibliothek 51

4. Die Authentisierungsbibliothek **53**
4.1 Ort des Geschehens - die Domänen 54
4.2 Principals als Akteure im Domänenraum 55
4.3 Die Sicherheiten der Principals 57
4.4 Authentisierung von Principals 60
4.5 Signierung und Verschlüsselung von Daten 62
4.5.1 Datensignierung 63
4.5.2 Datenverschlüsselung 63
4.6 Die Bindung an Kerberos 64
4.7 Sichere Socket-Kommunikation in Tycoon 65

5. Zusammenfassung und Ausblick **68**

Zusammenfassung

Die zunehmende Bedeutung von *Online-Dienstleistungen* und deren Realisierung mittels offener Computernetzwerke stellt eine wachsende Herausforderung an die Sicherheit dieser Systeme dar. Die hier anfallenden Sicherheitsaspekte in Datenbank- und Kommunikationssystemen stehen der geforderten Flexibilität (die Systeme sollen *offen* bleiben) entgegen. Die Heterogenität der im Einsatz befindlichen, miteinander verbundenen Systeme bildet eine zusätzliche Erschwernis.

Die persistente Programmierumgebung **Tycoon** (*Typed Communicating Objects in Open Environments*) bietet mit ihrer polymorphen Programmiersprache höherer Ordnung **TL** (*Tycoon Language*) einen idealen architektonischen Rahmen für die Entwicklung abstrakter, generischer Dienste für offene Systeme. Damit erfüllt dieses System alle Anforderungen, um als Implementierungsbasis für leistungsfähige und gleichermaßen einfach zu benutzende Sicherheitsdienstleistungen zu dienen.

Das Ziel dieser Studienarbeit ist die Entwicklung einer Bibliothek für Sicherheitsdienstleistungen in Tycoon. Als technisches Fundament für Sicherheits- und Authentisierungsprotokolle soll das am MIT (*Massachusetts Institute of Technology*) entwickelte **Kerberos-System** dienen, wobei von den Möglichkeiten der Sprache TL Gebrauch gemacht wird, bestehende externe Dienste in das persistente Objektsystem Tycoon einzubinden. Der in Tycoon mögliche hohe Abstraktionsgrad soll schließlich dazu führen, daß das sichere, aber auch komplexe Authentisierungssystem Kerberos über eine einfache, leicht zu benutzende Entwicklungsschnittstelle zur Verfügung steht.

1. Einführung

In der heutigen Zeit ist ein zunehmendes öffentliches Interesse an Online-Diensten und weltweiten Computernetzwerken, wie *CompuServe* [LAUE95] oder *Internet* [KROL94] zu verzeichnen. Die Informationsbeschaffung oder der Meinungsaustausch, die Suche nach Software oder die Unterstützung in den verschiedensten Interessengebieten, die Planung und Buchung von Reisen und die Beanspruchung von Multimedia-Dienstleistungen (Videokonferenzen, interaktives Fernsehen (*Video on demand*), bis hin zur Abwicklung verschiedenster Geschäftsvorgänge (Einkauf, Bankgeschäfte (*Home banking*), Aktien- und Devisenmarkt, etc.) sind die zentralen Anforderungen, die von einer immer breiteren Masse der Bevölkerung an Kommunikations- und Multimedia-Anbieter gestellt werden.

Mit offenen, weltweit zugänglichen Informationssystemen und deren Integration in vorhandene Telekommunikationsinfrastrukturen (z.B. Telefon- oder Kabelnetze), sowie der Schaffung neuer Hochgeschwindigkeitsnetze [1] und dem Angebot immer neuer Dienstleistungen versuchen die Anbieter diesen Anforderungen in den **Information Superhighway** gerecht zu werden. Dabei erweist sich diese *Datenautobahn* zunehmend als ein lukratives Geschäftsfeld für die daran beteiligten Unternehmen [RESN94].

In diesem Kontext gewinnen Sicherheitsaspekte in offenen Computernetzen wie die **Geheimhaltung** (Verschlüsselung, z.B. von Kreditkartennummern) und **Unversehrtheit** von Daten (Zugriffskontrollen, Datenschutz) oder die **Authentisierung** (Echtheit, Rechtsgültigkeit, Glaubwürdigkeit) eine zunehmende Bedeutung.

1.1 Sicherheit in offenen Netzen

Das heute weltweit größte Computernetzwerk, das **Internet**, entstand in seiner Historie aus dem **ARPAnet**, einem militärischen Forschungsnetz des US-Verteidigungsministeriums, sowie in der weiteren Entwicklung aus dem **NSFNET** (*National Science Foundation*), einem Verbundsnetz verschiedener Forschungseinrichtungen [KROL94]. Bei diesen Netzen spielten gerade solche Aspekte wie *Autonomie* und *Offenheit* sowie die Möglichkeit der *unbehinderten Kommunikation* eine wesentliche Rolle. Das ARPAnet sollte beispielsweise auch Teilausfälle (z.B. durch Anschläge) verkraften können.

Aus diesen Anforderungen entstand als Mechanismus zur Datenübertragung das *Internet-Protokoll* (**IP**), bei dem Datenpakete unterschiedlicher Größe von einem Sender über

[1] Wie beispielsweise Glasfaser und FDDI (*Fibre Distributed Data Interface*) [HEIN94], oder ISDN (*Integrated Services Digital Network*) [GOLD92, MORI93, WIEH95].

Zwischenstationen, sogenannte *Router*, zum Empfänger gelangen [COME95, COME94, STEV92a]. Dieser Mechanismus tritt den jetzt zunehmend an Bedeutung gewinnenden Sicherheitsaspekten zunächst unvereinbar entgegen, denn beim Transport der Datenpakete durch das weltweite Netz läßt sich der Weg, den diese Pakete über die verschiedenen Zwischenstationen nehmen, nicht vorhersagen — so bleibt nur zu hoffen, daß an den auf der Route liegenden Rechnern nur vertrauenswürdige Personen sitzen, welche die Daten der Pakete nicht öffentlich machen (Geheimhaltung), den Inhalt der Pakete unverändert lassen (Unversehrtheit), und keine fingierten Pakete hinzufügen (Echtheit) ...

Dieser sorglose Umgang mit den Datenpaketen gewinnt zusätzlich dadurch an Bedeutung, daß sämtliche heute relevanten Kommunikationsanwendungen, wie beispielsweise die elektronische Post (*Electronic mail*), Nachrichtenbretter (*News*), das Dateiübertragunsprotokoll (*File Transfer Protocol*, **FTP**) oder das gerade in letzter Zeit immer stärker wachsende *World Wide Web* (**WWW**) auf diesem Internet-Protokoll aufsetzen. Bis sich dieser Zustand durch den globalen Ersatz der IP-Schicht mit einer neuen, sicheren Version verbessert, in der die Datenpakete verschlüsselt und authentisiert versendet werden [2], ist in diesem Gebiet ein Wettlauf zwischen neuen Sicherheitsmaßnahmen und den entsprechenden Angriffsmethoden entbrannt (analog zu Viren und Virenscannern).

Die aktuelle Struktur im Internet mit der vorhandenen IP-Schicht erlaubt Sicherheitsmaßnahmen auf folgenden Ebenen:

- Auf Ebene der IP-Schicht kann man unmittelbar die konzeptuelle Freiheit dieser Schicht einschränkt. Eine Möglichkeit hierfür besteht in dem Filtern der Datenpakete mittels sogenannter *Firewalls* [BELL94, CHAP95]. Innerhalb der durch diese Datenpaketfilter geschützten, bzw. geschlossenen Systeme kann über Zugangskontrollen (*Access path control*) dafür gesorgt werden, daß nur autorisiertes Personal Zugriff auf die sicherheitssensitiven Daten hat.

  Zu beachten ist, daß eine derartige Sicherheitsmaßnahme lediglich die Abschottung lokaler, z.B. firmeninterner Netzwerke erlaubt. Die Wirksamkeit der Datenpaketfilter sei hier einmal in Frage gestellt (*was* soll gefiltert werden; wird zuviel oder zuwenig gefiltert ?), flexiblere Sicherheitsmaßnahmen, wie beispielsweise Authentisierungen sind mit den Datenpaketfiltern nicht möglich.

- Auf Ebene der Anwendungsschicht können spezifische Sicherheitsmaßnahmen direkt in die Kommunikationsanwendungen einfließen, z.B. Signierungsmechanismen in Nachrichtensysteme, oder Verschlüsselungen in WWW-Navigatoren (*Browser*).

- Als *Middleware* zwischen IP- und Anwendungsschicht können anwendungsunabhängige, sichere Kommunikationsmechanismen angeboten werden, die aber noch auf der IP-Schicht als "Anwendung" aufsetzen. Beispiele hierfür sind der *Secure-*

[2] Version 6 des Internet Protokolls: *IP next generation*, entwickelt von der *Internet Engineering Task Force* (IETF) [ATKI95, IPNG]

RPC von Sun [RFC1057, RFC1094, SUNOS40] [3], oder der von Netscape entwickelte, und in deren WWW-Navigator eingesetzte *Secure Socket Layer* (**SSL**) [4].

Auch die in dieser Studienarbeit entwickelte Authentisierungsbibliothek mit anwendungsunabhängigen Sicherheitsdienstleistungen fällt in diese Ebene. Ein Beispielmodul dieser Bibliothek realisiert einen eigenen, einfachen *Secure Socket Layer* auf Basis des Kerberos Authentisierungssystemes (siehe Kapitel 4.7).

Kernpunkt aller Sicherheitsmaßnahmen, unabhängig von deren Anwendungsebene, bilden sichere Verfahren zur **Datenverschlüsselung** (folgende Kapitel). Erst mit Hilfe dieser Verfahren kann man die verschiedensten Aspekte einer sicheren Kommunikation, wie z.B. Echtheit, Unversehrtheit oder Nichtabstreitbarkeit realisieren, wobei hier dann **Sicherheitsprotokolle** (ab Kapitel 1.1.7) im Mittelpunkt stehen, die klären, wie man die Verfahren zur Datenverschlüsselung anwendet.

1.1.1 Kryptographische Verfahren

Sichere *kryptographische Verfahren* [SALO90, KAUF95] bilden die Grundlage für die Realisierung von Sicherheitsaspekten in Kommunikations- und Informationssystemen. Sie bestehen im Kern aus einem kryptographischen Algorithmus (welcher offen bekannt sein kann) und einem dazugehörigen, geheimen Wert, den **Schlüssel**. Mit dem Algorithmus und dem Schlüssel K kann man eine Verschlüsselungsfunktion e_K bilden, mit der lesbare Daten, der **Klartext** w (*Plaintext*), in eine unlesbare, verschlüsselte Form, den **Kryptotext** c (*Ciphertext*) gebracht wird (**Chiffrierung**):

$$e_K(w) = c$$

Mit Hilfe der ebenfalls aus dem Algorithmus und dem Schlüssel ableitbaren Entschlüsselungsfunktion d_K kann der Kryptotext wieder in die lesbare Form gebracht werden (**Dechiffrierung**):

$$d_K(c) = w$$

Es gilt stets $d_K(e_K(w)) = w$, und in diesem Sinn ist d_K eine inverse Funktion zu e_K. In einem guten kryptographischen Verfahren sind $e_K(w)$ und $d_K(c)$ für sich genommen jeweils leicht und kostengünstig zu berechnen; auch sollte der Kryptotext nicht viel größer als der Klartext sein. Dagegen sollte es unmöglich sein, ohne Kenntnis von d_K den Klartext aus dem Kryptotext zu dechiffrieren.

Man unterscheidet **Datenstromchiffrierer** (*stream cipher*), die aus einem Benutzerschlüssel einen langen Schlüssel bzw. Schlüsselstrom generieren, welcher mindestens genau so lang ist, wie der Klartext, und **Blockchiffrierer** (*block cipher*), die dagegen mit

[3] Viele Abhandlungen über das Internet, wie neue Protokolle oder Systemarchitekturen finden sich in einer Serie von Reports unter dem Titel *Request For Comments*, bzw. **RFC's**. Derartige RFC's findet man im Internet auf dem Server des *Internet Network Information Center* unter der Adresse `SRI-NIC.ARPA`.

[4] Informationen über die aktuelle SSL Spezifikation 3.0 findet man derzeit nur im *World Wide Web*, z.B. direkt bei Netscape [SSL30a, SSL30b]. Eine frei verfügbare Implementation von SSL, **SSLeay**, findet man unter [YOU95, YOU96]

relativ kleinen Schlüssellängen auskommen (z.B. 64 Bit) und den Klartext in entsprechend passende Blöcke aufteilen (siehe Kapitel 1.1.6). Die Blockchiffrierer liegen dabei immer hart an der Grenze zwischen möglichst hoher Sicherheit (lange Schlüssel) und einer möglichst guten Performance der Verfahren (kurze Schlüssel).

Zur Realisierung verschiedener Sicherheitsaspekte lassen sich kryptographische Verfahren im Rahmen verschiedener **kryptographischer Methoden** anwenden. Eine solche kryptographische Methode ist die **Datenverschlüsselung**, für die sich die kryptographischen Verfahren unmittelbar durch ihre Ver- und Entschlüsselungsfunktionen anbieten. Mit Hilfe der Datenverschlüsselung stellt man die *Geheimhaltung* von Daten sicher, die auf einer unsicheren Kommunikationsstrecke übertragen, oder auf einem unsicheren Medium abgespeichert werden sollen.

Die **Datensignierung** ist eine weitere kryptographische Methode, bei der mit einem allgemein bekannten Verfahren und einem geheimen Schlüssel aus einem öffentlich zugänglichen Text ein Wert generiert wird (der *Message Integrity Code* (MIC), siehe auch Kapitel 1.1.5), der den Text in möglichst eindeutiger Weise umschreibt. Eine solcher MIC stellt die *Originalität* der öffentlichen Daten sicher, da eine Manipulation der Daten die neuerliche Generierung des MIC notwending machen würde, was aber ohne den geheimen Schlüssel unmöglich ist [5].

Eine dritte kryptographische Methode ist die **Authentisierung**, die durch ein Sicherheitsprotokoll (siehe Kapitel 1.1.7) die *Glaubwürdigkeit* einer Person bescheinigt. Wie man noch sehen wird, muß diese Person dabei beweisen, daß sie das Wissen um ein Geheimnis besitzt, ohne jedoch dieses Geheimnis selbst (auf der unsicheren Kommunikationsstrecke) zu verraten.

1.1.2 Symmetrische und asymmetrische Verschlüsselung

Ein Problem bei den klassischen Verschlüsselungsverfahren (vor 1975) besteht darin, daß falls man die Verschlüsselungsfunktion e_K kennt, man relativ einfach d_K berechnen kann, was nicht zuletzt daran liegt, daß in diesen Verfahren sowohl d_K, wie auch e_K auf dem selben, *geheimen Schlüssel* (K) aufbauen. Diese Verfahren bezeichnet man deshalb auch oft als *symmetrische Verschlüsselungsverfahren*, oder, da man neben d_K auch stets e_K verschweigen mußte, **private-key Systeme**. Beispiele für solche Systeme sind DES (*Data Encryption Standard*) oder IDEA (*International Data Encryption Algorithm*).

Wesentlich flexibler sind neuere **public-key Systeme**, deren bekanntester Vertreter das von *Rivest*, *Shamir* und *Adleman* entwickelte (und nach deren Namen benannte) RSA-Verfahren ist. Bei solchen Systemen basieren die Verschlüsselungsfunktion e_{K_1} und die Entschlüsselungsfunktion d_{K_2} auf unterschiedlichen Schlüsseln, die jedoch in einem mathematischen Zusammenhang stehen (siehe unten). Diese Verfahren bezeichnet man auch als *asymmetrische Verschlüsselungsverfahren.*

Im Gegensatz zu private-key Systemen, in denen sich ja mindestens zwei Parteien das selbe Geheimnis (K) teilen müssen, besteht bei public-key Systemen keine Notwendigkeit in der Verteilung geheimer Schlüssel. Stattdessen erhält jeder Teilnehmer des

[5] Für eine Überprüfung der Integrität einer Nachricht ist der geheime Schlüssel nötig. Dies kann man aber auch flexibler lösen, wie man noch bei der Betrachtung der public-key Systeme sehen wird.

Systemes zwei Schlüssel, K_1 und K_2, von denen man einen Schlüssel geheim hält (den **private-key**), und den anderen Schlüssel veröffentlicht (den **public-key**). Mit einem derartigen System kann man eine ganze Reihe verschiedener Aufgaben aus dem Bereich der Datensicherheit, z.B. für die elektronische Post, lösen:

- Hält man K_2 geheim, und veröffentlicht K_1, ist man in der Situation, daß die Benutzer eines Computernetzwerkes eine geheim zu haltende Nachricht jederzeit mit e_{K_1} herstellen können. Ausschließlich der Benutzer, für den die Nachricht bestimmt ist, kann jetzt den Kryptotext mit d_{K_2} entschlüsseln, da (hoffentlich) nur er den privaten Schlüssel K_2 kennt.

- Hält man K_1 geheim, und veröffentlicht K_2, kann man in einem offenen Computernetzwerk Nachrichten signieren. Dabei fertigt man mit Hilfe des privaten Schlüssels K_1 eine *digitale Signatur* der Nachricht an. Im Gegensatz zum MIC (*Message Integrity Code*) benötigt man zur Kontrolle der digitalen Signatur nicht den selben Schlüssel. Statt dessen kann jeder die digitale Signatur mit Hilfe des öffentlichen Schlüssels K_2 kontrollieren, und damit die Originalität der Nachricht beweisen. Trotzdem ist niemand in der Lage, nach einer Manipulation der Nachricht eine neue Signatur anzufertigen, da K_1 nicht bekannt ist [6].

Diese Beispiele demonstrieren, warum public-key Systeme die Basis für Sicherheitssysteme aus den Bereichen der elektronischen Post bilden. Prominente Vertreter für solche Sicherheitssysteme sind **PEM** (*Privacy Enhanced Mail*), ein seit 1993 existierender Standard für eine sichere Kommunikation mit *electronic Mail* im Internet (siehe Reports RFC 1421 - RFC 1424 [LINN93, KENT93, BALE93, KALI93]), und **PGP** (*Pretty Good Privacy*), eine weit verbreitete Public-Domain Software von Philip Zimmerman (z.B. [STAL95]).

Aufgrund der einfachen Handhabung der Schlüssel bei der Generierung und Kontrolle der Signatur, werden public-key Systeme bevorzugt bei der *Datensignierung* eingesetzt. Der Nachteil der public-key Systeme ist ihr in der Regel schlechtes Laufzeitverhalten, symmetrische Verschlüsselungsverfahren arbeiten deutlich schneller und lassen sich oft sogar relativ einfach auf Hardware implementieren (z.B. *DES-Chips*). Deshalb bevorzugt man bei der *Datenverschlüsselung*, insbesondere bei großen Datenmengen private-key Systeme. Bei der *Authentisierung* finden beide Systeme gleichermaßen Anwendung, obwohl auch hier die public-key Systeme durch eine Vereinfachung der Sicherheitsprotokolle leichte Vorteile verbuchen.

Im Folgenden werden zwei kryptographische Verfahren vorgestellt, ein private-key Verfahren (DES) und ein public-key Verfahren (RSA). Dies kann nur im Überblick geschehen, eine detailliertere Ausführung, wie und warum diese Verfahren funktionieren, findet man z.B. in [KAUF95]. Dort finden sich auch Informationen zu den Themenkomplexen PGP, PEM und weiteren kryptographischen Verfahren.

[6] Eine solche Signatur sichert nicht nur die Integrität elektronischer Post. Sie könnte auch Programme absichern, um diese z.B. vor Virenbefall zu schützen (die Signatur würde im Falle einer Änderung am Programmobjekt ungültig werden). Dazu muß das verwendete Signierungssystem auch die Signierung von Binärdaten zulassen, was z.B. bei PGP der Fall ist.

1.1.3 Der Data Encryption Standard (DES)

Der *Data Encryption Standard* (**DES**) ist ein von IBM entwickeltes Verschlüsselungsverfahren. Es ist seit 1977 nationaler amerikanischer Verschlüsselungsstandard für nichtklassifizierte sensible Daten [DES77], die Zertifizierung wurde von einer amerikanischen Bundesbehörde des *Department of Commerce*, dem *National Institute for Standards and Technology* (NIST) Mitte 1993 um weitere fünf Jahre verlängert.

Dieses Verfahren ist das derzeit einzige standardisierte Verschlüsselungsverfahren und wird vor allem im Bereich von Banken und Versicherungen eingesetzt. Es bildet als kryptographisches Verschlüsselungsverfahren die Grundlagen im Kerberos-System, wo es im Rahmen von Authentisierungsprotokollen eingesetzt wird [7].

DES zählt zu den symmetrischen Blockchiffrierern mit einer Schlüssellänge von 56 Bit und einer Blocklänge von 64 Bit. Es ist ein private-Key Verfahren und benutzt zum Ver- bzw. Entschlüsseln der einzelnen 64-Bit großen Datenblöcke den selben Schlüssel. Es arbeitet intern in 16 Runden mit Substitutionen (Ersetzen einzelner Bitmuster über eine Funktionstabelle) und Permutationen (Position einzelner Bits im Datenblock vertauschen). Im Rahmen der Substitutionen werden die Daten in jeder Runde mit jeweils wechselnden 48-Bit Teilschlüsseln XOR-verknüpft. Die dafür notwendigen 16 Teilschlüssel werden vorab aus dem eigentlichen DES-Schlüssel abgeleitet.

Wie man an dieser Beschreibung vielleicht sehen kann, eignet sich dieser Algorithmus insbesondere für die Realisierung auf einer Hardware (DES-Chips), die entsprechende Softwareimplementierung ist jedoch nicht sehr effizient (ca. 300 KByte/s) [KAUF95]. Dagegen ist das Verfahren recht sicher, es sind allerdings außerhalb der USA Exportbestimmungen zu beachten (Exportgenehmigungspflicht bei bestimmten Schlüssellängen).

1.1.4 Verschlüsselung nach Rivest, Shamir und Adleman (RSA)

Während sich die private-key Verfahren durchweg in allen kryptographischen Methoden anwenden lassen, sind die public-key Verschlüsselungsverfahren oft schon spezifisch an bestimmte Methoden gebunden. Das public-key Verfahren **RSA** erlaubt zwar sowohl die Datenverschlüsselung, wie auch die Signierung von Daten, dagegen lassen sich aber die Algorithmen von El Gamal und das **DSS**-Verfahren (*Digital Signature Standard*) nur für die Signierung von Daten anwenden [KAUF95]. Das *Zero knowlege proof system*, ebenfalls ein public-key Verfahren, läßt sich sogar nur zur Authentisierung einsetzen.

Aufgrund der Vielseitigkeit ist das wahrscheinlich am meisten benutzte public-key System RSA, welches 1978 von den Autoren *Rivest*, *Shamir* und *Adleman* veröffentlicht und nach deren Namen benannt wurde [RSA78]. Die Schlüssellänge bei RSA ist flexibel und wird allein durch die notwendige Sicherheit (langer Schlüssel) bzw. der geforderten Effizienz (kurzer Schlüssel) bestimmt. Ein guter Kompromiß aus beiden ist bei einer Schlüssellänge von 512 Bit zu erreichen, wobei schon hier RSA deutlich mehr Zeit benötigt, als symmetrische Verschlüsselungsverfahren (z.B. DES oder IDEA).

