Zertifikatbasierte Authentisierung für CORBA

Christophe Andrey

Diplomarbeit

Ecole Polytechnique Fédérale de Lausanne

Freie Universität Berlin

Februar 1999

Dieser Bericht beschreibt die Erfüllung der Aufgabe meiner Diplomarbeit: die Implementierung der Authentisierung in JacORB, eine Java-Implementierung von CORBA, mit Hilfe von SPKI-Zertifikaten.
Der Bericht erklärt zuerst die theoretischen Hintergründe : CORBA-Sicherheit und Zertifikatinfrastrukturen. Dann wird ein Authentisierungsprotokoll fuer verteilte Objekte begründet. Schliesslich werden die Architektur der Implementierung und die Gründe für meine Designentscheidungen vorgestellt.

Einleitung

Obwohl Sicherheit eine wesentliche Anforderung für grosse und kritische verteilte Systeme ist, wurde bis jetzt der Sicherheitdienst von CORBA nur von wenigen Implementierungen unterstützt und auf jeden Fall nie vollständig. Diejenigen, die ihn teilweise implementiert haben, haben meistens keine neue Technik dafür entwickelt, sondern haben existierende Verfahren verwendet, wie z.B. DCE.
Parallel dazu hat sich Public-Key-Kryptographie vor ungefähr 10 Jahren als ein zuverlässiges Mittel erwiesen, Vertrauensbeziehungen zwischen gegenseitig Unbekannten durch Zertifizierungseinrichtungen aufzunehmen. Trotzdem sind die heutigen Zertifikatinfrastrukturen entweder zu steif (X.509) oder zu spezialisiert (PGP), um auf die allgemeine Sicherheit heterogener, verteilter System angewendet zu werden.
Eine neue Zertifikatinfrastruktur, SPKI, vereinigt Organisationsflexibilität und universelle Aussagekraft.
Diese habe ich auf den Sicherheitsdienst von JacORB, eine Java Implementierung von CORBA, angewendet. Genauer gesagt habe ich Authentisierung, Grundlage für die Erfüllung der meisten anderen Sicherheitsanforderungen auf einem System, mit Hilfe von SPKI-Zertifikaten implementiert.

Die drei ersten Kapitel dieses Berichtes erklären den theoretischen Hintergrund meiner Diplomarbeit, während die drei letzten Kapitel beschreiben, was ich genau realisiert habe.
In Kapitel 1 wird CORBA grob vorgestellt, der Sicherheitsdienst genauer beschrieben und eine Implementierung (JacORB), mit der ich gearbeitet habe, eingeführt. Dann werden die Anforderungen meiner Aufgabe dargestellt: die Realisierung der Authentisierung in JacORBs Sicherheitsdienst.
In Kapitel 2 werde ich mich auf eine Teilmenge von Lösungen zu diesem Problem beschränken: Zertifikatbasierte Lösungen. Nach einer Erklärung der Prinzipien von Zertifikaten werde ich zwei existierende Zertifikatinfrastrukturen (X.509, PGP) und ein Modell (PolicyMaker) vergleichen.
In Kapitel 3 wird eine neue, vierte und für meine Implementierung gewählte Infrastruktur vorgestellt: SPKI. Genauer erkläre ich ihre Prinzipien, die Signaturverfahren, die sie unterstützt und die Grammatik, die sie zur Serialisierung der Zertifikate definiert.
In Kapitel 4 wird ein auf SPKI basierendes Protokoll für die Authentisierung des Aufrufers mit einem Serverobjekt, bei einem Fernaufruf in JacORB, detailliert und begründet.
Kapitel 5 beschreibt meine Implementierung dieser Protokolle und eines Teils der SPKI-Infrastruktur in Java. Die Architektur meiner Implementierung wird dargestellt, meine Designentscheidungen begründet und die wichtigsten Klassen vorgestellt.
Als Veranschaulichung der Protokolle und meiner gesamten Implementierung habe ich eine Beispielanwendung entwickelt: Authentisierung zum Zugriffschutz des JacORB Namensdienstes. Am Ende des fünften Kapitels wird sie vorgeführt.

Nach einem Schluß, der Wege für zukünftige Arbeiten entwirft, enthält dieser Bericht noch drei Anhänge. Der erste ist die Liste aller Grammatikregeln, die die Serialisierung von SPKI-Zertifikaten definieren. Der zweite enthält mehrere Klassendiagrammen, aus denen meine Implementierung resultiert, und der dritte enthält die Dokumentation meines Codes in Javadoc Format.

1.0 Problemstellung: Authentisierung in JacORB

1.1 CORBA

1.1.1 Überblick

Die *Object Management Infrastructure* [29] (OMA) ist eine standardisierte Spezifikation der *Object Management Group* (OMG) für Plattformen, die Objektfernaufrufe in heterogenen und verteilten Systemen unterstützen. Die OMG [30] ist ein Konsortium von mehr als 700 Unternehmen weltweit. Der Kern der OMA ist die *Common Object Broker Architecture* [22] (CORBA). Ihre heutige Version ist 2.3.
Verschiedene CORBA Implementierungen [23] sind auf dem Markt verfügbar. Sowohl Programmiersprachen, die sie unterstützen, als auch das Ausmaß der Spezifikation, das sie tatsächlich implementieren, sind unterschiedlich.

Eines der Hauptziele von CORBA ist es, Ortstransparenz der Fernaufrufe zwischen verteilten Objekten anzubieten. So soll der Aufrufer (Klient) nicht bemerken, ob das Objekt (Server), das er aufruft, auf dem lokalen Rechner liegt oder anderswo im Internet, auf einem Rechner, auf dem ein ganz anderes Betriebsystem läuft. Anders als in RPC sind die Klient-/Serverrollen nicht festgelegt: das Serverobjekt kann seinerseits ein anderes Objekt aufrufen (als Teil des ursprünglichen Aufrufs oder davon unabhängig) und sich somit als Klient verhalten.

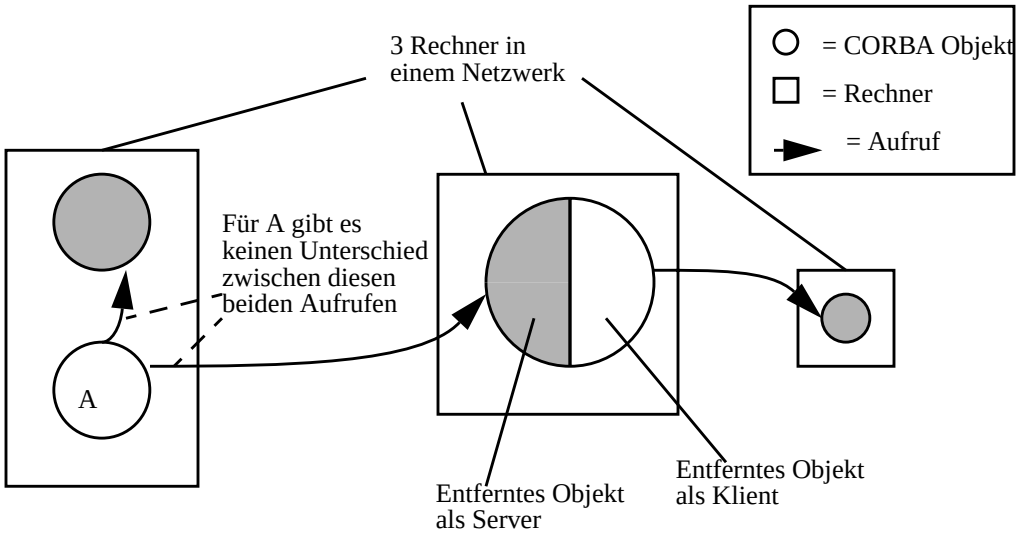


Abbildung 1. CORBA : Ortstransparenz und Klient-/Serverrollen

Außerdem soll die Interoperabilität von Systemen zwischen verschiedenen CORBA-Plattformen gelten. Als Erweiterung der Orts- und Betriebsystemstransparenz soll letztlich ein Aufruf unabhängig von der Sprache sein, in der der Klient und der Server programmiert sind.

Diese drei Ziele sind sicher nicht die einzigen, die die Entwicklung von CORBA bestimmen (wahrscheinlich hat jedes Mitglied der OMG seine eigenen Ziele), aber sie sind die Existenzberechtigung für mehrere Mechanismen, die im Rest dieses Berichts erklärt werden.

1.1.2 Software-Bus

Um das erste Ziel (die Ortstransparenz) zu erreichen verfügt CORBA über einen *Object Request Broker* (ORB), der sich als Software-Bus verhält. Abbildung 2 zeigt ein typisches und einfaches Szenario eines Fernaufrufs.

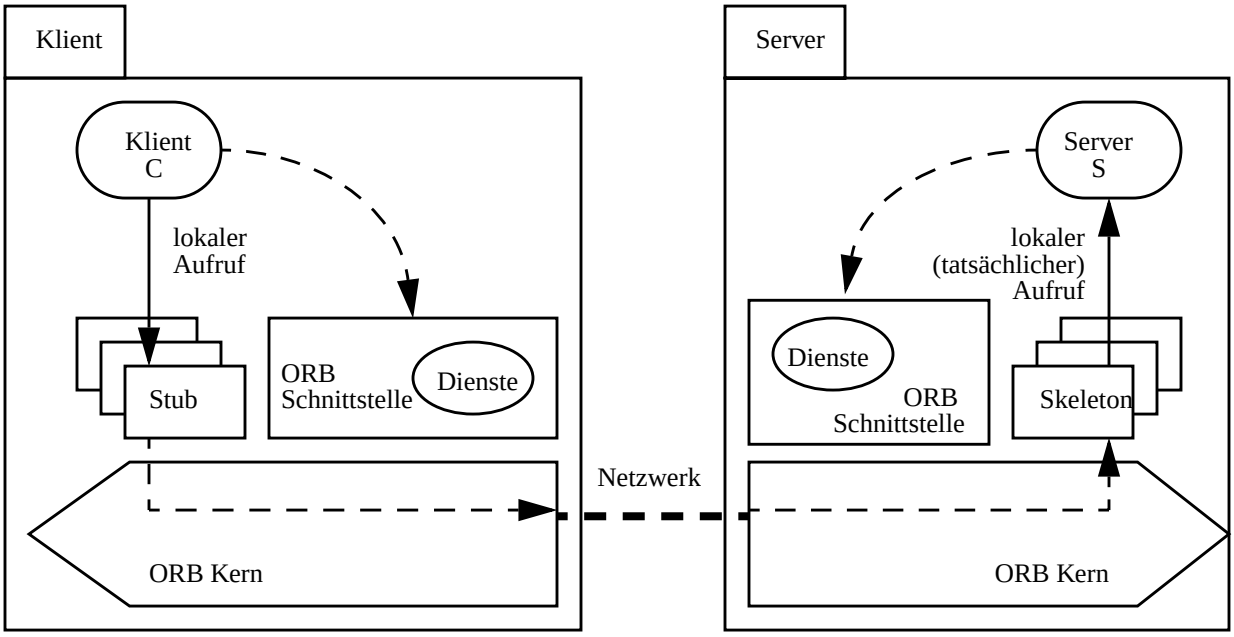


Abbildung 2. Der ORB

Ein Klientenprozess C (selbst nicht notwendig ein Objekt) will ein entferntes Objekt S aufrufen. Sowohl der Klient als auch das Serverobjekt benutzen den ORB. Durch einen vorigen Fernaufruf oder durch eine Anfrage an den Namensdienst des ORBs (später erklärt) besitzt der Klient *die Interoperable Object Reference* (IOR) von S, d.h. eine ortstransparente Referenz auf S. In der Umgebung von C gibt es auch einen Stub von S, d.h. ein Objekt, das die gleiche Schnittstelle wie S hat, das aber lokal ist. Um S aufzurufen wird C eigentlich den Stub lokal aufrufen. Dieses verhält sich als Proxy : es gibt

den Aufruf an den ORB weiter, der ihn serialisiert und ihn seinem Gegenstück ORB bei S sendet. Da deserialisiert der ORB den Aufruf und gibt ihn an ein Skeleton weiter. Ein Skeleton ist ein Objekt, das nebenläufige Aufrufe von S behandelt und letztlich S lokal aufruft. Der Rückgabewert wird danach entsprechend von S bis C vermittelt. Alle Skeletons haben die gleiche Schnittstelle zum ORB.
Das Format des serialisierten Aufrufs wird von CORBA im *General Inter-ORB Protocol* (GIOP) genau spezifiziert. So können zwei ORBs ganz verschiedener Hersteller kollaborieren, und so wird das zweite erwähnte Ziel (Interoperabilität) erfüllt.

Wir haben schon gesehen, daß C lokal die Schnittstelle von S kennen muß. Das wäre schwierig, wenn C in einer anderen Sprache als S implementiert wäre. Deshalb müßen diese Schnittstellen sprachunabhängig sein. Dafür hat die OMG eine *Interface Definition Language* (IDL) sowie ihre Mapping zu existierenden, bekannten Programmiersprachen als Teil CORBAs definiert.

1.1.3 IDL

IDL ist rein deklarativ, d.h. er beschreibt nur Typen und Methodensignaturen.

Verwendung

Abbildung 3 zeigt den typischen Prozeß, ein CORBA-fähiges Objekt zu programmieren.

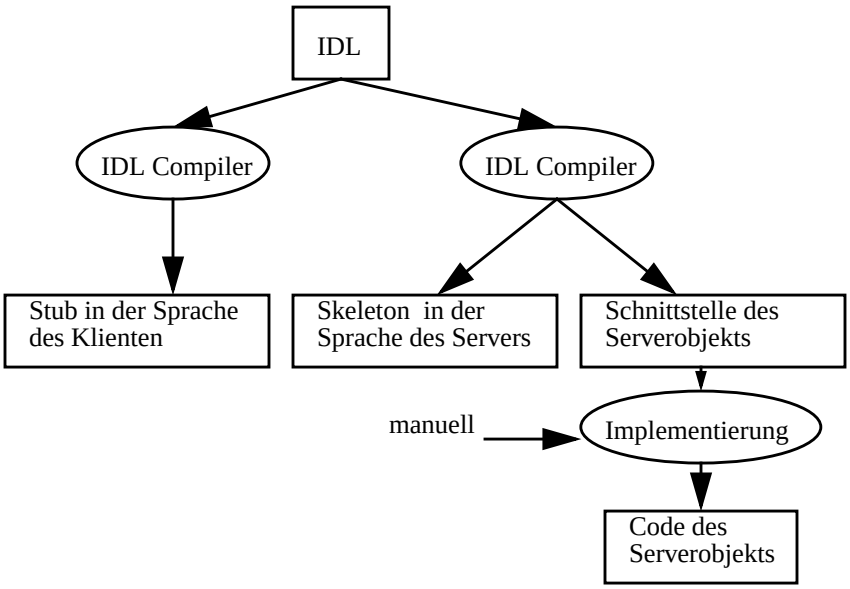


Abbildung 3. Prozess, ein CORBA-fähiges Objekt zu programmieren

Erstens schreibt man seine IDL Schnittstelle. Dann werden von einem IDL Compiler (ein Werkzeug, das gewöhnlich mit CORBA-Implementierung zusammen geliefert wird) drei Dokumente automatisch erzeugt: der Code des Stubs (in der Sprache des Klienten), der Code des Skeletons und der Schnittstelle des Objektes selbst (beide in der Sprache des Servers). Schließlich implementiert man diese Schnittstelle.

Eigenschaften

Eine grundsätzliche Eigenschaft von IDL ist Objektorientierung. So kann man Schnittstellen definieren, die Methoden enthalten. Diese sind Objekttypen. Auch strukturierte Typen können definiert werden, mit Hilfe von den Schlüsselwörtern `struct`, `array`, `union`, `sequence` (Array variabler Lange), `enum`. Aber sie sind keine ADT, sondern Aliases, wie von `typedef` definierte Typen in C.
Diese Definitionen stützen sich auf vordefinierte, elementare Typen, wie `short`, `long`, `float`, `double`, `char`, `boolean`, `octet`, `string`, `any` (entspricht irgendeinem Typ), `exception`. Schnittstellen (sogenannte *interfaces*) sind in *Modules* gruppiert, die sich wie *Packages* in Java verhalten.
Außerdem unterstützt IDL mehrfache Vererbung, aber kein Überladen von Operationsnamen.
Schnittstellen können als *locally constrained* definiert werden, um zu verhindern, daß sie entfernt aufgerufen werden. Das wird z.B. benutzt um Komponente des ORBs sprachunabhängig zu spezifizieren. En solcher Fall ist das *Principal Authenticator* Objekt, das im Kapitel 6 auftauchen wird.
Die OMG hat für C, C++, Java, Smalltalk, Ada und Cobol eine genaue Abbildung aus IDL definiert, die von IDL Compilern implementier wird.

So erfüllt IDL das dritte Ziel, das unter 1.1.1 erwähnt wurde

1.1.4 Dienste

Allgemeines

Abbildung 2 zeigt, daß sowohl Klienten als auch Serverobjekte den ORB direkt aufrufen können. In der Umgebung des ORBs werden noch Dienste (sogenannte *Common Object Services* (COS)) in der OMA angeboten. Außerdem spezifiziert die OMA auch *Common Facilities*, die die Standardisierung auf das Anwendungsniveau bringen, und nicht mehr nur aufs Systemniveau wie Dienste. *Common Facilities* sind als *vertical* oder *horizontal* unterschieden, je nachdem, ob sie geschäftsspezifisch sind oder nicht.

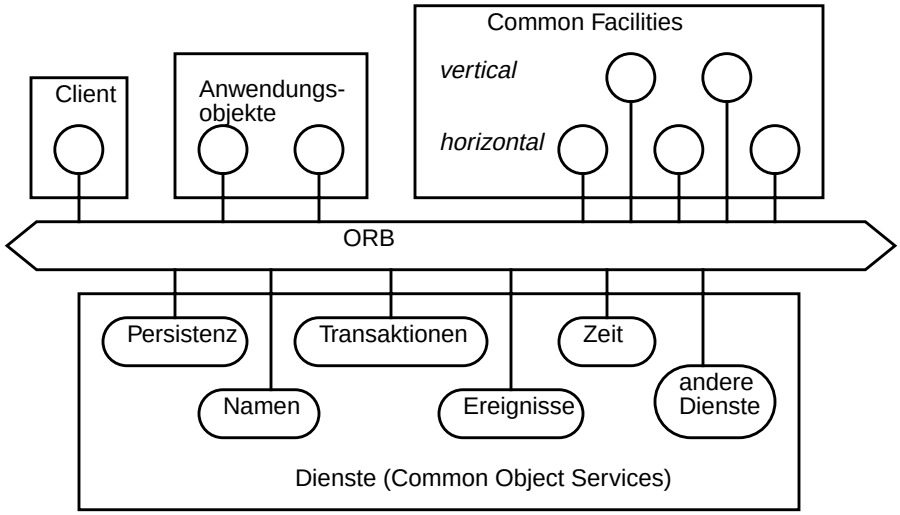


Abbildung 4. Die Umgebung des ORBs : OMA

Die OMG hat eine Anzahl von Diensten spezifiziert [18], z.B. für die Persistenz, für Transaktionen oder für die Nebenläufigkeitskontrolle.
Hier werden nur zwei von ihnen eingeführt: der Namensdienst, der als Experimentierfeld für meine Implementierung im Kapitel 6 genutzt wird, und der Sicherheitsdienst, Rahmen meiner ganzen Arbeit.

Namensdienst

Der Namensdienst ist ein Verzeichnis, das Namen mit Objektreferenzen verbindet. Er ist hierachisch strukturiert, d.h. daß Verzeichnisse ineinander geschachtelt sind. Für einen Benutzer des ORBs stellt er sich als entferntes Objekt dar, auf dem sich Anfrage- und Bindungsoperationen aufrufen lassen. So kann ein Klient die IOR eines Objektes erhalten, von dem er ursprünglich nur den vollständig qualifizierten Name kennt.
Ein ORB muß am Anfang seiner Ausführung auch die IOR dieses Serverobjektes kennen. Weil IOR sich in eine Standardform serialisieren lassen, kann man z.B. die IOR des Namensservers in einer vordefinierten und bekannten URL speichern.
Abbildung 5 illustriert die beiden hier erwähnten Mechanismen.

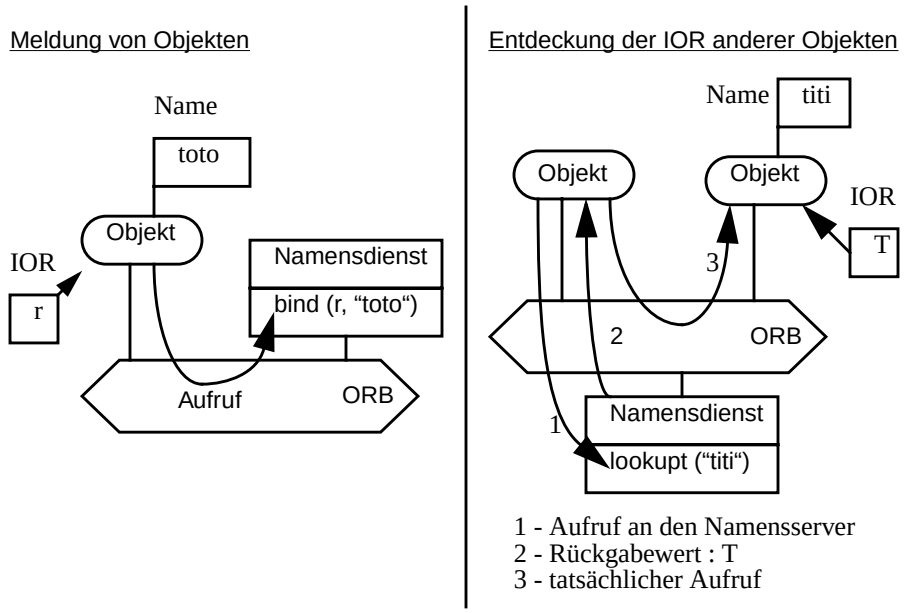


Abbildung 5. Bindungs- und Anfragemechanismen beim Namensserver

Schluß

Anders als der Namensdienst lässt sich der Sicherheitsdienst nicht als ein einziger Server implementieren, sondern ist in verschiedene Komponenten aufgelöst, die im nächsten Teil dieses Kapitels erklärt werden.
Schließlich sie bemerkt, daß dieses letzte Unterkapitel über CORBA nur die Gesichtspunkte vorgestellt hat, die für das Verständnis dieses Berichts nötig sind. Die ganze Spezifikation enthält noch mehrere Komponenten, die hier nicht erwähnt wurden.

1.2 Sicherheit in CORBA

Es gibt mehrere Anforderungen an den Sicherheitsdienst. In diesem Unterkapitel werden sie aufgelistet, erklärt und ihre spezifizierte Erfüllung vorgestellt.

Als der Hauptziel von CORBA ist Transparenz der Fernaufrufe soll folglich die Sicherheit dieser Aufrufe tranparent gewährleistet werden. Trotzdem wollen manche Anwendungen ihre Sicherheit selbst, dann explizit, durchsetzen.
Deswegen definiert die OMA 2 Sicherheitsniveaus:

- *Level 1* ist transparent; der ORB nimmt auf sich, die konfigurierten Sicherheitsanforderungen zu erfüllen. Deshalb ist ein Objekt Sicherheitsunbewusst.
- In *Level 2* nimmt selbst eine Anwendung die Massnahmen, ihre Sicherheit zu gewährleisten. Sie kann alle Mechanismen benutzen, die im *Level 1* vom ORB selbst behandelt werden. Ausserdem kann sie ihre eigene Mechanismen einführen. Aber dann sind diese anwendungsspezifisch und werden nicht in CORBA definiert.

Deshalb stelle ich in den nächsten Absätzen nur *Level 1* vor.

1.2.1 Authentizität

Authentizität ist gewährleistet, wenn der Aufrufer der ist, der er zu sein behauptet.

