Hauptseminar: Mobile Agenten
— am Beispiel des Agentensystems Tacoma —

Stefan Schäckeler

28. Januar 1997

Inhaltsverzeichnis

1 Motivation für Agentensysteme

Modelle um verteilte Anwendungen zu schreiben, gibt es wie Sand am Meer: C/S, RPC, Shared Memory, Gruppenkommunikation ... Vor ein paar Jahren kamen dann *Agentensysteme* hinzu. Ob Agentensysteme das Allheilmittel der Informatik sind, wage ich zu bezweifeln, aber zumindest sind sie ein interessanter Ansatz.

Agenten sind an sich nichts anderes als Prozesse, also

- Code
- Daten
- Zustand

die im Rahmen einer verteilten Anwendung sich in Netzen bewegen können. Der Unterschied zur normalen Prozeßmigration in verteilten Systemen ist, daß nicht das Betriebssystem die *Politik* (wann, welcher Prozeß wohin migriert) bestimmt, sondern der Agent selbst. Die Aufgabe des Agentensystems ist es, den *Mechanismus* (Mirgrationsverfahren) bereitzustellen.

Manche Agenten sind an einen bestimmten Ort gebunden und bieten Dienste an andere Agenten an. Zur Unterscheidung zu *mobilen Agenten* heißen diese *Service Agenten*. Im allgemeinen kann aber diese Unterscheidung unterbleiben.

Es gibt zwei Kriterien, die ein Agentensystem erfüllen muß:

1. Agenten müssen migrieren können
2. Agenten müssen sich lokal unterhalten können

Natürlich ist das nur eine Minimalforderung, weitere wünschenswerte Eigenschaften wären

- Sicherheit durch digitale Unterschriften und Verschlüsselung
- Fehlerbehandlung durch z.B. Transationen um Abstürze zu überleben
- Kommunikationskonzepte die sich der Problemstellung anpassen
- Programmiersprachenunterstützung, so das für jeden etwas dabei ist
- Dienstevermittlung

Vorteile von Agentensystemen

- **(natürliche) Abstraktion:** die Kommunikationskanäle sind vor dem Anwendungsprogrammierer verborgen, nicht aber die Orte. Dies vereinfacht die Einarbeitungszeit für den Anfänger und stellt i.a. keine Einschränkung für den Fortgeschrittenen dar.
- **heterogene Umgebung:** der Anwendungsprogrammierer braucht sich nicht um verschiedene Rechnerarchitekturen zu kümmern, sondern nur um das eigentliche, nämlich seine Anwendung.
- **Mobilität:** die Berechnung kann auf einen angemessenen Platz[1] ausgeführt werden.
- **Kommunikationsvielfalt:** je nach dem, was die Modellierung der realen Welt verlangt, kann die Kommunikation synchron oder asynchron erfolgen.

[1] z.B. angemessene Peripherie/Hardware (Co-Prozessor, Crypto-Prozessor) oder entsprechender Service Agent

- **Realzeit Interation:** zwar nimmt die Bandbreite der Computernetze stetig zu, aber die Anfangsverzögerung (Latenzzeit) ist durch Naturgesetze festgelegt (→ Lichtgeschwindigkeit). Was auf LANs nicht unbedingt stört, ist auf WANs ärgerlich — insbesondere bei interaktiven Anwendungen bekommt das der Benutzer zu spüren. Abhilfe schaffen Agentensysteme: der Client migriert zum Server (oder umgekehrt).
- **einfache Fehlerbehandlung:** da der Client und der Server sich auf *einem* Knoten befinden, vereinfachen sich die meisten Fehlerbehandlungsalgorithmen.
- **Performancegewinn** durch Reduzierung der Netzlast (lokales „Trading“) (→ Abb. 1).