[7] Die neue Kerberos Version 5 unterstützt auch die Einbindung anderer Verschlüsselungsverfahren als kryptographische Basis.

Mathematisch gesehen basiert RSA auf dem Faktorisierungsproblem: Es ist leicht, zwei Primzahlen p und q zufällig zu generieren und zu multiplizieren, um das Produkt $n = pq$ zu erhalten, aber es ist sehr schwer, p und q aus n zu berechnen, wenn n groß ist.

Zur Festlegung des privaten und des öffentlichen Schlüssels berechnet man zunächst $\phi(n)$, was sich aus $\phi(n) = (p-1)(q-1)$ ergibt [KAUF95]. Die Funktion $\phi(n)$ steht dabei für die Anzahl der zu einer natürlichen Zahl n teilerfremden Zahlen, die selbst nicht größer als n sind.

- Für den öffentlichen Schlüssel sucht man dann zunächst eine natürliche Zahl e, die zu $\phi(n)$ *relativ prim* ist. Mit dieser Zahl bildet man den öffentlichen Schlüssel $< e, n >$.
- Für den privaten Schlüssel berechnet man anschließend die natürliche Zahl d, die sich aus $1/(e \bmod \phi(n))$ ergibt. Diese bildet den privaten Schlüssel $< d, n >$.

Um einen Klartextblock m ($|m| < |n|$) mit dem öffentlichen Schlüssel zu chiffrieren, braucht man nur $c = m^e \bmod n$ zu berechnen. Nur mit dem privaten Schlüssel ist man jetzt in der Lage, den Kryptotextblock c wieder zu dechiffrieren: $m = c^d \bmod n$.

Ähnlich findet das Anfertigen einer Signatur s aus einem Klartextblock m ($|m| < |n|$) mit dem privaten Schlüssel statt: $s = m^d \bmod n$. Die Kontrolle der Signatur kann mit dem öffentlichen Schlüssel wie folgt durchgeführt werden: $m = s^e \bmod n$

Insgesamt gesehen ist RSA ein sehr sicheres Verschlüsselungsverfahren — solange die Faktorisierung großer Zahlen schwierig bleibt. In [KAUF95] wird beschrieben, daß das beste heute bekannte Verfahren zur Faktorisierung großer Zahlen bei 512 Bit so lange dauert, daß es effizienter ist, einige hundert Jahre zu warten, bis es ein besseres Faktorisierungsverfahren gibt.

Ein Problem bei RSA sind ähnlich wie bei DES Lizenzbestimmungen, die es außerhalb der Vereinigten Staaten einzuhalten gilt. Das Patent auf das RSA-Verfahren läuft allerdings am 20.10.2000 aus.

1.1.5 Hash-Algorithmen

Die bisher betrachteten kryptographischen Verfahren, insbesondere die private-key Systeme, basieren auf Algorithmen, die unmittelbar die Datenverschlüsselung zum Ziel haben. Immer wenn es um die Signatur von Daten ging, war bisher nur die Rede von *allgemein bekannten Verfahren*, die einen MIC (*Message Integrity Code*) oder eine *digitale Signatur* generieren.

Solche allgemein bekannten Verfahren sind beispielsweise die **sicheren Hash-Algorithmen** (*Message Digest* (MD) und *Message Authentication Check* (MAC)), die das Ziel haben, aus einem Klartext einen deutlich kürzeren und möglichst eindeutigen Wert zu generieren, den sogenannten **Hash-Wert**. Zu den heute weit verbreiteten Hash-Algorithmen zählt **MD5**, ein Verfahren, daß sich aus seinen Vorgängern MD2 und MD4 entwickelte [KALI92, RIVE92a, RIVE92b, KAUF95]. Dieses Verfahren generiert aus einem Klartext einen 128-Bit langen Hash-Wert, den *Message digest* (woraus sich auch die Bezeichnung MD für das Verfahren ableitet).

Der Kernpunkt dieser Verfahren besteht dabei darin, daß die zugrundeliegende Generatorfunktion e_K nicht umkehrbar ist — der generierte Wert läßt keinen Rückschluß auf den ursprünglichen Klartext zu. Er charakterisiert diesen Klartext in eindeutiger Weise, ohne dabei etwas über dessen Inhalt auszusagen, er stellt faktisch ein **Fingerabdruck** des Klartextes dar. Eine Funktion f, deren inverse Funktion f^{-1} faktisch nicht (oder nicht ohne zusätzliche Informationen) aus f berechenbar ist, nennt man auch eine **Einwegfunktion** (*One way function*).

Um die Integrität einer Nachricht zu garantieren, reicht jedoch die einfache Anwendung einer Einwegfunktion auf die Nachricht, um z.B. einen MIC zu generieren, nicht aus. Schließlich sind die Hash-Algorithmen, wie schon angedeutet wurde, *allgemein bekannt*, so daß es einem potentiellen Angreifer leicht fällt, zu einer manipulierten Nachricht einen neuen MIC zu generieren. Die Lösung besteht hier (ähnlich wie bei den kryptographischen Verfahren) in der kombinierten Anwendung des Hash-Algorithmus zusammen mit einem Schlüssel oder einem Paßwort.

Beispielsweise kann man die Nachricht vor Anwendung der Einwegfunktion einfach mit einem Paßwort verketten, um dann nur die Nachricht und den so generierten MIC auf die unsichere Kommunikationsstrecke zu bringen. Der Empfänger muß zwar jetzt zur Überprüfung der Integrität der Nachricht auch das Paßwort kennen, aber niemand kann jetzt ohne dieses Paßwort den neuen MIC einer manipulierten Nachricht herstellen.

Erfreulicher Weise unterliegen im Gegensatz zu den unterschiedlichen Lizenz- und Exportbestimmungen der bisher vorgestellten kryptographischen Verfahren, die sicheren Hash-Algorithmen keinerlei Beschränkungen.

1.1.6 Blockverschlüsselung

Um die Erörterung kryptographischer Verfahren abzuschließen, wird noch einmal folgendes Problem untersucht: Wie kann man mit Blockverschlüsselungsverfahren einen Klartext chiffrieren, der größer ist, als die vorhandene Schlüssellänge ?

Die Lösung dieses Problems scheint zunächst einfach: Man unterteilt den Klartext in Blöcke, die so groß sind wie der zur Verfügung stehende Schlüssel, wobei man den letzten evtl. kleineren Block mit Füllbits in der Größe angleicht. Dann chiffriert man jeden Block separat mit dem Schlüssel, und fügt die verschlüsselten Blöcke zum Kryptotext zusammen. Bei der Entschlüsselung geht man ebenfalls wieder mit dieser Aufteilung in einzelne Blöcke vor. Dieses Verfahren ist in der Praxis unter dem Namen **Electronic Code Book** (ECB) bekannt; es ist einfach und intuitiv in der Anwendung, aber bietet auch verschiedene Angriffsmöglichkeiten für den *Kryptoanalysten*.

Die Problematik liegt in der symmetrischen Blockstruktur von Kryptotext und Klartext. Enthält der Kryptotext zwei gleiche Blöcke, so kann man darauf schließen, daß die entsprechenden Blöcke im Klartext ebenfalls gleich sind. Dies kann zu Problemen führen, wenn der Kryptoanalyst über weitere Informationen verfügt: Besitzt er beispielsweise die Datenstruktur einer Datei, die alphabetisch sortiert Angestellte mit deren Gehältern enthält, könnte er trotz einer ECB-Verschlüsselung gewisse Rückschlüsse ziehen. Beispielsweise kann er auf die Gleichheit von Gehältern schließen. Wenn ihm zusätzlich Angestellte mit hohen Gehältern bekannt sind, kann er auch leicht die Gehälter der

Angestellten, die weniger verdienen (ohne Kenntnis der genauen Zahlen), entsprechend *anpassen*.

Um diese Angriffsmöglichkeiten zu verhindern, muß man die Symmetrie der Blockstruktur aufbrechen. Hierzu könnte man die Klartextblöcke vor der Verschlüsselung mit Zufallszahlen (deren Bitgröße der Blockgröße entsprechen) XOR-verknüpfen. Den Kryptotext bildet man dann aus Blöcken von Paaren, bestehend aus der Zufallszahl und dem dazugehörigen, verschlüsselten Block (die Zufallszahl braucht man nicht zu verschlüsseln). Jetzt werden gleiche Klartextblöcke zu unterschiedlichen Kryptotextblöcken, und die Analyse wird für den Kryptoanalysten ohne das Wissen um den verwendeten Schlüssel sehr schwer (da helfen ihm auch die lesbaren Zufallszahlen nicht weiter).

Es ist jedoch leicht die Problematik in diesem Verfahren zu erkennen: Die Effizienz leidet erheblich, da der Kryptotext in seiner Größe immer doppelt so groß ist, wie der Klartext. Außerdem sind die einzelnen Paare (Zufallszahl, Kryptoblock) trotz der jetzt bestehenden Asymmetrie immer noch untereinander unabhängig und damit für Vertauschungen anfällig.

Die Lösung bietet das CBC-Verfahren, das **Cipher Block Chaining**, mit einem einfachen Trick: Es benutzt nicht irgendwelche Zufallszahlen, sondern statt dessen den jeweils vorher generierten Kryptoblock. Jetzt werden also die einzelnen Klartextblöcke mit dem jeweils vorher generierten Kryptoblock XOR-verknüpft, und anschließend verschlüsselt. Es ist nur noch für die erste Verschlüsselung eine initiale Zufallszahl notwendig, der **Initialization vector**. Dieser Vektor wird zusammen mit dem ersten Kryptoblock in den Kryptotext integriert, so daß der gegenüber dem ECB-Verfahren nur um eine Blocklänge anwächst. Da die XOR-Funktion in der Regel kostengünstig ist, hat man beim CBC-Verfahren keinen besonders großen Effizienzverlust gegenüber dem ECB-Verfahren, man ist aber durch die asymmetrischen Abhängigkeiten vor einfachen Blockvertauschungen sicher.

Natürlich ergeben sich auch für das CBC-Verfahren eine Reihe von Angriffsmethoden zur Manipulation des Kryptotextes — auch wenn diese deutlich komplexer sind, als eine einfache Blockvertauschung. Deshalb gibt es auch noch weitere Verfahren, die an ECB und CBC anknüpfen, wie z.B. die **Output Feedback Mode** (OFB), oder **Cipher Feedback Mode** (CFB) Methoden. Diese und weitere Verfahren findet man in [DES81]. Sie beziehen sich dort zwar auf DES, sind aber auf alle Blockverschlüsselungsverfahren mit fester Schlüssellänge anwendbar.

1.1.7 Authentisierungsprotokolle

Nach der Betrachtung kryptographischer Methoden, wie die Verschlüsselung oder die Signierung von Daten, bleibt noch die Frage, wie man die kryptographischen Verfahren im Rahmen von Sicherheitsprotokollen einsetzen kann, um eine Authentisierung zwischen zwei Kommunikationspartnern realisieren zu können.

Beispielsweise möchte man mit einem private-key Verfahren in einem Netzwerk mit n Knoten eine Authentisierung betreiben, so daß sich jeder Knoten gegenüber jedem anderen Knoten authentisieren kann, und eine sichere, verschlüsselte Kommunikation untereinander möglich ist. Dazu müßte jeder Rechnerknoten $n - 1$ Schlüssel kennen,

insgesamt würde es also $n^2 - n$ geheime Schlüssel geben. Doch schon für die Einführung eines einzigen neuen Knotens müßten n neue Schlüssel generiert und verteilt werden. Dieser Aufwand ist besonders bei großen Computernetzen unhaltbar.

Eine Lösung für dieses Problem ist die Einführung eines absolut vertrauenswürdigen Knotens in diesem Netzwerk, der die Schlüssel aller anderen Knoten kennt und verwaltet, ein **Key Distribution Center** (KDC). In einem solchen Netzwerk wären zunächst nur $n - 1$ Schlüssel nötig, die alle im KDC hinterlegt sind. Beim Hinzufügen eines neuen Knotens, benötigt man zunächst nur einen neuen geheimen Schlüssel im KDC. Alle Knoten können in der Kommunikation mit dem KDC ihren geheimen Schlüssel benutzen, und mit ihm *sicher* Nachrichten austauschen. Möchten zwei Knoten untereinander in Verbindung treten, generiert das KDC einfach einen für beide Knoten temporär gültigen **Sitzungsschlüssel**.

Somit hat man mit dem KDC ein erheblich geringeres Schlüsselaufkommen, und ist einer pragmatischen Lösung für eine authentisierte Kommunikation nahe. Bleibt nur noch zu klären, mit welchen Protokoll die Knoten untereinander und mit dem KDC kommunizieren. Dabei möchte man beispielsweise Fälle vermeiden, in denen sich einfach ein Knoten a beim KDC anmeldet, um einen Sitzungsschlüssel für die Kommunikation zwischen den Knoten p und q zu beantragen.

Hier findet das klassische Authentisierungsprotokoll von **Needham** und **Schroeder** Anwendung [NEED78], welches ebenfalls auf einem *Key Distribution Center* zur Generierung temporärer Sitzungsschlüssel basiert. Dieses Protokoll findet in der Praxis eine große Verbreitung; viele Weiterentwicklungen fanden auf Basis dieses Protokolls statt. Auch das Kerberos-System benutzt dieses Protokoll (mit kleinen Erweiterungen).

Im Folgenden wird dieses Protokoll kurz vorgestellt. Für eine ausführlichere Diskussion, warum das Protokoll gegen verschiedene Angriffe sicher ist und welche Schwächen es dennoch hat, wird auf [KAUF95] verwiesen.

- Zunächst sendet der Knoten a an das KDC eine Nachricht, in dem er ihm mitteilt, daß er mit dem Knoten b kommunizieren möchte. Diese Nachricht ist um eine zufällig generierte Prüfsumme n_1 erweitert.

- Das KDC sendet an a eine Nachricht zurück, die komplett mit dem geheimen Schlüssel von a chiffriert ist. Diese Nachricht enthält n_1, den vom KDC generierten Sitzungsschlüssel für a und b, sowie ein **Ticket** für b, welches mit dessen geheimen Schlüssel chiffriert ist. Dieses Ticket enthält nochmals den Sitzungsschlüssel, und den Namen von a. Niemand außer b kann dieses Ticket dechiffrieren und damit kann niemand die Integrität dieses Tickets verletzen.

- Jetzt sendet a das Ticket und eine neue, mit dem Sitzungsschlüssel chiffrierte Prüfsumme n_2 an b.

- Mit Hilfe seines geheimen Schlüssels dechiffriert b das Ticket, bekommt damit den Sitzungsschlüssel, und sendet $n_2 + 1$ und eine dritte Prüfsumme n_3, alles mit dem Sitzungsschlüssel chiffriert, an a zurück.

- Wenn a von b die Antwort bekommt, und diese die korrekte Prüfsumme $n_2 + 1$ enthält, hat sich b gegenüber a authentisiert. Jetzt chiffriert a eine neue Nachricht bestehend aus $n_3 + 1$ mit dem Sitzungsschlüssel und sendet diese an b zurück.
- Wenn b von a die Antwort bekommt, und diese die korrekte Prüfsumme $n_3 + 1$ enthält, hat sich a gegenüber b authentisiert.

1.2 Das Kerberos System

Ein Forschungsprojekt mit dem Ziel, eine *sichere Netzwerkumgebung* zu schaffen, geht auf das vom MIT (*Massachusetts Institute of Technology*) 1983 gegründete Projekt **Athena** zurück. Das Ergebnis dieser Forschungen wurde auf der USENIX Winterkonferenz 1988 unter dem Namen **Kerberos** [8] vorgestellt [SNS88, NEUM87].

Die Basis von Kerberos bilden unabhängige, nicht nur an elektronischer Post gebundene Sicherheitsprotokolle, die eine generelle Client-/Server-Authentisierung in offenen Netzen ermöglichen. Das System erfüllt dabei die grundlegende Forderung, daß sich ein Teilnehmer des Systems nur einmal (zu Beginn einer Sitzung) zu identifizieren braucht, und daß Paßwörter nie im Klartext über das Netz gesendet werden. Das dabei zum Einsatz kommende Authentisierungsprotokoll geht auf das von Needham und Schroeder zurück [NEED78] (Kapitel 1.1.7). Als kryptographisches Verschlüsselungsverfahren für alle sensiblen Daten benutzt Kerberos den *Data Encryption Standard* DES (Kapitel 1.1.3), also ein symmetrisches *private-key* Verfahren.

Die initiale Identifikation eines Benutzers erfolgt über eine Instanz, die in einem gewissen Gültigkeitsbereich die Paßwörter aller Benutzer und Dienste kennt, und der alle Beteiligten trauen müssen (*Trusted third party*), dem **Authentication Server** (AS). Diese *vertrauenswürdige Instanz* sollte immer auf einer physisch sicheren Maschine untergebracht sein.

Nach der initialen Identifikation gegenüber dem AS kann ein Benutzer abgesicherte, authentisierte Dienstleistungen beanspruchen, indem er von Kerberos (auch in mehrfacher Folge) Berechtigungen, sogenannte **Tickets** anfordert. Dabei wird von Kerberos zusätzlich ein **Sitzungsschlüssel** generiert, der in der Kommunikation zwischen dem Diensterbringer und dem Benutzer, dem das Ticket gehört, eingesetzt werden kann. Dieses zusammengehörige Paar, Ticket und Sitzungsschlüssel, bezeichnet man in Kerberos als eine **Sicherheit** (*Credential*).

Für eine Authetisierung des Benutzers gegenüber dem Diensterbringer sind die Tickets immer nur zusammen mit einer Benutzeridentifikation, dem sogenannten **Authentifikator** des Benutzers gültig. Die eigentliche Authentisierung wird im Rahmen der Sicherheitsprotokolle von Kerberos dadurch garantiert, daß sowohl der Benutzer, wie auch der Diensterbringer, immer nur bestimmte Teile von Ticket und Authentifikator entschlüsseln kann. Dadurch wird eine Manipulation (verschiedene geheime Schlüssel, die alle gemeinsam nur die vertrauenswürdige Instanz kennt), eine Replikation (Prüfsum-

[8] Benannt nach dem dreiköpfigen Hund der griechischen Sagen, der den Eingang zur Unterwelt bewacht.

men, mit einem nur dem Benutzer und und dem Diensterbringer bekannten Sitzungsschlüssel chiffriert) oder Mißbrauch durch Dritte (die Absenderadresse und ein Zeitstempel sind verschlüsselter Bestandteil eines Tickets) verhindert. Neben der Authentisierung des Benutzers gegenüber dem Diensterbringer erlaubt das Kerberos-Protokoll auch die Authentisierung des Diensterbringers gegenüber dem Benutzer, so daß dieser sicher sein kann, daß die gewünschte Dienstleistung auch erbracht wurde.

Um sich bei der Beanspruchung mehrerer Dienste nicht immer wieder von neuem gegenüber dem AS identifizieren zu müssen, erhält der Benutzer bei der initialen Identifikation gegenüber dem AS ein Ticket für den **Ticket Granting Server** (TGS), das **Ticket Granting Ticket** (TGT). Mit diesem Ticket kann der Benutzer beim TGS weitere Tickets für Dienste anfordern, ohne jedes mal sein Paßwort eingeben zu müssen. Trotz dieser logischen Trennung zwischen AS und TGS haben beide Instanzen, wie wir noch sehen werden, eine ähnliche Funktionalität. Wie jedes andere Ticket auch, ist das TGT einer spezifischen Lebensdauer unterworfen — ist diese abgelaufen, ist das Ticket ungültig, und der Benutzer muß sich von neuem gegenüber dem AS identifizieren.

Obwohl Kerberos einer ständigen Weiterentwicklung am MIT unterworfen ist, hat das System auch heute noch mit einigen Schwächen zu kämpfen, z.B. die unbefriedigende Verfügbarkeit auf verschiedenen Plattformen (insbesondere PC's), oder der Verwendung von Zeitstempeln, die Tickets einer definierten Lebensdauer unterwerfen, und eine systemübergreifende Zeitsynchronisation notwendig machen. Es gibt auch noch einige Sicherheitslücken für potentielle Angriffe, die in der Fachwelt diskutiert werden (z.B. [BELL90]), es scheint aber nur eine Frage der Zeit, bis auch hier Abhilfe geschaffen ist.

So sind in der aktuellen Protokollversion 5 von Kerberos [KOHL93] einige der anfänglichen Mängel beseitigt worden. Kerberos besitzt mit dieser Version eine wohldefinierte Programmierschnittstelle (API — *Application Programmers Interface*), welche zudem auch noch *thread-safe* ist. Die Authentifikation über mehrere Gültigkeitsbereiche hinweg (*cross-realm*) ist in dieser Version besser unterstützt.

Insgesamt kann man Kerberos durchaus als ein stabiles, in der Praxis erprobtes Authentisierungssystem bezeichnen, welches sich teilweise sogar als *De-facto-Standard* etabliert (eingesetzt in DCE, X11-Fenstersystemen, `rlogin`, oder in neuen Versionen vom NFS). Aus diesen Gründen soll Kerberos auch das Fundament für die im Rahmen dieser Studienarbeit zu entwickelnde Authentisierungsbibliothek bilden.

1.2.1 Die Architektur von Kerberos

Kerberos besteht in seiner Architektur aus einer Reihe von Authentifikationsservern, die sich intern einer NDBM-Datenbank (ein Standardsystem unter UNIX) bedienen, und über eine Socket-Schnittstelle ansprechbar sind. Neben dem eigentlichem Kerberos-Server, dem *Authentication Server* (AS), ist auch der *Ticket Granting Server* (TGS) ein solcher Authentifikationsserver.

Sowohl der AS wie auch der TGS sind *vertrauenswürdige Instanzen* in einem Gebiet, dem **Realm**. Sie stellen in diesem Gebiet das *Key Distribution Center* (KDC) dar (Kapitel 1.1.7), und sind mit der Erzeugung und Verteilung von Sitzungsschlüsseln betraut (jedes Gebiet verfügt über ein eigenes KDC). Außerdem können die Authentifikations-

server innerhalb eines Gebietes Sicherheiten an alle in diesem Gebiet bekannten Teilnehmer, den **Principals**, vergeben. Principals können sowohl aktive Benutzer (*Klienten*), wie auch Prozesse, z.B. Diensterbringer (*Server*) sein.

Der AS und der TGS kennen als vertrauenswürdige Instanzen die geheimen Schlüssel aller in einem Gebiet registriert Principals. Die hierfür notwendigen Daten über die Principals befinden sich in der schon erwähnten NDBM-Datenbank. Während die Authentifikationsserver nur einen lesenden Zugriff auf diese Datenbank haben, gibt es auch noch **Administrationsserver** (KDBM-Server) in jedem Gebiet, von denen immer genau eine Instanz auf einem sicheren Basisrechner läuft, wo er Lese- und Schreibzugriff auf die Datenbank besitzt.