In CORBA trägt die ORB auf der Klientseite die Bürde, die Authentizität des Klienten zu beweisen. Diese Authentisierung teilt sich in 2 Phasen.
In der ersten erhält der ORB die *Credentials* des Klienten. Die Credentials sind eine Menge von Attributen über den Klienten, die seine Identität und damit verbundene Privilegien beweisen. Wie Credentials implementiert werden ist nicht spezifiziert. In der zweiten Phase werden die Credentials dem Serverobjekt vermittelt, das sie dann überprüft. Schauen wir diese Phasen genauer an.

In der ersten Phase verfügt der ORB über ein identifizierendes Geheimnis, wie z.B. ein Passwort, das der Klient ihm vermittelt hat. Mit diesem als Parameter ruft er *authenticate* auf dem *Principal Authenticator* auf. *Principal Authenticator* ist ein Objekt, der dem ORB und genauer dem Sicherheitsdienst gehört, und der die Credentials des Klienten erzeugt und sie ihm zurückgibt, vorausgesetzt, daß die vermittelten Authentisierungsdaten richtig waren.
Danach werden diese Credentials in *Current* eingefügt. *Current* ist auch Teil des ORBs, aber diesmal Teil des Transaktionsdiensts, und stellt den gegenwärtigen Aufrufskontext dar. Der ORB gewährleistet, daß das gleiche Objekt auch auf der Serverseite verfügbar ist.

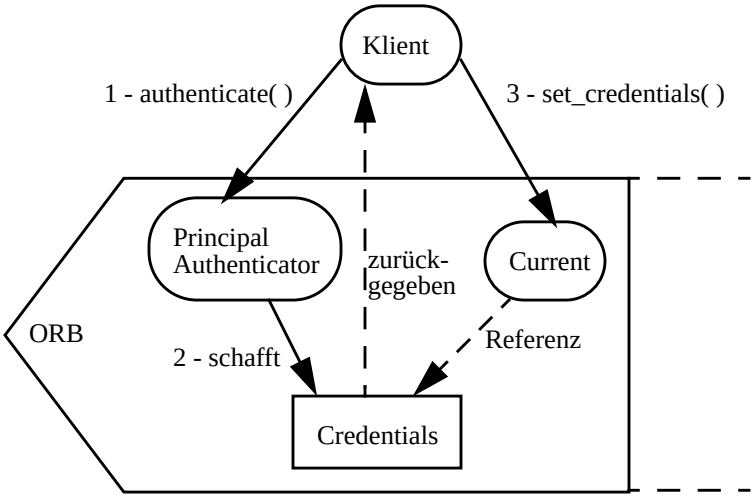


Abbildung 6. Erzeugung und Nutzung der Credentials

In einer zweiten Phase macht der Klient den Aufruf selbst. Die Credentials werden mit *Current* transparent auf die Server Seite vermittelt und können dann dort überprüft werden, so daß der Server entscheiden kann, ob er den Aufruf erlaubt.

Es ist zu bemerken, daß die erste Phase nicht bei jedem Aufruf zu geschehen braucht, sondern nur einmal, wenn der Klient erstmals beim ORB authentisiert wird. Im übrigen sind sowohl *Current* als auch *Principal Authenticato*r in IDL spezifiziert.

Um einen Aufruf durchzuführen muß manchmal ein Serverobjekt seinerseits andere Objekte aufrufen (Abbildung 1). Die Frage ist dann, ob er seine eigene Credentials oder diese seines Klienten verwendet. Im zweiten Fall sagt man, daß der Klient seine Priviliegien dem Serverobjekt *delegiert*.
Ein sicherheitsbewusstes Objekt kann diese Frage selbst beantworten und entscheiden, wie seine eigene Credentials mit den bekommenen kombiniert werden. Für *Level 1* Anwendungen definiert der Administrator eine Standarddelegationspolitik, die vom ORB durchgesetzt wird und die einer von den drei folgenden Möglichkeiten entspricht, wie in Abbildung 7 gezeichnet :

- Keine Delegation : das Zwischenobjekt verwendet seine eigene Credentials.
- Einfache Delegation : es verwendet die Credentials seines Klienten.
- Zusammengesetzte Delegation : es verwendet beide Credentials. Sie sind dann entweder getrennt oder gemischt. Im letzten Fall kann der Zielobjekt nicht unterscheiden, welche Credentials gehören dem Zwischenobjekt und welche dem ursprünglichen Klienten.

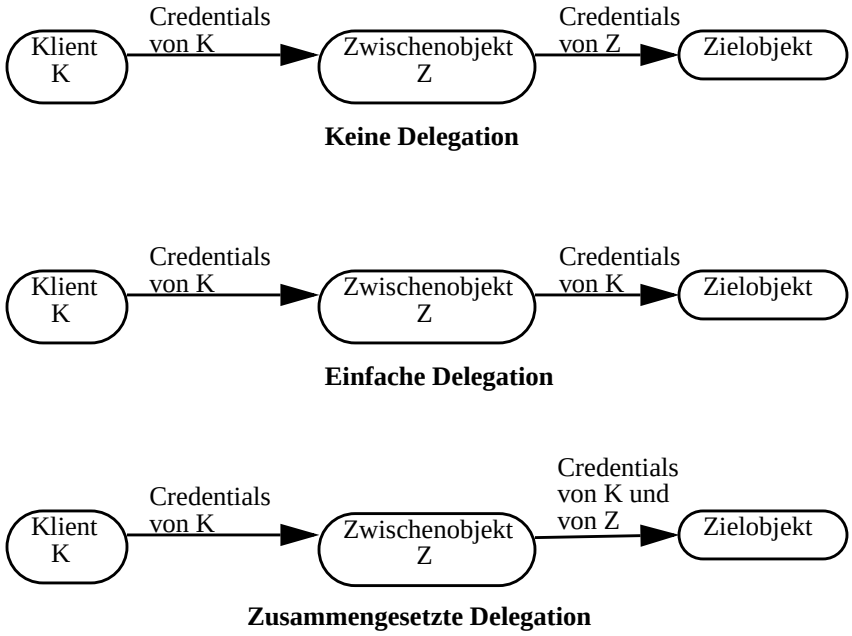


Abbildung 7. Privilegiendelegationschemen

1.2.2 Zugriffsschutz

Zugriffsschutzpolitiken

Für jedes Objekt wird nach einer Zugriffsschutzpolitik verfahren, die Subjekte mit Rechten verbindet. Subjekte sind Entitäten (menschliche Benutzer, gewisse Rechner, usw.), die auf das Objekt zugreifen dürfen oder nicht. Die Politik ist eine Menge von Regeln, die bestimmen, ob ein Subjekt eine gewisse Methode aufrufen darf und die in verschiedenen Formaten ausgedrückt werden können.

Access Control Lists (ACLs) verbinden jedes Objekt mit der Liste von Subjekten, die darauf zugreifen dürfen. Ein Beispiel dafür sind die Erlaubnisse auf Dateien in Unix.
Umgekehrt kann jedes Subjekt mit einer Liste von (Objekt, erlaubter Zugriff auf dieses Objekt) Paaren, eine sogenannte *capability-list*, verbunden werden.

Die OMA spezifiziert eine spezielle Anwendung von ACLs.
Zunächst werden Objekte von einem Administrator in Sicherheitsdomänen gruppiert. Dann weist er in jeder Domäne und für jeden Typ von Objekten Rechte zu.. CORBA definiert selbst eine Familie von Rechten (`get, set, manage, use`). Sicherheitsadministratoren können ihrerseits neue Familien definieren.
Einerseits wird jede Methode mit einer Reihe von Rechten verbunden, von denen ein Subjekt alle besitzen muß, um diese Methode aufrufen zu dürfen. Andererseits wird jedes Subjekt mit einer Menge von Rechten verbunden.

Abbildung 8. Beziehungen, die eine CORBA Sicherheitspolitik für einen bestimmten Typ in einer bestimmten Sicherheitsdomäne organisieren

Dieses Format erlaubt keine feinkörnige Sicherheitspolitik, weil Objekte eines gleichen Typs können nicht unterschiedlich geschützt werden. Ausserdem lässt sich eine für einen bestimmten Typ definierte Politik nicht einfach für einen anderen Typ wiederverwenden.

Ein typbasiertes Format, das OO-Programmierung besser paßt, wurde vorgeschlagen [3]. Dort werden Politiken selbst als Schnittstellen (sogenannte Sichten) ausgedrückt, die einem bestimmten Typ entsprechen und die sich vererben lassen. Dann kann eine Politik nicht nur allen Objekten eines gleichen Typs zugewiesen werden, aber kann auch jedes einzelnes Objekt von einer verschiedener Politik geschützt werden. So können definierte Sicherheitspolitiken feinkörnig wiederverwendet werden.

Durchsetzung der Politik

Unabhängig davon, wie die Politik ausgedrückt ist, braucht man einen Mechanismus, um sie anzuwenden. Dazu spezifiziert CORBA *Interceptors*.
Bevor und nachdem ein Fernaufruf vermittelt wird, wird sowohl auf Klientenseite als

auch auf Serverseite ein *Request* Objekt im ORB erzeugt, der Informationen über den Aufruf (den Befehl selbst, die Zeit, seinen Ursprung, usw.) enthält.

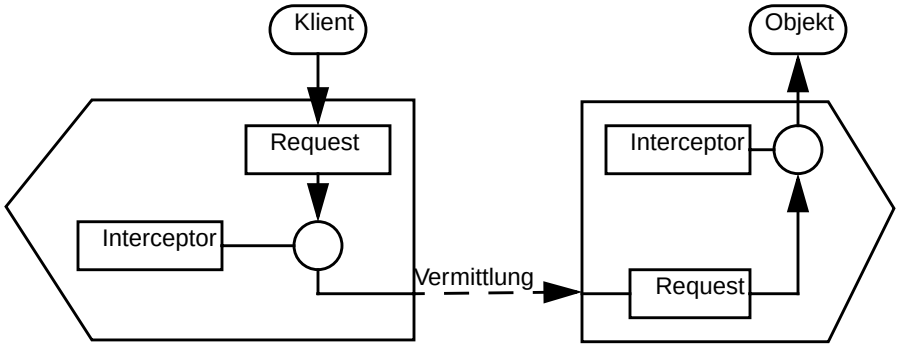


Abbildung 9. *Interceptors* und *Request* Objekte bein einem Aufruf

Ein *Interceptor* nimmt ein *Request* Objekt als Input und hält es entweder an oder vermittelt es weiter. Der Sicherheitsadministrator eines ORBs kann ihn so konfigurieren, daß sowohl ausgehende als auch einkommende *Requests* eine ganze Reihe von *Interceptors* durchqueren müßen. Folglich sind diese *Interceptors* dafür zuständig, die Sicherheitspolitik des ORBs durchzusetzen und einen *Request* abzubrechen, der dieser Politik nicht entspricht.

Bemerkung

Für die meisten Systeme ist Zugriffsschutz der Kern ihrer Sicherheit, weil er tatsächlich entscheidet, ob eine auszuführende Operation sicher ist oder nicht. Aber diese Entscheidung begründet sich ganz auf einem Attribut (z.B. einem Benutzernamen) des Subjekts (siehe Abbildung 8). Darum ist Authentisierung die Grundlage jener Sicherheitspolitik.

1.2.3 Ereignisprotokollierung

Keine Sicherheitspolitik ist perfekt, weil ein Administrator Fehler machen kann und weil es externe Sicherheitsfaktoren gibt, die ihm entgehen. Deshalb reicht es nicht, eine Politik zu spezifizieren und sie einfach durchzusetzen. Vielmehr ist es nötig, die Aktivität der Benutzer zu überwachen, um Politikkonfigurationsfehler zu entdecken und auf Angriffe zu reagieren, und um Benutzer abzuschrecken, mit der Sicherheit zu experimentieren. Grundlage dieser Überwachung ist es zu wissen, wer was macht. Daher ist Authentisierung auch bei der Ereignisprotokollierung (engl. *Audit*) grundsätzlich.
Weitere technische Details über Ereignisprotokollierung sind in der Spezifikation selbst [18] zu finden.

1.2.4 Zurechenbarkeit

In Geschäftsanwendungen von CORBA kann ein Aufruf eine Einkaufstransaktion darstellen. Dann ist es wichtig, daß weder der Käufer noch der Verkäufer sie später widerlegen und also seinen Teil des Vertrags nicht erfüllen kann. Dafür spezifiziert CORBA

ein Zurechenbarkeitsmodell [19]. Auch hier, wie bei der Ereignisprotokollierung, spielt Authentisierung eine grundsätzlich Rolle, denn Zurechenbarkeit nützt wenig, wenn eine der Seiten (oder beiden) sich hinter einer falschen Identität versteckt.

1.2.5 Schluß über Sicherheit in CORBA

CORBA spezifiziert ausserdem, wie Aufrufe zwischen ORBs verschlüsselt vermittelt werden können und wie Objekte sich in Domänen gruppieren lassen, um ihr Sicherheitsmanagement zu vereinfachen [19]. Aber da diese Themen meine Diplomarbeit nicht betreffen, werden sie nicht weiter erklärt.

Obwohl das Herz der CORBA Sicherheit in meisten Anwendungen letztlich Zugriffsschutz ist, verhält sich Authentisierung als notwendige Grundlage für die Garantie der anderen Sicherheitseigenschaften, wie es schon erwähnt wurde. Deshalb ist Authentizität die grundsätzlichste Sicherheitsanforderung eines verteilten Systems und deshalb ist Authentisierung der allererste Sicherheitsmechanismus, den eine Implementierung von CORBA anbieten muß.
Nicht anders ist es mit JacORB, der vor meiner Arbeit noch über keinen Sicherheitsdienst verfügte. Denn war die Hauptanforderung meiner Diplomarbeit: JacORB um eine Authentisierungskomponente zu erweitern.

1.3 JacORB

JacORB [5] ist eine reine Java Implementierung von CORBA, die unter der *GNU general public license* veröffentlicht wurde und kostenlos verfügbar [25] ist. JacORB wurde seit 1997 von Gerald Brose an der Freien Universität Berlin für Forschungs- und Lehrzwecke entwickelt.
Java wurde als Implementierungssprache verschiedenen Gründen gewählt. Das Java-IDL Mapping ist geradlinig [4]; klare objektorientierte Konzepte von Java erlauben, ein sauberes Design der Plattform anzustreben; Java erlaubte auch eine fast totale Portierbarkeit der Plattform und letztlich erleichterte die Implementierung durch *automatische Speicherverwaltung, Ausnahmebehandlungskonzepte, Feldgrenzenkontrolle, Verbot expliziter Zeiger-manipulation und Integration von Multi-Threading in die Sprache* [5].

Die heutige Version von JacORB ist 0.9g. Sie besteht aus einer Bibliothek von Java Klassen, die eine vollständige Implementierung von CORBA darstellen, und aus Werkzeugen. Diese sind ein IDL Compiler (`idl2j`), der IDL Schnittstellen nach Java Schnittstellen übersetzt, und ein Skeleton- und Stubgenerator (`jgen`), der aus der generierten Schnittstelle den Kode des Stubs und des Skeleton generiert. Die Anwendung dieser Werkzeuge in einem Implementierungsprozess ist in Abbildung 10 illustriert.

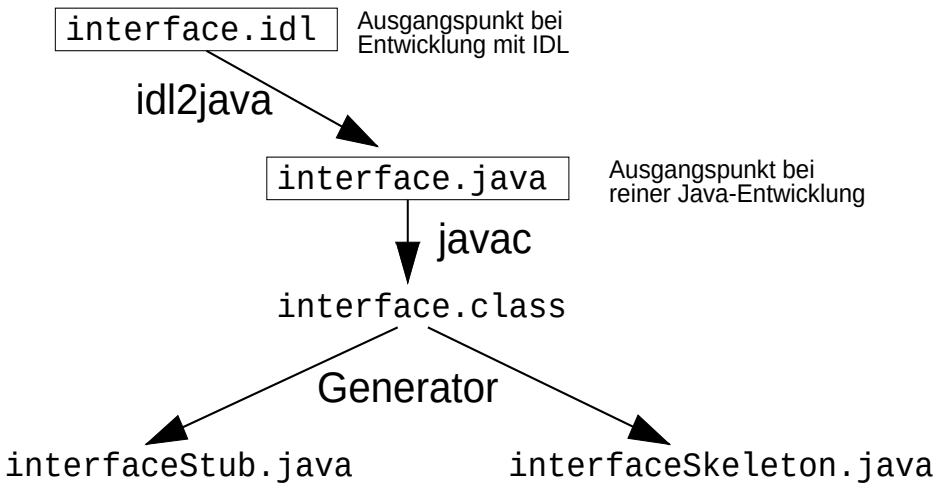


Abbildung 10. Vertretergenerierung mit JacORB [5]

Die von mir implementierte Sicherheitskomponenten sind der ersten Teil der Implementierung des JacORB Sicherheitsdienstes. Außerdem wird gegenwärtig an der Implementierung der Zugriffsschutzkomponente gearbeitet.

Unterkapitel 1.2 hat gezeigt, daß Authentisierung die Grundlage für Sicherheit in CORBA war. Denn war das Ziel meiner Arbeit, sie in JacORB zu realisieren. Dafür sollte ich, auf der CORBA-Ebene, die Objekte *Principal Authenticator* und *Credentials* so implementieren, daß das erste die authentisierte Identität eines Klienten wiederfindet und daß das andere sie enthält.
Eine weitere Anforderung war, diese Identität mit digitalen Zertifikaten darzustellen. Gründe dafür und Erklärung davon sind die Themen des nächsten Kapitels.

2.0 Verschiedene Zertifikatbasierte Lösungen

2.1 Alternativen zu Zertifikaten

Der Sicherheitsdiensts ist ziemlich neu in der gesamten Geschichte von CORBA. Deshalb ist er von mehreren ORBs noch nicht implementiert. Diese, die Authentisierung implementieren, entweder verwenden ad-hoc Mechanismen oder SSL.
Die ersten beruhen sich auf einem bestimmten Betriebsystem oder auf proprietäre Eigenschaften des ORBs, was Interoperabilität nicht entspricht.
Die andere Lösung verwendet eigentlich X.509 Zertifikate. Sie hat den Nachteil, daß der Schutz, den sie bringt, ist binär. Entweder benutzt man eine unsichere Verbindung oder eine mit SSL authentisierte und verschlüsselte, aber nie könnte man z.B. Authentisierung und gleichzeitig keine Verschlüsselung verlangen. Andere Male will man einfach Zugriffskontrolle machen. Aber wenn man dafür SSL verwenden will, dann muss man auch dazu die Verbindung verschlüsseln.
Dieser schneller Überblick von Lösunggen zu Authentisierung in CORBA zeigt keine skalierbare und vernünftige Alternative zu Zertifikaten.

Da SSL schon X.509 verwendet und da diese Infrastruktur verbreitet ist, hätte es praktisch gewesen, X.509 für mein Problem zu verwenden. Aber ein Ziel meiner Arbeit was also, mit der neuen SPKI-Infrastruktur zu experimentieren.

Der Rahmen meiner Verwendung von SPKI Zertifikate ist, sie als Darstellung von Credentials zu benutzen. Allgemein enthalten Credentials Information für Autorisierung und für Authentisierung, aber in dieser Arbeit kümmere ich mich nur um den zweiten Fall.

2.2 Prinzipien von Zertifikaten

Dieses Kapitel setzt voraus, daß der Leser Hashmechanismen und Public-Key Kryptographie [20] kennt, auf der Zertifikaten basiert sind.

Zertifikate in unterschiedlichen Infrastrukturen haben unterschiedliche Formate, Bedeutungen und Verwaltungen. Trotzdem benutzen sie einen minimalen allgemeinen Wortschatz.

Ein *Principal* ist eine Entität (Mensch, Prozeß, Einrichtung, usw.), die Zertifikaten ausstellt oder die von Zertifikaten qualifiziert wird.

Eine *Zertifikatinfrastruktur* ist eine Menge von Programmen, Einrichtungen und Regeln, die zusammen die Austeilung von Zertifikaten und ihre Bedeutung bestimmen.

Ein Zertifikat besteht aus mindestens 5 Komponenten:

1. Eine Identifizierung seines Ausstellers:

Der Aussteller ist ein Principal, der ein Schlüsselpaar besitzt.
Die Identifizierung muß im ganzen System eindeutig sein, sonst kann ein Principal von Rechten profitieren, die einem anderen Principal bewilligt wurden.
Diese Eindeutigkeit kann von einer systemweiten Organisation gewährleistet werden, die z.B. eindeutige Namen ausgibt. Dies ist die Lösung, die PGP und X.509 sich zu eigen gemacht haben.
SPKI andererseits verwendet Schlüssel an Stelle von globalen eindeutigen Namen. Für die Eindeutigkeit der Schlüssel wird ein kollisionsfreies Schlüsselerzeugungsverfahren angenommen.

2. Sein Subjekt:

Ein Principal, über den das Zertifikat etwas sagt. Wie der Aussteller wird er von seiner Identifizierung beschrieben.

3. Seine Gültigkeit in der Zeit:

Sie kann in 3 Weisen ausgedruckt werden:

- durch bestimmte Zeitgrenzen
- durch ein *on-line* Test, d.h. ein Befehl, der ausgeführt werden soll, um die Gültigkeit zu bestätigen.
- durch *Certificate Revocation Lists* (CRLs), die irgendwie geholt werden müßen und die die als ungültig erklärte Zertifikate listen und die von den Zertifikatenaussteller selbst veröffentlicht werden.

Diese 3 Weisen können auch kombiniert werden.

4. Sein Inhalt

Etwas über das Subjekt, wie z.B.:

- sein ID im System
- eine Eigenschaft von ihm: seine Adresse, sein öffentlicher Schlüssel oder seine Mitgliedschaft in einer Gruppe von Principals, usw.
- ein Recht des Subjekts

5. Seine Signatur

Sie ist der mit dem privaten Schlüssel des Ausstellers verschlüsselte Hash des Rests des Zertifikats [11]. So kann jeder, der den öffentlichen Schlüssel des Ausstellers (und zusätzlich den Hashalgorithmus, den er benutzt) kennt, die Signatur überprüfen.
Zertifikate müßen in einem kanonischen Format ausdrückbar sein, sonst kann man sie bei ihrer Vermittlung nicht kodieren und entkodieren, und sonst kann man auch nicht ihre Signatur überprüfen, weil sie auf der genauen Text des Zertifikat abhängt.

Aus diesem gemeinsamen Stamm unterscheiden sich die Zertifikatinfrastrukturen in 3 Hauptpunkte:

- Die Breite des Anwendungsgebiets der Zertifikate, d.h. wie verschieden sein kann, was ein Zertifikat über ein Subjekt sagt.
- Wie sie Principals identifizieren.
- Welche gesellschaftliche Organisation sie brauchen, um verwendet zu werden.

Diese Unterschiede werden durch die Vorstellung von 3 Zertifikatinfrastrukturen illustriert: Die zwei ersten werden heute breit verwendet, die dritte ist neu, teilweise noch in Entwicklung und meines Wissens noch nicht verwendet.

2.3 X.509

2.3.1 Infrastruktur

X.509 Zertifikate wurden schon 1988 von der CCITT empfohlen [6]. Es ist zur Zeit in der Version 3 vorhanden und z.B. in SSL [12], *Privacy Enhanced Mail* (PEM) [17], Netscape Communicator, Microsoft's Internet Explorer verwendet.
Die X.509 Principals sind unter 2 Gesichtspunkten stark hierachisch geordnet.
Erstens trägt jeder Principal einen eindeutigen Namen. Namen sind weltweit gemäss einer Baumstruktur (das X.500 Verzeichnis [11] (S. 377-389)) organisiert und entsprechen Identitäten in der "echten" (nicht digitalen) Welt. Ein Beispiel [15] (S.117) davon ist der Name der Firma Vineyard.Net:

```
C=US
S=Massachusetts
L=Vineyard Haven
O=Vineyard.NET,Inc
```

Die Buchstaben C, S, L, O deuten auf Niveaus des Namensbaums hin : *Country, State, Location, Organisation*. Vom allgemeinen (C) zum besonderen (O) wird der Name (sogenannte *Distinguished Name*) an einem Blatt des Baums eindeutig gemacht.