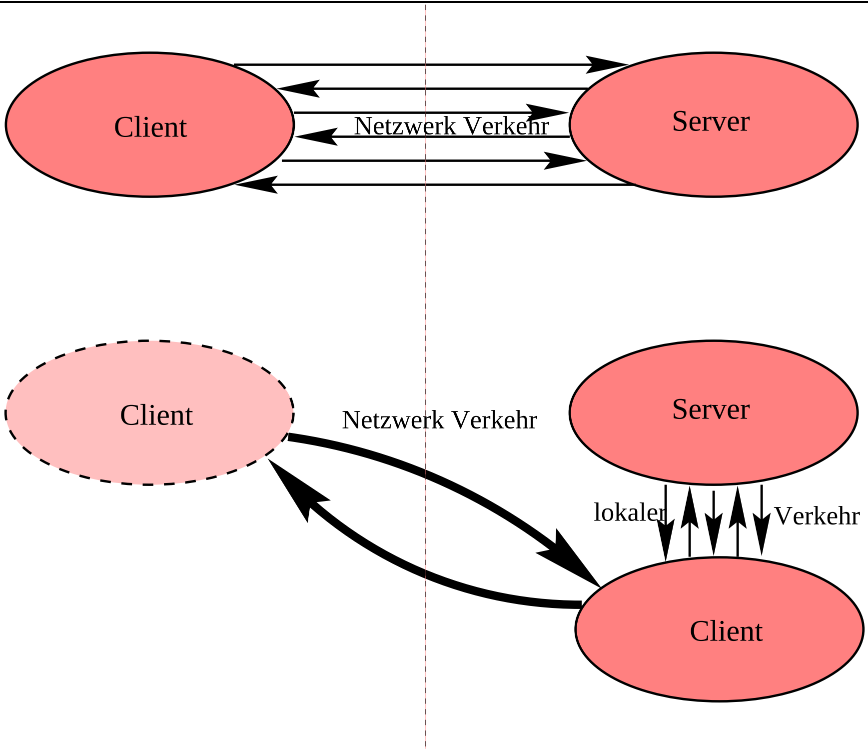


Abbildung 1: C/S Paradigma, Agenten Paradigma

Nachteile

- zur Zeit sind die verfügbaren Agentensysteme noch nicht ausgereift: es mangelt oft an Authentifikations- und Sicherheitsmaßnahmen. Es ist wohl nur noch eine Frage der Zeit, bis die ersten ausgereiften Agentensysteme verfügbar sind
- es gibt keine zentrale Instanz, die außer Kontrolle geratene Agenten wieder zur *Raison* bringt
- es gibt so gut wie keine Kommunikationsmöglichkeit unter Agentensystemen unterschiedlicher Hersteller[2]

Abschließend sei noch erwähnt: es gibt kein Problem, das zur Lösung unbedingt Agentensysteme erforderlich macht, es gibt gewöhnlich immer eine alternative konventionelle Lösung. Die oben erwähnten Vorteile von Agentensystemen sollten aber eine große Motivation *für* Agentensysteme sein.

[2] möglich wäre eine Kommunikation via Filesystem oder via IPC des Betriebssystems

2 Das Tacoma-Projekt

Das Tacoma[3] Projekt [JvRS95a] wurde 1993 von den Universitäten Tromsø und Cornell ins Leben gerufen. Tacoma steckt heute (Anfang 1997) noch in den Kinderschuhen, denn es fehlen ihm noch viele wichtige Features, aber es wird momentan viel implementiert und evaluiert. Wir können also gespannt sein was die Zukunft bringt.

3 Überblick über Tacoma

3.1 Daten: Briefcases, Cabinets and all that

Agenten müssen Daten manipulieren können. Sie müssen

1. Daten auf andere Orte mitnehmen können und
2. Daten auf den einzelnen Orten lesen und schreiben können,

um abhängig von ihnen Berechnungen durchführen zu können.