Für die Beanspruchung einer authentisierten Dienstleistung benötigt ein Principal ein Ticket, welches immer nur für genau einen Diensterbringer gültig ist. Ein solches Ticket enthält den Namen und die Netzwerkadresse des Principals, den Namen des Dienstes, für den es bestimmt ist, einen vom TGS oder AS zufällig gewählten Sitzungsschlüssel für die Kommunikation zwischen Principal und Diensterbringer, sowie eine Gültigkeitsdauer und einen Startzeitpunkt, die gemeinsam das Zeitfenster definieren, in dem das Ticket gültig ist.

Das Ticket bildet, wie schon angedeutet, zusammen mit dem Sitzungsschlüssel eine *Sicherheit*, von der ein Principal zu einem Zeitpunkt auch mehrere besitzen kann (für verschiedene Dienstleistungen). Alle Sicherheiten werden in dem **Ticket-File** des Principals verwaltet, oder, wie es in der neuen Version von Kerberos heißt, dem *Credential cache*.

Mit einem Authentifikator legt ein Principal seine Identität dar. Dieser besteht aus dem Namen und der Netzwerkadresse des Principals, sowie einem Startzeitpunkt für die Gültigkeit des Authentifikators. Diese Daten werden mit dem Sitzungsschlüssel chiffriert und sind im Gegensatz zu einem Ticket nicht wiederverwendbar. Der Authentifikator stellt damit sicher, daß ein Paar, bestehend aus Ticket und Authentifikator, nicht mehrfach benutzt werden kann.

1.2.2 Authentisierungsprotokolle

In Kerberos findet ein sicheres Authentisierungsprotokoll mit Hilfe der schon genannten Tickets statt. Dabei wird allerdings noch keine **Autorisierung** betrieben, durch die festgestellt wird, ob ein Principal einen Dienst benutzen darf — es wird lediglich die Authentizität der beteiligten Principals sichergestellt. Dieses Authentisierungsprotokoll besteht in all seinen Variationen aus den beiden folgenden grundlegenden Schritten:

- Durch Präsentation eines Geheimnisses (z.B. ein Paßwort) bei Kerberos ein Ticket für den gewünschten Dienst beschaffen.
- Sich gegenüber dem Diensterbringer mit Hilfe dieses Tickets für den gewünschten Dienst authentisieren.

Am Anfang besitzt ein Benutzer noch kein Ticket. Er muß also immer dann, wenn er ein Ticket für eine Dienstleistung beantragen möchte, im ersten Protokollschritt sein

Paßwort benutzen. Um hier ein erhöhtes Risiko zu verhindern (ein Paßwort wird nur selten geändert, es ist damit ein *permanenter geheimer Schlüssel*), kann ein Benutzer die schon genannte initiale Authentisierung gegenüber Kerberos durchführen, um ein begrenzt gültiges *Ticket Granting Ticket* (TGT) zu bekommen (*temporärer geheimer Schlüssel*). Dies kann beispielsweise mit dem Kerberos-Programm `kinit` erfolgen, und gliedert sich in folgende Schritte auf:

- Der Benutzer gibt seinen Namen an. Dieser wird mit der Anforderung an den AS gesendet, ein Ticket für den TGS (das TGT) zu erhalten:

  Klient $\rightarrow$ Kerberos : $<$ Name, *"Ich möchte ein TGT"* $>$

- Der AS überprüft daraufhin, ob ihm der Benutzer bekannt ist, ob er also über einen gültigen Eintrag in der Kerberos-Datenbank verfügt. Falls dies der Fall ist, generiert er einen zufälligen Sitzungsschlüssel K_{S_1}, der zur Kommunikation zwischen dem Benutzer und dem TGS genutzt werden kann.

- Er bildet dann das Ticket, indem er den Benutzernamen, den Namen des TGS, die aktuelle Netzwerkadresse des Benutzers und den Sitzungsschlüssel verkettet, und mit dem geheimen Schlüssel des TGS (K_{TGS}) verschlüsselt. Der Benutzer kann damit die Integrität dieses Tickets nicht gefährden.

- Schließlich bildet er die für den Benutzer bestimmte Sicherheit, welche mit dem geheimen Schlüssel des Benutzers K_B (den der AS ja kennt) chiffriert ist, und aus dem eben erstelltem Ticket und einer weiteren Kopie des Sitzungsschlüssels besteht.

- Das so verschlüsselte Paket sendet der AS zu der Workstation, an dem der Benutzer das Programm `kinit` aufgerufen hat, zurück:

  Kerberos $\rightarrow$ Klient : $< K_B\{\ K_{S_1}\ ,\ K_{TGS}\{\text{Ticket}\}\ \} >$

- Der Benutzer wird nun an der Workstation gebeten, sein Paßwort einzugeben. Aus diesem kann (lokal) der geheime Schlüssel des Benutzers generiert werden, womit das vom AS gesendete Paket entschlüsselt werden kann.

Nach dieser Prozedur ist der Benutzer im Besitz eines Sitzungsschlüssels und des TGT (in welches er allerdings keine Einsicht hat). Mit diesem Ticket kann jetzt der TGS als Diensterbringer genutzt werden, wobei seine Dienstleistung darin besteht, neue Tickets für andere Dienstleistungen zu vergeben. Der oben genannte, erste Protokollschritt findet nun zwar immer noch statt, aber das vom Benutzer zu präsentierende Geheimnis ist jetzt nicht mehr sein permanent gültiges Paßwort, sondern das temporär gültige TGT.

Der zweite grundlegende Protokollschritt in Kerberos ist die authentisierte Nutzung einer Dienstleistung mit Hilfe eines Tickets. Ob das Beantragen eines neuen Tickets beim TGS mit Hilfe des TGT, oder das Abbuchen eines Geldbetrages bei einem Bankserver mit Hilfe eines anderen Tickets — der Ablauf der Authentisierung entspricht immer dem gleichen, im folgenden beschriebenen Protokoll:

- Der Benutzer baut einen Authentifikator auf (der unter anderem den Namen des Benutzers und dessen Netzwerkadresse enthält). Die Angaben im Authentifikator müssen mit denen im Ticket übereinstimmen. Jede Manipulation oder Falschangabe in diesem Authentifikator kann vom Diensterbringer mittels der Gegenangaben im Ticket aufgedeckt werden.

- Das Ticket, der mit dem Sitzungsschlüssel chiffrierte Authentifikator und evtl. zusätzliche Nutzdaten (welche auch mit dem Sitzungsschlüssel chiffriert sein können) werden jetzt an den Diensterbringer gesendet. Diese Nutzdaten bestehen z.B. für den TGS aus dem Klartextnamen des Diensterbringers, für den der Benutzer ein neues Ticket haben möchte:

  Klient $\rightarrow$ TGS : $< \text{Dienst} , K_{TGS}\{\text{Ticket}\} , K_{S_1}\{\text{Authenticator}\} >$

  oder für einen anderen Dienst:

  Klient $\rightarrow$ Server : $< \text{Daten} , K_{Service}\{\text{Service-Ticket}\} , K_{S_2}\{\text{Authenticator}\} >$

- Der Diensterbringer entschlüsselt mit Hilfe seines geheimen Schlüssels das Ticket, und erhält so den Sitzungsschlüssel, mit dem er den Authentifikator und evtl. verschlüsselte Nutzdaten dechiffrieren kann.

- Die Authentisierung des Benutzers gegenüber dem Diensterbringer erfolgt durch einen Vergleich von Benutzernamen und Netzwerkadresse im Ticket und im Authentifikator, sowie einer Kontrolle der Gültigkeit mit Hilfe der im Ticket und im Authentifikator enthaltenen Zeitangaben.

- Nach einer erfolgreichen Authentisierung kann der Diensterbringer noch eine Antwort an den Benutzer senden. Diese ist nicht mehr an ein bestimmtes Protokoll gebunden und kann anwendungsspezifisch aufgebaut sein. So wird beispielsweise der TGS dem Benutzer das angeforderte Ticket mit dem neu generierten Sitzungsschlüssel zurücksenden:

  TGS $\rightarrow$ Client : $< K_{S_1}\{ K_{S_2} , K_{Service}\{\text{Service-Ticket}\} \} >$

Der nach diesem Protokollschritt ablaufende Dialog zwischen Diensterbringer und Benutzer ist ebenfalls individuell, und kann je nach Sicherheitsvorgaben anwendungsspezifisch gelöst werden. Ob nachfolgende Datenpakete mit dem Sitzungsschlüssel chiffriert, signiert, oder gar nicht verschlüsselt sind, bleibt frei gestellt.

Der letzte Protokollschritt, die Antwort des Diensterbringers zum Benutzer, kann auch für eine *wechselseitige Authentisierung* verwendet werden. Dafür muß sich der Diensterbringer noch gegenüber dem Benutzer authentisiert, was er schon implizit macht, in dem er in der Lage ist, das Ticket zu entschlüsseln um an den Sitzungsschlüssel zu gelangen. Eine vom Benutzer mitgesendete und mit dem Sitzungsschlüssel chiffrierte Prüfsumme kann vom Diensterbringer korrekt dechiffriert, nach einer vorher verabredeten Vorgabe verändert, und an den Benutzer zurückgesendet werden.

Server $\rightarrow$ Klient : $< K_{S_2}\{\text{Prüfsumme} - 1\} >$

Der Empfang der vom Benutzer erwarteten Prüfsumme authentisiert dann den Diensterbringer gegenüber dem Benutzer.

1.3 Realisierung in Tycoon

Aufgrund der vielfältigen Dienstleistungen in offenen Computernetzen und der Heterogenität der heutigen Informationssysteme sind an Sicherheitsdienstleistungen, wie sie in der Authentisierungsbibliothek angeboten werden, erhöhte Anforderungen an Flexibilität und Portabilität zu stellen. Deshalb wurde für die Realisierung dieser Bibliothek das persistente Objektsystem **Tycoon** (*Typed Communication Objects in Open Environments*) [MATT93, MMS94] benutzt, welches durch seine skalierbare, persistente Systemarchitektur diesen Anforderungen gerecht wird.

Dies kann man sich leicht dadurch verdeutlichen, indem man sich die Innovationszyklen heutiger Informatiksysteme in Abbildung 1.1 betrachtet. Dabei bilden die Informatiksysteme mit ihren Abstraktionen ("Verdeckung von Komplexität") eine Basis, die die flexible Entwicklung von Anwendungssystemen ermöglicht. Bei solchen Entwicklungsprozessen wird häufig aber auch der Bedarf an neuen Abstraktionen deutlich, was die Hersteller von Informatiksystemen dazu bewegt, ihre Systeme zu erweitern bzw. neue Systeme zu entwickeln.

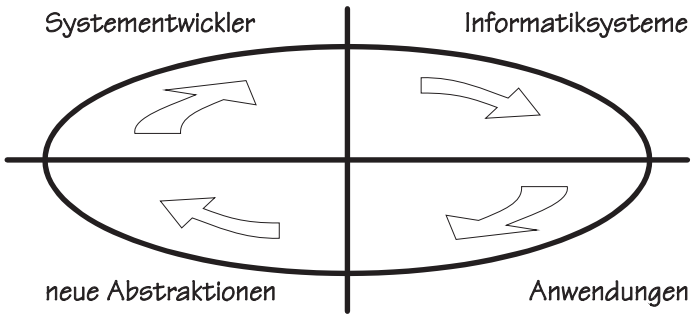

Abbildung 1.1: Innovationszyklen heutiger Informatiksysteme

In diesem Kontext stellt das persistente Objektsystem Tycoon mit seiner polymorphen *higher-order* Programmiersprache, der **Tycoon Language** (TL), ein *Metasystem* dar, welches sowohl die effiziente Entwicklung von Anwendungssystemen erlaubt, als auch durch seine stark reguläre Systemarchitektur in der Lage ist, neue Abstraktionen in den Entwicklungszyklus aufzunehmen. Damit besteht die Aussicht, daß man derartige Innovationszyklen in der Konstruktion von Anwendungs- und Informationssystemen innerhalb *einer* Entwicklungsumgebung auch *mehrfach* durchlaufen kann. Zudem reicht dabei die Beschränkung auf *eine* Programmiersprache durch die Sprachkonzepte höherer Ordnung in TL (Kapitel 1.3.1) aus, was die Produktivität in Tycoon erhöht. Eine unnötige Komplexität durch die Integration heterogener Systeme (wie Programmier-

sprachen, Datenbanken, Fenstersysteme, ...) auf niedriegstem Abstaktionsniveau wird dadurch vermieden.

Auch in der Entwicklung der Authentisierungsbibliothek wurde der Bedarf an neuen Abstraktionen deutlich, z.B. von Berechtigungshierarchien (behauptete bzw. bewiesene Identitäten) oder von Asymmetrien im Authentisierungsprozeß. Wie sich noch zeigen wird, kann man diesen neuen Abstraktionen innerhalb der Tycoon-Umgebung gerecht werden, beispielsweise über Mechanismen wie *Subtypisierung* oder *Polymorphismen*.

1.3.1 Das Tycoon-System

Das persistente Objektsystem **Tycoon** (*Typed Communication Objects in Open Environments*) [MATT93, MMS94] ist eine an der Universität Hamburg entwickelte Datenbankprogrammierumgebung, die durch eine skalierbare Systemarchitektur die Definition generischer Dienste in offenen Systemumgebungen erlaubt. Dabei wird die Definition neuer und die Einbindung externer Dienste in das Tycoon-System durch eine strikt typisierte, polymorphe Programmiersprache höherer Ordnung, der **TL** (*Tycoon Language*) ermöglicht.

Diese Programmiersprache ist eine Weiterentwicklung der Sprachen Quest [CARD90] und P-Quest [MATT91], wobei weniger Wert auf die Entwicklung neuer Sprachkonzepte gelegt wurde, als vielmehr in der orthogonale Integration bekannter Konzepte. Diese orthogonalen Konzepte bilden heute das Fundament des Tycoon-Systems und der Sprache TL:

- Die Sprache ist **funktional**, jeder Ausdruck evaluiert zu einem Wert. Funktionsdefinitionen und Funktionsapplikationen sind die wesentlichen Programmelemente in TL — dabei sind Funktionen *Objekte erster Klasse*, sie sind also gleichberechtigte Werte.

- Da die Auswertung eines Ausdrucks in TL von veränderlichen Bindungen im Objektspeicher abhängen können und diese Bindungen durch Zuweisungen änderbar sind (Seiteneffekte möglich!), ist TL aber auch **imperativ**.

- Die Sprache ist **strikt evaluativ**, da alle Terme als Funktionsargumente vor einer Funktionsapplikation vollständig ausgewertet werden.

- Da alle Basisterme in der Sprache TL einen deterministischen Wert besitzen, alle Zustandsvariablen initialisiert sind und die Auswertung von Subtermen eines Ausdrucks in einer sequentiell definierten Reihenfolge erfolgt, ist die Sprache TL **deterministisch**.

- Werte beliebig komplexer Datenstrukturen können über die Dauer einzelner Betriebssystemprozesse hinweg im Objektspeicherzustand erhalten bleiben. Da sich diese Eigenschaft der *Peristenz* auf alle Objekte der Sprache TL bezieht, spricht man auch von **orthogonaler Persistenz**.

- Die Sprache ist **polymorph**, was dadurch erreicht wird, daß auch Typen *Objekte erster Klasse* sind und damit als Funktionsparameter auftreten dürfen (*parametrischer Polymorphismus*). Daneben unterstützt die Sprache TL durch ihr hierarchisches Typsystem auch den *Subtyppolymorphismus*.

- In der Sprache TL sind Konsistenzbedingungen über alle Ausdrücke durch Typregeln und Typausdrücke definiert, deren Einhaltung durch die *statische Programmanalyse* garantiert wird. Eine explizite Auslagerung dieser Konsistenztests zur Programmlaufzeit ist möglich. TL ist damit **strikt typisiert**.

Wie in anderen modernen Programmiersprachen (wie z.B. Modula-2) üblich, unterstützt auch die TL die *Programmierung im Großen* durch Modularisierung. Dabei hat man in der TL die Möglichkeit, große Programme in **Schnittstellen** (*Interfaces*) und **Module** (*Modules*) aufzuteilen, wobei man beide Komponenten in Form von **Bibliotheken** (*Libraries*) strukturieren und verwalten kann. Die in den Schnittstellen enthaltenen Funktionssignaturen, Typen und Typoperatoren, werden durch eine Bindung an Module (*Werte*) implementiert.

Bei der in dieser Studienarbeit beschriebenen Authentisierungsbibliothek steht die Dokumentation der Schnittstellen im Mittelpunkt, da diese den globalen Sichtbarkeitsbereich der Entwicklungsschnittstelle definieren. In dem Fall, in dem neben der Schnittstelle auch die Implementierung von Interesse ist (z.B. bei Beispielimplementationen), wird der Begriff der **Komponente** benutzt. Unter einer Komponente versteht man dabei das zusammengehörige Paar, Schnittstelle und Modul [9].

1.3.2 Überblick: Authentisierung in Tycoon

Die Realisierung einer offenen, generell verwendbaren Authentisierungsbibliothek in Tycoon auf Basis von Kerberos ist das Ziel dieser Studienarbeit. Dabei soll der Zugriff auf diese Bibliothek über eine Entwicklungsschnittstelle erfolgen, die durch den hohen, in Tycoon möglichen Abstraktionsgrad leicht und intuitiv zu benutzen ist.

Grundlage hierfür bilden, neben den C-Bibliotheken für Kerberos und DES, die Tycoon-Schnittstellen zur Kommunikationsumgebung `commenv`, welche insbesondere die Kommunikation über *Sockets* anbieten (siehe Kapitel 2). Darauf aufbauend besteht die Authentisierungsbibliothek aus zwei Schichten:

- In der unteren Schicht wird die direkte Anbindung von Tycoon an die C-Bibliotheken von Kerberos und DES realisiert (Kapitel 3). Dabei wird schon hier eine rudimentäre Typisierung verschiedener Konzepte von Kerberos vorgenommen (z.B. Namen, Realms, Instanzen oder Sicherheiten).

- Darauf setzt eine höhere Schicht von Authentisierungsdiensten auf, mit denen von verschiedenen, systemnahen Konzepten von Kerberos (z.B. der Sockets) abstrahiert wird (Kapitel 4). Hiermit soll eine höhere Flexibilität und Nutzbarkeit der

[9] In Tycoon besteht die Möglichkeit, zu einer Schnittstelle auch mehrere Module für verschiedene Implementationen zuzuordnen. Dieser Fall wird hier jedoch vernachlässigt.

Bibliothek in den verschiedensten Anwendungsgebieten und Systemumgebungen erreicht werden.

Den schematischen Aufbau der Authentisierungsbibliothek zeigt Abbildung 1.2 in der Übersicht. Die in dieser Übersicht grau unterlegten Kästchen werden sich noch in den entsprechenden Kapiteln mit den Schnittstellen der Authentisierungsbibliothek füllen.

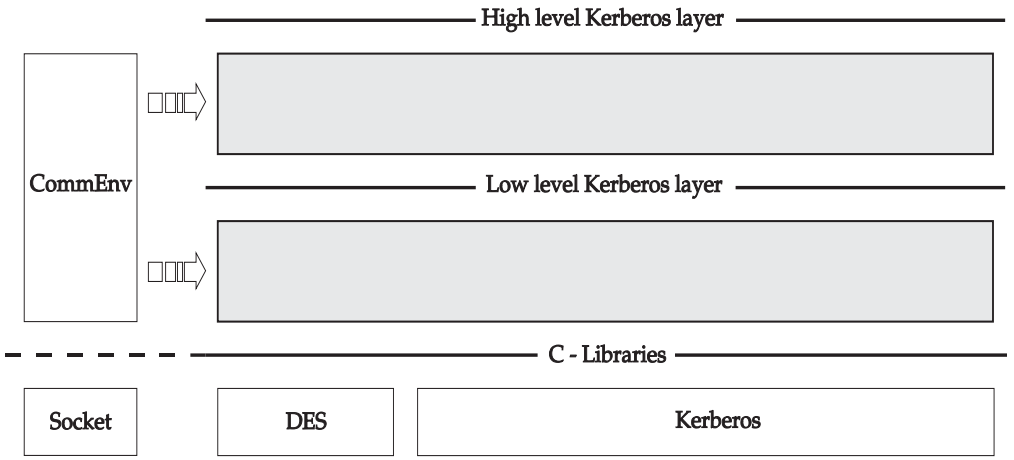

Abbildung 1.2: Die Authentisierungsbibliothek im Überblick

2. Die Kommunikationsumgebung

Entwicklungsziel von Authentisierungssystemen ist die Sicherheit in offenen Computernetzen. Verschiedene Kommunikationsmechanismen sind daher oft zentraler Bestandteil solcher Systeme, wie z.B. der **RPC-Mechanismus** bei *Suns Secure RPC* [RFC1057, RFC1094, SUNOS40] oder die **Sockets** bei Kerberos. Diese Mechanismen treten dabei sowohl in der Programmierschnittstelle dieser Systeme auf, wie auch intern in der Kommunikation der Systemkomponenten untereinander. Auch bei Kerberos erwarten viele Funktionen in ihrer Signatur Deskriptoren für Sockets, sowie weitere Strukturen der *BSD UNIX network database library*, beispielsweise Rechneradressen.

Zur Nutzung und Implementierung einer Kerberos-Anbindung in Tycoon ist daher als Handwerkszeug eine Kommunikationsbibliothek, die insbesondere das Socket-Konzept in Tycoon anbietet, notwendig. Eine solche Bibliothek wird über verschiedene Schnittstellen der Kommunikationsumgebung `commenv` angeboten. Diese Umgebung deckt dabei die Grundfunktionalität des Socket-Konzeptes ab, so daß mit ihrer Hilfe eine unmittelbare Kerberos-Schnittstelle in Tycoon entwickelt werden kann. In der darauf aufbauenden, abstrakteren Authentisierungsbibliothek wird dann jedoch eine Unabhängigkeit vom konkret verwendeten Kommunikationsmechanismus erreicht.

An dieser Stelle werden zunächst die für Kerberos relevanten Funktionen und Schnittstellen der Kommunikationsumgebung in Tycoon vorgestellt, sowie das dahinterstehende Konzept der Sockets kurz erörtert. Eine detailliertere Abhandlung über Sockets findet man z.B. in [COME95, COME94, STEV92a, STEV92b, BU93] oder der *Bibel* für BSD Programmierer [LEFF89].

2.1 Sockets, Domänen und Protokolle

Die **Sockets** entstanden im BSD-UNIX [LEFF89] als Methodik zur hardwareunabhängigen Interprozesskommunikation. Sie bilden **Kommunikationsendpunkte** zwischen den miteinander kommunizierenden Prozessen und abstrahieren dabei von der darunter liegenden Netzwerkarchitektur. Mit dieser Abstraktion ermöglichen die Sockets eine *transparente Kommunikation* zwischen den Prozessen, unabhängig davon, ob die Kommunikation tatsächlich im Netzwerk (*INET Domäne*) oder lokal, auf dem selben Rechner (*UNIX Domäne*) stattfindet. Die **Kommunikations-Domäne** spezifiziert für einen Socket die Standardsemantik und die Adressierungsart des zugrundeliegenden Netzwerkes.