Aber um eine Signatur zu überprüfen reicht der Name seines Austellers nicht; man braucht auch seinen öffentlichen Schlüssel. Deshalb soll jeder Name mit einem Schlüssel verbunden werden. Dazu braucht X.509 eine Hierarchie von Zertifizierungsstellen, sogenannten *Certification Authorities* (CA). In der digitalen Welt ist eine CA ein Principal und in der echten Welt eine Einrichtung. Jede Person oder Einrichtung, die ein Principal werden will, muß sich mit einer CA so einverstanden, daß sie ihre Schlüssel einander vertrauen und Zertifikate ausstellen, die die Name-Schlüssel Verbindung bestätigen.
Das gleiche passiert zwischen einer CA und ihrer hierarchisch übergeordneten CA, bis zur einzigen CA am Gipfel. Eine solche Lage ist in Abbildung 11 illustriert.

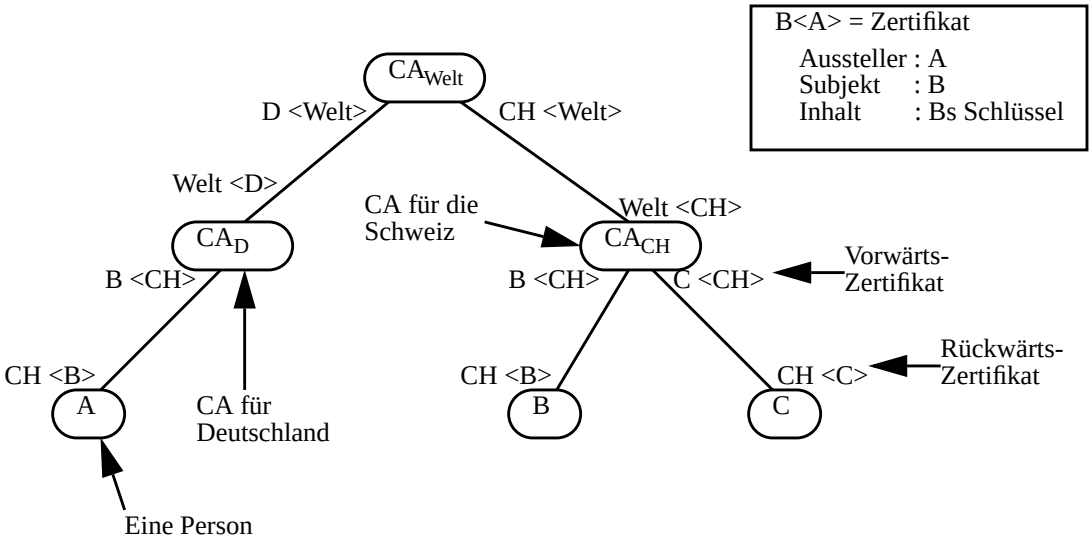


Abbildung 11. Beispiel einer X.509 Hierarchie von CAs

Wenn C einen von A signierten Text bekommt, dann braucht er einen Zertifizierungspfad (C->CH, CH->Welt, Welt->D, D->A), um den Schlüssel von A sicher zu kennen und dann die Signatur zu überprüfen. Die eindeutige Wurzel der Struktur gewährleistet, daß eine solche Kette immer existiert und einfach zu bilden ist. Übrigens entspricht oft die Hierarchie von CA dieser von Namen. X.509 spezifiziert selbst diese Hierarchien nicht, sondern konzentriert sich auf das Format der Zertifikate.

2.3.2 Format

Ein X.509 Zertifikat enthält die folgenden Felder [16]:

- *Version*: Die Version von X.509 (v1, v2 oder v3), die auch den Inhalt des Feldes *Extensions* festlegt.
- *Serial Number* : Diese Nummer wird von der CA zugewiesen und muß für jedes Zertifikat einer CA einmalig sein, da sie für die Widerrufung von Zertifikaten verwendet wird.
- Signatur
- *Issuer* : Der Name des Ausstellers
- *Validity* : Die Gültigkeitsdauer, als 2 Zeitgrenzen ausgedrückt.
- *Subject Name*
- *Subject Public Key Info* : Der öffentlicher Schlüssel (Wert und Algorithmus) des Subjekts.
- *Issuer Unique ID*
- *Subject Unique ID*: Version3-spezifisch und selten genutzt.

- *Extensions* : Version3-spezifisch. *Damit können zusätzliche Attribute mit der Person verbunden werden. Es gibt* Standard Extensions *aber auch* Private Extensions, *die letzteren können beliebige Informationen enthalten.* Extensions *sind außerdem kritisch oder nicht-kritisch, wobei ein Zertifikat, das eine kritische* Extension *hat, deren Bedeutung man nicht kennt, nicht akzeptiert werden darf.* [16]

Zwei bemerkenswerte *Standard Extensions* Felder sind die *Certificate Policies* und die *CRL Distribution Points.*
Das erste *beinhaltet die Konditionen unter denen die CA arbeitet. Dies soll Applikationen ermöglichen, anhand einer Vergleichsliste zu entscheiden, ob ein Zertifikat den geforderten Sicherheitsanforderungen entspricht, oder zurückgewiesen werden soll.*
Das zweite *bezeichnet den Punkt (IP-Adresse), wo Information über widerrufene Zertifikate (die CRL) der für dieses Zertifikat zuständigen Zertifizierungsstelle gefunden werden können.* [16]

2.3.3 Nachteile

Nachteile von X.509 Zertifikate liegen an ihrem Format und an der Hierarchie von CAs.

Ohne ihres *Extensions* Feld waren sie nur dazu geeignet, Schlüssel-Name Bindungen zu bestätigen. Zertifikatbasierte CORBA *Credentials* erfordern aber, beliebige Information an einem Principal zu binden. Dieses Problem soll durch *Private Extensions* in X.509 v3 gelöst werden. Das bedingt aber, daß das X.509-Zertifikat nur noch eine Hülle ist und die gesamte Information in den *Extensions* steckt.

Andererseits entspricht eine hierarchische Struktur den Vertrauensbeziehungen mancher kommerziellen Organisationen nicht [21]. Außerdem, um X.509 in einer weltweiter Umgebung von verteilten Objekten anzuwenden, wird es nötig sein, nicht nur jedem menschlichen Benutzer einen *Distinguished Name* zu geben, sondern auch manchen Rechnern. Das hätte zur Folge, daß die höchste CA eine durchschauende Übersicht auf die registrierten Unternehmen und Personen hätte, was jedoch in vielen Fällen nicht akzeptabel ist [7].

2.4 PGP

Der Zweck von PGP [14] ist dergleiche wie von X.509 : öffentliche Schlüssel zu verbreiten, aber fuer eine sehr bestimmte Anwendung: signierte oder verschlüsselte Email. Statt wie X.509 einen hierarchischen Namensraum zu verwenden, benutzt PGP die Emailadresse eines Principals als sein *Distinguished Name*.
Außerdem darf ein Principal jeden anderen Principal zertifizieren, und nicht nur seine ober- oder untergeordnete CA wie in X.509.

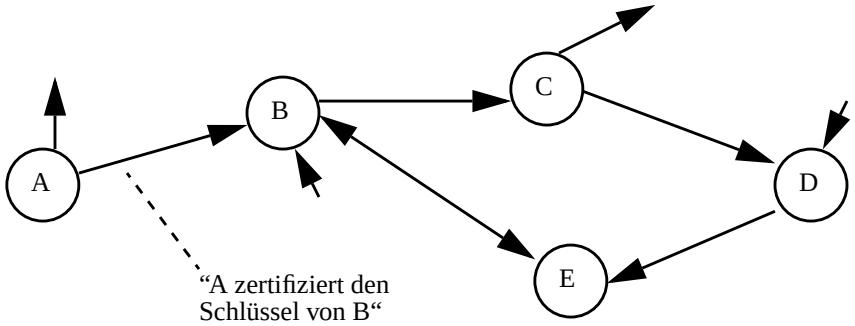
Ein PGP Zertifikat enthält:

- den öffentlichen Schlüssel, der zertifiziert wird
- die ID des Besitzers des Schlüssels
- das Datum, wann der Schlüssel geschafft wurde

- eine Liste von Signaturen: Mehrere Principals können eine Name-Schlüssel Bindung durch einen einzigen Zertifikat bestätigen.
- die Namen der Unterszeichner

Jeder Principal besitzt einen Schlüsselring. Jedem Schlüssel weist er ein Vertrauensniveau zu. Diese Niveaus sind *undefined* (unbestimmt), *untrusted* (nicht vertrauenswuerdig), *marginal* (meistens vertrauenswürdig), *complete* (vertrauenswürdig) und *ultimate* (total; für seinen eigenen Schlüssel). Sie stellen nicht dar, ob man meint, den echten Schlüssel eines betreffenden Principals zu besitzen, sondern ob man ihm traut, andere Name-Schlüssel Bindungen zu zertifizieren. Das Vertrauensniveau eines Schlüssels kann jederzeit vom Besitzer des Schlüsselrings willkürlich und unabhängig von anderen Principals geändert werden.
Einen Zertifizierungspfad entlang werden diese Niveaus kombiniert, um das Vertrauen in den Schlüssel am Ende des Pfades einzuschätzen.

Übrigens laufen diese Pfade nicht nur oben und unten einer Baumstruktur wie in X.509, sondern in ungeordneter Weise in einem Vetrauensnetz, deren Knoten Principals und deren Kanten Zertifikate sind, wie Abbildung 12 es illustriert.
Zertifizierungspfade (A<B>, B<C>, ..., Y<Z>) dürfen durch die Ausstellung von einem A<Z> Zertifikat verkürzt werden. Eigentlich existiert dieser Mechanismus auch bei X.509, wo er *crosscertificate* genannt wird [21].



Beispiel : Wenn man A, B, C vertraut und den Schlüssel von A kennt, dann kann man die Schlüssel von D und E sicher kennen.

Abbildung 12. Vertrauensnetz in PGP

Die Netzstruktur von PGP hat Vorteile gegenüber X.509.
Erstens ist PGP äußerst flexibel (jeder darf jeden zertifizieren, wenn er ihm traut). Zweitens braucht PGP keine schwere, zentralisierte Struktur.
Durch die Möglichkeit, das Vertrauen eines Schlüssels jederzeit zu verringern oder zu erhöhen, wird das Vertrauen ganz dezentralisiert, in Gegensatz zu X.509, wo es in der Politik der zentralen CA liegt. Das ist allgemein weder ein Nachteil noch ein Vorteil; es hängt von der Anwendung und von ihrer sozialen Umgebung an.

Andererseits hat PGP auch klare Nachteile. Erstens kann die Suche nach einem Pfad zur Zertifizierung eines Schlüssels eine nicht-triviale Suchstrategie im Vertrauensnetz erfordern. Zweitens macht die Dezentralisierung von dem Vertrauen es schwer, eine verbindliche Garantie der Authentizität eines Zertifikats von einem Principal zu erhalten, wie es von einer CA möglich wäre. Noch schwieriger wird es, wenn man eine Garantie einen ganzen Pfad entlang fordert.

2.5 PolicyMaker

Obwohl PolicyMaker keine Zertifikatinfrastruktur ist, wird dieser Ansatz in diesem Kapitel vorgestellt, weil er Probleme von X.509 und PGP hervorhebt und dazu Lösungen bringt, die von SPKI teilweise uebergenommen werden.

2.5.1 Ziel

PolicyMaker [2] ist ein sogenanntes *Trust Management System* (Vertrauensmanagementsystem). Das ist eine Anwendung, die für das Management von Zugriffsschutzpolitiken und von Vertrauensbeziehungen in einer anwendungsunabhängigen Weise geeignet ist.
Praktisch soll eine Anwendung, die sich schützen will, nicht mehr selbst ihre Zugriffskontrolle durchführen, sondern diese Aktivität PolicyMaker delegieren. Diese Delegation hat zwei Vorteile. Erstens wird es nicht mehr nötig, als Teil jeder neuen Anwendung Zugriffsschutzkomponenten zu programmieren, sondern es wird ein spezifisches System wie PolicyMaker dafür benutzt. Zweitens werden die anwendungsspezifischen Mechanismen von den Sicherheitsaspekten getrennt; dann können beide unabhängig und spezialisiert entwickelt werden, was das Risiko von Design- oder Implementierungsfehlern reduziert.
Bevor angeschaut wird, wie diese Ziele praktisch erfüllt wurden, wird erklärt, welche Probleme PolicyMaker löst.

2.5.2 Probleme mit Namen

Systeme, die PGP oder X.509 benutzen, verwenden namenbasierte Zugriffsschutzpolitiken, wie Abbildung 13 zeigt.

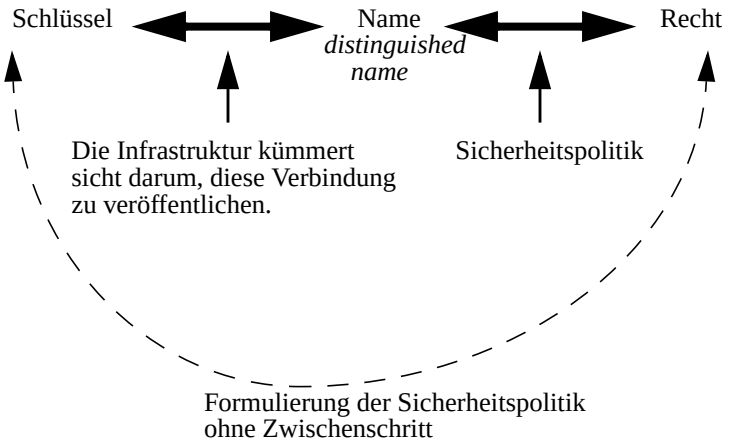


Abbildung 13. Rolle von Namen

Eine Zugriffskontrolloperation muß eine Entscheidung treffen: den erforderten Aufruf erlauben oder nicht. Bekannt wird die aufgerufene Operation, der Name des Aufrufers und ob er seinen Aufruf mit einem gewissen Schlüssel unterschrieben hat. Um die Entscheidung zu treffen sollen zwei zusätzliche Informationen geholt werden: der Schlüssel des Aufrufers (und dafür muß man manchmal ganze Ketten von Zertifikaten verarbeiten) und die Rechte, die mit dem Namen verbunden sind.
Im Vergleich, wenn man Politiken als eine direkte Verbindung zwischen Schlüsseln und Rechten (gestrichelte Linie in Abbildung 13) ausdrückt, bekommt man den Schlüssel bei dem Aufruf und braucht nicht mehr zu suchen, ob der erhaltene Name dem Schlüssel entspricht, der den Aufruf unterschrieben hat. So wird eine nutzlose Indirektionsstufe abgeschafft.

Außer der hier erklärten technischen Bürde haben Namen auch andere Nachteile. Erstens brauchen sie eine systemweite Infrastruktur, die ihre Bindung mit einem Schlüssel veröffentlicht und die ihre Eindeutigkeit gewährleistet. Zweitens gehören Namen zu einem “kleine Gemeinschaft”-Paradigma [8], wo jeder, jeden anderen mit seinen Eigenschaften zu kennen kann und wo die Eindeutigkeit von Namen praktisch von selbst gewährleistet ist.

Deshalb verzichtet PolicyMaker auf Namen und drückt eine Zugriffsregel als Beziehung zwischen einem Recht und einem Schlüssel aus. Es wird nicht nur die Zugriffskontrolle einfacher, sondern es wird dem Aufrufer erlaubt, anonym zu bleiben.

2.5.3 Architektur

Praktisch läuft PolicyMaker entweder selbständig als *daemon* oder als eine Softwarekomponente, die in die Anwendung integriert wird.
In beiden Fällen verhält er sich als eine Datenbank von *Assertions*. *Assertions* sind Beziehungen der folgenden Form, die eine Erlaubnis darstellen:
Source ASSERTS *AuthorityStruct* WHERE *Filter*

- *Source* bezeichnet entweder die lokale Sicherheitspolitik (dann ist die *Assertion* ein Eintrag in einer ACL) oder ist der öffentliche Schlüssel eines Principals (dann ist die *Assertion* ein gespeichertes Zertifikat und muß unterschrieben werden).
- *AuthorityStruct* ist der öffentlicher Schlüssel von dem (oder den) Subjekt(en) der *Assertion*.
- Ein Filter baut eine Beziehung zwischen einer anwendungsspezifischen Aktion (von einem sogenanntem *ActionString* beschrieben) und der Entscheidung, ob diese Aktion erlaubt ist, auf. Dann kann ein Filter einfach ein (Aktion, Boolean) Paar, aber auch ein eigenes komplexes Programm sein, dessen Sprache von PolicyMaker nicht festgelegt wird.

Wenn eine Anwendung einen Zugriff zu kontrollieren hat, dann delegiert sie diese Kontrolle an PolicyMaker durch eine Anfrage an die Datenbank. Anfragen haben die Form:
key1, key2,..., keyn REQUESTS *ActionString*

Das bedeutet, daß die von den gegebenen Schlüsseln identifizierten Principals zusammen erfordern, daß die von *ActionString* beschriebene Aktion durchgeführt wird.

Dann werden die Schlüssel mit den gespeicherten *AuthorityStructs* und die *ActionStrings* mit den Filtern verglichen, um die Zugriffsentscheidung zu treffen.

2.5.4 Technische Details

Außer der vorgestellten Architektur und dem Format der Anfragen und des Inhalts der Datenbank lassen sich alle Komponenten von PolicyMaker dem “Plug-and-Play“-Prinzip gemäß integrieren.

Filtersprachen
PolicyMaker-Filter können theoretisch in jeder Sprache ausgedrückt werden, die sicher interpretiert werden kann. Ihre Sicherheit bedeutet, daß sie nicht erlaubt, die Funktion der PolicyMaker-Anwendung selbst zu manipulieren. Ursprünglich werden reguläre Ausdrucke (*regexp*) und eine spezielle Erweiterung von AWK (AWKWARD) unterstützt.

Signaturverfahren
Signaturen werden von externen Programmen (PGP, PEM, usw.) überprüft. So können neue Verfahren oder Programme angewendet werden, sobald sie verfuegbar sind.

Anpaßung an eine bestimmte Anwendung
Um den Zugriffsschutz einer Anwendung an PolicyMaker zu delegieren, muß der Programmierer zuerst den Wortschatz für die Vertrauensbeziehungen und Aktionen seiner Anwendung definieren. Mit diesem Wortschatz werden dann Filter und *ActionStrings* geschrieben. Zweitens muß er PolicyMaker als Komponente oder als Server in sein Programm integrieren. Drittens muß der Sicherheitsadministrator der Anwendung die Datenbank von PolicyMaker so bevölkern, daß sie die gewünschte Sicherheitspolitik enthält.

Das hat auch den Vorteil, daß eine Sicherheitspolitik sehr komplex werden kann, ohne daß die Durchsetzungsmechanismen dieser Politik geändert werden müßen.

2.6 Anpassung an CORBA

In diesem Unterkapitel wird eingeschätzt, ob die Zertifikatinfrastrukturen und das Vertrauensmanagementsystem, die bis zu diesem Punkt vorgestellt wurden, an die Implementierung von CORBA-Credentials angepaßt sind. Dafür werden sie zuerst kurz klassifiziert. Danach werden zwei entscheidende Fragen beantwortet: ob Namen (*distinguished names*) nützlich sind und welche Information Zertifikate über ihr Subjekt vermitteln müßen.

2.6.1 Allgemeine Sicht

Die vorgestellten Systeme lassen sich in 2 Dimensionen unterscheiden (Abbildung 14): welches Vertrauensmodell sie verwenden und was sie über ein Subjekt ausdrücken können.
Die Aussagekraft kann entweder gering (Zertifikate können nur einen Schluessel mit

einem Namen binden) oder hoch (Zertifikate geben einem Subjkt beliebigen Privilegien) sein. Das Vertrauenmodell ist entweder ein Netzt (wie mit PGP), wobei jeder Principal entscheiden, wem er vertraut, oder eine Hierarchie von CAs, die sich als vertrauenswuerdig vorstellen und dafuer Garantien anbieten.

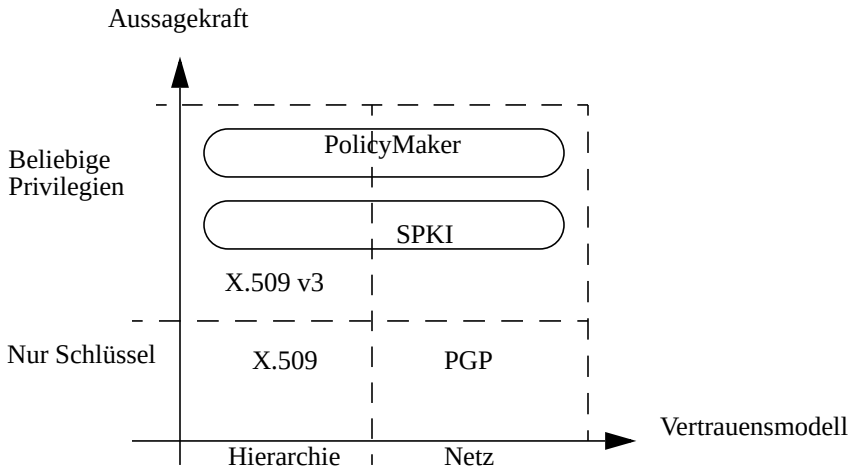


Abbildung 14. Klassifikation von Zertifikatinfrastrukturen

2.6.2 Nützlichkeit von Namen

Die *distinguished names* haben eine Existenzberechtigung, weil sie Objekte außerhalb der digitalen Welt darstellen, wie z.B. Emailadressen. Deshalb ist ein Name immer eine Referenz : eine Emailadresse ist z.B. eine Referenz zum Mensch, von dem sie benutzt wird. Sowohl mit X.509 Namen als auch mit EmailAdressen sind die Namen so aufgebaut, daß man durch eine Hierarchie von Organisationen immer die bettrefende Person finden kann.
Aber in einer Umgebung von verteilten Objekten, wie in CORBA, sind die Teilnehmer, die miteinander kommunizieren, nicht mehr Menschen, sondern Prozesse und Objekte. Deshalb ist es nicht mehr nötig, daß ihre identitizierende Kennzeichen von einem Mensch verständlich sind und daß sie einer Entität in “menschlicher“ Welt entsprechen. Da Namen außerdem eine technische Bürde sind, sind sie für die CORBA Sicherheit definitiv nutzlos.

2.6.3 Schlüsse

CORBA ist extrem generisch; damit soll jede mögliche verteilte Anwendung entwickelt werden können. Daher müßen Zertifikate (als Credentials) jedes mögliches Privileg oder alle Authentisierungsdaten darstellen können.
Deshalb ist PGP gar nicht geeignet. X.509 könnte verwendet werden, weil seine *Extensions* alles mögliche enthalten können. Aber dann wird die Struktur dieser Daten von jeder Anwendung separat spezifiziert. Aber für CORBA, wo neue Anwendungen

(Objekte) sich im Laufe der Zeit anmelden, und wo die Umgebung heterogen ist, ist diese Lösung sehr unpraktisch.
PolicyMaker hat diese Probleme nicht und ist genau so generisch wie CORBA es erfordert. Aber seine einzige Tätigkeit ist, Zertifikate an einem bestimmten Ort zu speichern, organisieren und behandeln, um Anfragen zu beantworten. Ein PolicyMaker Server verfuegt nach keinem Mechanismus, um Zertifikate mit anderen Servern auszutauschen und um sie zu vermitteln. Deshalb passt er an verteilte Systeme schlecht an. Folglich kann auch nicht PolicyMaker in unserem Fall verwendet werden. In der Abbildung 14 bleibt nur ein System, das noch nicht ausgeschlossen wurde: SPKI.