Datenhierachie

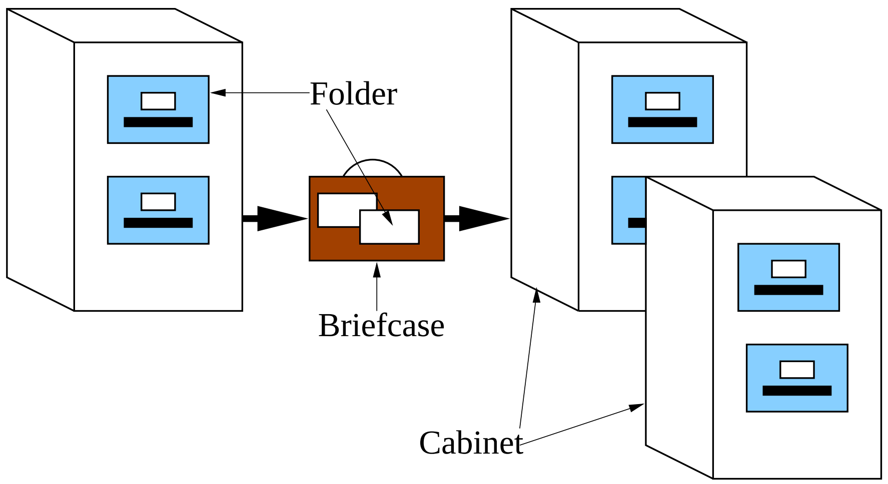


Abbildung 2: Datenhierachie

Folder: Auf unterer Ebene werden Daten in Folder abgelegt. In neueren Versionen von Tacoma sind Daten eines Folders als Listen strukturiert.

Cabinet: Folder, die an einen Ort abgelegt werden, werden in Cabinets abegelegt. Implementiert werden Cabinets als Verzeichnisse und die darin enthaltenen Folder als Dateien. Cabinets überleben somit Abstürze. Aus diesem Grund sollte man Agenten auch sauber programmieren: terminiert ein Agent, so werden seine ggf. temporär angelegten Cabinets nicht automatisch gelöscht.

Briefcase: Folder, die an einen entfernten Agenten übergeben werden sollen, werden in ein Briefcase gepackt. Briefcases sind eine Datenstruktur im Hauptspeicher und überleben somit keine Abstürze. Nur Briefcases können auf andere Orte gebracht werden.

[3] Tromsø and Cornell Moving Agents

Auf Folder gibt es drei allgemeine Operationen[4]:

- `store`
- `fetch`
- `delete`

Auf die Listenelemente eines Folders gibt es u.a. die vier folgenden Operationen:

- `push/top`
- `pop/add`

Die Strukturierung der Folder in Listen unterstützt diverse alltägliche Konstrukte. In das `HOST`-Folder eines Briefcase werden alle zu besuchenden Rechner aufge*list*et. Als Zielrechner betrachtet TACOMA immer den ersten Rechnernamen im `HOST`-Folder. Am Zielrechner angekommen, entfernt TACOMA automatisch den ersten `HOST`-Folder Eintrag. Schickt man des Briefcase weiter, so wird es also an den zweiten Rechner geschickt, ad infinitum. Agenten die einen festgelegten Weg nehmen, wären z.B. Systeminformationssammelagenten oder Softwareinstallationsmanagementagenten.

Desweiteren gibt es noch Befehle um ein Briefcase oder ein Cabinet zu erzeugen oder zu löschen.

Anzumerken ist vielleicht noch, daß Operationen auf Cabinets und ihre Folder leider nicht atomar sind[5].

Verschicken von Briefcases

Briefcases müssen zum Verschicken verpackt[6] werden. Das Format soll zurVereinfachung an einem Beispiel erklärt werden:

Das Briefcase

```
my_bc
   HOST : host1 host2 host3
   DATA : data1 data2
```

wird automatisch (via `archive` und anschliessendem Anhängen von `0D040D`) in folgende Form gebracht:

```
HOST{{host1}{host2}{host3}}DATA{{data1}{data2}}0D040D
```

Natürlich müssen noch bestimmte Bytes *escaped* werden.

3.2 `meet` — der Treffpunkt

Die (leider) einzige Möglichkeit der Kommunikation unter Agenten ist das Versenden von Briefcases via `meet`.

```
Agent_A: meet <Agent_B> <Briefcase>
```

Agent_B ist ein entfernter Agent, der noch nicht aktiviert ist, d.h. es liegt nocht auf Platte! Zwei lokale Agenten können sich also nicht unterhalten! — man kann das höchstens über Cabinets realisieren, die als eine art *shared memory* dienen! Nach meiner Definiton von Agentensystemen ist TACOMA also gar kein Agentensystem!

Dem `meet` Konstrukt können noch ein paar *Flags* übergeben werden, so daß `meet` synchron oder asynchron, entfernt oder lokal ausgeführt werden kann.