Die Kommunikation über Sockets kann mit Hilfe zweier unterschiedlicher Protokolle

durchgeführt werden. Mit dem **TCP**-Protokoll (*Transmission Control Protocol*) kann eine **virtuelle Verbindung** aufgebaut werden, bei der die Übertragung und die Reihenfolge der Informationseinheiten sicher ist. Dagegen besteht bei dem **UDP**-Protokoll (*User Datagram Protocol*) keine feste Verbindung. Die Informationseinheiten werden hier in Form von **Datagrammen** auf den Weg gebracht, wobei ihr tatsächliches Ankommen beim Adressaten nicht garantiert ist. Auch die Reihenfolge, in der sie beim Empfänger eintreffen ist undefiniert [1].

Bei einer verbindungsorientierten Kommunikation nennt man den Prozeß, der die Verbindung aufbaut, den **Klienten** (*Client*). Dagegen bezeichnet man den Prozeß, der auf einen solchen Verbindungsaufbau reagiert, den **Diensterbringer** (*Server*).

2.2 Kommunikation mit Sockets

Die Handhabung von Sockets stellt eine Generalisierung des UNIX Dateizugriffes da, welche als Erweiterung die Semantik von Kommunikationsendpunkten unterstützt. Für Sockets muß genau so wie für Dateien des Betriebssystems ein *Deskriptor* angefordert werden. Dieser Deskriptor kann mit den traditionellen Dateioperationen `read` und `write` gelesen bzw. beschrieben werden, ferner kann er mit `close` wieder geschlossen werden.

In den spezifischen Eigenschaften der Sockets als Kommunikationsendpunkte ergeben sich allerdings einige Unterschiede und Erweiterungen gegenüber den UNIX Datei-Handles. So erfolgt das Anlegen eines Socket-Deskriptors mit der eigenen Funktion,

```
newSocket = socket(domain, type, protocol);
```

Hierbei wird im Gegensatz zur `open` Funktion für Datei-Handles (erzeugt lediglich eine Referenz auf ein bestehendes Objekt (Datei oder Gerät)) eine neue Instanz angelegt, wobei einige Zusatzinformationen für Sockets (die oben genannte Domäne, der Verbindungstyp und das zu verwendende Protokoll) mit dieser Instanz abgespeichert werden.

Ein weiterer Unterschied zu Datei-Handles besteht darin, daß Sockets *adressierbar* sein müssen, sie benötigen für die Kommunikation einen öffentlichen Namen, der innerhalb einer Kommunikations-Domäne eindeutig sein muß, eine **Portnummer**. Diese bekommen sie automatisch beim Anlegen eines Deskriptors mit der Funktion `socket` durch das Betriebssystem zugewiesen. Dabei wird der Socket-Deskriptor an eine **Socket-Adresse** gebunden, welche aus der Portnummer und aus der aktuellen Rechneradresse besteht. Mit Hilfe der Funktion `bind` kann man einen Socket-Deskriptor aber auch *explizit* an eine Socket-Adresse binden. Dies ist z.B. für Diensterbringer sinnvoll, die ihre Dienste unter einer spezifischen, vordefinierten Adresse anbieten wollen.

Ein Diensterbringer muß unter Umständen sehr viele ankommende Anfragen befriedigen. Gerade bei begehrten Dienstleistungen kann es dann vorkommen, daß neue Anfragen

[1] Sockets setzen mit diesen Protokollen auf Schicht 3 (*Network Layer*) des OSI-Referenzmodells auf. Detailliertere Informationen über das OSI-Referenzmodell, verbindungsloser bzw. verbindungsorientierter Übertragung und den damit verbundenen Problemen wie *routing*, *segmentation* oder *flow control* findet man z.B. in [HEAP94, KERN89].

schon eintreffen, obwohl der Diensterbringer noch mit der Bearbeitung der letzten Anfrage beschäftigt ist. Für solche Situationen gibt es eine mit dem Socket-Deskriptor verbundene *Request-Warteschlange*, in der ausstehende Anfragen gepuffert werden können. Mit der Funktion `listen` kann man vor einem Verbindungsaufbau die Größe dieser Warteschlange, also die maximale Anzahl ausstehender Anfragen, einstellen.

Der eigentliche Verbindungsaufbau zwischen zwei Sockets erfolgt auf der Seite des Klienten durch Aufruf der Funktion `connect` mit dem *Anwählen* des Diensterbringers und auf der Seite des Diensterbringers durch Aufruf der Funktion `accept` mit dem *Warten* auf einen neuen Verbindungsaufbau. Der Diensterbringer benutzt nach dem Verbindungsaufbau zur Kommunikation mit dem Klienten einen neuen, durch `accept` zurückgegebenen Socket-Deskriptor, um sich den ursprünglichen Kommunikationsport für weitere Anfragen offen zu halten. Dieser Mechanismus erlaubt auch die Implementierung komplexerer Kommunikationsaufgaben mit Hilfe von Sockets, z.B. von *Multithreaded control processes*, welche mehrere Klienten parallel bedienen können.

Die Abbildung 2.1 verdeutlicht noch einmal den prinzipiellen Ablauf der Interprozeß-Kommunikation mit Sockets (hier mit einer virtuellen Verbindung).

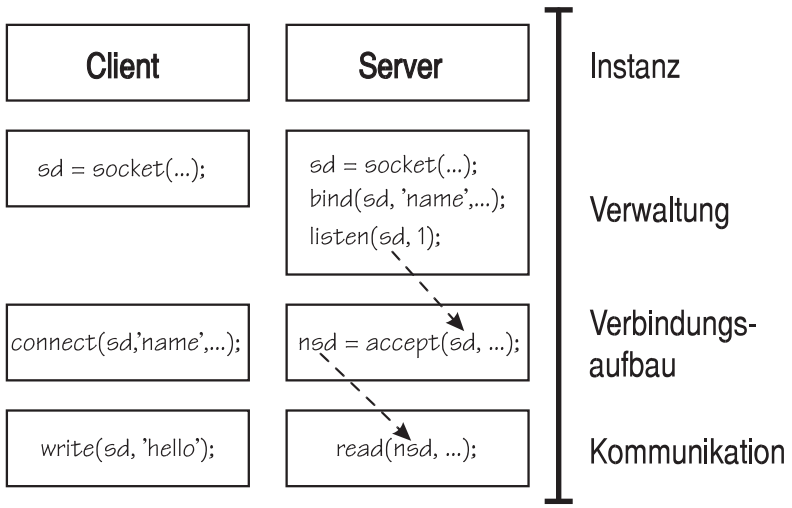

Abbildung 2.1: Kommunikation mit Sockets

Wie auch in der Abbildung ersichtlich ist, findet die eigentliche Kommunikation über Sockets meist mit den traditionellen Dateioperationen `read` und `write` statt. Es gibt hierfür aber auch neuere, für unsere Belange allerdings nicht relevante Operationen `send` und `recv`, welche speziell für Sockets implementiert wurden, und weitere Parameter für die Angabe socket-spezifischer Optionen anbieten. Für eine genauere Beschreibung dieser Funktionen sei auf [LEFF89] verwiesen.

2.3 Die Schnittstelle zur Kommunikationsumgebung

In folgendem Kapitel werden die von Kerberos benötigten Schnittstellen aus der Kommunikationsumgebung `commenv` vorgestellt. Hierzu gehören Schnittstellen, die Netzwerkadressen in Form abstrakter Datentypen anbieten (`INetAddress` und `NetAddress`), die Anfragen auf die *BSD UNIX network database library* erlauben (`NetDB`), sowie schließlich die eigentliche Socket-Schnittstelle (`Socket`).

2.3.1 Adressierung im Internet

Ein wichtiger Bestandteil einer Socket-Adresse ist die eigentliche **Netzwerkadresse**, die auf den Rechnerknoten zeigt, auf dem der Socket-Deskriptor gebunden bzw. angelegt wurde. Im Internet wird diese Netzwerkadresse durch einen 32-Bit Wert repräsentiert, der auch sehr häufig in Form einer Zeichenkette, der *dotted notation*, benutzt wird.

Ein Beispiel für eine solche Notation ist die Adresse `134.100.11.122`, welche auf den Rechner `dbis2` an der Universität Hamburg verweist. Jede der vier Zahlen, die durch die Punkte abgetrennt sind, hat dabei den Wertebereich eines Bytes ($[0..2^8 - 1]$). Die Umrechnung zu dem korrespondierenden 32-Bit Wert geschieht einfach durch eine Verkettung dieser vier Bytes zu einem 32-Bit Wort.

In der Tycoon Schnittstelle `INetAddress` wird für diese Internetadresse ein Datentyp `T` in Form eines *Word-Handles* definiert. Zu diesem Datentyp wird auch eine Funktionen zur Umrechnung einer Internetadresse in die *dotted notation* (`toString` bzw. `fromString`), sowie eine Funktion, die die aktuelle Internetadresse des aufrufenden Prozesses bestimmt, angeboten. Weiterhin stellt die Schnittstelle mit `nil` eine Konstante für eine *ungültige Internetadresse* zur Verfügung, und ferner eine Funktion `valid`, die kontrolliert, ob `not(equal(address nil))` gilt.

```
interface INetAddress
export

  Let T = word.Handle(word.T)

  nil :T

  fromString(netAddress :String) :T
  toString(netAddressHandle :T) :String

  current() :T

  equal(netAddress1 :T netAddress2 :T) :Bool
  valid(netAddress :T) :Bool
end;
```

Im Zuge der zu implementierenden Authentisierungsbibliothek wird eine weitgehende Abstraktion von physikalischen Gegebenheiten, wie z.B. das benutzte Kommunikationsmedium oder das zugrundeliegende Netzwerk, angestrebt. Die Schnittstelle von `INetAddress` ermöglicht aber zunächst nur einen spezifischen Zugriff auf Netzwerkadressen für das Internet, obwohl eine Schnittstelle für andere Netzwerkarchitekturen sicher ähnlich aussehen wird.

Hier bietet die Schnittstelle `NetAddress` einen abstrakteren Ansatz an, indem aufsetzend auf `INetAddress` ein abstrakter Datentyp (`Network`) angeboten wird, der eine standardisierte Handhabung von Netzwerkadressen erlaubt, und für jede physikalische Netzwerkarchitektur einen entsprechenden Wertetupel besitzt. Neben der von `INetAddress` her bekannten Funktionalität bietet der abstrakte Datentyp noch einen Konstruktor bzw. Destruktor für Netzwerkadressen an (im Internet ohne weitere Funktion), sowie Funktionen zur Konvertierung der *Byte-order* (siehe Kapitel 2.3.4).

```
interface NetAddress
export

  Let Network =
      Tuple
        T <: Ok

        new() :T
        free(:T) :Ok

        current() :T
        valid(:T) :Bool
        equal(:T :T) :Bool

        fromString(:String) :T
        toString(:T) :String

        byteOrderLocalToNet(:word.T):word.T
        byteOrderNetToLocal(:word.T):word.T
      end

  internet :Network
end;
```

Wie man sieht, gibt es hier bisher nur einen Wertetupel für das Internet, die Implementation weiterer Wertetupel ist für die Zukunft geplant.

2.3.2 Die Netzwerkdatenbank

Im Gegensatz zu den Netzwerkadressen bleibt der Zugriff auf die *BSD UNIX network database library*, wie der Name schon sagt, absolut systemspezifisch. Der Begriff *database* ist dabei jedoch etwas übertrieben, geht es doch hauptsächlich nur um Zugriffe auf die UNIX-Dateien `/etc/services` und `/etc/hosts`.

Die dafür angebotenen Funktionen zur Abfrage dieser "Datenbank" stehen über die Schnittstelle `NetDB` auch in Tycoon zur Verfügung. Diese Schnittstelle definiert für die Ergebnisse solcher Anfragen zunächst zwei Tupeltypen:

- Für Anfragen auf die Datei `/etc/services` enthält das Tupel `ServiceEntry` das Resultat. Dabei wird aufgeschlüsselt, wie ein bestimmter Dienst (*Service*) über eine Socket-Kommunikation ansprechbar ist. Hierzu gehören Informationen über die zu benutzende Portnummer und das dabei zu verwendende Protokoll (*TCP* oder *UDP*). Für solche Anfragen ist die Funktion `getServiceByName` zu benutzen.
- In dem Tupel `HostEntry` stehen schließlich Resultate von Anfragen auf die Datei `/etc/hosts`. Diese Datei ist im Zusammenhang mit der Bestimmung von Internetadressen von elementarer Bedeutung [2]. Mit der Funktion `getHostByName` wird aus dieser Datei zu einem Rechnernamen (dem *Host*) in dem Ergebnistupel abgelegt, welcher *Aliasname* und welche *Internetadresse* diesem Rechner zugeordnet ist.

Als weitere Funktion stellt `NetDB` mit `getHostName` eine Möglichkeit zur Verfügung, den Standardrechnernamen des aktuellen Prozessors abzufragen (dieser lautet beispielsweise `dbis1` oder `rzspc1`).

```
interface NetDB
export

  Let HostEntry =
      Tuple
        hostName        :String        (* Offizieller Rechnername *)
        firstAlias      :String        (* Erster Aliasname        *)
        hostAddressType :Int           (* Adress-Domaene          *)
        hostAddress     :iNetAddress.T (* Rechneradresse          *)
      end

  Let ServiceEntry =
      Tuple
        serviceName :String  (* Offizieller Dienstname *)
        socketPort  :Int     (* Portnummer             *)
        protocol    :String  (* Protkoll               *)
      end
```

[2] Eine flexiblere Verwaltung der Zuordnung eines Rechnernamens zu seiner Internetadresse erfolgt über das *Domain Name System* (DNS) [KROL95].

```
  getHostName() :String
  getHostByName(host :String) :HostEntry
  getServiceByName(name :String protocol :String) :ServiceEntry
end;
```

Ein Beispiel für einen Eintrag in die Datei `/etc/services` soll hier abschließend noch aufgeführt werden. Dabei wird festgelegt, daß ein Beispielprogramm von Kerberos unter dem Namen `sample` über die Portnummer `906` und dem Protokoll `TCP` ansprechbar ist:

```
...
sample     906/tcp     # Kerberos sample app server
...
```

2.3.3 Die Schnittstelle zu Sockets

Die Schnittstelle `Socket` bietet die Handhabung von Socket-Deskriptoren in Tycoon an. Hierzu gehört das Öffnen und Schließen, der Verbindungsaufbau, sowie der Nachrichtenaustausch mit Sockets.

Zunächst werden hierfür eine Reihe von Konstanten definiert, die den Verbindungstyp, die Kommunikations-Domäne und die zur Verfügung stehenden Protokolle definieren. Diese Konstantendefinition erfolgt dabei äquivalent zu den Konstanten aus der C-Schnittstelle `socket.h`.

```
interface Socket
export

  (* Socket types *)
  SOCK_STREAM()    :Int      (* Stream Socket              *)
  SOCK_DGRAM()     :Int      (* Datagramm Socket           *)
  SOCK_RAW()       :Int      (* Raw Protokoll              *)
  SOCK_RDM()       :Int      (* Reliably Delivered Message *)
  SOCK_SEQPACKET() :Int      (* Paketsequenz               *)

  AF_UNSPEC      :Int        (* Adress Spezifikation       *)
  AF_UNIX        :Int
  AF_INET        :Int
  ...

  PF_UNSPEC      :Int        (* Protokoll Spezifikation    *)
  PF_UNIX        :Int
  PF_INET        :Int
  ...
end;
```

Öffnen und Schließen von Sockets

Ein Socket-Deskriptor wird in der Schnittstelle von `Socket` als ein *Handle* definiert, der eine ähnliche Semantik wie ein Datei-Handle hat. Für einen ungültigen Wert des Handles wird hier die Konstante `nil` definiert.

```
interface Socket
export
  ...
  Let T = word.Handle(Int)        (* Socket Handle      *)
  nil :T                          (* Ungueltiger Handle *)

  new(domain, type, protocol :Int) :T
  destroy(sock :T) :Ok
  ...
end;
```

Die Funktion `new` ruft die Systemfunktion `socket` auf, und legt einen neuen Socket-Deskriptor an. Der Parameter `domain` spezifiziert dabei die Kommunikations-Domäne innerhalb dessen die Kommunikation stattfinden soll. Hierzu sind die schon vorher definierten Konstanten (`AF_*`) zu benutzen, beispielsweise `AF_UNIX`, zur rechnerinternen, lokalen Kommunikation oder `AF_INET`, zur netzwerkweiten Kommunikation über das IP-Protokoll.

Die Parameter `type` und `protocol` geben den Verbindungstypen und das Protokoll an. Der Verbindungstyp legt fest, ob eine *virtuelle Verbindung* aufgebaut werden soll, oder eine Kommunikation mit *Datagrammen* gewünscht ist (eine der `SOCK_*` Konstanten). Mit `protocol` wird ein spezielles Protokoll (eine der `PF_*` Konstanten) innerhalb einer Kommunikations-Domäne ausgewählt. Dieses überläßt man meist dem System, indem man als Standardwert '0' übergibt (diese praktische Übertragung der Verantwortlichkeit an das System entspricht der hier gebräuchlichen Programmierpraxis).

Zum Schließen eines Socket-Deskriptors gibt es noch die Funktion `destroy`, welche sich der Systemfunktion `close` bedient. Eine beim Schließen des Socket-Deskriptors bestehende Verbindung wird vorher abgebrochen, so daß die Portnummer einige Zeit später dem System wieder zur Verfügung steht. Beispiel zum Öffnen und Schließen eines Socket-Deskriptors:

```
let mySocket = socket.new(socket.PF_INET socket.SOCK_STREAM() 0);
...
socket.destroy(mySocket);
```

Bindungen

Zur Adressierung eines Kommunikationsendpunktes stellt die Schnittstelle `Socket` den Datentyp `Address` zur Verfügung. Dieser Datentyp enthält einen in der Kommunkations-

Domäne eindeutigen Namen eines Sockets in Form einer Portnummer und der aktuellen Rechneradresse des Sockets. Dabei kann das Format dieser Netzwerkadresse von Domäne zu Domäne unterschiedlich sein, weshalb dieser Datentyp noch zusätzlich einen *Domänenindikator* (`addressType`) enthält.

Die explizite Festlegung der Zuordnung "Socket auf Adresse" ist mit Hilfe der Funktion `bind` möglich. Zwar erhält ein Socket beim Öffnen mit der Funktion `new` eine implizite Adresse, aber oft wünschen z.B. Diensterbringer eine Kommunikation unter einer fest vorgegebenen Adresse. Viele Systemdienste sind über solche explizit vorbelegten Adressen erreichbar. Man kann diese Adressen mit Hilfe der in `NetDB` (*BSD UNIX network database library*) zur Verfügung gestellten Funktionen abfragen (siehe Kapitel 2.3.2).

```
interface Socket
export
  ...
  Let Address =
      Tuple
        addressType :Int            (* Adress Spezifikation *)
        port        :Int            (* Portnumber           *)
        inAddress   :iNetAddress.T  (* Internetadresse      *)
      end

  bind(sock :T name :Address) :Ok
  ...
end;
```

Die Abfrage der eigenen (evtl. implizit vorgegebenen) Adresse ist jederzeit mit Hilfe der Funktion `getMyAddress` möglich. Weiterhin kann nach einem erfolgreichen Verbindungsaufbau die Adresse des Kommunikationspartners mit der Funktion `getMyAddress` ermittelt werden.

```
interface Socket
export
  ...
  getMyAddress(sock :T) :Address
  getPeerAddress(sock :T) :Address
  ...
end;
```

Verbindungsaufbau

Zur Vorbereitung eines Verbindungsaufbaues benutzt der Diensterbringer, wie schon in Kapitel 2.2 erwähnt, die Funktion `listen`, um für einen Socket-Deskriptor die Größe

der Warteschlange für Anfragen zum Verbindungsaufbau festzulegen. Vielfach hat sich hierfür der Wert 5 eingebürgert.

```
interface Socket
export
  ...
  listen(sock :T requestQueue :Int) :Ok
  ...
end;
```

Anschließend wendet der Diensterbringer die Funktion `accept` auf einen geöffneten, gebundenen Socket-Deskriptor an, um den nächsten Verbindungsaufbau abzuwarten. Dabei wird zunächst die mit dem Socket verbundene Warteschlange überprüft — ist diese leer, blockiert `accept` den Diensterbringer bis eine Verbindung zustande kommt. Erst der Aufruf der Funktion `connect` durch einen Klienten löst ein beim Diensterbringer blockierendes `accept`.

Die Funktion `accept` gibt bei einem erfolgreichen Verbindungsaufbau neben der Adresse des rufenden Klienten auch einen neuen Socket-Deskriptor mit den gleichen Eigenschaften des ursprünglichen, für den Verbindungsaufbau benutzten Deskriptors zurück. Dieser neue Socket-Deskriptor wird für die eigentliche Kommunikation zwischen dem Klienten und dem Diensterbringer benutzt, während der Kommunikationsport für weitere Anfragen offen bleibt. Dieser ursprüngliche Kommunikationsport macht eigentlich nichts anderes, als die Warteschlange von Verbindungsanfragen zu verwalten.

Mit der Funktion `connect` initiiert der Klient einen Verbindungsaufbau zu dem durch `toSocket` spezifizierten Diensterbringer. Im Falle einer verbindungslosen Kommunikation (Angabe des Parameters `SOCK_DGRAM()` bei `new`) definiert `toSocket` lediglich die Empfängerspezifikation, wie sie im aufzubauendem Datagramm hinterlegt wird.

```
interface Socket
export
  ...
  connect(sock :T toSocket :Address) :Bool
  accept(sock :T var accepted :Address) :T
  ...
end;
```

Ein-/Ausgabefunktionen

Für den Austausch von Daten zwischen zwei Kommunikationsendpunkten nach einem erfolgreichen Verbindungsaufbau bietet die Schnittstelle `Socket` zunächst den Zugriff auf die einfachen Dateioperationen `read` und `write` an. Dabei ist der Austausch der Daten an Blöcke mit fest vorgegebener Länge gebunden, was zur Folge hat, daß die `read` Funktion evtl. auf noch fehlende Bytes wartet (*blockierende* Ein-/Ausgabe). Will man diesen Fall eines blockierenden Lesens verhindern, ist die Funktion `checkRead` hilfreich.

Sie stellt in einem über den Parameter `waitTime` einstellbaren Zeitraum fest, ob auf einem Socket noch weitere Daten zum Lesen anstehen.

Dieser insgesamt einfache Datenaustausch über Sockets mit den elementaren Dateioperationen `read` und `write` ist deshalb möglich, weil die Handhabung von Sockets auf eine Generalisierung des UNIX Dateizugriffes basiert. Um in Tycoon hiervon vollends zu profitieren, bietet die `Socket`-Schnittstelle Funktionen an, die einen Socket-Deskriptor in einen Eingabekanal (mit `toReader`) bzw. in einen Ausgabekanal (mit `toWriter`) umwandeln. Damit können Socket-Deskriptoren den in Tycoon gewohnten Ein- bzw. Ausgabeoperationen der Schnittstellen `Reader` bzw. `Writer` unterworfen werden.

```
interface Socket
export
  ...
  checkRead(sock :T waitTime :Int) :Bool

  write(sock :T data :word.T size :Int) :Ok
  read(sock :T size :Int) :word.T

  (* Konvertierung nach Reader / Writer *)
  toReader(sock :T) :reader.T
  toWriter(sock :T) :writer.T

  put(sock :T msg :message.T) :Ok
  get(sock :T msg :message.T) :Ok
end;
```

Die Kommunikation über Ein- bzw. Ausgabekanäle orientiert sich an dem *Stream*-Konzept, bei dem die zu übertragenden Daten aus einem Strom einzelner Zeichen bestehen. Dies mag in vielen Anwendungsfällen ausreichend sein, bei einer Kommunikation mit Sockets wird aber häufig eine paketorientierte Datenübertragung benutzt. Deshalb bieten die zuletzt aufgeführten Funktionen `put` und `get` eine effiziente Möglichkeit an, komplette Nachrichtenpakete über Sockets auszutauschen, wobei der Datentyp eines Nachrichtenpuffers aus der Schnittstelle `Message` benutzt wird (nächstes Kapitel).