3.0 Gewählte Lösung: SPKI

3.1 Eigenschaften

3.1.1 Überblick

Die *Simple Public Key Infrastructure* (SPKI) [9] ist eine Zertifikatinfrastruktur, die ein Zertifikatformat, einen Namensmechanismus und Operationen, die Principals unterstützen müßen, definiert. Da sie noch in Entwicklung ist, ist das Format noch nicht ganz stabil und die Bedeutung einiger Elemente der Infrastruktur bleiben noch umstritten.

Wie in PolicyMaker wird in SPKI ein Principal von seinem öffentlichen Schlüssel identifiziert. In diesem Bericht wird oft von dem Schlüssel eines Principals gesprochen; es ist als “öffentlichen Schlüssel” zu verstehen. Im Gegensatz zu X.509 gibt es keine CA, sondern jeder kann beliebige Zertifikate ausstellen, wie in PGP.
SPKI definiert verschiedene Objekte. Die wichtigsten davon sind Erlaubniszertifikate, Namenszertifikate, ACLs und CRLs. Es sei zu bemerken, daß das Wort “SPKI Objekt” “eine Datenstruktur in SPKI” bedeutet und nichts, was mit Objektorientierung oder verteilten Objekten zu tun hat.

3.1.2 Erlaubniszertifikate

Eine ACL ist als ein sogenanntes *5-Tuple* (eine Struktur mit 5 Feldern) dargestellt:

- Ein Aussteller (“self“ im Fall eines ACLs)
- Ein Subjekt: gewöhnlich ein Schlüssel. Aber es kann auch eine Menge von n Schlüsseln sein, von denen man verlangt, daß k (0<= k <=n) anwesend sind.
- Ein Delegationsflag : Ob das Subjekt die ihm gegebene Erlaubnis an andere Subjekte delegieren darf.
- Das *Tag-Body*: Die Kodierung der Erlaubnis selbst. Die Form dieses Feldes ist wenig spezifiziert und seine Semantik gar nicht. Wie es gebraucht werden soll, ist anwendungsspezifisch, wie bei PolicyMaker.
- Die Gültigkeit: Entweder zwei Daten oder ein o*n-line* Test, der eine CRL holt.

Im Gegensatz zu ACLs werden Erlaubniszertifikate vermittelt, haben aber trotzdem das gleiche Format wie ACLs, mit 2 Unterschieden. Das Ausstellerfeld enthält nicht “self“, sondern den Schlüssel des Ausstellers. Ein Erlaubniszertifikat ist eine signierte ACL. Die Signatur ist aber kein zusätzliches Feld, sondern ein separates SPKI-Objekt.

Es wurde schon erwähnt, daß in CORBA-Privilegien auf verschiedene Weise delegiert werden können. In PGP und X.509 kam dieses Thema nicht vor, da es meistens sinnlos wäre, eine Namenschlüssel-Bindung zu delegieren. Aber mit Erlaubniszertifikaten lassen sich Delegationsketten bauen. Was eine solche Kette ist und welche Operationen damit verbunden sind, lässt sich am einfachsten mit einer Kette von 2 Zertifikaten illustrieren.
Seien 2 richtig unterschriebene Zertifikate:
Z1 = (I1, S1, D1, A1, V1)
Z2 = (I2, S2, D2, A2, V2)
Wenn S1 = I2 und D1 = wahr, dann kann diese Kette reduziert werden. Der Resultat der

Reduktion ist ein einziges Zertifikat: (I1, S2, D2, Schnitt (A1, A1), Schnitt (V1, V2)). Der Schnitt(A1, A2) ist die gemeinsame Erlaubnis, die in Z1 sowie in Z2 bewilligt wird. Der Schnitt (V1, V2) ist der Schnitt beider Zeitintervalle.

Übrigens können Ketten beliebiger Länge reduziert werden. Reduktion erfüllt zwei Ziele.
Erstens ist es nicht nötig, eine lange Kette von delegierten Zertifikaten oft zu benutzen und folglich eine grössere Datenmenge zu vermitteln. In diesem Fall wird der Aussteller des ersten Zertifikats der Kette gebeten, ein neues Zertifikat auszustellen, das die ganze reduzierte Kette darstellt.
Die zweite Anwendung ist Autorisierung. Abbildung 15 zeigt das Beispiel eines Kontrolleurs K, der Daten gegen “write“-Zugriffe schützt. Er stellt ein Zertifikat mit der Erlaubnis, es zu delegieren, aus. Das Zertifikat wird weiter bis zum Principal C delegiert.
1. C baut die Kette Z1, ..., Zn. Dafür muß er sich alle Zertifikate Zi besorgen.
2. Zusammen mit dem “write“-Aufruf an K vermittelt er die Kette.
3. K reduziert die Kette zusammen mit der ACL
4. Wenn er ein *5-Tuple* (selbst, C, `d`, “write“ (mindestens), ein nicht leeres Zeitintervall) erhält, dann erlaubt er es, daß Cs Aufruf durchgeführt wird. (`d` kann Delegation erlauben oder nicht.)
In diesem Fall gibt es nicht nur eine Kette, sondern einen ganzen Zyklus.

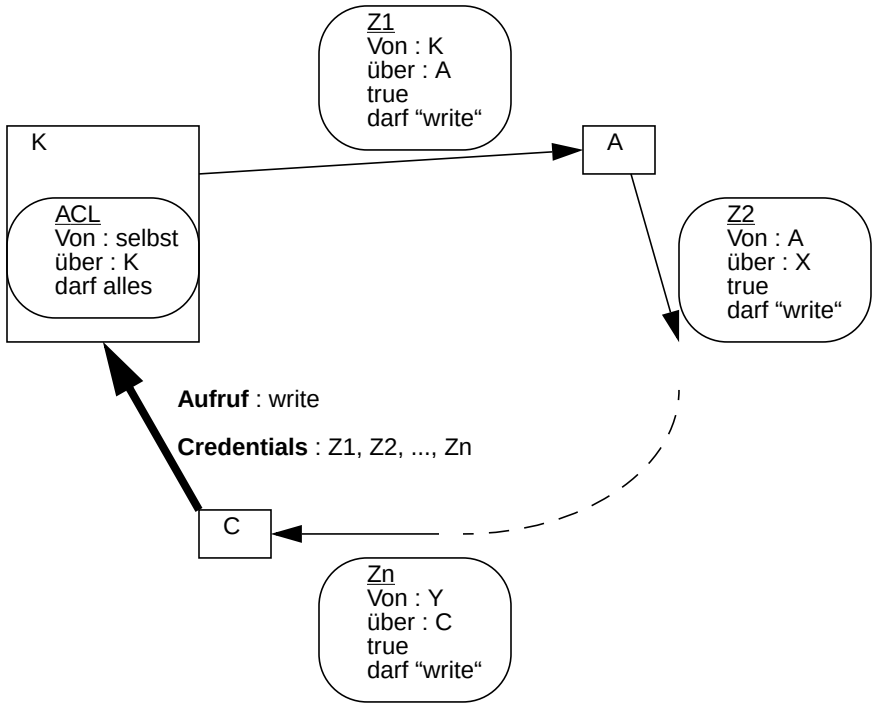
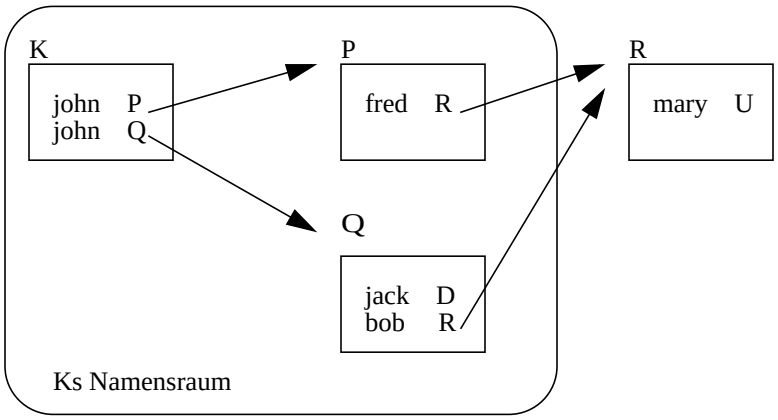


Abbildung 15. Delegation und Zertifikatketten für Autorisierung

3.1.3 Namenszertifikate

Ein Principal P kann jedem anderen Principal einen willkuerlichen Namen geben. Dieser Name ist kein distinguished name, weil er nicht global gueltig ist, sondern nur lokal, relativ zu P. Dieser Name ist wie ein Spitzname (ein Alias), den P einem anderen Schluessel gibt. So kann jeder Principal seinen eigenen Namensraum definieren.
Fuer einen bestimmten Principal brauchen diese Namen nicht, eindeutig zu sein, sondern können auch Gruppen von Schlüsseln definieren.
In SPKI beschreibt die S-Expression `(name john)`, wobei "name" ein Schluesselwort ist, die Menge von Schluesseln, mit denen der Alias "john" verbunden ist. Diese Menge kann eigentlich leer sein. Ausserdem hängt die Definition dieser Menge vollständig von dem Principal, der diese S-Expression liest, ab.
Obwohl ein Name zu einem Principal lokal ist, kann er global verwendet werden, wenn man vorher sagt, relativ zu welchem Schlüssel er definiert ist. Z.B. `(name K john)` ist die Menge von Schlüsseln, die mit dem Namen John verbunden sind, aber diesmal nicht bei dem Principal, der diese S-Expression liest, sondern beim Principal K.
Außerdem kann man aufeinanderfolgende Namen verwenden und so mehrere Namensräume durchqueren, um letztlich einen bestimmten Principal zu beschreiben.
Der Resultat davon ist ein vollständig qualifizierter Name, wie z.B. `(name K john fred)`. Diese Sequenz beschreibt den Principal, der von John Fred genannt wird, wo der Schlüssel von John bei dem Principal mit Schlüssel K zu finden ist.
So werden Ketten von Namen geschaffen. Sie lassen sich auch reduzieren, aber nicht eindeutig wie Ketten von Erlaubniszertifikaten. Z. B. könnte John Fred mit verschiedenen Schlüsseln verbinden. Oder K könnte mit John dasselbe machen, wobei einige "Johns" selbst "Freds" definieren würden und andere nicht. (Ein solcher Fall ist in Abbildung 16 illustriert.) Folglich lassen sich Namensketten zu einer unvorhersehbaren Zahl von Schlüsseln reduzieren.



Die Auflösung von `(name K john fred)` ist R

Abbildung 16. Beispiel von SPKI Namen

Um die Definition von Namen zu veröffentlichen stellt ein Principal Namenszertifikate aus. SPKI spezifiziert sie als einen eigenen Typ von Zertifikaten, obwohl sie sich von Erlaubniszertifikaten nur wenig unterscheiden: statt des *Tag-Body* Feldes besitzen sie ein “Name“ Feld.

3.2 Grammatik

3.2.1 Einführung

Um vermittelt und unterschrieben zu werden, müßen Zertifikate einem bestimmten Format entsprechen. Bei SPKI wird es von einer Grammatik definiert [10], und nicht dem Implementierer freigelassen. So sind zwei verschiedene, korrekte Implementierungen von SPKI interoperabel. Folglich ist SPKI für CORBA, das Interoperabilität anstrebt, gut geeignet.
Übrigens ist die Grammatik noch nicht ganz stabil, da SPKI noch entwickelt wird. Darüber diskutiert eine Mailingliste [31].

3.2.2 Struktur

Alle SPKI Objekte und ihren Komponenten sind als *S-Expressions* ausgedrückt. Sie sind Ausdrücke der Form “(Schlüsselwort Argument_1 ... Argument_n)“, wo jedes Argument ein String oder wieder eine S-Expression sein kann. Z. B. ist auch die Lisp Grammatik auf S-Expressions basiert.

Ein Beispiel dafuer ist der Ausdruck eines öffentlichen Schlüssels:

```
(public-key
     (rsa-pkcs1-md5
         (e #03#)
         (n
          |ANHCG85jXFGmicr3MGPj53FYYSY1aWAue6PKnpFErHhKMJ
          a4HrK4WSKTOYTTlapRznnELD2D7lWd3Q8PD0lyi1NJpNzMk
          xQVHrrAnIQoczeOZuiz/yYVDzJ1DdiImixyb/Jyme3D0UiU
         Xhd6VGAz0x0cgrKefKnmjy410Kro3uW1| )))
```

- Dieser Ausdruck entspricht der Regel:
  `<pub-key>:: "(" "public-key" <pub-sig-alg-id> <s-expr>* <uris> ")" ;`
  wobei “public-key“ das Schlüsselwort ist, das erlaubt, den Typ des Objekts zu erkennen,
- wobei `<pub-sig-alg-id>` von der folgenden Regel beschrieben wird:
  `<pub-sig-alg-id>:: "rsa-pkcs1-md5" | "rsa-pkcs1-sha1" | "rsa-pkcs1" | "dsa-sha1" | <uri> ;`
- wobei `<uri>` (*uniform resource identifier*) beschreibt eine beliebige Ressource und `<s-exp>` ist eine beliebige S-Expression. Die Interpretation von beiden ist dem Benutzer der Zertifikate und dem Implementierer von SPKI freigelassen.

3.2.3 Kanonische Form

Im vorigen Beispiel hat der String, der den Wert des Exponenten (e) des Schlüssels beschreibt, eine beliebige Länge und kann jedes Zeichen enthalten. Folglich ist es unmöglich, zu wissen, ob "| "das Ende des Strings bedeutet oder nur ein Teil davon ist. Dann kann ein solcher Ausdruck nicht parsiert werden.

Eigentlich lassen sich S-Expressions unter 2 Formen ausdrücken: einer fortgeschrittenen und einer kanonischen Form.
Das obere Beispiel ist in fortgeschrittener Form dargestellt. Wörter werden von Leerzeichen getrennt, und Strings sind in "Base 64"-Kodierung geschrieben (Jedes Zeichen ist auf 7 Bits kodiert).
Die kanonische Form ist im Gegensatz kompakter. Zeichen in Strings sind auf 8 Bits kodiert. Statt Wörter mit Leerzeichen zu trennen, werden sie mit ihrer Länge präfigiert und danach miteinander verkettet. Z.B. wird der Delegationstag "(propagate)" zu "(9:propagate)" in kanonischen Form.
Der Vorteil der kanonischen Form ist, daß sie sich parsieren lässt (kein Problem mit Begrenzungszeichen) und daß sie kompakter ist. Deshalb wird sie für die Serialisierung (und Vermittlung) von Zertifikaten verwendet.
Aber diese Form lässt sich wegen der auf 8 Bits kodierten Zeichen von einem Menschen schlecht lesen. Deshalb zeigt dieses Bericht nur fortgeschrittene Formen von S-Expressions.

3.3 Hash von Objekten

3.3.1 Verarbeitung von Sequenzen

Zwischen Principals werden nie direkt Schlüssel oder Zertifikate vermittelt, sondern meistens Sequenzen. Sie enthalten beliebigen SPKI-Objekte.
Ein Principal verarbeitet diese Objekte sequentiell. Was er mit ihnen tut ist nicht allgemein spezifiziert: er kann sie speichern, sie ignorieren oder eine entsprechende Aktion durchführen.
Aber in einem bestimmten Fall ist sein Verhalten genau spezifiziert: wenn das Objekt zu dem "Operation"-Typ gehört. Im Moment unterstützt SPKI nur zwei Operationen. Die erste ist nur eine generische S-Expression, deren Bedeutung anwendunsspezifisch ist. Die zweite ist die Hashoperation. Sie bedeutet, daß der Principal das vorige Objekt in der Sequenz speichern und es mit dem Hash seiner kanonischen Form referenzieren muß. Als Parameter der Operation wird ein Hashalgorithmus gegeben. SPKI unterstützt MD5 und SHA1 [20].

3.3.2 Hash von Objekten

So besitzt jeder Principal eine Hashtabelle von SPKI-Objekten. Obwohl alle Objekte gehasht werden können, werden meistens nur Zertifikate und Schlüssel gehasht.
Zuerst kann diese Hashtabelle als eine Datenbank von Zertifikaten (wie eine PolicyMaker Anwendung) und von Schlüsseln (wie ein Keyring in PGP) angesehen sein. So können Ketten aufgebaut und ausgelöst werden , und Signaturen überprüft werden, ohne daß der Schlüssel des Unterzeichners jedesmal im Zertifikat steht.
Seine zweite Rolle ist, den Ausdruck eines Zertifikats zu verkürzen. Eigentlich können die Hashwerten nicht nur intern in der Hashtabelle verwendet werden, sondern auch zwischen Principals. Wenn zwei Principals einen Schlüssel gehasht und gespeichert

haben, dann können sie diesen Schlüssel mit seinem Hash in allen Zertifikaten, die sie ausstellen, ersetzen und sich gleichzeitig noch verstehen. Dieser Mechanismus nimmt trotzdem an, daß die benutzten Hashfunktionen kollisionsfrei sind, sonst identifiziert ein Hash einen Principal nicht mehr eindeutig. Folglich ist in SPKI das identifizierende Kennzeichen eines Principals nicht nur sein Schlüssel, sondern auch jedes Hash dieses Schlüssels.

Ein Principal ist allein dafür zuständig, wie er seine Hashtabelle verwaltet. Deshalb trägt er die Verantwortlichkeit, die Hashbefehle durchzuführen oder nicht, und deshalb kann seine Hashtabelle nicht von anderen Principals gelesen werden. Diese können nur Befehle geben, bestimmte Objekte in ihr zu speichern, ohne direkt bestätigen zu können, ob dieser Befehl durchgeführt wurde oder nicht.

3.3.3 Beispiel

Hier ist ein Beispiel (aus [10]) der Verarbeitung einer Sequenz, die eine Hashoperation enthält:

```
(sequence
  (public-key
    (rsa-pkcs1-md5
      (e #11#)
      (n |ALNdAXftavTBG2zHV7BEV59gntNlxtJYqfWIi2kTcFIgIPSjKlHleyi
         9s5dDcQbVNMzjRjF+z8TrICEn9Msy0vXB00WYRtw/7aH2WAZx+x8erOW
         R+yn1CTRLS/68IWB6Wc1x8hiPycMbiICAbSYjHC/ghq2mwCZO7VQXJEN
         zYr45| )))
  (do hash md5)
  (cert
    (issuer (hash md5 |+gbUgUltGysNgewRwu/3hQ==|))
    (subject
      (keyholder (hash md5 |+gbUgUltGysNgewRwu/3hQ==|)))
    (tag
      (* set
        (name "Carl M. Ellison")
        (street "207 Grindall St.")
        (city "Baltimore MD")
        (zip "21230-4103")))
        (not-after "1998-04-15_00:00:00"))
  (signature
    (hash md5 |54LeOBILOUpskE5xRTSmmA==|)
    (hash md5 |+gbUgUltGysNgewRwu/3hQ==|)
    |HU6ptoaEd7v4rTKBiRrpJBqDKWX9fBfLY/MeHyJRryS8iA34+nixf+8Yh/
    buBin9xgcu1lIZ3Gu9UPLnu5bSbiJGDXwKlOuhTRG+lolZWHaAd5YnqmV9h
    Khws7UM4KoenAhfouKshc8Wgb3RmMepi6t80Arcc6vIuAF4PCP+zxc=| )
)
```

Das erste Element ist ein Schlüssel, der passiv empfangen wird. Das zweite Element ist eine Hashoperation, die anordnet, den vorigen Schlüssel mit MD5 zu hashen und zu speichern. Das dritte Element ist ein Zertifikat, dessen Aussteller von seinem Hashwert beschrieben wird. Der Wert |+gbUgUltGysNgewRwu/3hQ==| soll eigentlich dem ersten Schlüssel entsprechen. Das Zertifikat listet einige Eigenschaften über seinen Aussteller : deshalb ist es ein Selbstzertifikat. Das vierte Element ist die Signatur des Zertifikats. Sie braucht ihm nicht direkt in der Sequenz zu folgen, denn sie enthält den Hashwert des

Objekts, das sie unterschreibt und das deshalb wiedergefunden werden kann. Ausserdem enthält sie das Kennzeichen seines Unterzeichners, der hier nochmals von seinem Hashwert charakterisiert wird.

3.4 Typen von Schlüsseln

SPKI spezifiziert nicht nur kryptographische Schlüssel, sondern auch Signaturschlüsseln. Ein solcher Schlüssel wird von 3 Elementen charakterisiert:

- Der Verschlüsselungsalgorithmus, für den er geeignet ist.
- Sein Wert, der aus mehreren Parametern besteht.
- Der Hashalgorithmus wird verwendet, um Signaturen zu erzeugen.

Ausser den zwei erwähnten Hashalgorithmen unterstützt SPKI noch zwei Verschlüsselungsalgorithmen : RSA und DSA [20]. Dann sind die spezifizierte Signaturalgorithmen kombinationen dieser 4 Algorithmen : rsa-pkcs1-md5, rsa-pkcs1-sha1, rsa-pkcs1, dsa-sha1.
Die Kombination dsa-md5 existiert nicht, weil MD5 fuer DSA nicht geeignet ist. PKCS1 ist ein Standard, der ein Format für Schlüssel spezifiziert.
SPKI spezifiziert aber auch rsa-pkcs1 als Typ von Signaturschlüssel. Natürlich erlaubt dieser Typ nicht, eine Signatur zu überprüfen, weil man nicht weiss, welcher Hashalgorithmus zu verwenden ist. Aber die Frage stellt sich, ob rsa-pkcs1 reicht, um einen Schlüssel (und folglich ein Principal) zu beschreiben.
Allgemein ist dann die Frage: Ist ein Principal nur von seinem kryptographischen Schlüssel identifiziert oder von diesem und von seinem verbundenen Hashalgorithmus ? Oder, mit anderen Worten : Sind zwei Principals, die RSA-Schlüssel mit gleichen Werten besitzen, gleich? Diese Frage ist grundsätzlich, denn sie definiert die Gleichheit zweier Principals. Diese Operation wird ausserdem ständig benutzt, um Sicherheitsentscheidungen zu treffen, wie z.B. in einer Kettenreduktion oder bei einem Vergleich mit einer ACL.
In der SPKI-Mailingliste wurde diese Frage diskutiert, aber meines Wissens keine Entscheidung getroffen. Da ich keinen Grund dafür sah, daß ein Principal einmal mit MD5 und einem anderen Mal mit SHA1 unterschrieb, habe ich für meine Implementierung beschlossen, daß Schlüssel verschiedener Hashalgorithmen unterschiedliche Principals identifizieren. Folglich wird der rsa-pkcs1 Typ nicht unterstützt.

3.5 Grund für eine nur partielle Implementierung von SPKI

Meine Implementierung von SPKI ist nur dazu geeignet, die Authentizität von CORBA-Credentials und ihre Integrität während ihrer Vermittlung zu gewährleisten. Genauer gesagt müßen diese Credentials Attribute von Principals ausdrücken können. Dafür ist es nicht nötig, alle Aspekte von SPKI zu implementieren. Stattdessen habe ich nur das implementiert, was fuer die Authentisierung in JacORB nützlich war. Teile von SPKI (wie z.B. Kettenreduktion, *Subject Thresholds*, Tagstrukturen für Erlaubnisse, usw.) bleiben unimplementiert, aber nie wurde die SPKI-Spezifikation verletzt, damit z.B. die Infrastruktur zur Authentisierung in JacORB besser passt. So besteht meine Implementierung aus Komponenten, auf denen man weiter aufbauen kann, um eine vollständigere Implementierung von SPKI zu realisieren.
Welche Teile von SPKI im einzelnen implementiert wurden, wird im Laufe der folgen-

den Kapitel erklärt werden, parallel zu der Vorstellung meiner Lösung für die Authentisierung in JacORB.

4.0 Authentisierungsprotokoll

Dieses Kapitel stellt ein Protokoll vor, das ich für die Authentisierung von JacORB Klienten entworfen habe, und begründet es.