[4] beim ersten Zugriff auf ein Folder wird es automatisch angelegt
[5] wahrscheinlich sind das UNIX-Aufrufe — die UNIX Dateisemantik ist ja nicht transaktionsorientiert
[6] marshalling

3.3 Kommunikation in Tacoma

Vergleicht man die Semantik von verteilten Programmiersprachen wie *SR* mit der von TACOMA stellt man fest, daß es in TACOMA nichts Vergleichbares zu `in/ni` gibt (→ Tab. 1 & Abb. 3).

Effekt	SR		TACOMA	
	Aufruf	Service	Aufruf	Service
Remote Procedure Call	`call`	`proc`	`meet sync`	`proc`
Rendezvous	`call`	`in`	`meet sync`	—
dynamische Prozeßerzeugung	`send`	`proc`	`meet`	`proc`
asynchrone Nachrichten	`send`	`in`	`meet`	—

Tabelle 1: Vergleich der IPC Mechanismen von *SR* und TACOMA

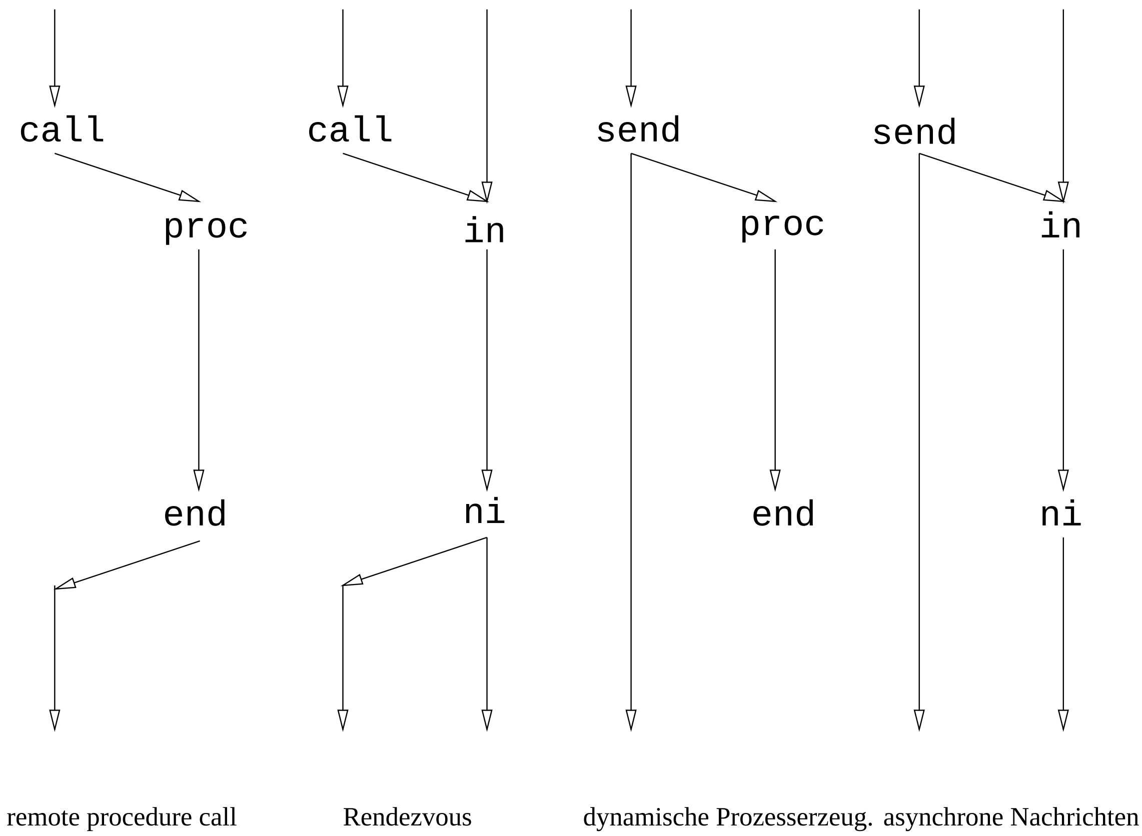


Abbildung 3: Interaktionsmuster in SR

Glücklicherweise wird zur Zeit gerade an Rendezvous gearbeitet, so daß TACOMA bald ein universal einsetzbares Agentensystem wird. Übergeben werden dann via Rendezvous Briefcases.