2.3.4 Nachrichten und Datenpakete

Die letzte hier beschriebene Modulschnittstelle aus der Umgebung `commenv` ist `Message`. Diese Schnittstelle stellt einen Nachrichtenpuffer varianter Größe auf dem C-Heap zur Verfügung, in dem unter anderem beliebige Tycoon Datenobjekte abgespeichert werden können. Dies ermöglicht nicht nur eine effiziente, paketorientierte Datenübertragung über Sockets, sondern ist auch für andere Situationen von Interesse, in denen man Tycoon Datenobjekte auf dem C-Heap benötigt (z.B. bei der Datenverschlüsselung oder Datensignierung — siehe Kapitel 3.4).

Der in dieser Schnittstelle definierte Nachrichtenpuffer `T` besitzt neben dem Konstruktor `new` und dem Destruktor `free` auch noch eine Rücksetzfunktion `reset`. Diese setzt den aktuellen Bytezähler eines Nachrichtenpuffers (entspricht der Anzahl relevanter Bytes, die sich gerade im Puffer befinden) auf 0 zurück. Ganz nach der Mimik der Schnittstelle `Word` aus der Umgebung `stdenv` gibt es noch zwei Funktionen, um Zeichenketten in einen Nachrichtenpuffer abzulegen (`putString`) bzw. um Zeichenketten aus einem Nachrichtenpuffer zu extrahieren (`getString`).

Entsprechende Funktionen zum Datentransfer gibt es auch für den Datentyp `word.T` (`putWord` bzw. `getWord`). Diese beiden Funktionen haben aber auch noch eine Besonderheit: Sie benutzen den in `T` enthaltenen abstrakten Datentypen `netAddress.Network`, um eine Korrektur der (maschinenabhängigen) *Byte-order* durchzuführen. Daten, die mit diesen Funktionen in einen Nachrichtenpuffer transferiert werden, werden in die *Network standard byte order* gebracht [COME95, COME94] und umgekehrt, so daß ausgelesene Datenworte automatisch das richtige, plattformabhängige Zahlenformat haben.

```
interface Message
export

  Let T =
    Tuple
      net      :netAddress.Network (* Netzwerk                  *)
      var len  :Int                (* Aktuelle Laenge im Puffer *)
      var size :Int                (* Groesse des Puffers       *)
      data     :word.T             (* Datenpuffer               *)
    end

  new(network :netAddress.Network maxSize :Int) :T
  free(msg :T) :Ok
  reset(msg :T) :Ok

  putString(msg :T offset :Int val :String) :Ok
  getString(msg :T offset :Int) :String

  putWord(msg :T offset :Int val :word.T) :Ok
  getWord(msg :T offset :Int) :word.T

  from(V <:Ok value :V) :T
  to  (msg :T   V <:Ok) :V
end;
```

Die letzten beiden Funktionen, `from` und `to` ermöglichen den Transfer beliebiger Tycoon Datenobjekte in einen Nachrichtenpuffer (Austausch beliebiger Datenobjekte über Sockets — *orthogonale Mobilität*). Die Funktion `from` transferiert jedes Tycoon Datenobjekt (`V <:Ok`) in einen Nachrichtenpuffer, und legt diesen auch noch vorher selbständig

mit der richtigen (notwendigen) Größe an. Das entsprechende Datenobjekt kann später mit der Funktion `to` wiedergewonnen werden.

Der einzige Nachteil ist dabei noch, daß dies derzeit unter Ausnutzung der Tycoon-Komponente `Unsafe` geschieht und damit nicht typsicher ist. Dies wird sich aber mit Einführung dynamischer Datentypen in Tycoon ändern.

2.3.5 Beispielklient

Zu einem besseren Verständnis der Schnittstelle zur Kommunikationsumgebung folgt hier noch ein kleines Anwendungsbeispiel, welches man in ähnlicher Form auch im Modul `socketTest` ausprobieren kann. An dieser Stelle wird lediglich der Beispielklienten aus diesem Modul kurz vorgestellt. Dieser sendet zu einen Diensterbringer auf einem entfernten Rechner (`serverHost`) unter der Portnummer `serverSocket` die Nachricht *"Ist Deine Studienarbeit endlich fertig ?"* und liest die entsprechende Antwort.

```
let serverHost = "dbis25"
let serverSocket = 4711
let msg = "Ist Deine Studienarbeit endlich fertig ?"
```

Internetadresse des entfernten Rechners bestimmen.

```
let hp = netDB.getHostByName(serverHost)
```

Aliasnamen des Rechners auf die aktuelle Standardausgabe ausgeben.

```
print.string("Nehme Verbindung mit dem Server ")
print.string(hp.firstAlias)
print.string(" auf ...")
print.ln()
```

Anfordern eines Socket-Deskriptors für die Kommunikation.

```
let sock = socket.new(hp.hostAddressType
                      socket.SOCK_STREAM()
                      0)
```

Adresstupel des Diensterbringers konstruieren und Verbindung aufnehmen.

```
let sin :socket.Address =
    tuple
      hp.hostAddressType
      serverSocket
      hp.hostAddress
    end

socket.connect(sock sin)
```

Nachricht an den Diensterbringer bereitstellen ...

```
let myMessage = message.new(netAddress.internet 1024)
message.putString(myMessage 0 msg)
```

... und senden:

```
socket.put(sock myMessage)
```

Antwort lesen. Es wird komplett der Datenblock gelesen, der seitens des Diensterbringers mit `socket.put` versendet wird.

```
socket.get(sock myMessage)
```

Antwort des Diensterbringers auf die aktuelle Standardausgabe ausgeben.

```
print.string("Der Server antwortet mit:")
print.string(message.getString(myMessage 0))
print.string(result)
print.ln()

socket.destroy(sock);
```

Für das dazu passende, etwas umfangreichere Beispiel des Diensterbringers sei auf die Implementation im Modul `socketTest` verwiesen.

3. Die Kerberos-Schnittstelle

In diesem Kapitel wird die untere Schicht der Authentisierungsbibliothek beschrieben, welche eine unmittelbare Schnittstelle zum Kerberos-System darstellt.

Die Basiskomponenten dieser Schicht bestehen neben der Schnittstelle `LowKrb`, welche die grundlegenden Datentypen von Kerberos in Form *abstrakter Datentypen* (ADT's) definiert, aus zwei DES Schnittstellen, und einer Komponente, mit der man externe Daten, die sich auf dem C-Heap befinden, in ein untypisiertes *Bytearray*-Datenobjekt transferieren kann. Dadurch verfügt man über die Möglichkeit, mit DES verschlüsselte Datenblöcke oder Tickets (die Kerberos oft als "Datencontainer" benutzt) unter die Kontrolle des Objektspeichersystemes zu bringen, wo sie der *Garbage collection* unterliegen und sich in die von Tycoon unterstützten Konzepte der *orthogonalen Persistenz* und der *orthogonalen Mobilität* einpassen.

Aufbauend auf diese Basisschnittstelle können dann in einer übergeordneten Schicht verschiedene Funktionalitäten von Kerberos angeboten werden. Zu diesen Funktionen gehören die Verwaltung von Ticket-Files und des *Ticket granting tickets*, die Akquisition von Tickets für Dienstleistungen und deren Authentisierung, sowie die Verschlüsselung und Signierung von Daten. In Kapitel 4 wird diese Schnittstelle zum Kerberos-System dann dazu benutzt, abstraktere, einfach zu benutzende Authentisierungs- und Sicherheitsdienstleistungen zu realisieren.

Schließlich wird auch noch die in Kerberos realisierte Authentisierung über eine virtuelle Kommunikationsverbindung mit Sockets zur Verfügung gestellt, sowie eine Beispielkomponente mit einigen Anwendungen für diese Bibliotheksschicht. Eine Übersicht über die Authentisierungsbibliothek einschließlich der Schnittstellen, die in diesem Kapitel noch vorgestellt werden, findet man in Abbildung 3.1.

3.1 Bytearrays in der Schnittstelle zu C

Wie schon angedeutet, werden für die C-nahen Kerberos-Funktionen untypisierte **Bytearrays** fester Länge benötigt, beispielsweise um verschlüsselte Tycoon-Datenobjekte unter Kontrolle des Objektspeichersystemes zu halten.

Diese Bytearrays existieren schon auf Ebene des **TSP** (*Tycoon store protocol*), ein Protokoll, welches den Zugriff auf das Objektspeichersystem steuert. Sie bilden dort ein fundamentales Element zur Datenspeicherung, und enthalten z.B. compilierten Tycoon-Bytecode. Um diese Bytearrays in der Schnittstelle zu C-Funktionen nutzen zu können,

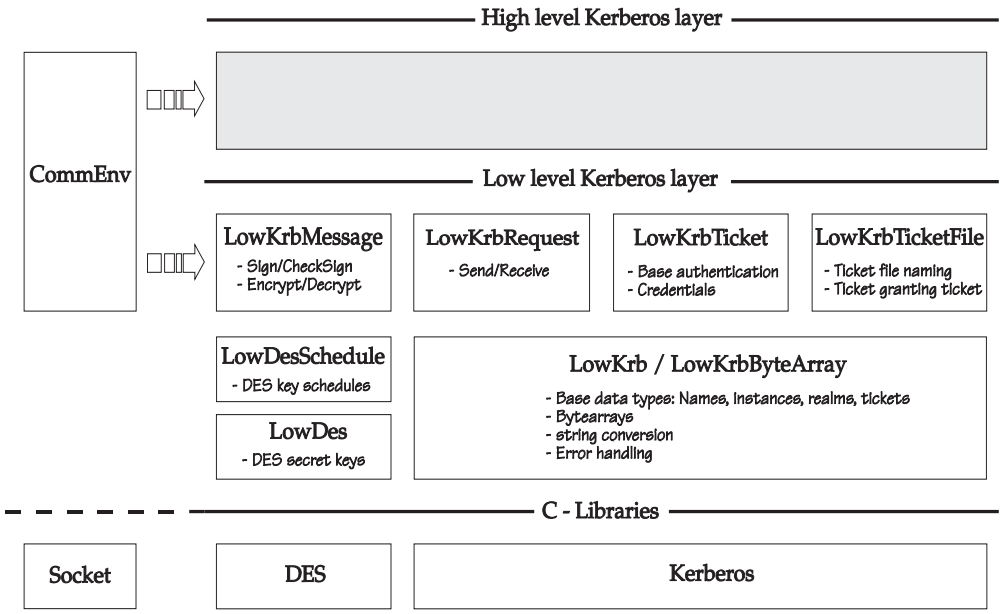

Abbildung 3.1: Die Kerberos-Bibliothek

werden zwei Funktionen benötigt, die den Datentransfer zwischen dem C-Heap und den Bytearray-Objekten realisieren.

Hierzu wird in der Schnittstelle `LowKrbByteArray` zunächst die Funktion `toCHeap` angeboten, welche ein Tycoon-Bytearray auf den C-Heap kopiert. Dabei wird auf dem C-Heap ein Speicherbereich mit der notwendigen Größe reserviert, der später wieder vom Anwendungsprogrammierer explizit freigegeben werden muß (mit der Funktion `free(...)` aus der Schnittstelle `Word`).

Um von einer Datenstruktur auf dem C-Heap zu einem Bytearray zu gelangen, stellt die Schnittstelle weiterhin die Funktion `fromCHeap` zur Verfügung. Diese benötigt als Parameter neben der Angabe des Zeigers auf die C-Struktur noch eine Größenangabe (Anzahl der zu kopierenden Bytes) und einen Offset, ab dem die Daten in das Bytearray kopiert werden sollen (gleiche Mimik wie in der Schnittstelle `Word`). Schließlich gibt es noch mit `size` über eine Funktion zur Bestimmung der Größe eines Bytearrays.

```
interface LowKrbByteArray
export
  T <:Ok           (* Bytearray *)

  toCHeap(byteArray :T) :word.T
  fromCHeap(data :word.T size :Int offset :Int) :T
  size(byteArray :T) :Int
end;
```

3.2 Die DES Schnittstellen

Kerberos benutzt als kryptographisches Verfahren zur Datenverschlüsselung DES (den *Data Encryption Standard*, siehe Kapitel 1.1.3). Für dieses Verfahren wird in der Schnittstelle `LowDes` zunächst ein 64-Bit großer DES-Schlüssel (von dem allerdings nur 56 Bit effektiv genutzt werden) als Datentyp bereitgestellt.

Diese Schnittstelle bietet neben den Funktionen `get` und `toWord`, die einen Datentransfer des DES-Schlüssels zwischen Tycoon und C Funktionen erlauben, die Einwegfunktion `fromString` an, welche aus einer beliebig langen Zeichenkette (z.B. ein Paßwort) einen DES-Schlüssel (einen 56-Bit Hash-Wert) generiert. Dies geschieht nach dem Prinzip der Einwegfunktionen (Kapitel 1.1.5) derart, daß faktisch keine Umkehrung möglich ist, um von dem DES-Schlüssel auf die Zeichenkette zu schließen (keine "`toString`" Funktion). Diese Funktion wird unter anderem auch von Kerberos dazu benutzt, um die geheimen Schlüssel der Benutzer aus deren Paßwörtern abzuleiten.

```
interface LowDes
export
  T <:Ok

  get(addr :word.T offset :Int) :T
  toWord(key :T) :word.T

  fromString(secretString :String) :T
end;
```

Wie schon in Kapitel 1.1.3 angedeutet, ist DES ein Verschlüsselungsverfahren, welches auf 16 Iterationen von Substitutionen und Permutationen aufbaut. Dabei werden die Daten blockweise bei jeder Iteration mit einem sich wechselndem 48-Bit Schlüssel XOR-verknüpft. Die dazu notwendigen 16 Schlüssel werden durch verschiedene Shift-Operationen aus dem eigentlichem 56-Bit DES Schlüssel (`LowDes.T`) gewonnen.

Dieses Feld aus 16 Schlüsseln nennt man auch **Schlüsselverzeichnis** (*Key-Schedule*). Sobald ein DES-Schlüssel bekannt ist, kann man aus diesem ein Schlüsselverzeichnis generieren, und direkt als Basis für die Ver- bzw. Entschlüsselung der Daten benutzen (bessere Performance). Auch viele Kerberos-Funktionen erwarten ein solches Schlüsselverzeichnis direkt als Parameter.

In der Schnittstelle `LowDesSchedule` wird ein solches Schlüsselverzeichnis als Datentyp `T` zur Verfügung gestellt. Für die Generierung dieses Verzeichnisses aus dem DES-Schlüssel wird die Funktion `fromKey` bereitgestellt und für die Benutzung des Schlüsselverzeichnisses für andere C-Funktionen gibt es noch die Funktion `toWord`.

```
interface LowDesSchedule
export
  T <:Ok

  fromKey(key :lowDes.T) :T
  toWord(keySchedule :T) :word.T
end;
```

3.3 Das Basisschnittstelle zu Kerberos

Die Komponente `LowKrb` bildet aufbauend auf die Bytearrays aus Kapitel 3.1 die Tycoon-Basisschnittstelle zu Kerberos. Sie stellt verschiedene Hilfsfunktionen (Fehlerbehandlung, Datumskonvertierung) sowie die Basisdatentypen zur Verfügung, die für den Zugriff auf die C-Bibliothek von Kerberos nötig sind.

```
interface LowKrb
export

  KSUCCESS :Int         (* Rueckgabewert  --  OK       *)
  KFAILURE :Int         (* Rueckgabewert  --  Fehler   *)

  getErrorText(err :Int) :String
  convTime(timeStamp :word.T) :date.T
  ...
end;
```

Zu den Basisdatentypen von Kerberos gehören die Komponenten, aus denen sich ein Principal zusammensetzt, also der **Name**, die **Instanz** und der **Realm**, sowie das **Ticket**, welches von Kerberos generell als "Datencontainer" für Authentifikatoren, Sicherheiten, etc. genutzt wird.

Es hat sich gezeigt, daß auf dieser niedrigen Bibliotheksebene für alle Basisdatentypen die gleichen Operationen benötigt werden. Zu diesen Operationen gehören die Feststellung der belegten Speicherplatzgröße (`size`), die Konvertierung von und nach Zeichenketten (`fromString` und `toString`), Schnittstellenfunktionen für den C-Heap (`toWord`, `get` und `put`) und eine Funktion zur Feststellung eines aktuellen Initialwertes (`current`). Aufgrund dieser Übereinstimmungen sind alle Basisdatentypen über den selben *abstrakten Datentypen* (ADT) definiert.

```
interface LowKrb
export
  ...
```

```
  Let ADT = Tuple
              T <:Ok

              size() :Int

              fromString(str :String) :T
              toString(value :T) :String

              toWord(value :T) :word.T
              get(addr :word.T offset :Int) :T
              put(addr :word.T offset :Int value :T) :Ok

              current() :T
            end
  ...
end;
```

Ein Name verweist unter Kerberos auf einen Principal, er steht also für einen Benutzernamen (z.B. "Hugo") oder eine Dienstleistung (z.B. "sample"). Ein Name darf nicht leer sein, und kann über die Funktion `current` von der aktuellen Betriebssystemidentität des laufenden Betriebssystemprozesses bestimmt werden.

Eine Instanz bezeichnet einen Zustandsraum, innerhalb dessen ein Principal agieren kann. Dies kann z.B. der Workstation-Name sein, an dem ein Benutzer gerade arbeitet.

Unter einem Realm versteht man das schon in Kapitel 1.2.1 definierte Gebiet, innerhalb dessen ein Principal gegenüber Kerberos registriert ist. Das aktuelle Gebiet kann mit der Funktion `current` aus der Kerberos-Datenbank abgefragt werden. Dieses aktuelle Gebiet lautet beispielsweise bei DBIS an der Universität Hamburg:

```
DBIS.INFORMATIK.UNI-HAMBURG.DE
```

Für ein Ticket gibt es unter Kerberos keinen initialen Wert. Ein Aufruf der Funktion `current` gibt ein Ticket mit der Zeichenkette "NIL" als Inhalt zurück.

```
interface LowKrb
export
  ...
  name :ADT
  Let Name = name.T

  instance :ADT
  Let Instance = instance.T

  realm :ADT
  Let Realm = realm.T
```

```
  ticket :ADT
  Let Ticket = ticket.T
  ...
end;
```

Um einen Principal eindeutig zu identifizieren benötigt man alle drei Komponenten: Den Namen, die Instanz und das Gebiet. Um die Angabe solcher Identifikatoren zu vereinfachen, bietet Kerberos einen Zeichenketten-Parser an, der diese Komponenten aus folgender Zeichenkette ableitet:

```
"<Name>[.<Instanz>][@<Gebiet>]"
```

Die Funktion `parse` nutzt diese Funktion, und gibt die drei Komponenten als Tupel zurück. Komponenten, die in der Zeichenkette nicht angegeben sind, werden über die entsprechende `current`-Funktion expandiert. Im Gegensatz dazu konstruiert die Funktion `generate` aus den drei Komponenten wieder die oben angegebene Zeichenkette.

```
interface LowKrb
export
  ...
  Let UserName = Tuple
                   userName  :Name
                   userInst  :Instance
                   userRealm :Realm
                 end

  parse(userName :String) :UserName
  generate(userName :UserName) :String
end;
```

3.4 Verschlüsselung und Signierung

Die Komponente `LowKrbMessage` nutzt die bisher dokumentierten Basisschnittstellen, um die Verschlüsselung und Signierung von Daten zu ermöglichen. Dabei finden diese Prozesse auf Bytearrays statt, damit sich alle Daten, auch die verschlüsselten bzw. signierten Datenpakete, unter der Kontrolle des Objektspeichersystems von Tycoon befinden.

Zur Verschlüsselung bzw. Signierung der Daten wird immer direkt der DES-Schlüssel benutzt, die Key-Schedules werden intern in den Funktionen generiert. Außerdem werden die Socket-Adressen von Sender und Empfänger der Daten benötigt, da beim Entschlüsseln der Daten bzw. Prüfen der Signatur die angegebene Senderadresse von Kerberos immer kontrolliert wird (Prüfung der Senderintegrität).

Die Funktion `sign` signiert eine Nachricht, wohingegen die Funktion `proveSign` diese Signatur überprüft, und die originale Nachricht zurückgibt. Die Nachricht bleibt durch die Signierung unverschlüsselt, ist aber über die Signatur gegen Manipulationen geschützt. Eine illegal manipulierte Nachricht würde in der Funktion `proveSign` eine Exception auslösen.

```
interface LowKrbMessage
 export

   sign(in       :lowKrbByteArray.T
        key      :lowDes.T
        sender   :socket.Address
        receiver :socket.Address) :lowKrbByteArray.T

   proveSign(in       :lowKrbByteArray.T
             key      :lowDes.T
             sender   :socket.Address
             receiver :socket.Address) :lowKrbByteArray.T
   ...
 end;
```

Die Funktion `encrypt` verschlüsselt die angegebene Nachricht, so daß dessen Geheimhaltung gewährleistet bleibt. Zum Entschlüsseln dieser Nachricht ist die Funktion `decrypt` zu benutzen. Das Ver- und Entschlüsseln ist auch mehrfach, mit unterschiedlichen Schlüsseln möglich.

```
interface LowKrbMessage
 export
   ...
   encrypt(in       :lowKrbByteArray.T
           key      :lowDes.T
           sender   :socket.Address
           receiver :socket.Address) :lowKrbByteArray.T

   decrypt(in       :lowKrbByteArray.T
           key      :lowDes.T
           sender   :socket.Address
           receiver :socket.Address) :lowKrbByteArray.T
 end;
```

3.5 Kerberos Login und Ticket-Files

Die Verwaltung von Ticket-Files, in denen ein Principal seine Sicherheiten sammelt, und die Akquisition des *Ticket granting tickets* finden in der Komponente `LowKrbTicketFile` statt. Dabei wird beim Erwerb des TGT (dies ist meist das erste Ticket) das Ticket-File automatisch durch Kerberos angelegt.

Normalerweise erzeugt Kerberos solche Ticket-Files auf dem Verzeichnis `/tmp` und konstruiert dabei den Dateinamen aus der aktuellen Betriebssystemidentität des Principals. Dies ist für die Authentisierungsbibliothek in Tycoon ungünstig, da man hier auch unabhängige Abläufe modellieren möchte (z.B. mit *Threads*), denen unterschiedliche Principals zugeordnet sind, die aber in einem einzigen Objektspeichersystem (und damit unter der selben Betriebssystemidentität) ablaufen.