4.1 Anforderungen

Das Szenario ist, das ein CORBA Klient (oder sein ORB für ihn) sich bei einem Serverobjekt authentisieren will. Sich authentisieren bedeutet, Credentials vorzulegen, die Attribute enthalten, die einen Klienten charakterisieren. Außerdem sollen diese Attributen echt sein, und der Server muß auf jeden Fall fähig sein, sie zu überprüfen.
Dann stellt sich die Frage, was die Bedingungen sind, damit der Server ein Attribut als echt betrachtet. Ein Attribut ist echt, entweder weil der Server es selbst ausgegeben hat oder weil es von einem Principal ausgegeben wurde, dem der Server dafür traut. In diesem Fall ist dieser Principal ein vertrauenswürdiger Dritter (engl. *Trusted Third Party*) und der Server hat ihm eigentlich das Recht delegiert, Attribute zu vergeben. In beiden Fällen gibt es eine Einheit, die Attribute veröffentlicht und verwaltet, egal ob sie beim dem Server selbst liegt oder davon entfernt.

4.2 Spezifikation

4.2.1 Teilnehmer

Aus den Anforderungen wurden mehrere Teilnehmer des Protokolls identifiziert:

- Der Klient, der sich (allein oder durch seinen ORB) seine authentisierte Attributen besorgt.
- Das Serverobjekt, das (allein oder durch seinen ORB) die Authentizität von vorgestellten Attributen überprüft.
- Eine Komponente, die Attributen verwaltet und besorgt. Eigentlich lässt sie sich in 2 Einheiten unterteilen:
  Eine Menge von Attributanbietern (engl. *attribute provider*): Quellen von Attributen, die sie als Zertifikate (Attributzertifikate) ausstellen.
  Ein Attributserver (engl. *attribute server*): Er ist der Vermittler zwischen Klienten und Attributanbietern. Er stellt einen Punkt dar, an dem Klienten ihre Attributen nachfragen können. Der Attributserver holt sie aus den verschiedenen Attributanbietern, setzt sie zusammen und gibt sie dem Klienten zurück.

4.2.2 Prinzip des Protokolls

Abbildung 17 zeigt die Beziehungen zwischen den Teilnehmern und das Szenario der Authentisierung.

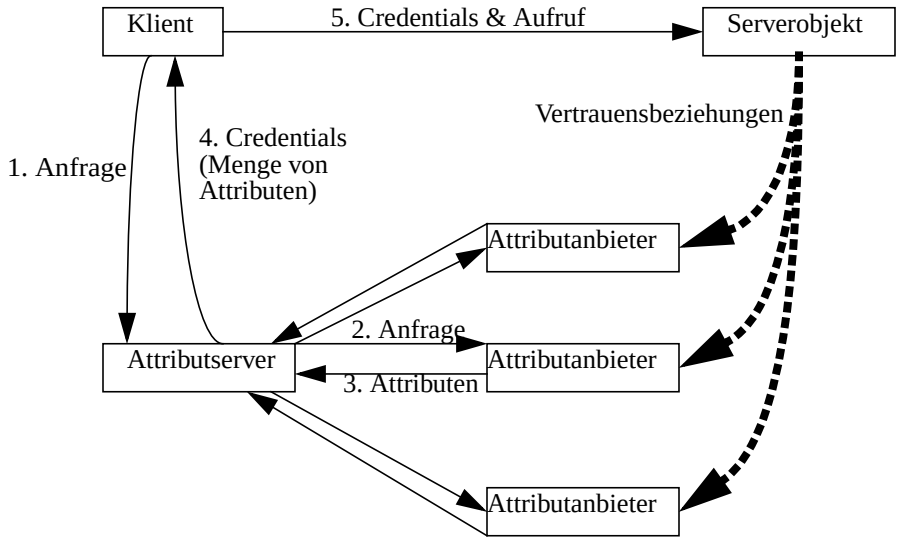


Abbildung 17. Teilnehmer an dem Protokoll

Erstens bittet der Klient den Attributserver um seine Attribute. Dafür ruft der Attributserver jeden Attributanbieter, den er kennt. Diese geben, drittens, die Attribute zurück, die dem Klienten entsprechen. Viertens sammelt der Attributserver alle diese Attributen und gibt sie dem Klienten zurück. Letztlich vermittelt der Klient dem Serverobjekt seine Attribute als Credentials zusammen mit seinem Aufruf.
Weder der Klient noch der Attributserver brauchen, die Unterschrift und die Gültigkeit der Attributzertifikate zu überprüfen, obwohl sie es können.
Beim 5. Schritt ist der Aufruf (und übrigens jeder späterer Aufruf, der noch von der Authentifikation profitiert, nicht Teil dieses Protokolls. Mein Protokoll ist folglich davon unabhängig, ob dieser Aufruf gestützt (signiert oder verschlüsselt) wird oder nicht.
Das Szenario der Abbildung 17 ist das einzige und hinreichende, das dieses Protokoll unterstützt.

Damit es funktioniert, müßen seine Teilnehmer die folgenden Bedingungen erfüllen:

- Das Serverobjekt muß allen Attributanbieter vertrauen, von denen er Attribute annimmt. Natürlich kann er auch nur einer Teilmenge von ihnen vertrauen (d.h. ihnen das Recht delegiert zu haben Attributzertifikate auszustellen). Aber dann wird er einen Teil der erhaltenen Attributen ablehnen müßen.
- Der Klient muß eine Referenz auf den Attributserver besitzen, damit er ihn aufrufen kann.
- Ebenso braucht der Attributserver, Referenzen auf die Attributanbieter.
- Jeder Attributanbieter soll fähig sein, Attributen zu vergeben. Wie er diese Funktionalität erfüllt, bleibt ihm ueberlassen. Z.B. könnte er eine Datenbank von Attributen

haben und Zertifikate erst bei einer Anfrage ausstellen, eine festgelegte Menge von vorbereiteten Zertifikaten besitzen oder Attribute gar dynamisch generieren, wenn um sie gebeten wird. Meine Implementierung definiert Schnittstellen, die die Integration eines beliebigen Mechanismus, der die Funktionalität des Attributanbieters erfüllt, erlaubt.

Praktisch läuft der Attributserver als ein eigenes CORBA-Serverobjekt, genau wie der Namensdienst. Er braucht nicht einzig im System zu sein, da er nur ein Vermittler ist. In meinem Modell wird er trotzdem einzig bleiben, denn mehrere Attributserver bringen vielleicht eine Verbesserung der Leistung durch Dezentralisierung, aber sicher keine neue Funktionalität. Die Eindeutigkeit des Attributservers macht es auch einfacher für ORBs, eine Referenz auf ihn am Anfang ihrer Ausführung zu bekommen. Dann könnte der Attributserver eines Systems einer vordefinierte Objektreferenz oder einem vordefinierten Namen beim Namensdienst entsprechen, oder, genauso wie für den Namensdienst in mehreren CORBA Implementierungen (wie z.B. in JacORB), seine Objektreferenz in einem vordefiniertem Ort im Netzwerk gespeichert haben.
Ein Attributprovider ist ein entferntes Objekt, entweder auf dem gleichen ORB wie der Attributserver oder als eigener Server. Meine Implementierung erlaubt beides.

Es ist zu bemerken, daß weder der Attributserver noch der -anbieter einen Zustand behält, über wen ihn nachgeschlagt hat. Deshalb ist die Aktion, seine Credentials zu holen, von sich selbst keine Authentisierung wie z.B. ein Einloggen. Die Authentisierung geschieht tatsächlich erst, wenn das Serverobjekt die Credentials des Klienten empfängt.

Die vorgestellte Organisation der Teilnehmer an dem Authentisierungsprozess ist sicher nicht die einzige mögliche. Alternativen werden jetzt eingeführt, zusammen mit den Gründen, warum ich sie nicht in Betracht gezogen habe.

- *Nur ein einziger Attributanbieter.*
  Attributen haben sehr wahrscheinlich unterschiedliche Bedeutungen: z. B. sind einige von ihnen IDs und andere stellen Mitgliedschaft in einer bestimmten Gruppe dar. Vielleicht werden diese Attribute von verschiedenen, voneinander unabhängige CAs ausgestellt, oder mindestens von Einheiten, die unterschiedliche Aufgaben haben. Deshalb ist es unrealistisch, die Ausstellung von Attributzertifikaten zu zentralisieren. Außerdem wäre dann die Vertrauenspolitik eines Serverobjektes rein binär: entweder vertraut er den einzigen Attributanbieter oder nicht, aber er könnte nicht mehreren Attributanbietern unterschiedlich vertrauen.
- *Kein Attributserver, sondern nur Attributanbieter, die von Klienten direkt aufgerufen werden.*
  Der einzige Vorteil davon wäre, zwei Vermittlungen (zwischen Attributserver und Klienten) zu ersparen. Aber der Klient muesse jetzt alle verschiedenen Attributanbieter selbst aufrufen und danach alle erhaltene Attribute kombinieren. Nicht nur müßte dann jeder Klient die Referenz von jedem Attributanbieter besitzen, dazu er müßte auch vom Erscheinen neuer Anbieter informiert werden.
  Im Gegensatz erlaubt ein eigener Attributserver, das Erhalten von Credentials (das eine Funktionalität des ORBs ist) von der Verwaltung verschiedener Attributanbieter (die zu einer eigenen Infrastruktur gehören können) zu trennen, was die Flexibilität des ganzen Systems verbessert.

Nach der Vorstellung der Teilnehmer schauen wir an, was genau vermittelt wird und welche Sicherheitsgarantien das Protokoll anbietet.

4.3 Detail des Protokolls

4.3.1 Überblick

Abbildung 18 zeigt die betreffenden Komponenten der Teilnehmer und die ausgetauschten Meldungen genauer. Zur Vereinfachung wird nur ein Attributanbieter dargestellt.

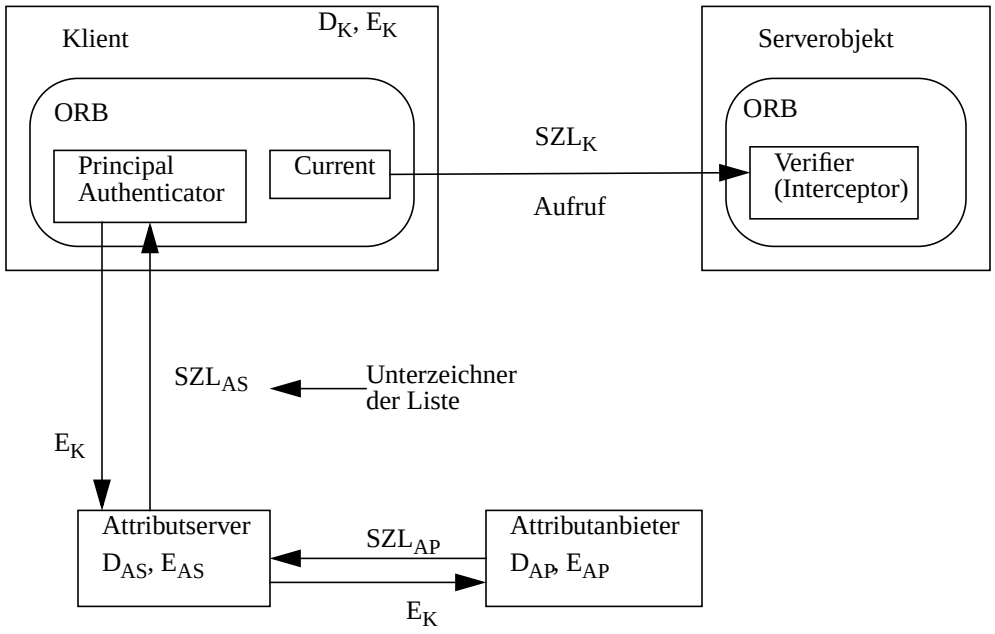


Abbildung 18. Detaillierte Sicht der Elemente des Protokolls

Außer dem Serverobjekt, das nichts zu unterschreiben hat, sind alle Teilnehmer an dem Protokoll SPKI-Principals und besitzen folglich ein Schlüsselpaar (D, E), wobei D privat und E öffentlich ist.
Auf der Klientenseite ist der *Principal Authenticator* dafür zuständig, sich die Attribute zu besorgen. Dann werden die Credentials in *Current* eingefügt, um vermittelt zu werden. Auf der Serverseite werden sie von einer *Verifier* Komponente überprüft. *Verifier* ist eine von mir implementierte Überprüfungseinheit, die in einem CORBA-*Interceptor* integriert ist.

4.3.2 Format der Meldungen

Beim Aufruf des Attributservers wird nur der Schlüssel des Klienten vermittelt. Dann wird er jedem Attributanbieter weitergegeben. Rückgabewerte aus den Anbietern durch den Attributserver bis zum Klienten sind signierte Zertifikatlisten (SZL). Eine SZL besteht aus zwei SPKI-Sequenzen. Die erste enthält alle Attributzertifikate über einen Principal, die vermittelt werden. Die zweite enthält nur die Signatur der ersten Sequenz von dem Sender der Sequenz. So sieht die S-Expression einer SZL aus:

```
(sequence
  (cert ..... )
  (signature ....)
       ......
  (cert ..... )
  (signature ....)
)
(sequence
  (signature .....)
)
```

Man hätte erwartet, eine SZL als eine Sequenz von einer Sequenz und ihrer Signatur formattiert zu werden, aber die SPKI-Grammatik erlaubt keine eingeschachtelte Sequenz.

Der Aufruf des Serverobjekts selbst besteht aus zwei Teilen: die Credentials als eine vom Attributserver unterschriebene SZL und eine Liste, die den Aufruf selbst, die Zeit, wann er geschaffen wurde, und den Schlüssel des Klienten enthält und die vom Klienten unterschrieben wird.
In den SZL beschreiben Zertifikate ihren Aussteller und ihr Subjekt mit ihren Schlüsseln, lieber als mit ihren Namen oder mit dem Hash ihres Schlüssels. Die Wahl dieses Format verlängert den Ausdruck der Zertifikate, aber bringt andererseits Vereinfachungen: Principals brauchen keine Hashtabelle zu besitzen, und auch keine Auflösung von Namen auszuführen.

Um die Signaturen zu überprüfen müßen Teilnehmer den Schlüssel anderer Teilnehmer kennen.

- Anbieter brauchen keinen Schlüssel zu kennen.
- Der Attributserver muß den Schlüssel jedes Anbieters kennen, um die erhaltene SZL zu überprüfen.
- Der Klient braucht nichts zu überprüfen; deshalb braucht er auch keinen Schlüssel zu kennen.
- Das Serverobjekt braucht nicht nur alle Schlüssel der von ihm vertrauten Anbieter zu kennen, sondern auch den Schlüssel des Attributservers. Der Schlüssel des Klienten wird zusammen mit dem Aufruf vermittelt, da er nicht im voraus bekannt ist.

4.4 Sicherheit

4.4.1 Anforderungen

Ich habe dieses Protokoll so entworfen, daß es daran hindert, daß eine vermittelte Information unterwegs geändert wird. Unter anderen Konsequenzen kann ein Klient K nicht daran gehindert werden, alle Credentials, die ihn betreffen, einem Server vorzustellen. Aber kein Schutz gegen die Lektüre dieser Information wird angeboten.

4.4.2 Kein Schutz gegen Lektüre:

Ich habe gewählt, daß jeder Klient im System (CORBA-Objekt oder einfach Prozess) den Attributserver über die Attribute jedes möglichen Principals befragen darf. Dieser Klient braucht es eben nicht, selbst ein Principal zu sein.
Weil der Attributserver offentlich ist und weil alle Information, die er liefern kann, aus den Attributanbietern kommt, können diese auch von irgendeinem Klienten frei angefragt werden. In diesem Fall muß ein Objekt sich die Objektreferenz des Anbieters zuerst besorgen.

Diese Wahl hat den Nachteil, daß alle Attribute von allen Principals öffentlich sind. Deswegen kann ein Principal nicht persönliche Daten privat halten. Und genau das würde erfordert, wenn z.B. ein Principal einen menschlichen Benutzer darstellt, dessen Credentials Daten wie z.B. seine Sozialversicherungsnummer, der Betrag auf seinem Bankkonto oder einfach seine Adresse enthalten.
Obwohl Vertraulichkeit von meinem Protokoll nicht angeboten wird, kann das Format seiner Meldungen geändert werden, um diese Vertraulichkeit zu gewährleisten und ohne daß die Struktur des Protokolls geändert wird. Die einzige zusätzliche Anforderung, die Vertraulichkeit bringt, ist, daß das Serverobjekt (oder sein ORB) jetzt selbst auch ein Principal sein muß und folglich ein Schlüsselpaar besitzen muß.
Die Idee ist, jede Vermittlung von Attributen mit dem öffentlichen Schlüssel ihres Empfängers zu verschlüsseln. Abbildung 19 zeigt, mit welchem Schlüssel die Liste von Zertifikaten vom Attributanbieter bis zum Serverobjekt verschlüsselt wird.

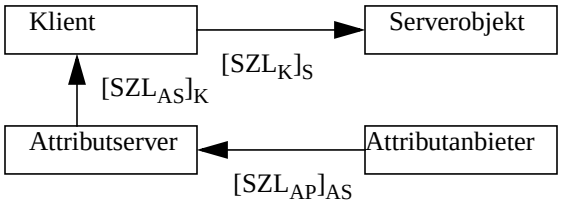


Abbildung 19. Verschlüsselte Vermittlung der Attribute

Der Anbieter kann nicht seine Liste direkt mit KK verschlüsseln, weil der Attributserver die verschiedene Listen, die er empfängt, lesen muß, damit er sie zusammenfassen kann.

Diese Verschlüsselung könnte eigentlich transparent sein, denn der CORBA Sicherheitsdienst bietet automatische Verschlüsselung als Teil des ORBs an. Aber diese Funktionalität ist in JacORB nicht implementiert. Sie zu besorgen hätte meine Arbeit kompliziert, ohne daß die Authentisierung selbst verbessert würde.
Außerdem ist es fraglich, ob die Attribute bei den Vermittlungen geheim zu halten einen Sinn hat. Letztlich werden die Credentials dem Serverobjekt geliefert. Was es mit diesen Zertifikaten genau machen wird (sie zerstören, sobald sie nicht mehr nötig sind? Oder sie speichern, veröffentlichen und verbreiten?), ist in jedem Fall außer Kontrolle des Klienten und sogar des Protokolls.

Aus diesen beiden Gründen habe ich beschlossen, die Attribute öffentlich zu halten.

4.4.3 Schutzmechanismen

Um die Schutzmechanismen des Protokolls zu zeigen, betrachten wir jede Vermittlung um herauszufinden, was bei ihr geschehen kann, so daß die erwähnten Anforderungen nicht mehr erfüllt werden, und wie das Protokoll es verhindert.

Bei Anfragen an den Attributserver kann nur den Schlüssel geändert werden. Später beim Empfang der Attribute, wird der Klient das bemerken. Deshalb wird die Vermittlung des Schlüssels nicht geschützt.
Ebenso ist es bei der Vermittlung des Schlüssels zwischen Attributserver und Anbieter.

Bei der Vermittlung der Liste von Zertifikaten vom Anbieter zum Server können zwei Angriffe durchgeführt werden.
Der erste ändert den Inhalt der Liste von Zertifikaten. Der Attributserver wird dann die Signatur der Liste überprüfen und bemerken, daß sie nicht stimmt.
Der zweite ändert auch die Liste, aber zusätzlich ersetzt die Signatur mit einer neuen, mit einem Schlüssel des Angreifers generierte. Da der Attributserver den Schlüssel jedes Anbieters kennt, wird auch diese Änderung entdeckt.
Dasselbe gilt für die Vermittlung der SZL vom Attributserver zum Klienten und dann zum Serverobjekt.

Weil die Credentials immer signiert vermittelt werden, kann ein Angreifer nicht verhindern, daß ein Klient einem Server alle seine Attribute vorstellt. So wird eine der Sicherheitsgarantien des Protokolls erfüllt.

Ein Angreifer kann sich die Credentials von einem Klienten K frei besorgen. Ob er danach mit dieser Information betrügerische Aufrufe ausführen kann, hängt vom Serverobjekt und besonders von seinem verlangten Schutz der Verbindung ab. So ist ein Serverobjekt, das nachdem sein Klient sich bei ihm authentisiert hat, immer signierte Aufrufe verlangt, gegen falsche Aufrufe und *Replay*-Angriffe gestützt.

Schließlich sorgt dieses Protokoll für das Widerrufen von Zertifikaten aus Vereinfachungsgründen nicht. Statt dessen werden von den Attributanbietern augestellte Zertifikate durch ihre Gültigkeitsdaten zeitlich begrenzt. Dieses Zeitintervall kann für jeden Anbieter konfiguriert werden.

Im nächsten Kapitel wird die Implementierung dieses Protokolls und besonders von den SPKI Zertifikaten, die es benutzt, beschrieben.

5.0 Beschreibung meiner Implementierung

Dieses Kapitel stellt das Design meiner Implementierung vor. Nur die wichtigsten Klassen werden beschrieben. Aber hervorgehoben werden die Designentscheidungen, die ich getroffen habe, und ihre Gründe. Details über die Rolle jeder Klasse und ihrer Methoden kann im kommentierten Code selbst oder in der JavaDoc-Dokumentation dieser Klassen (Anhang C) gefunden werden.
Dieses Kapitel beschreibt alles, was ich während meiner Diplomarbeit implementiert habe, d.h : eine Bibliothek von Klassen, die mein Authentisierungsprotokoll realisieren, und eine Beispielanwendung, die dieses Protokoll verwendet und die am Ende dieses Kapitels vorgestellt wird. Im folgenden Text, wird mit “meine Implementierung“ diese Bibliothek gemeint, und nicht die Beispielanwendung.
Die folgende Tabelle misst die Größe meiner Implementierung. Klassen sind in 2 Kategorien geteilt: diese, die ich manuell implementiert habe, und diese, die automatisch generiert wurden.
Die Zahl von Codelinien bezieht Kommentare ein, aber nicht *Copyright*-Headers.
Übrigens ist die Beispielanwendung in dieser Tabelle nicht dargestellt.

TABLE 1. Größe meiner Implementierung

	Zahl von Klassen und Schnittstellen	Zahl von Codelinien
1. Bibliothek: von mir implementiert	92	6’276
3. Bibliothek: automatisch generiert	108	15’184
GesamtSumme	200	21’460

5.1 Allgemeine Sicht: Schichten

Meine ganze Implementierung ist in Schichten getrennt. Eine Schicht entspricht einem Schritt im Prozeß, Zertifikate und Schlüssel zwischen den Teilnehmern des Protokolls zu vermitteln. Die oberste Schicht wird in JacORB benutzt. Am anderen Ende benutzt die untere Schicht direkt die Java I/O-Klassen.
Jede Schicht ist auch eine Menge von Klassen, die den gleichen Zweck erfüllen und die als ein Java-Package gesammelt werden.
Anhang B1 zeigt die gesamte Architektur meiner Implementierung und wird in diesem Unterkapitel erklärt.

5.1.1 Schichten

Alle Klassen, die ich implementiert habe, wurden in einem Package `spki` gesammelt. Für die Implementierung wird ein zweites Package (`cryptix`) benutzt, das Werkzeugklassen enthält und das später vorgestellt wird.

Die obere Schicht ist Teil von JacORB selbst: die Klassen haben dort ihre Signatur von einer IDL Schnittstelle definiert. Sie (*Principal Authenticator, Current,* Credentials) waren vor meiner Arbeit schon teilweise implementiert, aber unterstützten Authen-

tisierung nicht. Die Änderungen, die ich an diesen Klassen gemacht habe, sind sehr klein.

Schicht 4 stellt die Teilnehmer des Protokolls dar. Folglich enthält sie auch Stubs und Skeletons für die Attributserver und -anbieter.
Sie wird vom Package `spki.auth` (wie "Authentisierung") vertreten.
Die Klassen dieser Schicht behandeln und vermitteln drei Arten von Objekten: SZL, Schlüssel und Zertifikate.

Diese letzten Objekte werden in Schicht 3 implementiert und im Package `spki.certificate` gesammelt. Sie erfüllen hauptsächlich 3 Funktionalitäten: Zertifikate (verschiedener Typen) darstellen, Schlüssel generieren und Texte (wie Zertifikate oder SZL) signieren. Zertifikate und Schlüssel verfügen außerdem nach über Operationen, um sich aus S-Expressions Objekte zu erzeugen und bzw. sich als S-Expressions darzustellen.