3.4 Migration

Wie kann ein Agent nun migrieren, wenn er nur Briefcases verschicken kann? Da das eine wichtige Frage ist, die erst später umfassend geklärt wird, hier die Kurzfassung: ein Agent verschickt ein

Briefcase, in dem ein Folder Code, also einen weiteren Agenten, enthält, an einen bestimmten entfernten Agenten. Dieser Agent entpackt das Briefcase und führt es aus. Eigentlich ganz einfach!

3.5 Bridgehead: der „Brückenkopf"

Der Prozeß, der jedes Briefcase am Zielrechner empfängt, heißt Firewall[7]. Die Firewall ruft (je nach Agentensystem) weitere Agenten auf, die sich um Sicherheit und Fehlertoleranz kümmern. Zusammengenommen bilden sie den „Brückenkopf". Im Gegensatz dazu gibt es auf den Rechnern noch Service Agenten (vgl. Kapitel 4.1).

Die Semantik von `meet` soll aus Platzgründen nur am synchronen Treffen erläutert werden:

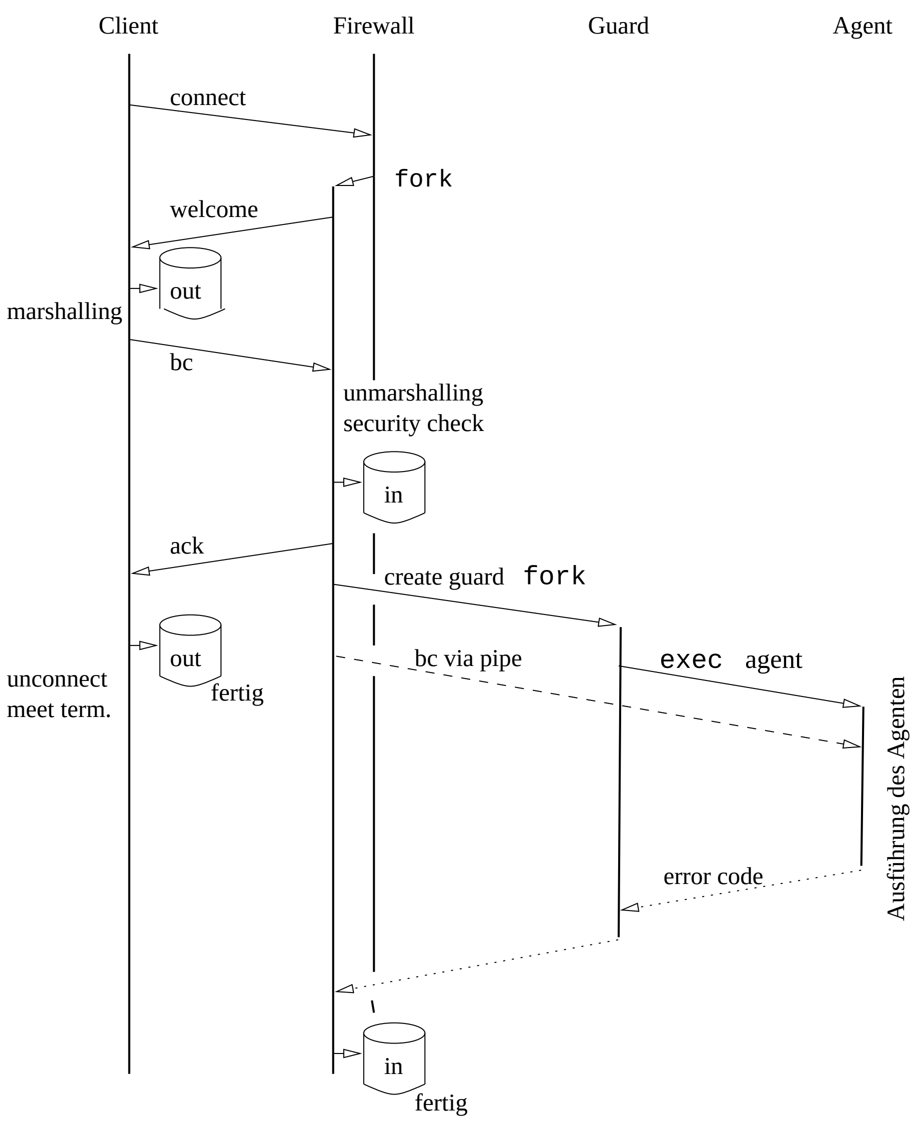


Abbildung 4: interne Abläufe in TAOCMA

Das meiste geht aus Abb. 4 hervor. Die genaue Funktion des Wächters und der Logdateien wird noch in Kapitel 4.3 erklärt.