Um solche *Multiple ticket files* zu ermöglichen, nimmt diese Komponente Einfluß auf die Namenskonvention für die Erzeugung von Ticket-Files in Kerberos. Diese Dateien werden jetzt zwar nach wie vor noch auf dem Verzeichnis `/tmp` angelegt, aber mit folgenden, an dem Namen des Principals orientierten Aufbau des Dateinamens:

```
/tmp/.tkt_<principalName>_<principalInstance>
```

Um vor Operationen, die das Ticket-File benutzen, den internen Verweis des Kerberos-Systemes auf die richtige Datei zu setzen, ist die Funktion `set` zu benutzen:

```
interface LowKrbTicketFile
 export
   set(principal :lowKrb.UserName) :Ok
   ...
 end;
```

Dies ist allerdings nur bei der Implementierung eigener Funktionen in Tycoon notwendig, die die C-Bibliothek von Kerberos benutzen. Die Tycoon-Funktionen aus dieser Bibliothek benutzen Kerberos schon in der korrekten Art und Weise, so daß immer das richtige Ticket-File benutzt wird.

Die initiale Identifikation eines Principals gegenüber dem Kerberos-System ist mit der Akquisition des *Ticket granting tickets* und dem Anlegen eines Ticket-Files verbunden. Dieser Identifikationsprozeß wird in zwei Fälle unterschieden:

- Im dem Fall, in dem der Principal ein aktiver Benutzer ist, findet die Identifikation dadurch statt, daß der Principal sein geheimes Paßwort vorweisen kann. In diesem Fall ist die Funktion `newClient` zu benutzen, die das Paßwort als Parameter erwartet. Wird für diesen Parameter eine leere Zeichenkette angegeben, erfolgt eine interaktive Abfrage des Paßwortes.

  Der Erwerb des TGT für aktive Benutzer findet nach dem in Kapitel 1.2.2 beschriebenen Protokoll statt. Der dabei am Ende benötigte geheime DES-Schlüssel des Benutzers wird mit der Einwegfunktion `des.stringToKey` aus seinem Paßwort abgeleitet.

- In dem Fall, in dem der Principal ein Prozeß ist, wäre ein Paßwort als Geheimnis für die initiale Identifikation jedoch ungünstig, da dieses Paßwort im Programmobjekt des Prozesses fest angegeben werden müßte.

  Ein Prozeß benutzt statt dessen eine **Service-Schlüsseldatei**, die unter dem Principalnamen des Diensterbringers den benötigten, geheimen DES-Schlüssel bereithält. Die Absicherung der geheimen Schlüssel in der Service-Schlüsseldatei erfolgt über die Zugriffsrechte des Prozesses auf diese Datei. Niemand, der nicht die entspechenden Zugriffsrechte auf diese Datei besitzt, kann sie zur Identifikation benutzen. Der zu identifizierende Prozeß muß mit der Betriebssystemidentität laufen, unter der er die Datei lesen kann, anderenfalls ist ein Erwerb des TGT nicht möglich.

  In dem Fall, in dem der Principal ein Prozeß ist, ist Funktion `newServer` zu benutzen. Sie erwartet den Dateinamen der Service-Schlüsseldatei als Parameter. Übergibt man hier die leere Zeichenkette, wird eine in Kerberos definierte Systemdatei (meist `/etc/srvtab`) benutzt. Wichtig ist bei dieser Funktion, daß der Principalname exakt so angegeben wird (mit Name und Instanz), wie er in der Service-Schlüsseldatei eingetragen wurde.

Beide Funktionen, `newClient` und `newServer`, erwarten eine Angabe für die Lebensdauer des TGT in Minuten und geben Werte vom Datentyp `T` zurück. Die Verwendung solcher Werte mit dem Datentypen `T` erzwingt eine erfolgreich durchgeführte initiale Identifikation des Principals.

```
interface LowKrbTicketFile
 export
   ...
   T <:lowKrb.UserName

   newClient(name     :lowKrb.UserName
             lifeTime :Int
             password :String) :T

   newServer(name     :lowKrb.UserName
             lifeTime :Int
             srvtab   :String) :T
   ...
 end;
```

Die unterschiedliche Identifikation der Principals über die Funktionen `newClient` und `newServer` ist auch in Abbildung 3.2 skizziert.

Den Abschluß der Schnittstelle `LowKrbTicketFile` bilden Funktionen zur Gültigkeitsüberprüfung des Datentypen `T` (Kontrolle, ob das aktuelle TGT noch gültig ist), und eine Funktion, die das Ticket-File eines Principals löscht. Das Löschen einzelner, spezifischer Sicherheiten im Ticket-File ist in Kerberos allerdings nicht möglich.

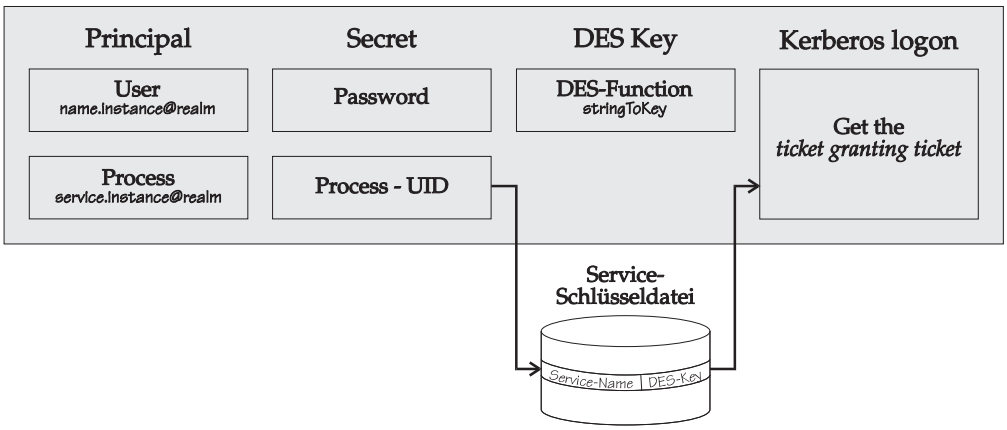

Abbildung 3.2: Akquisition des TGT in der Initialisierungsphase

```
interface LowKrbTicketFile
 export
   ...
   valid(ticketFile :T) :Bool
   validUntil(ticketFile :T) :date.T

   Let FreeResults =
       Tuple case
         destroyed,   (* Alle Tickets wurden geloescht.              *)
         noTickets,   (* Es sind keine weitern Tickets vorhanden.    *)
         notDestroyed (* Die Tickets koennen nicht geloescht werden. *)
       end

   free(ticketFile :T) :FreeResults
 end;
```

3.6 Authentisierung und Tickets

Nach der initialen Identifikation gegenüber dem Kerberos-System kann ein Principal Tickets für Dienstleistungen anfordern, und Authentifikatoren aufbauen. Er kann also das "Basisgeschäft" der Authentisierung, wie es mit der Schnittstelle `LowKrbTicket` angeboten wird, ausüben.

Hierzu werden zwei komplexere Datentypen benötigt: Einer für die Sicherheit, die ein Principal anfordern kann, und einer für den Authentifikator, der dem Diensterbringer Auskunft über den als Klienten auftretenden Principal gibt.

```
interface LowKrbTicket
 export
   Let Authenticator =
       Tuple
         pName      :lowKrb.Name     (* Name des Principals  *)
         pInst      :lowKrb.Instance (* Seine Instanz        *)
         pRealm     :lowKrb.Realm    (* Sein Gebiet          *)
         chksum     :Int             (* Seine Pruefsumme     *)
         session    :lowDes.T        (* Sitzungsschluessel   *)
         lifetime   :Int             (* Lebensdauer          *)
         createTime :word.T          (* Angelegt am ...      *)
       end

   Let Credential =
       Tuple
         service    :String          (* Dienst-Name          *)
         instance   :lowKrb.Instance (* Dienst-Instanz       *)
         realm      :lowKrb.Realm    (* Dienst-Gebiet        *)
         session    :lowDes.T        (* Sitzungsschluessel   *)
         lifetime   :Int             (* Lebensdauer          *)
         createTime :word.T          (* Angelegt am ...      *)
         kvno       :Int             (* Schluessel-Version   *)
         ticket     :lowKrb.Ticket   (* Das Ticket           *)
         pName      :lowKrb.Name     (* Name des Klienten    *)
         pInst      :lowKrb.Instance (* Instanz des Klienten *)
       end
   ...
 end;
```

Die Akquisition des Tickets für eine Dienstleistung und der für Kerberos nötige Aufbau eines Paares, bestehend aus Ticket und Authentifikator, geschieht über die Funktion `makeRequest`. Diese Funktion fordert allerdings nur dann ein neues Ticket an, falls es für die gewünschte Dienstleistung im Ticket-File des Principals noch kein entsprechendes Ticket gibt. Ansonsten wird das bestehende Ticket benutzt, und lediglich der Authentifikator wird neu aufgebaut (dieser ist nur einmal verwendbar).

Die von dieser Funktion zurückgegebene Datenstruktur ist nicht mit dem von Kerberos angeforderten Ticket zu verwechseln! Die Tycoon-Datenstruktur `lowKrb.Ticket` dient hier als "Datencontainer", und enthält eine Versionsangabe, das Kerberos-Ticket für den Diensterbringer (verschlüsselt mit dessen geheimen Schlüssel), und den Authentifikator (verschlüsselt mit dem Sitzungsschlüssel):

$< \text{VERSION} , K_{Service}\{\text{TICKET}\} , K_{Session}\{\text{AUTHENTICATOR}\} >$

```
makeRequest(id       :lowKrbTicketFile.T
            service  :String
            inst     :lowKrb.Instance
            realm    :lowKrb.Realm
            checksum :Int) :lowKrb.Ticket  (* <> Kerberos-Ticket *)
```

Der Authentifikator kann nur von jemandem eingesehen werden, der den Sitzungsschlüssel kennt, und dies kann nur der Diensterbringer sein (nach dem er das Kerberos-Ticket dechiffriert hat). Deshalb kann man die im Authentifikator enthaltene Prüfsumme, die der Klient über den Parameter `checksum` angibt, zur Authentisierung des Diensterbringers gegenüber dem Klienten nutzen (siehe Kapitel 1.2.2).

Das Aufschlüsseln eines solchen, mit `makeRequest` angefertigten Paketes kann der Diensterbringer komfortabel mit der Funktion `readRequest` durchführen. Diese Funktion dechiffriert das Kerberos-Ticket, entnimmt daraus den Sitzungsschlüssel, und entschlüsselt damit den Authentifikator, welcher als Funktionsresultat zurückgegeben wird.

```
readRequest(receivedData :lowKrb.Ticket  (* <> Kerberos-Ticket *)
            service      :String
            inst         :lowKrb.Instance
            client       :iNetAddress.T
            keyFile      :String) :Authenticator
```

Da für das Dechiffrieren des Kerberos-Tickets der geheime Schlüssel des Diensterbringers benötigt wird, muß dieser in der Signatur der Funktion `readRequest` seine Identität (Name und Instanz) und seine (schon in Kapitel 3.5 erwähnte) Service-Schlüsseldatei angeben. In dieser Service-Schlüsseldatei muß der geheime Schlüssel des Diensterbringers unter der angegebenen Identität auffindbar sein. Wird für den Dateinamen der Service-Schlüsseldatei eine leere Zeichenkette angegeben, erfolgt die Suche des geheimen Schlüssels in einer von Kerberos vordefinierten Datei (meist `/etc/srvtab`).

Damit ein Principal einen Überblick über seine bereits akquirierten Sicherheiten hat, kann er jederzeit mit der Funktion `getCredential` auf sein Ticket-File zugreifen, und damit alle relevanten Daten einer Sicherheit, wie z.B. den Sitzungsschlüssel oder die verbliebene Lebensdauer, ermitteln.

```
interface LowKrbTicket
 export
   ...
   getCredential(id      :lowKrbTicketFile.T
                 service :String
                 inst    :lowKrb.Instance
                 realm   :lowKrb.Realm) :Credential
 end;
```

3.7 Client-/Server-Authentisierung

Das Schnittstelle `LowKrbRequest` bietet einen Zugriff auf die in Kerberos realisierte Authentisierung über eine virtuelle Kommunikationsverbindung mit Sockets an. Die für die Kommunikation genutzten Socket-Deskriptoren müssen dabei vorher schon ordnungsgemäß geöffnet worden sein, und die Verbindung zwischen Klient und Diensterbringer muß bereits bestehen.

Die Funktion `send` versendet alle für die Authentisierung notwendigen Daten an den Diensterbringer, und geht dabei mit folgendem Protokoll vor:

- Das Ticket für den gewünschten Dienst wird beschafft. Ist im Funktionsaufruf die Option `KOPT_DONT_MK_REQ` gesetzt, wird kein neues Ticket angelegt. Statt dessen wird das im Parameter `ticket` angegebene Ticket benutzt.
- Der entsprechende Authentifikator wird aufgebaut (dies erfolgt ähnlich, wie bei der Funktion `makeRequest` aus der Schnittstelle `LowKrbTicket`).
- Der Sitzungsschlüssel wird mit Hilfe von `getCredential` ermittelt.
- Es wird folgendes Datenpaket an den Diensterbringer gesendet:

  $<$ VERSION , $K_{Service}\{\text{TICKET}\}$, $K_{Session}\{\text{AUTHENTICATOR}\}>$
- Anschließend wird die Antwort von Diensterbringer abgewartet. Diese muß wie folgt lauten:

  $< K_{Session}\{\text{CHKSUM+1}\}>$
- Die so empfangene Prüfsumme wird mit Hilfe des Sitzungsschlüssels dechiffriert und kontrolliert. Es findet also eine Authentisierung des Diensterbringers statt (diese letzten beiden Schritte finden allerdings nur dann statt, wenn die Option `KOPT_DO_MUTUAL` benutzt wird).

```
interface LowKrbRequest
 export

   KOPT_DONT_MK_REQ :word.T   (* Lege kein neues Ticket an.      *)
   KOPT_DO_MUTUAL   :word.T   (* Wechselseitige Authentisierung. *)

   send(options    :word.T              (* Eine der KOPT_.. Optionen *)
        sock       :socket.T            (* Vorbereiteter Socketport  *)
        ticket     :lowKrb.Ticket       (* Evtl. vorgegebenes Ticket *)
        service    :String              (* Dienst-Name               *)
        inst       :lowKrb.Instance     (* Dienst-Instanz            *)
        realm      :lowKrb.Realm        (* Dienst-Gebiet             *)
        chksum     :Int                 (* Pruefsumme fuer Dienst    *)
        mySocket   :socket.Address      (* Meine Adresse             *)
        toSocket   :socket.Address) :Ok (* Adresse des Dienstes      *)
```

```
  ...
end;
```

Um im Rahmen dieses Protokolls zu antworten, kann auf der Seite des Diensterbringers die Funktion `receive` benutzt werden. Diese arbeitet passend zu `send` mit folgendem Protokoll:

- Es werden (blockierend, also mit Warten auf die nächsten Daten) die vom Klienten versendeten Daten gelesen:

  $<$ VERSION , $K_{Service}\{\text{TICKET}\}$, $K_{Session}\{\text{AUTHENTICATOR}\}>$

- Die Version wird kontrolliert. Gegebenenfalls wird an den Klienten ein Protokollfehler zurückgemeldet.

- Der Authentifikator wird mit dem Sitzungsschlüssel aus dem Datenpaket gewonnen (ähnlich, wie bei der Funktion `readRequest` aus der Schnittstelle `LowKrbTicket`). Dabei findet auch die Authentisierung des Klienten statt, indem die Daten aus dem Authentifikator mit den Angaben im Ticket verglichen werden.

- Die Prüfsumme wird um eins hochgezählt und mit Hilfe des Sitzungsschlüssels chiffriert.

- Die so chiffrierte Prüfsumme wird an den Klienten zurückgesendet:

  $<K_{Session}\{\text{CHKSUM}+1\}>$

Die Funktion `receive` gibt den zwischenzeitlich gewonnenen Authentifikator des Klienten als Funktionswert zurück. Diesen benötigt der Diensterbringer evtl. noch, um mit dem Klienten weiter zu kommunizieren (wofür er unter Umständen den Sitzungsschlüssel benötigt).

```
interface LowKrbRequest
 export
   ...
   receive(options    :word.T          (* Eine der KOPT_.. Optionen  *)
           sock       :socket.T        (* Vorbereiteter Socketport    *)
           ticket     :lowKrb.Ticket   (* Zwischenspeicher            *)
           service    :String          (* Dienst-Name                 *)
           inst       :lowKrb.Instance (* Dienst-Instanz              *)
           fromSocket :socket.Address  (* Adresse des Klienten        *)
           mySocket   :socket.Address  (* Dienst-Adresse              *)
           keyFile    :String)         (* Service-Schluesseldatei     *)
          :lowKrbTicket.Authenticator  (* Authentifikator d.Klienten *)
 end;
```

3.8 Beispiele der Kerberos-Bibliothek

Die bisher beschriebenen Schnittstellen der Kerberos-Bibliothek sollen das Fundament für eine abstraktere Authentisierungsbibliothek bilden. Die Benutzung spezifischer, an das Konzept von Kerberos und der Socket-Kommunikation gebundener Authentisierungsdienstleistungen ist jedoch auch schon auf dieser Ebene möglich — wie die Beispielkomponente `LowKrbExamples` zeigen soll:

```
interface LowKrbExamples;
 export

   Principal <:Ok

   kinit(name :String lifetime :Int password :String) :Principal
   kdestroy(identity :Principal) :Ok

   client(identity :Principal host :String socket :Int)
   server(serverSocket :Int)
end;
```

Die beiden Funktionen `kinit` und `kdestroy` sind den gleichnamigen Kerberos-Hilfsprogrammen nachempfunden (siehe Kapitel 1.2.2) und zeigen wie mit Hilfe der Schnittstelle `LowKrbTicketFile` die initiale Authentisierung eines Benutzers gegenüber Kerberos durchgeführt werden kann.

Interessant ist hierbei, daß die Funktion `kinit` bei erfolgreicher Authentisierung einen Wert vom Typ `Principal` zurückgibt, der eine gegenüber Kerberos berechtigte Instanz, eine *bewiesene Identität* [1], repräsentiert. Da dieser Typ gekapselt ist, und `kinit` der einzige Wertkonstruktor für diesen Typ darstellt, kann die Benutzung dieses Typs als Parameter für andere Funktionen (wie hier für `kdestroy` oder `client`) eine erfolgreich durchgeführte Authentisierung erzwingen. Dies ist ein Beispiel dafür, wie mit Hilfe programmiersprachlicher Konzepte (Funktionen, Kapselung) eine erfolgreich durchgeführte Authentisierung erzwungen werden kann.

Auch die Serialisierung von Funktionsapplikationen kann durch einen solchen Typ erreicht werden: Die Funktion `kdestroy` zum Löschen aller Sicherheiten eines Principals kann erst dann aufgerufen werden, wenn mindestens einmal die Akquisition eines Tickets mittels `kinit` erfolgte.

Schließlich befindet sich in dieser Beispielkomponente noch die Demonstration einer authentisierten Kommunikation zwischen einem Klienten (über die Funktion `client`) und einem Diensterbringer (über die Funktion `server`). Da beide Funktionen auf jeweils unterschiedlichen Tycoon-Maschinen laufen können, bilden sie ein Beispiel für den authentisierten Dialog zweier entfernter Tycoon-Maschinen.

[1] Solche gekapselten Typen für Akteure, Identitäten oder Sicherheiten in Kerberos finden sich auch in der abstrakteren Authentisierungsbibliothek wieder (Kapitel 4.2 und Kapitel 4.3).

Beide Funktionen ähneln in ihrem Ablauf den Beispielprogrammen von Kerberos (dem *Sample-Client* und dem *Sample-Server*), wobei sich mit der Funktion `client` sogar direkt der Kerberos *Sample-Server* ansprechen läßt (wenn man als Wert für den Parameter `socket` eine 0 angibt). Die wechselseitige Authentisierung in diesen Funktionen wird im Kern, wie in Kapitel 3.7 zur Schnittstelle `LowKrbRequest` beschrieben, mit Prüfsummen durchgeführt. Dies ist hier noch einmal im groben Ablauf beschrieben:

- Für den Klienten (`client`):
  - Kommunikation mit dem Diensterbringer unter der gewünschten Rechneradresse eröffnen. Dies geschieht mittels des in Kapitel 2 schon beschriebenen Mechanismus für Sockets.
  - Senden der Authentisierung unter Angabe einer Prüfsumme an den Diensterbringer.
  - Antwort vom Diensterbringer lesen. Diese enthält evtl. einen Hinweis auf einen Fehler, falls z.B. die Authentisierung nicht korrekt war.
- Für den Diensterbringer (`server`):
  - Lesen auf den als Parameter angegebenen Socket-Port. Diese Leseoperation ist *blockierend*, es wird auf eine authentisierte Anfrage gewartet.
  - Empfangene Anfragen werden schon innerhalb der Kerberos-Funktion authentisiert. Ist diese Authentisierung erfolgreich (und nur dann), kann die vom Klienten gesendete Prüfsumme dechiffriert werden.
  - Die dechiffrierte Prüfsumme wird mit einigen zusätzlichen Daten an den Klienten zurückgesendet.

Betrachtet man sich die Realisierung dieser Beispiele im Modul `lowKrbExamples`, so fällt auf, daß die Benutzung der Tycoon-Schnittstellen auf dieser Ebene noch recht komplex ist. Insbesondere die weitere Kommunikation mit einer Verschlüsselung oder Signierung der Daten ist nicht so einfach und leicht zu realisieren, wie man es sich wünschen würde. Dies wird sich mit der im nächsten Kapitel beschriebenen Schicht der Authentisierungsbibliothek ändern, hier werden Sicherheitsdienstleistungen angeboten, die durch geeignete Abstraktionen wesentlich leichter zu verstehen und zu benutzen sind.

4. Die Authentisierungsbibliothek

Mit den in Kapitel 3 beschriebenen Schnittstellen kann man Authentisierungs- und Sicherheitsdienstleistungen anbieten, die auf Kerberos als Basissystem aufbauen, aber durch geeignete Abstraktionen wesentlich leichter zu verstehen und zu benutzen sind. Der Programmierer, der diese Entwicklungsschnittstelle benutzen möchte, soll von den systemspezifischen Details des Kerberos-Systems entbunden werden und mit einem überschaubaren Satz an Dienstleistungen die von ihm gewünschten Sicherheitsaskepte realisieren können.

Dazu werden zunächst eindeutige Identifikatoren für die Akteure, die als Teilnehmer eines Informationssystemes bestimmte Sicherheitsdienstleistungen beanspruchen wollen, definiert. Hierfür wird der schon in Kerberos geprägten Begriff des *Principals* benutzt. Diese Principals sind dabei immer an bestimmte Lokationen in einem potentiell weltweitem Netz gebunden, der *Domäne*, einem *Namensraum* innerhalb dessen ein Principal registriert ist.

Ein Principal kann leicht aus seinen Komponenten (Name, Instanz und Domäne) konstruiert werden, er steht zunächst für eine unbewiesene Identität in einem Informationssystem. Ein solcher Principal kann aber durch eine Authentisierung (Vorlage eines Geheimnisses gegenüber dem System, z.B. ein Paßwort) in eine bewiesene, *authentisierte Identität* überführt werden. Erst eine solche Identität ist in der Lage, *Berechtigungen* zu akquirieren, mit deren Hilfe bestimmte, durch Authentisierung geschützte Dienstleistungen beansprucht werden können.