S-Expression-Objekte bilden Schicht 2 (Package `spki.sexp`) und stellen unterschiedliche Typen von S-Expressions dar. Nicht alle SPKI-Objekte entsprechen einer Klasse, weil nicht alle von meiner Implementierung erfordert werden und weil Objekte dieser Schicht grobkörniger als SPKI-Objekte sind und deshalb mehrere von ihnen zusammen darstellen.
Jedes S-Expression-Objekt erfüllt 3 Funktionalitäten. Erstens bietet es die verschiedenen Felder der entsprechender S-Expression als Attribute des Objekts an. Zweitens lässt es sich in kanonischer oder fortgeschrittener Form serialisieren. Drittens lässt es sich aus einem Syntaxbaum (AST für *abstract syntax tree*) erzeugen. Was ein AST ist, wird später erklärt.

Diese AST aufzubauen ist eigentlich die Rolle der Schicht 1, die aus 3 Mengen von Klassen besteht. Die erste (`spki.syntaxtree`) enthält 95 Klassen. Jede von denen stellt ein SPKI-Objekt (von einer Regel der Grammatik definiert) dar. Das zweite Package (`spki.parsing`) enthält einen Parser, die ASTs aus kanonischen S-Expressions erzeugt. Die dritte Menge von Klassen (`spki.visitor`) sind Basisvisitoren für ASTs. Was *Visitoren* sind, wird auch später erklärt.
Die Klassen dieser Schicht wurden vollständig automatisch generiert.

5.1.2 Prinzipien für Trennungen zwischen Schichten

Die Wahl, eine Architektur in Schichten zu trennen, und der Entschluss, wo genau eine Trennung zu machen ist, beruhen auf einer Grundlage der Objektorientierung: Wiederverwendbarkeit.
Die Schichten sollen so getrennt sein, daß es möglich wäre, eine von ihnen durch eine andere Implementierung zu ersetzen, ohne die andere viel ändern müßen. Je genauer eine Schicht einer bestimmten Funktionalität entspricht, desto einfacher ist ihr Ersatz, genau wie mit Objekten.
Formeller gesagt, soll außerdem die Ordnung der Schichten 2 Bedingungen erfüllen:

- Eine gewisse Schicht ist nie von einer oberen Schicht abhängig, d.h. sie benutzt sie nie.
- Eine Schicht benutzt nur die Schicht, die direkt unter ihr liegt, und keine untere Schicht.

Die erste Bedingung erlaubt es, nur einen unteren Teil des “Stacks” von Schichten in einer anderen Anwendung wiederzuverwenden. Sie wird in meiner Implementierung streng respektiert. So können Schichten 1 und 2 als Implementierung der SPKI-Grammatik in einem System, das Zertifikate und Schlüssel anders als meines darstellt, wiederverwendet werden. Ebenso können Schichten 1, 2 und 3 in einer Anwendung, die SPKI-Principals und Zertifikate behandelt und die ganz anders als JacORB-Authentisierung ist, auch wiederverwendet werden.
Die zweite Bedingung erlaubt, daß eine Schicht ihre Schnittstelle ändern kann (z.B. weil sie ersetzt wird), ohne daß eine andere Schicht, außer der direkt darüberliegenden, ihre auch ändern muß. Sie wird in meiner Implementierung allgemein respektiert; Ausnahmen sind Objekte der Schichten 3 und 4, die Objekte der Schicht 2 direkt benutzen, und die besondere Form der Schicht 5, die Schicht 3 und 4 direkt benutzt. Dieser letzte Fall ist keine schwere Verletzung der Bedingung, weil Schichten 4 und 5 (im Vergleich mit den anderen) dünn sind.
Nachdem die Prinzipien, die zur Entscheidung über die Position einer Trennung führen, erklärt wurden, werden diese Trennungen selbst vorgestellt.

5.1.3 Gründe für die Trennungen zwischen Schichten

Die Trennung zwischen Schichten 5 und 4 entspringt aus der Sorge, JacORB Klassen so wenig wie möglich zu ändern und sie von meiner Implementierung zu trennen. Der Grund dieser Trennung von Packages war, eine gleichzeitige Entwicklung von meiner Implementierung und von JacORB zu erlauben, ohne daß eine der Entwicklungen Wirkungen auf die andere hatte. Die Schichtentrennung entspricht auch der Trennung zwischen den Klassen, die ich geschrieben habe, und den in JacORB existierenden, die ich nur geändert habe.

Obwohl Schichten 3 und darunterliegende in jedem SPKI-basierten System verwendet werden können, stellt Schicht 4 nur eine bestimmte Anwendung (nämlich Authentisierung in JacORB) dar. Mit anderen Worten sind Schichten 1, 2 und 3 eine Bibliothek von SPKI-Objekte, während Schichten 4-5 eine Anwendung sind.

Die Trennung zwischen Schichten 2 und 3 kann besser verstanden werden, wenn man den Unterschied zwischen Schichten 1 und 2 kennt.
Wie schon erwähnt, wurden die drei Packages der Schicht 1 automatisch generiert. Das heißt, daß jedesmal, wenn der Input des Klassengenerators geändert wird, auch alle Klassen erneut generiert werden. Also generiert man entweder diese Klassen ein für allemal und kann folglich ihren Code ändern, oder erlaubt man sich, den Input des Generators während der Entwicklung der ganzen Bibliothek von Zeit zu Zeit zu ändern und die Klassen neu zu generieren. Im letzten Fall würde jede Änderung der generierten Klassen von einer neuen Generierung zerstört.
In unserem Fall war der Input des Generators die Reihe von SPKI-Grammatikregeln. Da sie von ihren Authoren ständig weiterentwickelt werden und da ich bis spät im Entwicklungsprozeß über das Format des Inputs lernte, war es unbedingt erforderlich, die Klassen neu generieren zu können, ohne Teile meiner Implementierung damit zu verlieren. Deshalb habe ich Schicht 1, die auf jeder Zeit neu generiert werden kann, von Schicht 2, die ich implementiert habe, getrennt.

Im Gegensatz zu den vorher erklärten Trennungen, die einer Notwendigkeit entsprechen, könnte die Trennung zwischen Schichten 2 und 3 auch wegfallen. Objekte der Schicht 3 (Zertifikate, Schlüssel, SZL) könnten sich selbst serialisieren und sich direkt aus ASTs der Schicht 1 erzeugen, ohne jedesmal ein S-Expression Objekt als Zwischenschritt zu benutzen.
Trotzdem habe ich mich aus 2 Gründen entschlossen, zusätzlich Schicht 2 in die Architektur einzufügen.
Grundsätzlich entspricht dieser Trennung einer zwischen Syntax (S-Expressions Objekte) und Semantik (Zertifikatobjekte). So könnte die ganze SPKI Grammatik (aber nicht die SPKI-Infrastruktur) geändert werden, ohne daß die Objekte, die die Infrastruktur darstellen, geändert werden müßen. In diesem Fall würde nur Schicht 2 ersetzt. Obwohl diese Eventualität unwahrscheinlich ist, sind sicher (mindestens) kleine Änderungen der Grammatik zu erwarten. Die rufen dann nur Implementierungsänderungen hervor, die auf Schicht 2 begrenzt sind und einfach durchzuführen sind, weil die geänderten Objekte nicht andere Funktionalitäten (wie diese der Schicht 3) enthalten.
Der zweite Grund ist ein praktischer. Jeder Schicht entsprach einer Phase meiner Implementierung. Es war einfacher, zuerst sicherzustellen, daß die SPKI-Grammatik richtig parsiert wurde und die ASTs richtig traversiert wurden, bevor sich darum zu kümmern, wie Zertifikate zu unterschreiben oder Schlüssel zu behandeln sind. Eine Fusion der Schichten 2 und 3 hätte mich dazu gezwungen, alle diese Probleme gleichzeitig zu lösen.
Wie schon erwähnt entspringt aus der Existenz der Schicht 2 ein funktional nutzloser Zwischenschritt im (De)Serialisierungsprozess. Wenn einmal die Grammatik stabil geworden ist und wenn Konventionen über die Darstellung (die nicht immer eindeutig ist) von Zertifikaten als SPKI Objekte festgelegt sind, dann könnte ein Leistungsgewinn durch die Eliminierung der Schicht 2 erreicht werden.

5.1.4 Bemerkungen

Sowohl die Schichten als auch ihre Grenzen wurden in einer *Top-Down* Ordnung vorgestellt, die den Weg von den Anforderungen (Protokoll) aus bis zu ihren *low-level* Realisierung (Parsing) darstellt.
Im nächsten Kapitel werden die Schichten genau in der umgekehrten Richtung (*Bottom-Up*) detailliert vorgestellt. So wird es verständlicher, weil ich sie in dieser Ordnung implementiert habe und weil Operationen einer Schicht sich auf diese der unteren Schicht abstützen.

5.2 Schicht 1: Parsergenerator

5.2.1 Ziel

Das Ziel der zwei unteren Schichten meiner Architektur ist, Zertifikatobjekte (aber auch SZL und Schlüssel) zu serialisieren und deserialisieren.
Serialisierung mit der SPKI-Grammatik ist einfach: man nimmt in einer bestimmten Ordnung jede Komponente eines Objekts und drückt sie (auch rekursiv, wenn erforderlich) als String aus.
Im Gegensatz braucht Deserialisierung einen größeren Aufwand von Operationen, um Strings zu parsieren und damit Objekte zu erzeugen.

5.2.2 Prinzipien des Parsierens

Das Parsing einer S-Expression lässt sich ein drei Phasen teilen:

- In der lexikalischen Analyse schneidet ein sogenannter *Token Manager* (also Scanner) einen Text in *Tokens* (Wörter in der Grammatik) ab.
- Die syntaktische Analyse kontrolliert, ob die Sequenz dieser Tokens den Regeln der Grammatik entspricht.
- Die Erzeugung von Objekten aus den parsierten Ausdrücken beendet die Deserialisierung.

Dieser Prozeß kann in jeder dieser Phasen wegen eines Fehler angehalten werden. Er ist in Abbildung 20 illustriert.

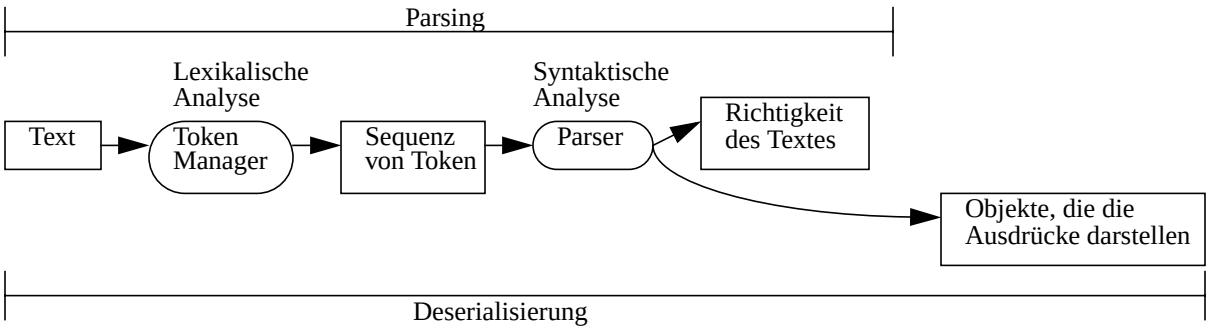


Abbildung 20. Abschnitte einer Deserialisierung

Einen Parser (und seinen *Token Manager*) zu programmieren ist keine einfache Aufgabe, besonders für eine Grammatik von circa 90 Regeln. Auch weil Parsing kein Thema meiner Diplomarbeit war, habe ich ein Werkzeug verwendet, das einen Java-Parser (zusammen mit einem *Token Manager*) aus einer Datei von Grammatikregeln automatisch generiert: JavaCC.
Alternative Werkzeuge habe ich nicht gesucht und abgeschätzt, weil JavaCC alle meine Bedürfnisse erfüllte.

5.2.3 JavaCC

JavaCC [27] von Sun Microsystems (tm) ist selbst eine kostenlose Java-Anwendung.
In der Inputdatei wird einerseits definiert, was ein Token ist, und andererseits, was die Regeln sind, die eine Sequenz von Tokens respektieren muß. Aus der Definition eines Tokens wird eine `TokenManager`-Klasse generiert, während eine `Parser`-Klasse aus der Menge von Regeln erzeugt wird.
Eine Option erlaubt es, nur den Parser zu generieren. Aber dann muß der Programmieren seinen eigenen *Token Manager* implementieren. Eine andere Option spezifiziert das *Lookahead* des Parsers (d.h. wieviele Tokens vorwärts er anschauen muß, bevor er sich im Fall eines zweideutiges Präfixes für den einen oder den anderen Ausdruck entscheidet).

Die einzige Aktion eines so generiertes Parsers ist, die Richtigkeit seines Inputs zu bestimmen. Um beim Parsing jedes Ausdrucks eine bestimmte Aktion durchzuführen (z.B. das entsprechende Objekt zu generieren und es so deserialisieren) kann man in jeder Regel der Inputdatei beliebigen Java-Code einfügen, der jedesmal durchgeführt wird, wenn diese Regel angewendet wird. Theoretisch wäre es so möglich, Objekte zu erzeugen.
Aber in unserem Fall würde ein solcher Code für jede Regel ganze Methoden enthalten, weil es für die meisten SPKI-Objekte Alternativen und Felder zu unterscheiden gibt, die nur unter bestimmten Umständen zu betrachten sind. Ein Stück Code in jeder der 90 Regeln würde die Inputdatei unüberschaubar und deshalb schwer veränderlich machen. Aber Einfachheit der Änderung der Grammatik ist eine grundsätzliche Anforderung an meine Implementierung, da die SPKI-Grammatik nicht stabil ist. Folglich mußte die Implementierung der Erzeugung von Objekten weiter automatisiert werden.

5.2.4 Syntaxbäume

Ein Syntaxbaum (engl. AST wie *abstract syntax tree*) [1] ist ein Baum dessen Blätter Tokens und dessen Knoten Strukturen sind. Strukturen sind entweder zusammengesetzte Objekte, die von einer Regel definiert werden, oder homogene Strukturen, wie z.B. Listen, optionale Listen, optionale Felder oder Alternativen. Ein Liste ist homogen, wenn sie nur Tokens eines gleiches Typs enthält.
Abbildung 21 zeigt das AST der folgenden *name* S-Expression:
`(name john mark edward),`
wobei die folgenden Regeln zutreffen:

```
<name>          :: <relative-name> | <fq-name> ;
<relative-name> :: "(" "name" <names> ")" ;
<names>        :: <byte-string>+ ;
```

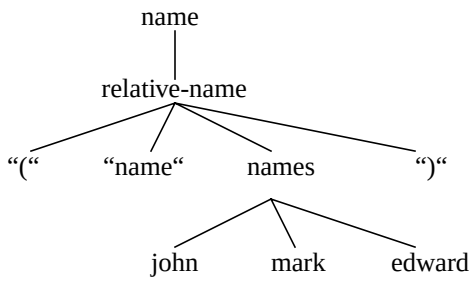


Abbildung 21. Beispiel eines Syntaxbaums

Während er einen Ausdruck parsiert, erkennt der Parser, welchen Regeln die verschiedenen Teile des Ausdrucks entsprechen. Deshalb kann er auch gleichzeitig den Baum aufbauen. Zwei Erweiterungen von JavaCC erfüllen genau diese Aufgabe: JJTree und JTB.

5.2.5 JJTree

JJTree ist mit JavaCC zusammengeliefert. Als Input nimmt er die gleiche Datei wie JavaCC. Als output generiert er eine Menge von Klassen und eine neue Datei in

JavaCC-Format. Die Klassen entsprechen Regeln der Grammatik und ihre Instanzen werden die Knoten der Syntaxbäume. Die neue Datei enthält die gleichen Regeln wie der Input, aber Befehle, um den Syntaxbaum beim Parsing aufzubauen, wurden eingefügt. Dann kann diese Datei als Input von JavaCC verwendet werden, der einen Parser generieren wird.
Eine Anwendung dieses Parsers gibt die Wurzel des aufgebauten Syntaxbaumes zurück. Information kann auf zwei verschiedenen Weisen aus diesem Baum herausgenommen werden, um z.B. Objekte zu erzeugen.
Mit der ersten Strategie sind die erzeugten Objekte die Knoten des Baumes selbst. Dafür modifiziert der Programmierer die von JJTree generierten Klassen, so daß sie nicht mehr nur Knoten darstellen, sondern auch vollständige Ausdrücke, deren Komponenten als Attribute eines Knotenobjekts gelesen werden können.
Im Beispiel der Abbildung 21 würde eine Klasse `Name` generiert. Sie könnte dann weiterimplementiert werden, so daß sie über eine Methode `getName(int index)` verfügt, die einen der Namen *John, Mark, Edward* direkt liefert, ohne daß man den Baum von Knoten zu Knoten traversiert. In unserem Fall würde so jede Knotenklasse eine S-Expression darstellen, deren Felder direkt als Attribute eines Objekts nachschlagbar sind. Dann könnten Zertifikatobjekte aus solchen S-Expression-Objekten leicht (einfach durch eine Kopie der betreffenden Attribute) erzeugen werden. Es würde auch die Notwendigkeit einer zweiten Schicht, wie in meiner Architektur, beseitigen.
Leider ist diese Lösung nicht möglich, weil sie voraussetzt, daß Klassen ein für allemal generiert werden. Es wurde schon erwähnt, daß diese Voraussetzung in meiner Implementierung nicht gilt.
Die zweite Lösung ist im Prinzip die gleiche awie die vorige, aber der neu geschriebene Code wird nicht direkt in Knotenklassen geschrieben, sondern in neuen Klassen, die Knoten und ihre unterliegende Bäume lesen und die die von diesen Knoten getragene Information in separaten Objekten widerspiegeln. Genau diese Rolle spielen in meiner Implementierung die Objekte der Schicht 2, da ich diese zweite Strategie angewendet habe.

JJTree unterstutzt auch das *Visitor Design Pattern* [13]. Dieses Pattern lässt sich auf jede heterogene Struktur, deren Elemente verarbeitet werden sollen, anwenden. Aber es wird hier nur im Rahmen einer Baumstruktur erklärt werden.
Die Teilnehmer des Patterns sind Visitoren und Knoten. Jeder Knoten enthält eine Methode `accept(Visitor v)`, die nur `v.visit(this)` zurückruft, wobei `this` die Referenz des Knotens ist. Ein Visitor enthält eine Reihe von Methoden `visit(NodeType n)`, wobei `NodeType` jedem Typ von Objekten entspricht, den der Visitor zu besuchen vorhat.

Als Beispiel betrachten wir einen Visitor, der alle Knoten in einer *DepthFirst*-Ordnung besucht und deren Inhalt ausdruckt. Der Code eines solchen Visitors ist in Abbildung 22 gezeigt.

```
class PrinterVisitor {

   public PrinterVisitor() {....}

   public visit(NodeType n){
      print(n);
       n.childNode.accept(this);
      Für jeden Kindknoten wiederholen.
      .....
   }

}
```

Eine solche Methode für jeden Typ von Knoten

Abbildung 22. Ein Visitor, der den Inhalt jedes Knotens ausdruckt.

Es ist zu bemerken, daß die Implementierung dieser Operation auf den Code der Knoten keine Auswirkungen hat. So können neue Funktionalitäten hinzukommen, ohne die Knotenklassen zu ändern.
Folglich sieht auf den ersten Blick das *Visitor Pattern* für meinen Fall ideal geeignet aus. Leider ist sein Anwendungsgebiet sehr beschränkt. Einen Visitor zu implementieren, statt explizit im Baum zu tauchen und von Knoten zu Knoten bis zur gewünschten Information zu navigieren, lohnt sich nur, wenn die durchzuführende Operation auf allen (oder die meisten) Knoten angewendet werden soll. Außer zwei Ausnahmen (einen Syntaxbaum in kanonischer oder in fortgeschrittener Form auszudrucken) sind eigentlich alle Operationen auf dem Baum lokal, d.h. sie kümmern sich nur um ein paar Knoten, die nur ein kleinenen Teil des gesamten Baums darstellen. Es wäre zum Beispiel sinnlos, einen Visitor zu implementieren, um den Wert eines bestimmten Feldes einer S-Expression in ihrem Syntaxbaum zu finden.

Schließlich war JJTree für meine Implementierung benutzbar, obwohl mehrere seiner Funktionalitäten (Visitoren, Änderung von generierten Klassen) von mir ungenutzt blieben.

5.2.6 JTB

JTB [28] wurde von J. Palsberg and K. Tao von der Purdue University entwickelt. Die Version, die ich benutzt habe, ist 1.1pre4.

Die Funktionalität von JJTree und von JTB sind ähnlich. Beide sind Präprozessoren für JavaCC, und beide unterstützen Visitoren.
JJTree hat einige Vorteile gegen JTB.
Erstens erlaubt er, daß Knotenklassen modifiziert werden, während diese Klassen bei JTB nur getypte Strukturen (Objekte, deren Instanzvariablen alle öffentlich sind) sind, die nie geändert werden sollen, sondern nur von Visitoren behandelt werden. Aber dieser Vorteil ist im Fall meiner Implementierung nutzlos.
Zweitens erlaubt JJTree, den Typ eines Tokens zu erkennen. Im Gegensatz kennt man von einem Token in einem von JTB generierten Baum nur den Wert und den Fakt, daß es ein Token ist.

Drittens ist JJTree flexibler: ein Benutzer kann genau sagen, welche Typen von Knoten im Syntaxbaum dargestellt werden müssen und welche ignoriert werden können, was in JTB nicht möglich ist. Aber diese Flexibilität hat einen Preis: das Format einer Inputdatei von JJTree ist auf jeden Fall komplizierter als das von JTB.

Dies ist der erste Vorteil von JTB, besonders weil ich keine große Flexibilität von einem Syntaxbaumerzeuger verlangte.
Ein zweiter Vorteil von JTB ist, daß er auch Basisvisitorklassen generiert, wie einen `DepthFirstVisitor`, der alle Knoten eines Baums in einer bestimmten Reihenfolge traversiert.
Drittens sind die Kinderknoten eines bestimmten Knotens mit ihrem eigentlichem Typ referenziert, und nicht alle in einem Vektor enthalten, wie bei JJTree. Diese Eigenschaft vereinfacht die Navigation durch den Baum.

Die zwei letzten Listen von Vorteilen zeigen, daß JTB besser als JJTree für meine Aufgabe passte. Deshalb habe ich ihn gewählt.

5.2.7 Packagestruktur

Schicht 1 ist genau das Resultat der Anwendung von JTB auf die im Anhang A aufgelistete Grammatik. JTB hat 3 Packages generiert:

- `spki.syntaxtree` enthält 95 Klassen. Die meisten dieser Klassen entsprechen SPKI-Objekten und folglich auch Regeln der Grammatik. Aber es gibt auch 6 Klassen universellen Zwecks: `NodeChoice` (eine Alternative) " x | y", `NodeList` (eine Liste) "(x)+", `NodeListOptional` "(x)", `NodeOptional` "[x]", `NodeSequence` "x y z", `NodeToken` (ein Token). (Zwischen Anführungszeichen sind die entsprechenden EBNF-Notierungen beschrieben.)
- `spki.parsing` enthält den Parser und seinen Token-Manager.
- `spki.visitor` enthält vordefinierte Visitoren, wie den schon erwähnten `DepthFirstVisitor`.

5.2.8 Schwierigkeiten mit dem Parsing

Praktisch habe ich mit dem Parsing drei Hauptprobleme gehabt.