[7] TACOMA: tac_firewall — nur genau dieser Prozeß muß übrigens auf jedem Rechner gestartet werden

Zu erwähnen ist vielleicht noch, daß der `meet` Befehl das Folder `HOST` im ihm übergebenen Briefcase erwartet. An ihn wird das Briefcase via *TCP/IP* versendet. Der an die `meet` Anweisung übergebene Zielagentenname wird zuvor heimlich in das Folder `CONTACT` geschrieben.

Hier noch etwas für Nostaligiker: Frühere Versionen von Tacoma implementierten `meet` als nur eine einzige Zeile Code:

```
cat briefcase | rsh targethost /usr/local/agent/targetagent
```

Das schaft wohl kein zweites Agentensystem.

4 Tacoma Goodies

Neben der im vorigen Kapitel besprochenen Kernfunktionaliät bietet Tacoma weitere Features, die in den folgenden Unterkapiteln besprochen werden.

4.1 Service Agenten

Hier sollen primär nur *Code Service Agenten* besprochen werden, weil ohne sie keine Agentenmigration möglich wäre — sie sind in Tacoma lebenswichtiger Bestandteil des Agentensystems!

4.1.1 Code Service Agenten für interpretierte Sprachen

Code Service Agenten sind für die Ausführung von Code zuständig, den sie über Briefcases von anderen Agenten bekommen. Der Code Service Agent extrahiert den Code der im `CODE`-Folder[8] angegebenen Folder.

Ein simpler migrierender Agent wäre z.B.

```
bc_create my_bc
   folder_store my_bc HOST utopia
   folder_store my_bc DATA data1 data2
   folder_store my_bc MYCODE "proc main {bc} {
                              upvar bc loc_bc;
                              exec echo \"Lunch now!\" > /dev/console
                              return 0
                           }"
   folder_store my_bc TCLCODE MYCODE
meet ag_tcl my_bc
```

Beispiel 1: ein einfacher Agent

In dem migrierenden Briefcase gibt es i.a. noch benutzerdefinierte Folder, also Datenfolder (oben: `DATA`). Damit der migrierte Agent nun darauf zugreifen kann bekommt er das ganze Briefcase als Referenzparameter übergeben: daher die ersten zwei Zeilen des `MYCODE`-Folder.

Der Agent `ag_tcl` sieht damit (verkürzt) folgendermaßen aus:

```
### Prozeduren der Benutergeschriebenen Agenten deklarieren
foreach folder [folder_fetch bc TCLCODE] {eval [folder_fetch bc $folder]}

### Ausfuehren des Codes, d.h. starten der main-Prozedur
set returncode [main bc]
```

Beispiel 2: `ag_tcl`

[8] der *Tcl*Agent bezieht sich auf das `TCLCODE`-Folder, der *C*Agent bezieht sich auf das `CCODE`-Folder, etc

4.1.2 Code Service Agenten für compilierte Sprachen

Hier müssen mehrere Schritte durchgeführt werden:

- `ag_c` schreibt alle im `CCODE`-Folder aufgelisteten Folder auf Platte (beim interpret. Ansatz wurde der Code direkt durch `eval` ausgeführt!)
- der *C*-Compiler wird aufgerufen
- der erzeugte Agent wird aufgerufen. Das Briefcase bekommt er durch Pipes

4.1.3 Weitere Service Agenten

Es gibt noch mehrere Service Agenten die sich beispielsweise um die Synchronisation der lokalen Agenten im begrenzten Umfang kümmern können. Auf die Besprechung wird aber verzichtet.

4.2 Authentifizierung und Krypthographie

Für die Authentifzierung in TACOMA ist `tac_firewall` höchstpersönlich zuständig.