Die oberste Schicht der Authentisierungsbibliothek stellt Sicherheitsdienstleistungen zur Verfügung, die mit Hilfe solcher Berechtigungen in Anspruch genommen werden können. Hierzu zählt beispielsweise die Signierung und Verschlüsselung von Daten, wobei die entsprechenden Daten bei der Überprüfung der Signatur bzw. der Entschlüsselung *typsicher* wiedergewonnen werden können. Auch die Authentisierung von Klienten und Diensterbringern zählt zu diesen Sicherheitsdienstleistungen, sie stellt einen ersten Schritt in Richtung eines generalisierten *Subjekt*-Begriffes dar, wie er in [RUD95] vorgestellt wird.

Den Abschluß bildet eine Beispielkomponente, die einen *Secure-Socket-Layer* in Tycoon auf Basis einer *Client-/Server-Authentisierung* realisiert. Die komplette Übersicht über die Authentisierungsbibliothek, einschließlich der in diesem Kapitel noch vorgestellten Schnittstellen, findet man in Abbildung 4.1.

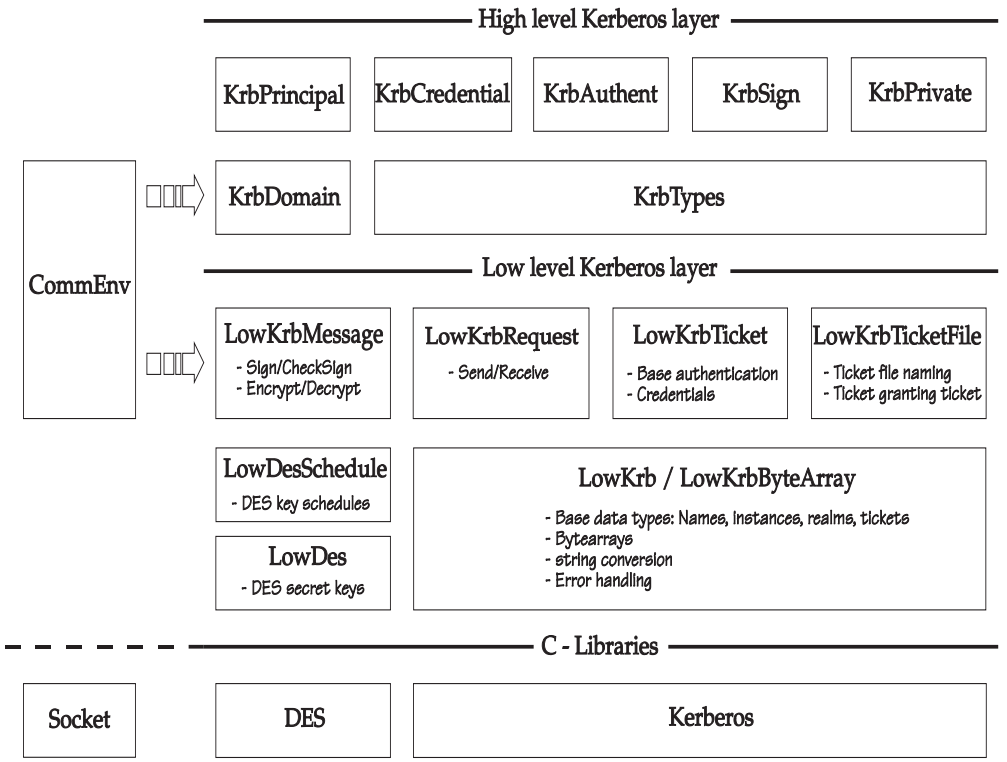

Abbildung 4.1: Die Authentisierungsbibliothek

4.1 Ort des Geschehens - die Domänen

Eine **Domäne** identifiziert einen *Namensraum*, innerhalb dessen ein *Principal* (siehe Kapitel 4.2) als Klient für Sicherheitsdienstleistungen registriert ist (solche Dienstleistungen sind z.B. die Durchführung von Authentisierungen oder die Beantragung von Berechtigungen). Die Domäne `T` aus der Schnittstelle `KrbDomain` ist ein wichtiger Bestandteil des Deskriptors, der einen Principal identifiziert (Kapitel 4.2).

Die Authentisierungsbibliothek nutzt diesen Domänentyp zusätzlich zur Abstraktion von der konkret verwendeten Netzwerkarchitektur innerhalb einer Domäne. Dazu enthält die Domäne neben einem Namen auch noch einen `Host`, ein Tupel, bestehend aus einem abstrakten Netzwerkdatentyp (aus der `commenv` Schnittstelle `NetAddress`) und einer aktuellen Netzwerkadresse. Die Funktion `current` die von der Schnittstelle `KrbDomain` angeboten wird, bestimmt zusätzlich zum Namen der Domäne diesen Netzwerkdatentyp (im Moment immer `netAddress.internet`) und initialisiert mit Hilfe dieses abstrakten Datentypen (der Funktion `netAddress.internet.current()`) die aktuelle Netzwerkadresse im Host-Tupel.

```
interface KrbDomain
export

  Let Host = Tuple
               network :netAddress.Network
               address :network.T
             end
  Let T    = KrbTypes.Domain

  new(name :String hst :Host) :T
  current() :T

  getHost(dom :T) :Host
  getName(dom :T) :String

  equal(dom1, dom2 :T) :Bool
end;
```

4.2 Principals als Akteure im Domänenraum

Ein **Principal** (*Hauptperson, Mandant*) identifiziert die Teilnehmer in einem Informationssystem (eines potentiell weltweiten Netzes), die bestimmte Sicherheitsdienstleistungen beanspruchen wollen. Ein solcher Identifikator setzt sich neben der Domäne (aus der Schnittstelle `KrbDomain`) aus einem Namen und einer Instanz (zur feineren Granularität eines Namensraumes) zusammen. Ein Principal kann sowohl Personen (*Benutzer, Klienten*), wie auch Prozesse (*Diensterbringer*) adressieren.

Die Schnittstelle `KrbPrincipal` stellt einen solchen Identifikator zunächst einmal als eine **behauptete Identität** (einen `Claim`, *Behauptung, Forderung*) zur Verfügung. Neben dem Konstruktor `new` und den entsprechenden Selektoren, erlaubt es die Funktion `iAm` aus dem Namen der aktuellen Betriebssystemidentität des aufrufenden Benutzers eine behauptete Identität zu konstruieren. Die Instanz bleibt dabei leer, und die Domäne wird mit `krbDomain.current()` vorbelegt.

Die Funktion `equal` stellt die Gleichheit zweier Principals fest, welche sich aus der Gleichheit von Namen, Instanz und Domäne des Principals ergibt.

```
interface KrbPrincipal
export

  Let Claim = KrbTypes.Claim

  new(name :String  instance :String  dom :krbDomain.T) :Claim
  iAm() :Claim
```

```
  getName    (principal :Claim) :String
  getInstance(principal :Claim) :String
  getDomain  (principal :Claim) :krbDomain.T

  equal(principal1, principal2 :Claim) :Bool
  ...
end;
```

Eine behauptete Identität kann sich gegenüber dem System (die durch die Domäne bezeichnete Instanz) durch Präsentation eines Geheimnisses authentisieren. Sie wird damit zu einer bewiesenen, **authentisierten Identität**, welche in der Lage ist, Berechtigungen zu akquirieren, Authentisierungen oder andere sicherheitssensitive Aktivitäten durchzuführen, die Bestandteil dieser Authentisierungsbibliothek sind.

Eine Identität ist ein Subtyp einer behaupteten Identität — die Selektoren und die Vergleichsfunktion für behauptete Identitäten sind auch auf authentisierte Identitäten anwendbar (*Subtyppolymorphismus*).

```
interface KrbPrincipal
export
  ...
  Let Identity = KrbTypes.Identity  (* Identity <: Claim *)
  Let Secret   = KrbTypes.Secret

  prove  (principal :Claim
          secret    :Secret
          lifetime  :Int) :Identity

  proveMe(secret   :Secret
          lifetime :Int) :Identity  (* prove(iAm() secret lifetime) *)
  ...
end;
```

Damit sich eine behauptete Identität gegenüber dem System authentisieren kann, wird sie meist ein *Paßwort* als Geheimnis angeben [1]. Da eine behauptete Identität aber auch einen Prozeß identifizieren kann und dieser in seinem Programmkode kein fest eingebautes Paßwort enthalten sollte, ist in manchen Authentisierungssystemen (wie z.B. in Kerberos) auch die Angabe einer *Service-Schlüsseldatei* als Geheimnis möglich, welche den geheimen Schlüssel des Prozesses bereithält. Die Authentisierung erfolgt in diesen Systemen dadurch, daß ein Lesezugriff auf diese Schlüsseldatei möglich ist, und

[1] Die Funktionen des eingesetzten Authentisierungssystemes sollten dafür sorgen, daß dieses Paßwort nicht, oder nur in verschlüsselter Form die lokale Workstation verläßt. Bei Kerberos ist dies sichergestellt (siehe Kapitel 3.5).

diese auch tatsächlich den geheimen Schlüssel der vom Prozeß vorgegebenen Identität enthält (siehe Kapitel 3.5).

Neben dem Vorweisen eines Geheimnisses ist auch noch die Angabe einer Lebensdauer für die Identität relevant, welche so klein wie möglich bemessen sein sollte. Um festzustellen ob, und wenn ja wie lange eine Identität noch gültig ist, werden über die Schnittstelle `KrbPrincipal` noch verschiedene Verifizierungsfunktionen zur Verfügung gestellt.

```
interface KrbPrincipal
export
  ...
  valid   (id :Identity) :Bool
  validFor(id :Identity target :Claim) :Bool

  validUntil   (id :Identity) :date.T
  validUntilFor(id :Identity target :Claim) :date.T

  invalidate(id :Identity) :Ok
 end;
```

Schließlich gibt es noch die Funktion `invalidate`, die eine Identität explizit zerstört, was mit dem Verlust sämtlicher, von dieser Identität erworbenen Berechtigungen verbunden ist.

4.3 Die Sicherheiten der Principals

Nachdem sich ein Akteur in einem Informationssystem als ein Principal benannt und identifiziert hat, kann er Berechtigungen akquirieren, oder an ihn gerichtete Berechtigungen anderer Akteure verifizieren. Solche Berechtigungen und Sicherheiten werden unter dem Oberbegriff des **Credentials** zusammengefasst, und dieser wird als abstrakter Datentypen `T` in der Schnittstelle `KrbCredential` zur Verfügung gestellt.

Diese Schnittstelle bietet für diesen Typen verschiedene Funktionen an, wie z.B. die Feststellung, *wem* eine Berechtigung gehört und für *wen* es als Sicherheit gedacht ist. Es existiert aber kein direkter Konstruktor für den Datentyp `T` — eine Berechtigung läßt sich nur in Form seiner konkreten Subtypen, als Berechtigung für einen Klienten (als `Credit`), oder als Sicherheit eines Diensterbringers, die er von einem Klienten bekommen hat (als `IdentityCard`), anlegen.

```
interface KrbCredential
export

  T <:Ok
```

```
  forWhom (id :krbPrincipal.Identity  cred :T) :krbPrincipal.Claim
  fromWhom(id :krbPrincipal.Identity  cred :T) :krbPrincipal.Claim
  ...
end;
```

Analog zu Identitäten haben auch Berechtigungen nur eine begrenzt gültige Lebensdauer, die deren Benutzung zeitlich limitiert. So stellt auch diese Schnittstelle Verifizierungsfunktionen zur Verfügung, die feststellen, *ob* und bis *wann* eine Berechtigung gültig ist. Die Lebensdauer einer Berechtigung kann allerdings in dieser Schnittstelle nicht direkt als Parameter angegeben werden, da in den meisten Authentisierungssystemen diese Zeitspanne durch den Systemadministrator, der die Kerberos-Datenbank verwaltet, festgelegt wird.

```
interface KrbCredential
export
  ...
  valid(id :krbPrincipal.Identity  cred :T) :Bool
  validUntil(id :krbPrincipal.Identity  cred :T) :date.T
  ...
end;
```

Durch weitere Funktionen können aus einer Berechtigung eine **Prüfsumme** und ein **Sitzungsschlüssel** gewonnen werden. Die Prüfsumme ist hier ein *Einmalwert* (ein **nonce**), und wird automatisch beim Anlegen einer Berechtigung über einen Zufallsgenerator erzeugt. Dieser Einmalwert ermöglicht die Authentisierung des Diensterbringers (siehe Kapitel 4.4). Der Sitzungsschlüssel erlaubt eine Datenverschlüsselung bzw. Datensignierung zwischen Klient und Diensterbringer (Kapitel 4.5).

```
interface KrbCredential
export
  ...
  getChecksum(id :krbPrincipal.Identity  cred :T) :Int
  getSecret(id :krbPrincipal.Identity  cred :T) :krbPrincipal.Secret
  ...
end;
```

Die konkreten Subtypen von `KrbCredential.T` werden durch `Credit`, als begrenzt gültige Erlaubnis für einen Klienten, eine bestimmte Dienstleistung zu nutzen, sowie durch `IdentityCard`, als eine begrenzt gültige Garantie, die ein Klient gegenüber einem Dienstleister erbracht hat, repräsentiert.

Damit eine Berechtigung zwischen einem Klienten und einem Diensterbringer auf einer offenen, unsicheren Netzwerkverbindung ausgetauscht werden kann, werden schließlich noch Sicherheiten in *transportfähiger Form* benötigt, was mit Hilfe des Datentypen **Ticket** realisiert wird. Ein Ticket besteht aus einem verschlüsselten Informationsblock, der nur eine sehr kurzzeitige Lebensdauer hat (in Kerberos 5 Minuten).

```
interface KrbCredential
export
  ...
  Credit <:T
  IdentityCard <:T

  Ticket <:Ok

  acquire(id     :krbPrincipal.Identity
          target :krbPrincipal.Claim)    :Credit

  getTicket(id :krbPrincipal.Identity cred :Credit) :Ticket

  proveTicket(id     :krbPrincipal.Identity
              ticket :Ticket
              client :krbPrincipal.Claim) :IdentityCard
end;
```

Ein Klient kann aus einem `Credit` (auch mehrfach) ein `Ticket` generieren. Der Diensterbringer kann aus einem solchen, für ihn bestimmten `Ticket` eine `IdentityCard` des Klienten gewinnen, mit der in Folge eine Authentisierung des Klienten möglich ist. Der Zusammenhang zwischen den verschiedenen Ausprägungen von Sicherheiten ist in der Abbildung 4.2 noch einmal in der Übersicht dargestellt.

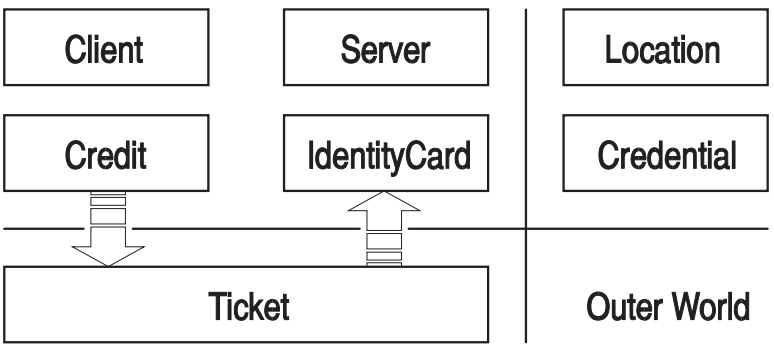

Abbildung 4.2: Sicherheiten und Tickets

Die Unterscheidung von `Credits` und `IdentityCards` garantiert die der Handhabung von Tickets zugrundeliegenden Asymmetrie. Ein Diensterbringer hat durch diese Asymmetrie keine Möglichkeit, aus einer `IdentityCard` in der Rolle eines Klienten ein neues Ticket zu generieren. Weiterhin unterstützt diese Unterscheidung (was im folgendem Kapitel 4.4 zu sehen ist) die Authentisierung von *fremden Identitäten*.

4.4 Authentisierung von Principals

Aufbauend auf die Sicherheiten aus der Schnittstelle `KrbCredential` kann man sich noch zusätzlich überlegen, mit Hilfe dieser Sicherheiten den entfernten Kommunikationspartner zu authentisieren.

Ein Principal würde Identifikatoren für solche Kommunikationspartner zunächst einmal mit Hilfe der Komponente `KrbPrincipal` aus deren Namen, Instanz und Domäne zusammensetzen — damit hätten alle Kommunikationspartner den Rang einer unbewiesenen Identität. Kann sich ein entfernter Kommunikationspartner aber authentisieren, darf er in Folge bestimmte, geschützte Dienstleistungen benutzen — ein entfernter, authentisierter Kommunikationspartner stellt damit eine **fremde Identität** dar.

Man erkennt deutlich die Parallele zur eigenen Identität, die ja auch aus einer behaupteten Identität mittels einer *Selbstauthentisierung* (Eingabe eines Paßwortes) gegenüber dem System entsteht. Äquivalent zu dieser eigenen Identität sind auch fremde Identitäten Subtyp von `krbPrincipal.Claim`.

Die Schnittstelle `KrbAuthentication` stellt zunächst mit dem Typ `T` eine abstrakte Fremdidentität zur Verfügung, auf die verschiedene Funktionen anwendbar sind. Hierzu zählen Verifizierungsfunktionen, die die Lebensdauer einer Fremdidentität überprüfen, sowie Funktionen die feststellen, *wessen* Fremdidentität referenziert wird, und von *wem* diese Fremdidentität nachgewiesen wurde.

```
interface KrbAuthentication
export

  T <:krbPrincipal.Claim

  valid     (id :krbPrincipal.Identity auth :T) :Bool
  validUntil(id :krbPrincipal.Identity auth :T) :date.T

  provedId(id :krbPrincipal.Identity auth :T) :krbPrincipal.Claim
  provedBy(id :krbPrincipal.Identity auth :T) :krbPrincipal.Claim
  ...
end;
```

Die konkreten Ausprägungen von fremden Identitäten werden in dieser Schnittstelle jeweils durch eine Fremdidentität für einen entfernten Klienten, der `ClientId`, und für einen entfernten Diensterbringer, der `ServerId` repräsentiert. Beide Fremdidentitäten

sind Subtyp von `T` und damit auch von `krbPrincipal.Claim`. Sie sind das Resultat der Authentisierungsfunktionen `proveClient` bzw. `proveServer`, über die sich der jeweilige Kommunikationspartner authentisiert.

```
interface KrbAuthentication
export
  ...
  ClientId <:T
  ServerId <:T

  proveClient(serverId :krbPrincipal.Identity
              idCard   :krbCredential.IdentityCard
              client   :krbPrincipal.Claim
             ) :ClientId

  proveServer(clientId        :krbPrincipal.Identity
              myCredit        :krbCredential.Credit
              server          :krbPrincipal.Claim
              serversChecksum :Int
             ) :ServerId
end;
```

Die Authentisierung des Klienten gegenüber dem Diensterbringer mittels der Funktion `proveClient` geschieht über die Vorlage der Berechtigung des Klienten (`IdentityCard`). In dieser steht die tatsächliche, vom Authentisierungssystem bestätigte Identität des Klienten, welche gegen die vom Kommunikationspartner vorgegebenen Daten abgeglichen werden. Die IdentityCard gewinnt der Diensterbringer vorher mit Hilfe der Funktionen aus der Schnittstelle `KrbCredential` aus dem Ticket, welches er vom Klienten bekommen hat. In einigen Authentisierungssystemen findet schon dabei die Authentisierung des Klienten statt — in diesen Systemen hat die Funktion `proveClient` nur noch den Charakter eines Konstruktors für die Fremdidentität des Klienten.

Die Authentisierung des Diensterbringers gegenüber des Klienten mittels der Funktion `proveServer` geschieht über die Vorlage einer Prüfsumme (ein vom Klient generierter Einmalwert), die der Diensterbringer dem Klienten (mit dem Sitzungsschlüssel chiffriert) zukommen läßt. Diese Prüfsumme gewinnt der Diensterbinger aus dem vom Klienten empfangenen Ticket, was nur dann funktioniert, wenn er der rechtmäßige Empfänger dieses Tickets ist (das Ticket ist mit dem geheimen Schlüssel des gewünschten Diensterbringers chiffriert). Die Vorlage der korrekten Prüfsumme garantiert dem Klienten, daß sein aktueller Kommunikationspartner auch der erwünschte Diensterbringer ist. Eine globale Übersicht über diese Authentisierungsprozesse findet man in Abbildung 4.3.

Eine Übersicht über die verschiedenen Ausprägungen von Identitäten, die aus einer behaupteten Identität abgeleitet werden können, stellt auch Abbildung 4.4 dar. Diese Abbildung zeigt, daß aus diesen verschiedenen Identitäten ein einheitlicher **Subjekt**-

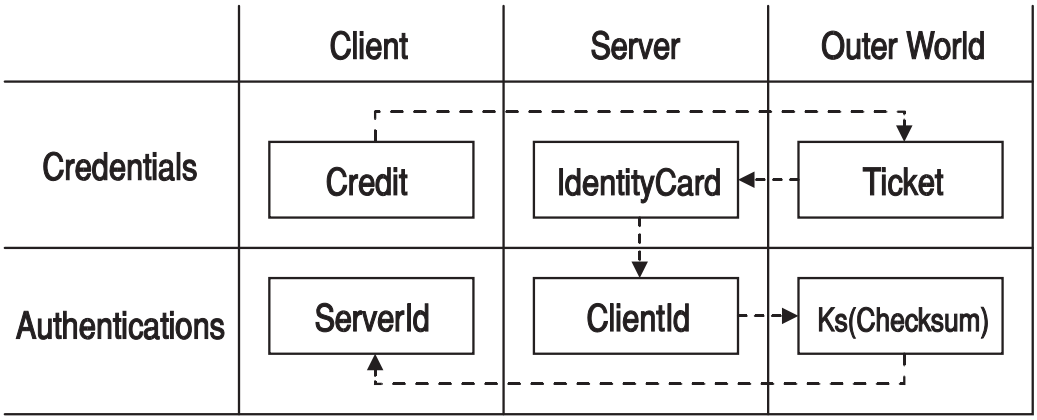

Abbildung 4.3: Sicherheiten und Authentisierung

Begriff aggregieren kann (Eigenschaften einer behaupteten Identität plus begrenzte Lebensdauer einer Identität). Ein solcher Subjekt-Begriff könnte eine Grundlage für die in [RUD95] beschriebenen Zugriffskontroll-Listen (*access control lists*) bilden.

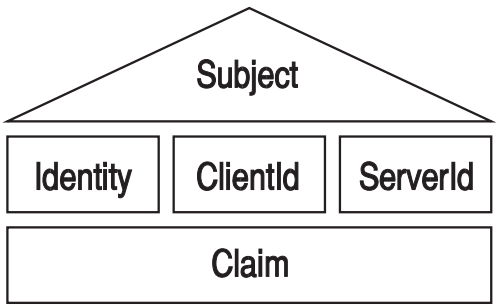

Abbildung 4.4: Principals und Identitäten

4.5 Signierung und Verschlüsselung von Daten

Die einheitliche Sicht auf den Datentyp `KrbCredential.T` sowohl für Klienten, wie auch für Diensterbringer und die Bereitstellung eines gemeinsamen Selektors für den Sitzungsschlüssel aus diesem Datentyp ermöglicht es, symmetrische Funktionen für die Signierung und die Verschlüsselung von Daten zu definieren. Dabei wird diese Signierung bzw. Verschlüsselung in der Signatur der entsprechenden Funktionen über *Typoperatoren* abgesichert (`Signed(A<:OK)` bzw. `Encrypt(A<:OK)`), so daß die strukturelle Typäquivalenz bei der Wiedergewinnung der Daten gewährleistet ist.