Das erste war, die richtige Definition eines Tokens in JTB auszudrücken. Für Schlüsselwörter und spezielle Zeichen war die Aufgabe trivial. Aber einen String beliebiger Länge, der mit der Angabe seiner Lange präfigiert wird, als ein Token zu erkennen und seinen Wert zu lesen, erfordert, Java Code in der Definition eines Tokens zu schreiben und eine kleine Zustandsmaschine zu programmieren. Dieser Code steht in der Datei `parser.jj` in meiner Implementierung.
Das zweite Problem wurde schon als Vorteil von JJTree erwähnt, daß nämlich der Typ eines Tokens in einer Instanz der Klasse `spki.syntaxtree.NodeToken` nicht zu finden ist. Als Lösung dieses Problem habe ich einige Tokens (diese, von denen man den Typ kennen mußte) auch als eine Grammatikregel ausgedrückt. So wurde ein solches Token in einem Syntaxbaum mit einem Knoten bestimmtes Typs und einem neuen Token ersetzt. Diese Transformation hat 6 neue Klassen geschaffen.
Das dritte Problem war zu bestimmen, welche Wert das *Lookahead* haben sollte. Je größer diese Zahl ist, desto langsamer läuft der Parser. Aber wenn sie zu klein ist, macht er Fehler, weil er zwei Ausdrücke nicht mehr richtig unterscheiden kann. Mehrere Experi-

mente haben gezeigt, daß der minimale Wert 8 ist. Diese Zahl ist ausserordentlich hoch. Sie könnte auf fast ihren Hälfte reduziert werden, wenn man nicht das gleiche Schlüsselwort “cert” sowohl für Namenszertifikate als auch für Erlaubniszertifikate verwenden würde. Das wäre um so vernünftiger, da beide ganz unterschiedliche Typen sind.

Die Autoren von SPKI behaupten [10]: *Parsing of a canonical form S-expression requires minimal look-ahead and no re-scanning of incoming bytes. As a result, the parsing code remains very small. Assuming each byte string is stored with a length field, generation of a canonical form from a data structure requires an extremely small amount of code.*

Meine Erfahrung, die sich auf dem Problem des *Lookaheads* und auf meiner gesamten Implementierung dieser Grammatik basiert, zeigt, daß das erforderte *Lookahead* gar nicht minimal ist und daß der Code für das Parsing, obwohl nicht sehr groß, auf jeden Fall nicht sehr klein (*very small*) bleibt. Es könnte eingewendet werden, daß mein Code viel kleiner wäre, wenn ich keinen Parsergenerator verwendet hätte, sondern den Parser an SPKI genau angepaßt und selbst programmiert hätte. Aber so hätte die Implementierung des Parsers länger als diese mit dem Generator gedauert, was das allgemeine Argument für die Einfachheit der Implementierung der SPKI-Grammatik auch widerlegt.

5.3 Schicht 2: S-Expressions

Diese Schicht serialisiert S-Expressions und stellt die aus Syntaxbäume gezogene Information als Objekte dar. Diese Objekte stellen trotzdem nicht beliebige S-Expressions dar, sondern unterstützen nur eine Untermenge von ihnen.

5.3.1 Nur partielle Implementierung

Abbildung 23 zeigt die Beziehung zwischen Schichten in ihrer Unterstützung von S-Expressions. Der größte Kreis ist die Menge aller von der Grammatik definierten S-Expressions. Da die Klassen der Schicht 1 automatisch aus der Grammatik generiert wurden, stellt diese Menge auch die von der Schicht 1 unterstützten S-Expressions dar. Das heißt, daß alle möglichen S-Expressions in meiner Implementierung parsiert werden können. Im Zentrum ist die Menge aller S-Expressions, die von den Zertifikat- und Schlüsselobjekten benutzt werden. Diese Menge wurde nach den Bedürfnissen aus der Implementierung des Protokolls festgelegt. Dazwischen ist die Menge von S-Expressions, die von Schicht 2 unterstützt werden.
Idealerweise sollte diese Menge gleich dem inneren Kreis sein, weil es nichts nützt, eine S-Expression in der Schicht 2 darzustellen und sie in Schicht 3 nicht zu benutzen. Die Unterstützung der Schicht 2 ist aus zwei Gründen größer.
Erstens habe ich meine Implementierung in einer *Bottom-Up* Richtung realisiert. Obwohl mein Design voraussah, was die Bedürfnisse der oberen Schichten waren, hat die deren Implementierung die Designpläne manchmal geändert. Deshalb sind einige schon entwickelte Klassen (wie z.B. Erlaubniszertifikat-S-Expressions) nutzlos geworden. Zweitens wurden einige Klassen aus Vollständigkeitsgründen implementiert. Z.B. war es kein großer Programmierungsaufwand, *relative names* S-Expressions zu unterstützen, nachdem ich *fully-qualified* names implementiert hatte.

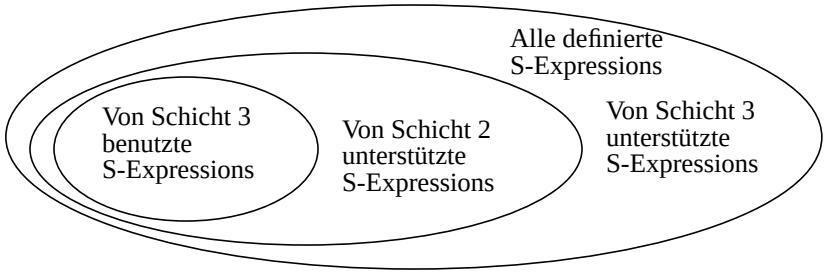


Abbildung 23. Unterstützte SPKI Objekte

Spezifisch sind alle S-Expressions, die als Klassennamen im Anhang B2 erscheinen, unterstützt. Auch andere S-Expressions sind unterstützt, aber wurden in eine der Klassen im Anhang B2 integriert, um die Darstellung der S-Expressions in Objekten zu vereinfachen. Zum Beispiel wurden die S-Expressions *deleg* oder *valid* als Felder der Klasse `AbstractCertSExp` unterstützt. Die wichtigsten S-Expressions, die nicht unterstützt werden, sind *crl*, *acl*, Tags, *subject-threshold* und generische *s-expressions.*
Sogar die von Objekten dargestellten S-Expressions sind nicht immer vollständig unterstützt, weil einige ihrer Felder mir nutzlos waren. Zum Beispiel ignorieren die Zertifikatklassen die *comment, cert-display, online-test* Felder. Schlüssel tun das gleiche mit *uris* und Bytestrings mit *display-type.*

5.3.2 Beschreibung der Klassen

Anhang B2 zeigt alle Klassen der Schicht 2, die S-Expressions darstellen. Die Organisation dieses Vererbungsdiagramms wird von 3 Prinzipien bestimmt.

Erstens wird eine S-Expression entweder von einer Klasse oder von einer Schnittstelle dargestellt. Wenn sie eine Sequenz von Tokens und andere S-Expressions (ihre Felder) ist, dann entspricht sie einer Klasse, deren Attribute ihre Felder darstellen und in einem möglichst genauen Typ (`java.util.Date` für ein Datum, `java.lang.String` für ein Bytestring, usw.) ausgedrückt sind.
Wenn ein S-Expression aber eine Alternative ist, dann entspricht sie einer Schnittstelle. Außerdem wird jede Klasse, die auch ein Zweig der Alternative ist, diese Schnittstelle implementieren. Weil die SPKI-Grammatik keine Alternative mit einer Sequenz als Zweig enthält, ist die Wahl zwischen Schnittstelle und Klasse immer eindeutig.
Schnittstellen in Java können zwei Zwecke erfüllen. Sie können Methodensignaturen definieren und damit Unterklassen dazu zwingen, diese Methoden zu implementieren. Das ist ihre häufigste Anwendung. Aber sie können auch Typen unter einem gleichen Supertyp gruppieren und so eine gleiche Sicht auf Objekte verschiedener Typen anbieten. Meine Schnittstellen `SeqEntSExp, KeyHolderObjSExp, SubjObjSExp, OpSExp und PrincipalSExp` gehören zur zweiten Kategorie.

Das zweite Prinzip ist, daß alle S-Expression-Klassen von der Klasse `SExpression` erben. Diese abstrakte Klasse spezifiziert und bietet Methoden für die Deserialisierung von S-Expressions und für ihren Ausdruck in kanonischer oder fortgeschrittener Form. Details über diesen Prozeß werden im nächsten Teil dieses Berichts erklärt.

Das dritte Prinzip ist über die Vererbung zwischen Klassen anders als der Klasse `S-Expression`. Obwohl die Vererbung unter Schnittstellen und zwischen Schnittstellen und Klassen Regeln der Grammatik (nämlich Alternativen) entspricht, nimmt die Vererbung zwischen Klassen (nämlich aus `AbstractCertSExp` und `NameSExp`) keine Rücksicht auf die Grammatik, sondern gruppiert Klassen, die gemeinsame Funktionalitäten besitzen, unter einer gleichen Superklasse. Zum Beispiel, weil sie einen ähnlichen Inhalt haben, erben `CertSExp` und `NameCertSExp` beide `AbstractCertSExp`, obwohl sie grammatisch keine Beziehung haben.

Noch nicht erwähnt wurde die Schnittstelle `SExpInterface`, von der alle Klassen und Schnittstellen erben. Sie widerspiegelt die Signatur der öffentlichen Methoden von `SExpression`. Ohne sie könnte man z.B. die kanonische Form eines Objekts von Typ `SubjObjSExp` nicht erhalten, obwohl dieses Objekt diese Funktionalität sicher anbietet, weil alle mögliche Unterklassen von `SubjObjSExp` von `SExpression` erben.

Weitere und genauere Erklärungen über die Klassen dieser Schicht sind im Anhang C zu finden.

5.3.3 Serialisierungsmechanismen

Jedes Objekt, das eine S-Expression darstellt, kann auf zwei Weisen erzeugt werden. Die eine erzeugt es durch das Parsing von Daten aus einem *InputStream* (Deserialisierung) und die andere aus einer Reihe von Parametern, die seinem Konstruktor vermittelt werden. Jede dieser Weisen bestimmt einen Zustand des Objektes, nämlich ob es einen Syntaxbaum enthält oder nicht.
Dieser Zustand bestimmt, wie das Objekt seine kanonische Form ausdruckt und, folglich, wie es serialisiert wird. Wenn es einen Syntaxbaum enthält, dann wird ein von mir implementierter Visitor verwendet: `NumbererVisitor`, der den ganzen Baum in einer *Depth First*-Ordnung traversiert und den Inhalt jedes Blattes ausdruckt. Aber wenn das Objekt keinen Syntaxbaum enthält, dann wird seine `toDirectCanonical()` Methode es direkt in seiner kanonischen Form ausdrucken. Der Nachteil dieser Methode ist, daß sie für jede Klasse unabhängig implementiert werden muß. Eine Alternative wäre, den entsprechenden Syntaxbaum eines Objekts explizit aufzubauen und danach einen Visitor darauf anzuwenden. Ich habe diese Möglichkeit ausgeschlossen, weil der explizite Aufbau eines Syntaxbaums im Vergleich zu direkter Serialisierung zu viel Arbeit ist.
Wie die fortgeschrittene Form ausgedruckt wird, hängt auch vom Zustand des Objekts ab. Wenn es einen Syntaxbaum enthält, dann wird ein anderer Visitor (`AdvancedForm`) verwendet. Wenn es aber keinen Syntaxbaum gibt, dann hätte ich wie für die kanonische Form eine Methode definieren können, die den Ausdruck direkt liefert. Um meine Implementierungsarbeit zu vereinfachen, habe ich aber beschlossen, diesen Prozeß zu automatisieren. Seine Funktion wird in Abbildung 24 gezeigt.

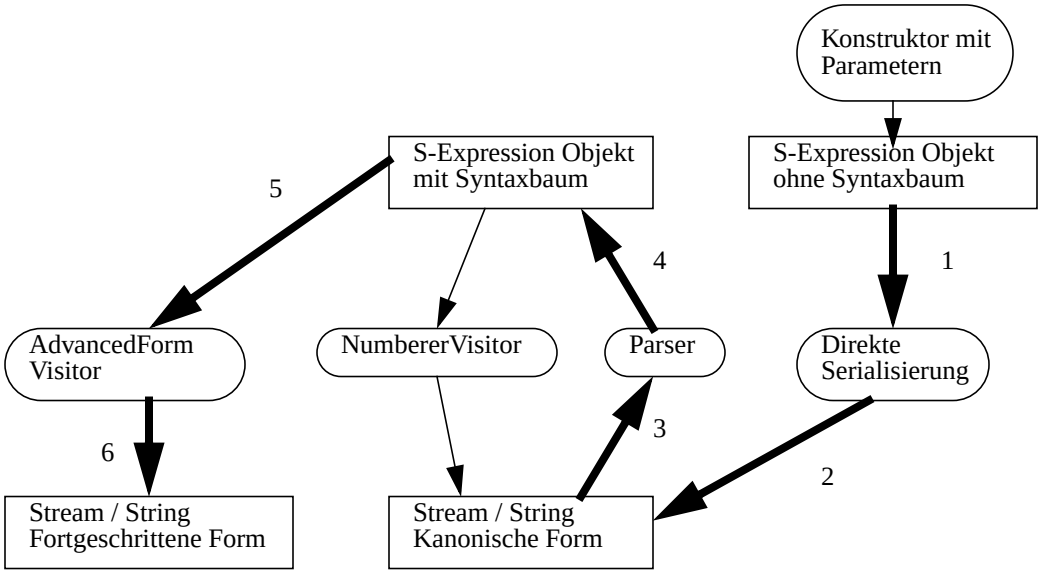


Abbildung 24.

Mögliche Behandlungen eines S-Expression Objekts
Dicke Pfeile: Prozeß des Ausdrucks der fortgeschrittenen Form eines Objektes ohne Syntaxbaum.

Diese Abbildung beschreibt alle existierenden Arten, ein S-Expression Objekt zu erzeugen und zu serialisieren. Der fette Weg ist der einzige, um die fortgeschrittene Form eines Objekts ohne Syntaxbaum automatisch auszudrucken.
Praktisch wird das Objekt zuerst in einen *Buffer* serialisiert. Dann wird der Inhalt dieses *Buffers* parsiert und damit der Syntaxbaum des Objekts aufgebaut. Schließlich lässt dieses Objekt einen `AdvancedForm`-Visitor seinen Syntaxbaum traversieren, um seine fortgeschrittene Form auszudrucken.
Obwohl diese ganze Operation zeitlich teuer ist, verursacht sie keinen Leistungsverlust bei der Vermittlung von Zertifikaten, weil diese immer in kanonischer Form vermittelt werden. Die fortgeschrittene Form wird nur ausgedruckt, wenn ein Mensch sich den Inhalt einer S-Expression anschauen will.

Die Entscheidungen, wie ein Objekt auszudrucken ist, und die Befehle, Syntaxbäume zu erzeugen und mit Visitoren zu besuchen, sind alle in der Wurzelklasse `SExpression` implementiert.

5.4 Schicht 3: Zertifikate

5.4.1 Allgemeines

Ziel dieser Schicht ist, die im Protokoll vermittelten Objekte (SZL und Schlüssel) anzubieten. Außerdem müssen sich diese Objekte serialisieren und deserialisieren lassen.

Objekte dieser Schicht teilen sich in drei Hauptkategorien: Öffentlichen Schlüssel, Zertifikate (zusammen mit SZL) und Hilfsobjekte.

Serialisierung:

Alle Hauptobjekte (d.h. SZL, Zertifikate und öffentliche Schlüssel) überschreiben die Methode `toString()` von `java.lang.Object`, so daß sie ihre kanonische SPKI-Darstellung als String zurückgeben. So wird die Serialisierungsfunktionalität eingebaut, ohne eine einzige neue Methode deklarieren zu müssen. Intern erzeugen sie ein S-Expression Objekt, das sie mit ihren eigenen Attributen initialisieren.
Alle Hauptobjekte können auch deserialisiert werden, weil alle einen Konstruktor besitzen, der S-Expression Objekte als Parameter annimmt.

Kompatibilität mit Java2:

Für meine Implementierung habe ich mit Java 2 (*Reference Implementation* für Solaris) [26] gearbeitet, weil seine Sicherheitspackages weiter entwickelt als die von Java 1.1 sind. Dabei war das Ziel, in den Packages `java.security.*` definierte Klassen und Schnittstellen so weit wie möglich wiederzuverwenden, so daß meine Implementierung den Java-Konventionen entspricht und so daß sie mit anderen Bibliotheken, die die gleiche Java-Schnittstellen auch respektieren, kombinierbar ist. Für jede Art von vorgestellten Klassen wird erklärt, ob dieses Ziel erreicht wurde.

Cryptix:

Um zu vermeiden, Signatur- oder Schlüsselgenerierungsalgorithmen implementieren zu müssen, habe ich zwei Quellen von Verfahren und Objekten benutzt: die Java Standard-Klassen und Cryptix.
Cryptix [24] ist eine von Systemics, Ltd entwickelte Bibliothek von Java-Klassen für Kryptographie. Sie ist kostenlos verfügbar und ihr Code darf geändert werden. Von dieser Bibliothek habe ich nur ein paar Klassen behalten und verwendet, die das Package `cryptix` bilden, das zusammen mit dem Package `spki` geliefert wird. Was ich vom gesamten Cryptix behalten habe sind:

- Klassen, die Datenformate behandeln und die für den Ausdruck fortgeschrittener Formen in Schicht 2 verwendet wurde.
- Klassen, die RSA-Schlüsselpaare generieren
- Klassen, die RSA-Schlüssel darstellen und RSA-Signaturen erzeugen.

Die übrigen benutzten Verfahren (DSA-Schlüsselpaare generieren und DSA-Signaturen erzeugen) werden von Java selbst angeboten.

5.4.2 Schlüssel

Die Anforderungen an Klassen zum Umgang mit Schlüsseln waren:

- Schlüssel als Daten darstellen. Die Objekte sollten die Parameter von sowohl öffentlichen Schlüsseln als auch privaten Schlüsseln enthalten und den Schlüsseltyp (Algorithmus) beschreiben. Außerdem sollte es möglich sein, öffentliche Schlüssel zu serialisieren.

- Signaturen erzeugen und überprüfen.
- Schlüsselpaare behandeln: sie generieren, laden und speichern, damit ein Principal *off-line* gehen kann, ohne seinen Schlüssel (und die damit verbundenen Rechte) zu verlieren.

Um die erste Anforderung zu realisieren, habe ich eine Hierarchie von Klassen, die öffentliche Signaturschlüssel darstellen, aufgebaut (siehe Anhang B5). Ich habe meine eigenen Klassen definiert und keine existierende Klasse wiederverwendet, weil diese Klassen die SPKI Serialisierungsverfahren enthalten mußten und auch nicht lediglich Verschlüsselungsschlüssel darstellen, sondern auch einem bestimmten Hashalgorithmus entsprechen. Die Integration mit Java 2 ist aber maximal, weil meine Klassen die Schnittstellen `java.security.Key`, `java.security.PublicKey`, `java.security.interfaces.RSAPublicKey` implementieren.
Im Gegensatz zu öffentlichen Signaturschlüsseln sind private Schlüssel nur Verschlüsselungschlüssel, d.h. daß sie keine Signaturoperation enthalten und keinem Hashalgorithmus entsprechen. Folglich sind sie nur Behälter für ihre Parameter. Deshalb konnte ich für RSA zwei Cryptix-Klassen wiederverwenden. Aber weder Java noch Cryptix verfügt über eine öffentliche Implementierung eines DSA-Schlüssels. Deshalb habe ich ihn selbst implementiert. Anhang B5 zeigt die Vererbungshierachie dieser Klassen, und besonders, wie sie sich mit Java-Schnittstellen integrieren.

Um die zweite der obigen Anforderungen zu erfüllen, verfügen öffentliche Schlüssel-Klassen über Operationen, um eine Signatur zu erzeugen oder zu überprüfen. Diese Algorithmen sind nicht in den Schlüssel-Klassen selbst implementiert, sondern benutzen externe Klassen:(`java.security.Signature` für DSA-SHA1, `cryptix.provider.rsa.MD5_RSA_PKCS1Signature` und `cryptix.provider.rsa.SHA1_RSA_PKCS1Signature`) für ihre entsprechende Algorithmen.

Die Klasse `spki.certificate.PersistentKeyPair` erfüllt die dritte Anforderung.
Ihre erste Aufgabe ist die Generierung von neuen Schlüsseln. Wie bei den Signaturverfahren habe ich die Schlüsselgenerierungsverfahren nicht selbst implementiert, sondern `java.security.KeyPairGenerator` für DSA und `cryptix.provider.rsa.BaseRSAKeyPairGenerator` für RSA verwendet. Da beide Verfahren bereits in Java verfügbar waren, habe ich auf die Möglichkeit verzichtet, externe Generatoren wie z.B. PGP zu benutzen.
Schlüsselpaare können in einer Datei gespeichert werden. Der öffentliche Schlüssel wird im Klartext geschrieben, während der private Schlüssel mit Hilfe eines vom Schlüsselsbesitzter gewähltes Paß-Satzes verschlüsselt wird. Beim Laden der Schlüssel wird der private Schlüssel durch Eingabe des Paß-Satzes entschlüsselt. Sowohl der öffentliche Schlüssel als auch sein privates Gegenstück werden als *pub-key* S-Expressions in kanonischer Form serialisiert und in diesem Format gespeichert.

Was über Public-Key-Objekte bis jetzt nicht erwähnt wurde, ist, daß sie auch Principal-Objekte darstellen können.

5.4.3 Principals

Anhang B4 zeigt, daß ein Principal eine von 2 Rollen spielt: Subjekt oder Aussteller eines Zertifikats. Ich habe die Wurzelschnittstelle `spki.certifikcate.PrincipalReference` genannt und nicht einfach Principal, weil es schon eine `java.security.Principal` Schnittstelle gibt. Diese hat aber den entscheidenden Nachteil, eine Methode `getName()` zu haben. Folglich entspricht sie dem Paradigma, wobei das Kennzeichen eines Principals sein *distinguished name* ist. Deshalb konnte ich sie nicht verwenden.
Der einzige Unterschied zwischen einem Aussteller und einem Subjekt ist, daß der Aussteller selbst nicht von seinem Namen identifiziert werden kann. Dieser Unterschied respektiert die SPKI-Grammatik.
Genau eine Instanz eines Subjekts und eine von einem Aussteller sind mit jedem Zertifikat Objekt verbunden.

5.4.4 Gebildete Objekte: Zertifikate

In meiner Implementierung gibt es vier Zertifikatsklassen. Ihre Hierarchie ist im Anhang B4 zu sehen.

- `Certificate` ist eine abstrakte Klasse, die gemeinsame Funktionalitäten für die beiden folgenden Typen bietet.
- `PermissionCert` stellt ein Erlaubniszertifikat dar, aber ist eigentlich nicht implementiert.
- `NameCert` stellt ein Namenszertifikat dar.
- `TypedNameCert` ist genau wie `NameCert`, außer daß sein “Name“-Feld einem bestimmten Format entspricht, nämlich “type:string”, wobei “string“ ein beliebiger String ist und “type“ einer von den Strings: “Public“, “AccessId“, “PrimaryGroupId“, “GroupId“, “Role“, “AttributeSet“, “Clearance“, “Capability“. Sie stellen CORBA-Sicherheitsattributtypen (*security attribute type*) dar und werden in der nächsten Schichte benutzt, um zu beschreiben, welchen Typ von Credentials ein Attributanbieter ausstellt.
  Eigentlich ist diese Klasse anwendungsspezifisch und sollte in einer höheren Schicht liegen, aber ich behalte sie zeitweilig in Schicht 3 zusammen mit den anderen Zertifikatklassen.

`spki.certificate.Certificate` erbt nicht von `java.security.cert.Certificate`, weil diese Klasse eine `getPublicKey()` Methode enthält. Aber in SPKI ist oft kein Schlüssel in einem Zertifikat enthalten, sondern nur Namen oder Hashwerte. Deshalb war die Integration mit dem Java-API unmöglich.

Schließlich hat ein Zertifikatobjekt immer eine richtige Signatur: sobald es erzeugt wird, wird es signiert, und sobald es deserialisiert wird, wird seine Signatur überprüft und die Erzeugung verweigert, falls sie nicht stimmt.