Leider ist es nicht *entscheidbar*, ob ein Agent sicherheitsgefährdend ist, daher gibt es in TACOMA zwei (schwächere) Kriterien für das akzeptieren / ablehnen eines Briefcases:

- Authentifikation des Authors[9] eines Briefcases. Momentan geschieht dies via *PGP*[10].
- Um einen Performancegewinn zu erzielen, kann man in einem sicheren Teilnetz auf oben genannte Authentifikation verzichten, wenn der Absenderrechner als „zuverlässig“ gilt.

Die Funktionsweise des zweiten Punktes ist die, daß wenn B ein Briefcase bekommt, dann stammt es entweder von A, also einem „sicheren“ Rechner, oder es stammt von W, aber A hat es nach Methode 1 schon authentifiziert. Ähnliches gilt somit für C, und somit auch für D, ad infinitum ($\rightarrow$ Abb. 5).

Ein Agent läuf unter denselben Unixrechten (Benutzer, Gruppe) wie `tac_firewall` ab. Innerhalb diesen Schranken kann ein Agent alles machen, wenn er erstmal die obige Authentifikation passiert hat:

- Löschen von fremden Cabinets
- `while 1 do fork` bis interne Betriebssystemtabellen überlaufen
- `while 1 do malloc(1000000)` um den virtuellen Speicher aufzubrauchen
- Fremde Agenten via `kill -9` killen — ja es kann sogar das ganze Agentensystem ge*kill*t werden ($\rightarrow$ Beispielprogramm 3)

Diese Gefahren könnte man noch etwas einschränken:

- Einführung eines Code Service Agenten für sichere Sprachen, wie *SaveTcl*. Da es schon einen Code Service Agenten für *Tcl* gibt, sollte das ein triviales Problem sein.
- Ausnutzung der UNIX Sicherheitsmechanismen:

[9] die Person, die das Briefcase geschrieben hat, nicht die, die es losschickt!

[10] im Anhang werden kurz asymmetrische Verschlüsselungsverfahren angrissen

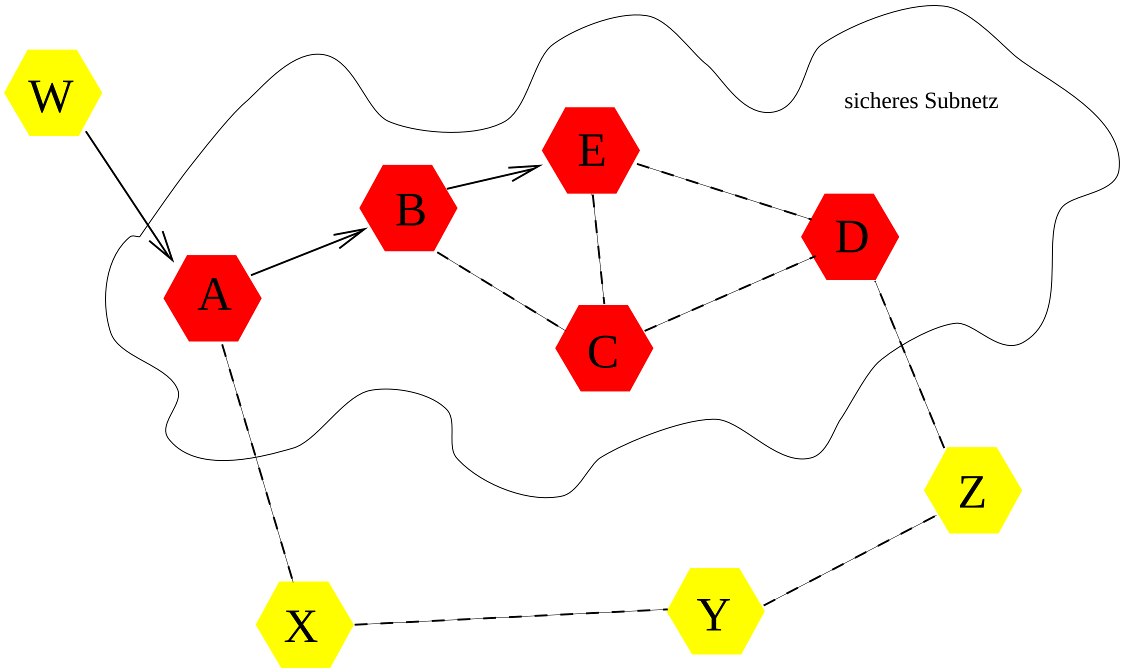


Abbildung 5: Authentifizierung in TACOMA

- in den noch unveröffentlichten Prototypen von *Tacoma* können die Agenten unter einer beliebigen *USER*-Id ausgeführt werden
- mit `chroot` könnte der `ag_c` nach dem Compilieren und noch vor dem Ausführen den Verzeichnisbaum einschränken