Es können alle Datenobjekte signiert bzw. verschlüsselt werden, die sich unter der Kontrolle des Objektspeichersystemes von Tycoon befinden. Auch die Resultate der Signierung und der Verschlüsselung stehen als *Bytearrays* (siehe Kapitel 3.1) unter der Kontrolle des Objektspeichersystemes, so daß sowohl die *orthogonale Persistenz* wie auch die *orthogonale Mobilität* der signierten bzw. verschlüsselten Datenobjekte gesichert ist.

4.5.1 Datensignierung

Die Signierung von Daten wird über die Schnittstelle `KrbSign` angeboten. Dabei besteht der mit der Funktion `signIt` ausgeübte Prozeß der Datensignierung in der Generierung eines *Message Integrity Codes* (MIC), was in Kapitel 1.1.5 genauer beschrieben ist. Der so generierte Hash-Wert schützt vor Manipulationen des Datenobjektes, verschlüsselt es aber nicht.

Bei der Wiedergewinnung des Datenobjektes mit Hilfe der Funktion `contents` muß der Hash-Wert korrekt wiederherstellbar sein. Ob dies möglich ist, kann vorab mit der Funktion `verify` überprüft werden.

```
interface KrbSign
export

  Signed(A <:Ok) <:Ok

  signIt(A     <:Ok
         iAm    :krbPrincipal.Identity
         cred   :krbCredential.T
         object :A) :Signed(A)

  contents(A     <:Ok
           iAm    :krbPrincipal.Identity
           cred   :krbCredential.T
           sign   :Signed(A)) :A

  verify(A    <:Ok
         iAm   :krbPrincipal.Identity
         cred  :krbCredential.T
         sign  :Signed(A)) :Bool
end;
```

4.5.2 Datenverschlüsselung

Die Verschlüsselung von Daten wird über die Schnittstelle `KrbPrivate` angeboten. Die Funktion `encrypt` verschlüsselt dabei ein Datenobjekt, so daß neben der Sicherheit gegen Manipulationen auch die *Vertraulichkeit* der verschlüsselten Daten garantiert ist — im Gegensatz zu signierten Objekten enthalten verschlüsselte Objekte keinen Klartext.

Besitzt man eine Berechtigung mit dem passenden Sitzungsschlüssel, kann man das Klartextobjekt mit der Funktion `decrypt` wiedergewinnen. Ob dies gelingt, kann mit der Funktion `verify` überprüft werden. Principals, die nicht im Besitz des passenden Sitzungsschlüssels sind, haben faktisch keine Möglichkeit das Klartextobjekt zurückzugewinnen.

```
interface KrbPrivate
export

  Encrypt(A <:Ok) <:Ok

  encrypt(A     <:Ok
          iAm    :krbPrincipal.Identity
          cred   :krbCredential.T
          object :A) :Encrypt(A)

  decrypt(A     <:Ok
          iAm    :krbPrincipal.Identity
          cred   :krbCredential.T
          crypt  :Encrypt(A)) :A

  verify(A    <:Ok
         iAm   :krbPrincipal.Identity
         cred  :krbCredential.T
         crypt :Encrypt(A)) :Bool
end;
```

4.6 Die Bindung an Kerberos

Die Authentisierungsbibliothek in Tycoon sollte möglichst unabhängig vom konkret verwendeten Authentisierungssystem sein (hier: Kerberos). Dies bedeutet insbesondere, daß keine systemspezifischen Attribute in den Schnittstellen dieser Bibliothek erscheinen sollten, was in Tycoon durch Kapselung bzw. *Information hiding* der Typdefinitionen in den Schnittstellen erreicht werden kann.

Dabei kommt es jedoch zu Problemen, wenn in verschiedenen Bibliotheksmodulen das Wissen um systemspezifische Datentypen notwendig wird. So benötigt das Modul zur Datenverschlüsselung (`krbPrivate`) als Sitzungsschlüssel den konkret in Kerberos eingesetzten `DES`-Schlüssel. In der Schnittstelle `KrbPrincipal`, indem mit `Secret` ein genereller Datentyp für solche Schlüssel definiert wird, sollten aber solche systemspezifischen Attribute gerade vermieden werden.

Um für die Authentisierungsbibliothek einen Kompromiß zu erreichen, wurden deshalb alle Datentypen, deren systemspezifischen Kerberos-Attribute modulübergreifend

benötigt werden, in der Schnittstelle `KrbTypes` offengelegt. Ein Beispiel hierfür ist der für den Sitzungsschlüssel benötigte Datentyp `Secret`:

```
Let Secret = Tuple
               case password with pWord  :String
               case keyFile  with kFile  :String
               case DESkey   with desKey :lowDes.T   (* <-- *)
             end
```

In der Schnittstelle `KrbPrincipal`, in der dieser Datentyp eigentlich als verborgener Datentyp definiert werden sollte, steht:

```
Let Secret = KrbTypes.Secret
```

Durch diese *Indirektion* wird in allen Schnittstellen der Authentisierungsbibliothek der direkte Zugriff auf systemspezifische Kerberos-Attribute vermieden, die Portabilität der Authentisierungsbibliothek zu anderen Authentisierungssystemen wird durch die zentrale Bindung dieser Attribute über die Schnittstelle `KrbTypes` insgesamt erhöht.

4.7 Sichere Socket-Kommunikation in Tycoon

Als kleines Anwendungsbeispiel für die Authentisierungsbibliothek soll die Implementierung einer Komponente zur sicheren Socket-Kommunikation in Tycoon dienen. Dabei sollen Teilnehmer in einem offenem Computernetzwerk in der Rolle von Identitäten (mit einer gültigen Identifikation gegenüber einer dritten Instanz) eine Kommunikation mit Sockets auf verschiedenen Sicherheitsstufen (Authentisiert, mit signierten Nachrichten, mit verschlüsselten Nachrichten) durchführen können.

Beispielsweise wollen Monika und Bob über eine sichere Socket-Verbindung Nachrichten austauschen. Diese Nachrichten sollen ruhig von jedem Benutzer im Netzwerk gelesen werden können, nur die Manipulation dieser Nachrichten durch Dritte soll mittels einer Signatur verhindert werden. Über eine weitere Socket-Verbindung möchte Bob mit seiner Bank Transaktionen tätigen, hier ist eine private, verschlüsselte Verbindung gefragt.

Wie in Kapitel 1.1 angedeutet, gibt es für eine sichere Kommunikation mit Sockets auch schon mit **SSL** ein standardisiertes, kommerziell eingesetztes Protokoll [SSL30a, SSL30b]. Dabei kann der Anwendungsprogrammierer zunächst wie gewohnt mit Sockets umgehen (mit den Funktionen `socket` anlegen, mit `bind` binden, etc.). Bei Bedarf kann dann ein Socket-Deskriptor an eine vorher initialisierte SSL-Struktur gebunden werden, mit der in Folge eine gesicherte Kommunikation möglich ist (Verbindungsaufbau über `SSL_connect` und `SSL_accept`, Kommunikation mit `SSL_read` und `SSL_write` [YOU95, YOU96]).

An diesem Konzept orientiert sich auch das hier als Anwendungsbeispiel entwickelte Tycoon Komponetne `SecureSocket`, welche einen sicheren Socket-Deskriptor in Form

des Typen `T` exportiert. Durch die Benutzung von Identitäten aus der Schnittstelle `KrbPrincipal` und begrenzt gültiger Sicherheiten aus der Schnittstelle `KrbCredential` entfällt jedoch die Handhabung von *Zertifikaten*, wie sie in SSL nötig sind. Der dort anfallende Verwaltungsaufwand wird hier quasi auf die Administration in Kerberos"abgewälzt". Ein weiterer Unterschied ist, daß SSL im Gegensatz zu Kerberos ein asymmetrisches Verschlüsselungsverfahren benutzt, was bei der Verschlüsselung längerer Nachrichten nicht sehr effizient ist.

Der Konstruktor für einen sicheren Socket-Deskriptor `T`, die Funktion `new`, benötigt neben der Identität des Benutzers und der gewünschten Sicherheitsstufe auf dem Kommunikationskanal, den schon geöffneten und gebundenen Socket-Deskriptor `handle`.

```
interface SecureSocket
export

  T <: Ok          (* Sicherer Socket Handle                        *)

  Let Security =
      Tuple case
        none,      (* Keine Sicherheit                              *)
        once,      (* Authentisierung bei Verbindungsaufbau         *)
        safe,      (* Wie 'once' mit Signierung aller Daten         *)
        private    (* Wie 'once' mit Verschluesselung aller Daten   *)
      end

  new(id       :krbPrincipal.Identity
      security :Security
      handle   :socket.T): T
  ...
end;
```

Die weitere Handhabung eines sicheren Socket-Deskriptors geschieht äquivalent zum SSL Protokoll. Ein Verbindungsaufbau ist mit den Funktionen `connect` und `accept` möglich, eine sichere Kommunikation mit `put` und `get`.

```
interface SecureSocket
export
  ...
  connect(secureSocket :T target :krbPrincipal.Claim) :Bool
  accept(secureSocket :T) :krbPrincipal.Claim

  put(A <:Ok secureSocket :T data :A) :Ok
  get(A <:Ok secureSocket :T) :A
end;
```

Mit dieser Schnittstelle könnte in unserem Beispiel mit Bob und Monika jetzt Bob wie Folgt auf eine Anfrage von Monika warten:

```
let sec = tuple_case safe of secureSocket.Security end;

let bobsSocket = secureSocket(bobsID, sec, socketHandle1);
let monika     = accept(bobsSocket);
let nachricht  = get(monika);
```

Anschließend könnte Monika mit Bob in Verbindung treten:

```
let monikasSocket = secureSocket(monikasID, sec, socketHandle2);
if connect(monikasSocket bobsClaim) then
   put(monikasSocket "Hallo Bob")
   let antwort = get(monikasSocket)
end;
```

Jetzt könnte Bob antworten. Er kann beliebige Tycoon-Daten senden, alle Daten werden vor dem Versenden signiert:

```
put(monika tuple "Hallo Monika" 4711 { fun(val :Int) val + 1 } end)
```

5. Zusammenfassung und Ausblick

Im Rahmen dieser Studienarbeit wurde eine zweischichtige Authentisierungsbibliothek in Tycoon geschaffen. Die untere Bibliotheksschicht stellt Sicherheitsdienstleistungen zur Verfügung, die sich unmittelbar an Kerberos orientieren, die obere Schicht stellt Authentisierungsdienste bereit, welche weitgehend von Kerberos und den Details der Kommunikationsschicht (z.B. Sockets oder Internetadressen) abstrahieren.

In der unteren Bibliotheksschicht wurden mit den *Bytrearrays* ein adäquates Mittel zur Abspeicherung aller C-nahen Daten im Tycoon Objektspeichersystem gefunden. Wer Authentisierung systemnah mit Kerberos betreiben möchte, wird schon hier reichlich mit Funktionen zur Datentenverschlüsselung, zur Akquisition von Berechtigungen oder zur Authentisierung über bestehende Socket-Verbindungen bedient. Dabei findet man auch in dieser Schicht schon höhere programmiersprachliche Abstraktionen vor, die die Benutzung dieser Schicht vereinfachen, wie beispielsweise die abstrakten Datentypen für alle Basisdatentypen von Kerberos.

Die Abstraktionen in der oberen Schicht der Authentisierungsbibliothek gehen aber in der einfachen und intuitiven Benutzung der angebotenen Sicherheitsdienstleistungen noch deutlich weiter. So bewirkt beispielsweise der Einsatz von Typoperatoren in der Schnittstelle bei den Verschlüsselungs- und Signierungsfunktionen, daß Werte aller Tycoon-Datentypen durch diese Operationen typsicher verschlüsselt bzw. signiert werden können. Der Benutzer solcher Funktionen braucht sich keine Gedanken über spezifische Datenstrukturen oder Typkonvertierungen zu machen, er profitiert hier von der *Orthogonalität* dieser Funktionen im selben Maße, wie er es in andernen Bereichen von Tycoon auch gewohnt ist (*orthogonale Persistenz* bzw. *orthogonale Mobilität*). An der Beispielkomponente für eine sichere Socket-Verbindung wird ferner demonstriert, wie einfach und intuitiv die Entwicklungsschnittstelle der Authentisierungsbibliothek dazu eingesetzt werden kann, um auch komplexere Sicherheitsmechanismen zu realisieren.

Für die weitere Entwicklung dieser Bibliothek erscheint es zunächst als sehr wichtig, die untere Bibliotheksschicht an die Kerberos Version 5 anzupassen, da die Weiterentwicklung und Portierung von Kerberos zunehmend nur noch diese neue Version betreffen. Als weiteren Schritt könnten noch zusätzliche Administrationsdienste von Kerberos in die untere Bibliotheksschicht aufgenommen werden, wie z.B. die Verwaltung der Masterdatenbank oder die Generierung von Service-Schlüsseldateien.

Auch die Realisierung eigener, besser abgesicherter Administrationsdienste in Tycoon wäre möglich (Persistenz/Skalierbarkeit). Die Haltung administrativer Daten in verschlüsselten, persistenten Objekten, welche sich in einem über eigene Zugriffskon-

trollen abgesicherten Objektspeichersystem befinden, bietet wesentlich mehr Sicherheit, als die zum Teil unverschlüsselte Ablage von Tickets und Verwaltungsinformationen im UNIX-Filesystem. Den Kerberos-Routinen könnte man diese neue Funktionalität über *C-Callbacks* "unterschieben".

Derartige Erweiterungen wären aber nicht ganz im Sinne der bestehenden Authentisierungsbibliothek, da diese ja vorhandene Funktionalität über *externe Dienste* einbinden, und dabei möglichst unabhängig vom konkret benutzten Authentisierungssystem bleiben will (neues System $\rightarrow$ neue Features). Die Realisierung eigener Administrationsfunktionen über C-Callbacks würde jedoch eine zu große Abhängigkeit vom konkret benutzten System, bzw. der aktuellen Systemversion schaffen.

Dagegen ist die obere Schicht der Authentisierungsbibliothek noch beliebig ausbaubar. Beispielsweise wäre noch eine weitere Erhöhung des Abstraktionsgrades möglich, um die Sicht auf die Kerberos-Typschnittstelle `KrbTypes` zu verbergen. Auch wäre noch eine Erweiterung der angebotenen Authentisierungsdienste möglich, z.B. um die in [RUD95] beschriebenen Zugriffskontroll-Listen (*access control lists*).

Schließlich wäre auch noch die Realisierung einer größeren Beispielanwendung, die die Authentisierungsbibliothek nutzt, wünschenswert, beispielsweise ein *sicherer Tycoon Mailreader*.

Literaturverzeichnis

[ATKI95] R.Atkinson; *Security Architecture for the Internet Protocol*; RFC 1825, August 1995.

[BALE93] D.Balenson; *Privacy Enhancement for Internet Electronic Mail; Part III: Algorithms, Modes, and Identifiers*; RFC 1423, February 1993.

[BAN89] M.Burrows, M.Abadi and R.Needham; *A logic of authentication*; Technical report, DEC System Research Center, 1989.

[BELL90] S. Bellovin and M.Merritt; *Limitations of the Kerberos Authentication System*; Computer Communications Review, Vol 20 #5, October 1990, S.119-132.

[BELL94] Steven Bellovin and William Cheswick; *Firewalls and Internet Security: Repelling the Wily Hacker*; Addison-Wesley 1994, ISBN 0-201-63357-4.

[BU93] Michael Busch; *Kommunikative Eleganz — Sockets unter BSD-UNIX*; Multiuser Multitasking Magazin, iX 2/1993, S.156-158, Heinz Heise Verlag, Hannover.

[CARD90] L.Cardelli; *The Quest Language and System (Tracking Draft)*; Technical Report, DEC Systems Research Center, Palo-Alto, California, 1990.

[CHAP95] D.Brent Chapman and D.Zwicky; *Building Internet Firewalls*; O'Reilly Associates Inc. 1995, ISBN 1-56592-124-0.

[COME95] Douglas Comer; *Internetworking with TCP/IP; Vol I: Principles, Protocols, and Architecture*; Prentice-Hall, Third Edition 1995, ISBN 0-13-216987-8.

[COME94] Douglas Comer and David L.Stevens; *Internetworking with TCP/IP; Vol II: Design, Implementation, and Internals*; Prentice-Hall, Second Edition 1994, ISBN 0-13-134677-6.

[DES77] *Data encryption standard*; Federal Information Processing Standards, no.46, National Bureau of Standards, U.S. Department of Commerce, 1977.

[DES81] *DES Modes of Operation*; Federal Information Processing Standards, no.81, National Bureau of Standards, U.S. Department of Commerce, 1981.

[GOLD92] Fred R. Goldstein *ISDN in Perspective*; Addison-Wesley 1992, ISBN 0-201-50016-7.

[HEAP94] Nicholas Heap; *OSI-Referenzmodell ohne Geheimnis*; Heinz Heise Verlag, Hannover 1994, ISBN 3-88229-045-5.

[HEIN94] Mathias Hein and Wolfgang Kemmler; *FDDI - Fibre Distributed Data Interface*; International Thomson Publishing 1994, ISBN 3-929821-52-4.

[IPNG] WWW-Page of the Internet Protcol Next Generation Working Group; *IP: Next Generation*; http://www.ietf.cnri.reston.va.us/ipng/ipng.html.

[KALI92] B.Kaliski; *The MD2 Message-Digest Algorithm*; RFC 1319, April 1992.

[KALI93] B.Kaliski; *Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services*; RFC 1424, February 1993.

[KAUF95] Charlie Kaufman, Radia Perlman and Mike Speciner; *Network Security - PRIVATE Communication in a PUBLIC World*; Prentice-Hall PTR 1994, ISBN 0-13-061466-1.

[KENT93] S.Kent; *Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management*; RFC 1422, February 1993.

[KERN89] H.Kerner; *Rechnernetze nach ISO-OSI, CCITT*; H.Kerner, A-3012 Wolfsgraben, 1989, ISBN 3-900934-10-X.

[KOHL93] J.Kohl and C.Newman; *The Kerberos Network Authentication Service (V5)*; RFC 1510, September 1993.

[KROL94] Ed Krol; *The Whole Internet User's Guide & Catalog*; O'Reilly/International Thomson Verlag, 2nd Edition April 1994, ISBN 1-56592-063-5.

[KROL95] Ed Krol; *Die Welt des Internet — Deutsche Ausgabe*; O'Reilly/International Thomson Verlag, 1.Auflage 1995, ISBN 3-930673-01-0.

[LAUE95] Thomas Lauer; *CompuServe professionell — Weltweit Informationen, Know-How und Daten austauschen*; Addison-Wesley, 2.Auflage 1995, ISBN 3-89319-937-3.

[LEFF89] Leffler, McKusick, Karels and Quarterman; *The Design and Implementation of the 4.3BSD UNIX Operating System*; Addison-Wesley 1989, ISBN 0-201-06196-1.

[LINN93] J.Linn; *Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*; RFC 1421, February 1993.

[MATT91] Florian Matthes; *P-Quest: Installation and User Manual*; DBIS Tycoon Report 101-91, Fachbereich Informatik, Universität Hamburg, Germany, October 1991.

[MATT93] Florian Matthes; *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*; Springer-Verlag 1993, ISBN 3-540-56581-7.

[MMS94] F. Matthes, S.Muessig, and J.W Schmidt; *Persistent Polymorphic Programming in Tycoon: An Introduction*; FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, Germany, August 1994.

[MORI93] Peter Moritz; *ISDN aufgelöst — Fähigkeiten und Grenzen des digitalen Netzes*; Magazin für Computertechnik, c't 8/1993, S.100ff., Heinz Heise Verlag, Hannover.

[NEED78] R.Needham and M.Schroeder; *Using Encryption for Authentication in Large Networks of Computers*; CACM 21, 12 (Dec 1978), p.993-999.

[NEUM87] S.P.Miller, B.C. Neuman, J.I. Schiller and J.H. Saltzer; *Kerberos Authentication and Authorisation System*; Project Athena Technical Plan, MIT 1987.

[SSL30a] Netscape Communications Corporation; *Standards Documentation: The SSL Protocol*; http://home.mcom.com/newsref/std/SSL.html.

[SSL30b] Netscape Communications Corporation; *The SSL 3.0 Specification (Index)*; http://home.netscape.com/eng/ssl3/index.html.

[RFC1057] RFC 1057, RPC: *Remote Procedure Call Protocol Specification Version 2.*

[RFC1094] RFC 1094, NFS: *Network File System Protocol Specification.*

[RESN94] Rosalind Resnick and Dave Taylor; *The Internet Business Guide — Riding the Information Superhighway to Profit*; Sams Publishing 1994, ISBN 0-672-30530-5.

[RIVE92a] R.Rivest; *The MD4 Message-Digest Algorithm*; RFC 1320, April 1992.

[RIVE92b] R.Rivest; *The MD5 Message-Digest Algorithm*; RFC 1321, April 1992.

[RSA78] R.Rivest, A.Shamir and L.Adleman; *A method for obtaining digital signatures and public key cryptosystems*; Communications of the ACM, Vol 21 #2, February 1978, S.120ff.

[RUD95] A.Rudloff, F.Matthes and J.W.Schmidt; *Security as an Add-On Quality in Persistent Object Systems*; FIDE Technical Report FIDE/95/138, Fachbereich Informatik, Universität Hamburg, Germany 1995.

[SALO90] A.Salomaa; *Public-Key Cryptography*; Springer-Verlag 1990, ISBN 3-540-52831-8.

[SNS88] J.G.Steiner, B.C.Neumann and J.I.Schiller; *Kerberos: An authentication service for open network systems*; Proceedings of the Winter 1988 Usenix Conference, February 1988.

[STAL95] William Stallings; *Datensicherheit mit PGP*; Prentice Hall 1995, ISBN 3-930436-28-0.

[STEV92a] W.Richard Stevens; *Programmierung von UNIX-Netzen*; Deutsche Übersetzung: Hauser-Verlag 1992, ISBN 3-446-16318-2.

[STEV92b] W.Richard Stevens; *Advanced Programming in the UNIX Environment*; Addison-Wesley 1992, ISBN 0-201-563177-7.

[SUNOS40] Sun OS 4.0 Manual; *Security Features Guide / Network Programming*.

[WIEH95] Lothar F.Wiehler; *ISDN — Eine praxisnahe Einführung*; Addison-Wesley 1995, ISBN 3-89319-905-5.

[YOU95] E.A. Young; *SSLeay v0.5.1 — A free implementation of SSL*; December 1995, ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL.

[YOU96] E.A. Young, T.J.Hudson; *SSL — The SSLeay Programmers Reference (Draft)*; ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL/SSLeay.doc-1.5.tar.gz.