Wie schon im letzten Kapitel erklärt, werden Zertifikate nur vermittelt, wenn sie in einer SZL (Klasse `spki.certificate.SignedCertList`) verpackt sind.
Wie für Zertifikate ist die Signatur der Sequenz immer richtig. Aber ein Objekt dieser Klasse kann sich in einem von zwei Zuständen befinden. Im ersten, der nur durch eine

Deserialisierung erreicht werden kann, sind die enthaltenen Zertifikate nur ein einziger String. Deswegen werden die Signaturen der einzelnen Zertifikate nicht überprüft. Dieser Zustand erspart Rechenzeit, wenn der Empfänger einer SZL folgendes will: prüfen, ob die Integrität der Vermittlung respektiert wurde, die SZL selbst signieren und sie schließlich weitervermitteln. In diesem Szenario braucht er selbst nichts, über die Zertifikate zu wissen. Ein Beispiel davon ist der Attributserver in meinem Protokoll.
Sobald jedoch auf ein Zertifikat der SZL zugegriffen wird, werden alle Zertifikate überprüft.

Mit der SZL sind jetzt alle Stücke bereit, die zwischen Teilnehmern an das Authentisierungsprotokoll vermittelt werden.

5.5 Schichten 4 und 5: Authentisierungsprotokoll in JacORB

Weil Schichten 4 und 5 eng kooperieren, werden sie in diesem Unterkapitel gemeinsam vorgestellt.

Im Vergleich mit den letzten Schichten sind diese Schichten sehr dünn. Sie enthalten Klassen, die die Teilnehmer des Protokolls (siehe Abbildung 18) darstellen. Zwei davon sind Teil des ORBs (*Principal Authenticator* und *Current*) während drei andere zur Schicht 4 gehören: `AttributeProvider`, `AttributeServer` und `Verifier`.

5.5.1 Attributanbieter und -server

Wie schon erwähnt, teilt `AttributeServer` Anfragen nach Credentials an die Attributanbieter aus und setzt ihre Antworten zusammen.
Die abstrakte Klasse `AttributeProvider` definiert eine Methode, die einen Schlüssel annimmt und (möglicherweise leere) Credentials zurückgibt. Wie ein Anbieter diese Credentials erzeugt, ist von seiner Implementierung abhängig. Als Beispiel habe ich eine Klasse `IDNumberProvider` implementiert. Ein solcher Anbieter verhält sich wie folgt. Er behält eine Hashtabelle von Schlüssel-Ganzzahl-Paaren. Wenn eine Anfrage über einen Schlüssel kommt, der sich in der Hashtabelle befindet, dann gibt er die verbundene Zahl zurück. Wenn der Schlüssel noch nicht in der Tabelle ist, dann wird sie darin zusammen mit einer noch nicht gegebenen Zahl gespeichert und diese Zahl zurückgegeben.
Sowohl `AttributeProvider` als auch `AttributeServer` sind CORBA-fähige Objekte. Im heutigen Stand meiner Implementierung läuft ein einziger Prozeß, der ein `AttributeServer`-Objekt und mehrere `AttributeProvider`-Objekte enthält. Aber diese Konfiguration könnte sehr einfach geändert werden, um z.B. separaten Prozesse zu haben, die `AttributeProvider` Objekte enthalten.
Außerdem verfügen beide Klassen über ein GUI, das erlaubt, den Inhalt der Attributanbieter und die Anfragen an den Attributserver anzuschauen. Die Klassen, die dieses GUI implementieren, sind im Package `spki.auth.gui` zu finden. Da sie noch in Entwicklung waren, als dieses Bericht vollendet wurde, werden sie hier nicht illustriert und auch nicht im Anhang C dokumentiert.

5.5.2 Komponenten des ORBs

Um die Implementierung dieser Komponenten zu erklären, wird das Detail eines Aufrufs, wie der in Abbildung 17, aus der Sicht des ORBs beschrieben.

Zuerst wird der *Principal Authenticator* des Klienten mit dem Schlüsselpaar dieses Klienten initialisiert. Dann wird der ORB auf ihm `authenticate()` aufrufen. Um diesen Befehl durchzuführen, sendet der *Principal Authenticator* dem Attributserver, den er durch seine Konfiguration kennt, seinen öffentlichen Schlüssel. Als Rückgabewert erhält er die Credentials des Klienten.
Danach bekommt der ORB diese Credentials und fügt sie in *Current* ein.
Dann erzeugt er ein *Request* Objekt, das den Aufruf selbst darstellt und dessen Struktur Abbildung 25 zeigt.

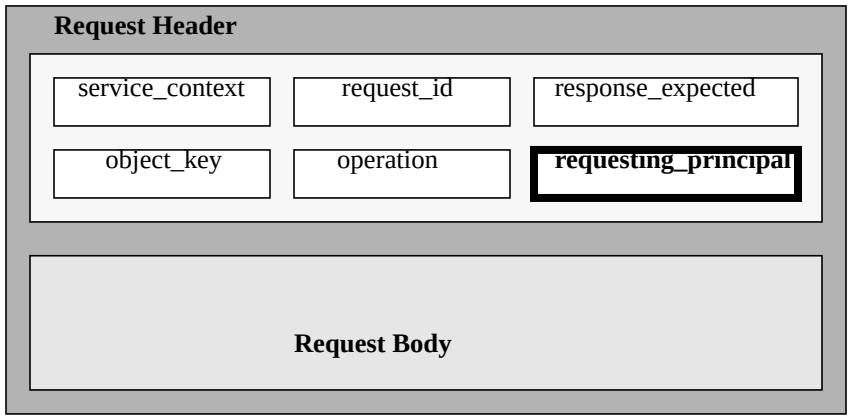


Abbildung 25. Struktur eines *Request* Objekts

Das Feld *requesting_principal* enthält eigentlich die Credentials, die deswegen mit dem Aufruf vermittelt werden.
Wie Abbildung 9 zeigt, verarbeitet ein Interceptor den ankommenden *Request* auf der Serverseite. Er verfügt außerdem über ein `Verifier`-Objekt.

Es ist zu bemerken, daß diese Weise, Credentials zu vermitteln ist nicht diese, die von der CORBA spezifiziert wird, sondern nur ein Trick, um es möglich machen, meine Implementierung zu testen.

5.5.3 Verifier

Ein `Verifier` wird mit einer Liste von Schlüsseln initialisiert. Danach wird er jeden dieser Schlüssel als einen vertrauenswürdigen Attributanbieter betrachten.
Seine Aufgabe ist es, die Credentials zu empfangen, ihre Signaturen zu überprüfen (durch Methoden der vermittelten Objekte selbst), die Schlüssel der Aussteller mit der Liste von Attributanbietern zu vergleichen und infolgedessen eine Aktion durchzuführen.
Was diese Aktion ist, kann vom Programmierer durch die Ableitung von `spki.auth.Verifier` beschlossen werden. Als Beispiel habe ich `Verifier` so programmiert, daß er eine erfolgreiche Authentisierung einfach protokolliert.

Mit `Verifier` kam das letzte Stück, das das Authentisierungsprotokoll vollständig implementiert. Von diesem Zeitpunkt an können JacORB-Anwendungen sich authentisieren.

5.6 Anwendung: Zugriffsschutz des Namensservers

Als Anwendung des Authentisierungsprotokolls habe ich einen Zugriffsschutz des Namensdienstes von JacORB implementiert.
Der Namensdienst ist ein kritischer Punkt in einem verteilten System, weil er Objekten erlaubt, die Referenz der Objekte, mit denen sie kommunizieren wolle, zu finden. Folglich erlaubt er, eine verteilte Menge von Objekte als System zu arbeiten. Wenn ein Angreifer den Namensdienst kontrollieren könnte, dann könnte er alle Objekte im System verwirren: er würde falsche oder gar keine Objektereferenzen bei einer Anfrage zurückgeben. So würde die anderen Objekte daran gehindert, zu kommunizieren, weil sie die nötige Referenzen nicht besitzen würden.
Deshalb ist es notwendig, den Namensdienst zu schützen.

Rahmen

Der Namensdienst erlaubt nur gewissen Klienten, bestimmte Operationen durchzuführen. Diese Sicherheitspolitik ist in einer ACL kodiert, wobei sie Rechte mit Attributen verbindet. Von den Schlüsseln seiner Klienten kennt der Namensdienst nichts im voraus. Aber er besitzt eine Liste von Schlüsseln der Attributanbieter, denen er vertraut, Attribute zu vergeben. Folglich braucht der Namensdienst authentisierte Attribute seiner Klienten, um eine Zugriffskontrollentscheidung über sie treffen zu können.

Implementierung

Beim Server (in diesem Fall der Namensdienst) wurde die Klasse `jacorb.Orb.Security.ACL` geändert, so daß sie sich als eine Tabelle von *AccessId*-Recht-Paare aus einer Datei erzeugen lässt. `jacorb.Orb.Security.AccessControlInterceptor` habe ich auch geändert, um darin die neue ACL und einen `spki.auth.Verifier` zu integrieren.
Aus Vereinfachungsgründen gibt es im System nur einen Attributanbieter, dem der Verifier vertraut und der Attributen des Typs *AccessId* vergibt. Die gleiche *AccessId* befinden sich in der ACL des Namensdienstes.
Zweitens braucht der Namensdienst auch Klienten, in diesem Fall Prozesse, die nur dazu geeignet sind, beim Namensdienst anzufragen.

Benutzerschnittstellen

Klienten werden aus einem GUI gesteuert, wobei der menschliche Benutzer entscheiden kann, mit welcher Operation und mit welchen Parametern er den Namensdienst aufrufen will. Der Namensdienst selbst verfügt auch über ein GUI, das nicht von mir implementiert wurde und das erlaubt, den Inhalt seiner Verzeichnissen anzuschauen. Der Verifier selbst protokolliert seine Entscheidungen in einer Datei oder auf dem Standard-Output.
Da die GUIs im Moment der Abfassung dieses Berichts noch in Entwicklung waren, werden sie hier nicht illustriert. Aber GUIs zu implementieren ist nicht die einzige Richtung, in der meine gesamte Arbeit erweitert werden kann.

6.0 Schluß

6.0.1 Ausblick

Meine Arbeit kann in mindestens zwei Richtungen fortgesetzt werden: Weiterimplementierung des Sicherheitsdienstes und Organisation der Attributanbieter.

Die erste zukünftige Arbeit ist die Verbesserung meiner Implementierung durch die Unterstützung von CRLs. Im Moment sind alle Zertifikate zeitlich begrenzt. Die zweite ist ihre Erweiterung zur Autorisierung. Dann würde das Anwendungsgebiet von SPKI maximal ausgenutzt: Credentials würden beliebige Erlaubnisse tragen und ACL in Interceptors könnten selbst als *acl* S-Expressions ausgedrückt werden. Weil das Protokoll, um Credentials zu holen, schon existiert, und weil die ganze Funktionalität, Zertifikate zu serialisieren, schon implementiert ist, würde diese Aufgabe kein großes Problem auf der Seite der Darstellung der Erlaubniszertifikatobjekte bedeuten. Aber die Austeilung von Zertifikaten sollte dann genauer überlegt werden, weil der Attributanbieter nicht mehr einzig ist, wie in unserem Protokoll, sondern jeder Server stellt selbst Zertifikate aus, die seine Sicherheitspolitik veröffentlichen.
Ein solches Problem weist auf die zweite Richtung einer Fortsetzung meiner Diplomarbeit hin: die Organisation der Attributanbieter.

Z.B. kann man in einem weit verteilten System nicht nur einen Attributserver haben. Wenn es mehrere von ihnen gibt, müssen sie dann die gleichen Attribute austeilen?
Selbst wenn nur Authentisierung unterstützt wird, wie weiß ein Principal, daß er einem bestimmten Principal als Attributanbieter vertrauen soll und nicht einem anderen? Weil dieser Attributanbieter Teil einer Hierarchie wie X.509 ist? Weil solche grundsätzlichen digitalen Vertrauensbeziehungen sich immer auf eine Verbindlichkeit in der “echten“ Welt stützen haben muß? Weil andere Principals haben den Attributanbieter für eine gewisse Zeit gewählt? (SPKI kann auch diese Art von Entscheidungen durch *subject-threshold* S-Expressions unterstützen).
Solche Fragen haben wahrscheinlich keine allgemeine Antwort. Die Aufgabe ist dann, Typen von Anwendungen zu identifizieren, für die eine bestimmte Organisation von Privilegienausstellern am besten paßt.

Eine dritte zukünftige Arbeit ist kein persönliches Unternehmen sondern eine Überzeugungsarbeit, deren Ziel die Verbesserung der SPKI Grammatik ist. Das erforderliche *Lookahead* kann reduziert werden, *cert* und *name-cert* SPKI-Objekte sollten irgendwie verbunden sein, weil sie ähnlich sind, usw.

6.0.2 Zusammenfassung

Das Ziel meiner Diplomarbeit war, die Authentisierung als Teil des Sicherheitsdienstes von JacORB mit SPKI Zertifikaten zu implementieren.
Deswegen habe ich ein einfaches Protokoll entwickelt, das einem Klienten des ORBs erlaubt, seine authentifizierte Attribute (Credentials) von einem Server, der sie zentralisiert, zu holen. Diese Attribute werden von einem vertrauenswürdigen Dritten ausgestellt, dem auch Serverobjekte vertrauen. Ich habe dieses Protokoll implementiert und, damit verbunden, eine Bibliothek von Klassen, die SPKI-Zertifikate in einem standardisierten Format serialisieren.

Diese Arbeit hat bewiesen, daß CORBA-Authentisierung mit Zertifikaten möglich ist, teilweise dank der Flexibilität und der Aussagekraft der SPKI-Objekte. Sie hat aber auch gezeigt, daß die SPKI-Grammatik zu parsieren keine triviale Aufgabe ist und daß einige Änderungen dieser Grammatik das Parsing beträchtlich vereinfachen könnten.

6.0.3 Danksagungen

Diese Diplomarbeit entstand in der Zeit vom November 1998 bis Februar 1999. Ich möchte mich bei meinem Betreuer Gerald Brose für seine Betreuungsarbeit und besonders seine ständige Verfügbarkeit bedanken. Mein Dank geht auch an Sebastian Staamann (EPFL-LSE), der es ermöglicht hat, daß ich meine Diplomarbeit in der Freien Universität Berlin durchführe.

Literaturverzeichnis

[1] A. Aho, R. Sethi, J.D. Ullmann, Compilers, Addison-Wesley, 1986

[2] M. Blaze, J. Feigenbaum, J. Lacy, *Decentralized Trust Management*, IEEE Symposium on Security and Privacy, 1996, S. 164-173

[3] Gerald Brose, *A View-Based Access Control Model for CORBA*, in: Jan Vitek, Christian Jensen (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Springer LNCS, 1999.

[4] Gerald Brose, *Java & CORBA - How close are they really ?* , Java Developers' Journal, Vol. 3, No. 1, January 1998.

[5] G. Brose, B. Bokowski, *Ein Object Request Broker für Java*, Informatik/Informatik, Zeitschrift der schweizerischen Informatikorganisationen, No. 3/97, 27-30

[6] CCITT, *Recommandation X.509 - The Directory Authentication Framework*, 1988

[7] C. M. Ellison, *Certification Infrastructure Needs For Electronic Commerce And Personal Use*, http://www.clark.net/pub/cme/nist-7-24/, Juli 1997

[8] C. M. Ellison, *What do you need to know about the person with whom you are doing business ?*, House Science and Technology Subcommitte, Hearing of 28 October 1997, http://www.clark.net/pub/cme/html/congress1.html

[9] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, T. Ylonen, *SPKI Certificate Theory*, Internet draft, http://www.clark.net/pub/cme/theory.txt, Oktober 1998

[10] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, T. Ylonen, *Simple Public Key Certificate*, Internet draft, http://www.clark.net/pub/cme/spki.txt, März 1998

[11] W. Ford, *Computer Communications Security*, Prentice Hall, 1994, S. 76-84

[12] A. Freier, P. Karlton, P. Kocher, *The SSL Protocol, Version 3.0*, Internet Draft, March 1996, http://home.netscape.com/eng/ssl3/ssl-toc.html

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[14] S. Garfinkel, *PGP : Pretty Good Privacy*, O'Reilly & Associates, 1995

[15] S. Garfinkel, G. Spafford, *Web Security and Commerce*, O'Reilly & Associates, 1997

[16] M. Hahsler, *X.509 v3, h*ttp://olymp.wu-wien.ac.at/usr/h92/h9251277/seminar/X509.htm, 1997

[17] Network Working Group, *RFC 1422 : Privacy for Internet Electronic Mail*, Part II Februar 1993

[18] Object Management Group, *CORBA services: Common Object Services Specification*, Updated: December 1998

[19] R. Orfali, D. Harkey, J. Edwards, *Instant CORBA*, Wiley, 1997, Kapitel 11

[20] B. Schneier, *Applied Cryptography*, Wiley, 2nd Ed., 1996

[21] Y. Yang, L. Brown, J. Newmarch, *Which One is for You : Issues of Trust in Public Key Certificates*, http://www.adfa.edu.au/~lpb/papers/auug98/trustmodels.html, August 1998

URLs von Ressourcen

[22] CORBA : http://www.omg.org/corba

[23] CORBA Implementierungen: http://adams.patriot.net/~tvalesky/freecorba.html

[24] Cryptix: http://www.systemics.com/software/cryptix-java/index.html

[25] JacORB : http://www.inf.fu-berlin.de/~brose/jacorb

[26] Java 2: http://www.javasoft.com/products/jdk/1.2/

[27] JavaCC : http://www.suntest.com/JavaCC/

[28] JTB: http://www.cs.purdue.edu/homes/taokr/jtb/index.html

[29] OMA : http://www.omg.org/oma/

[30] OMG : http://www.omg.org

[31] SPKI Mailing List Archive : http://www.sandelman.ottawa.on.ca/spki/

Anhang A: SPKI Grammatikregeln

Quelle: [10]

<5-tuple>:: <issuer5> <subject5> <deleg5> <tag-body5> <valid5> ;

<acl-entry>:: "(" "entry" <subj-obj> <deleg>? <tag> <valid>?
<comment>? ")" ;

<acl>:: "(" "acl" <version>? <acl-entry>* ")" ;

<byte-string>:: <bytes> | <display-type> <bytes> ;

<bytes>:: <decimal> ":" {binary byte string of that length} ;

<cert-display>:: "(" "display" <byte-string> ")" ;

<cert>:: "(" "cert" <version>? <cert-display>? <issuer> <issuer-loc>?
<subject> <subject-loc>? <deleg>? <tag> <valid>? <comment>? ")" ;

<comment>:: "(" "comment" <byte-string> ")" ;

<crl>:: "(" "crl" <version>? <hash-list> <valid-basic> ")" ;

<date>:: <byte-string> ;

<ddigit>:: "0" | <nzddigit> ;

<decimal>:: <nzddigit> <ddigit>* | "0" ;

<deleg5>:: "t" | "f" ;

<deleg>:: "(" "propagate" ")" ;

<delta-crl>:: "(" "delta-crl" <version>? <hash-of-crl> <hash-list>
<valid-basic> ")" ;

<display-type>:: "[" <bytes> "]" ;

<fq-name>:: "(" "name" <principal> <names> ")" ;

<general-op>:: "(" "do" <byte-string> <s-part>* ")" ;

<gte>:: "g" | "ge" ;

<hash-alg-name>:: "md5" | "sha1" | <uri> ;

<hash-list>:: "(" "canceled" <hash>* ")" ;

<hash-of-crl>:: <hash> ;

<hash-of-key>:: <hash> ;

<hash-op>:: "(" "do" "hash" <hash-alg-name> ")" ;

<hash-value>:: <byte-string> ;

<hash>:: "(" "hash" <hash-alg-name> <hash-value> <uris> ")" ;

<issuer-loc>:: "(" "issuer-info" <uris> ")" ;

<issuer-name>:: "(" "issuer" "(" "name" <principal> <byte-string> ")" ")" ;

<issuer5>:: <key5> | "self" ;

<issuer>:: "(" "issuer" <principal> ")" ;

<k-val>:: <byte-string> ;

<key5>:: <pub-key> ;

<keyholder-obj>:: <principal> | <name> ;

<keyholder>:: "(" "keyholder" <keyholder-obj> ")" ;

<low-lim>:: <gte> <byte-string> ;

<lte>:: "l" | "le" ;

<n-val>:: <byte-string> ;

<name-cert>:: "(" "cert" <version>? <cert-display>? <issuer-name>
<subject> <valid> <comment>? ")" ;

<name>:: <relative-name> | <fq-name> ;

<names>:: <byte-string>+ ;

<not-after>:: "(" "not-after" <date> ")" ;

<not-before>:: "(" "not-before" <date> ")" ;

<nzddigit>:: "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

<obj-hash>:: "(" "object-hash" <hash> ")" ;

<one-valid>:: "(" "one-time" <byte-string> ")" ;

<online-test>:: "(" "online" <online-type> <uris> <principal> <s-part>* ")" ;

<online-type>:: "crl" | "reval" | "one-time" ;

<op>:: <hash-op> | <general-op> ;

<principal>:: <pub-key> | <hash-of-key> ;

<pub-key>:: "(" "public-key" <pub-sig-alg-id> <s-expr>* <uris> ")" ;

<pub-sig-alg-id>:: "rsa-pkcs1-md5" | "rsa-pkcs1-sha1" | "rsa-pkcs1" | "dsa-sha1" | <uri> ;

<range-ordering>:: "alpha" | "numeric" | "time" | "binary" | "date" ;

<relative-name>:: "(" "name" <names> ")" ;

<reval-body>:: <one-valid> | <valid-basic> ;

<reval>:: "(" "reval" <version>? <subj-hash> <reval-body> ")"

<s-expr>:: "(" <byte-string> <s-part>* ")" ;

<s-part>:: <byte-string> | <s-expr> ;

<seq-ent>:: <cert> | <name-cert> | <pub-key> | <signature> | <op> |

<reval> | <crl> | <delta-crl> ;

<sequence>:: "(" "sequence" <seq-ent>* ")" ;

<sig-val>:: <s-part> ;

<signature>:: "(" "signature" <hash> <principal> <sig-val> ")" ;

<simple-tag>:: "(" <byte-string> <tag-expr>* ")" ;

<subj-hash>:: "(" "cert" <hash> ")" ;

<subj-obj>:: <principal> | <name> | <obj-hash> | <keyholder> | <subj-thresh> ;

<subj-thresh>:: "(" "k-of-n" <k-val> <n-val> <subj-obj>* ")" ;

<subject-loc>:: "(" "subject-info" <uris> ")" ;

<subject5>:: <key5> | <fq-name5> | <obj-hash> | <keyholder> | <subj-thresh> ;

<subject>:: "(" "subject" <subj-obj> ")" ;

<tag-body5>:: <tag-expr> | "null" ;

<tag-expr>:: <simple-tag> | <tag-set> | <tag-string> ;

<tag-prefix>:: "(" "*" "prefix" <byte-string> ")" ;

<tag-range>:: "(" "*" "range" <range-ordering> <low-lim>? <up-lim>? ")" ;

<tag-set>:: "(" "*" "set" <tag-expr>* ")" ;

<tag-star>:: "(" "tag" "(*)" ")" ;

<tag-string>:: <byte-string> | <tag-range> | <tag-prefix> ;

<tag>:: <tag-star> | "(" "tag" <tag-expr> ")" ;

<up-lim>:: <lte> <byte-string> ;

<uri>:: <byte-string> ;

<uris>:: "(" "uri" <uri>* ")" ;

<valid-basic>:: <not-before>? <not-after>? ;

<valid5>:: <valid-basic> | "null" | "now" ;

<valid>:: <valid-basic> <online-test>* ;

<version>:: "(" "version" <byte-string> ")" ;

Anhang B: Objektmodelle

Alle Modelle verwendet die *Unified Modelling Language* (UML)-Notation.

Anhang C: Dokumentation meines Codes als Javadoc Dateien