Leicht implementierbar wäre darüber hinaus:

- Zugriffskontrolle um nur bestimmte Serviceagenten aufzurufen
- Authentifikation des Senders eines Briefcases via *PGP*

```
bc_create my_bc
folder_store my_bc HOST utopia
folder_store my_bc my_code {
             proc main{xxx} {
             set psid [lindex [exec ps ax | grep tac_firewall | cut -c 1-5] 0];
             exec kill -9 $psid }}
meet ag_tcl my_bc
```

Beispiel 3: `kill-9`-Agent

Es ist momentan ein Krypthographieagent in Vorbereitung, der bei Bedarf von der Firewall aufgerufen werden kann.

4.3 Recovery

Recovery ist in der aktuellen *Tacoma*version wohl erst teilweise implementiert. Vorgestellt wird die angestrebte Implementierung.

Das zugrunde liegende Fehlermodell sieht zwei Fälle vor:

- Agent stürzt ab
  - Zugriff auf unerlaubte Resourcen
  - fehlerhaft programmiert
- Rechner stürzt ab

Durch geeignte Erweiterungen (insbesondere Time-Outs) könnte man auch mit Linkfehlern und Netzpartitionierungen zurecht kommen.

Agent stürzt ab

Der Agent wird durch einen Wächterprozeß gestartet. Der Wächter wertet den Returncode des Agenten aus. Stellt er einen Fehler fest, so hat er zwei Möglichkeiten darauf zu reagieren:

- stellt der Agent durch ein speziell ausgezeichnetes Folder seine eigene Recoveryprozedur bereit, so wird sie ausgeführt[11]
- ansonsten findet eine standardmäßige Reaktion statt
  - Bei einem synchronen Treffen wird ein Briefcase mit einem Errorfolder zurückgeliefert
  - Bei einem asynchronen Treffen wird eine Fehlermeldung auf der Console des Aufrufer ausgegeben

Rechner stürzt ab

Vorgestellt wird von mir ein modifizierter Algorithmus.

In Logging Dateien wird festgehalten, welcher Agent gerade aktiv ist. Ankommende Briefcases und deren Zielagentenname werden komplett in die Inlog Datei gedumpt, ausgehende Briefcases und deren Zielagentennamen werden komplett in die Outlog Datei gedumpt ($\rightarrow$ näheres Abbildung 6).

Nach dem Wiederhochfahren des Rechners kann die Firewall anhand des Inlogs nur feststellen, wo sie ungefähr vohrer war: entweder im Bereich 1 oder im Bereich 2. Stellt die Firewall fest, daß sie im Bereich 1 abgestürzt ist, so unternimmt sie nichts. Jetzt kann es sein, daß der Klient schon ein Briefcase losgeschickt hat und auf das *Acknowledge* wartet. Daher bekommt der Klient nach einer gewissen Zeit ein Time-Out. Der Klient entscheidet nach dem Time-Out, ob er es erneut mit dem selben oder mit einem anderen Server probiert.

Stellt die Firewall fest, daß sie im Bereich 2 abgestürzt ist, startet sie den Agenten erneut. Halt, was ist, wenn sie noch kein *Acknowledge* losgeschickt hat und beim Klienten der Timeout abgelaufen ist? Der Server muß sich also zuerst beim Klienten eine Erlaubnis zum Starten einholen. Der Klient entscheidet anhand seines Outlogs: Ist das Briefcase schon im Outlog als abgeschlossen makiert, so darf der Server den Agenten nicht wieder starten, ansonsten schon.

Was passiert, wenn ein Agent der auf dem Server einen `meet` ausgeführt hatte, neugestartet wird? Soll tatsächlich der `meet` wiederholt werden? Anhand des Outlogs kann festgestellt werden, ob ein `meet` ausgeführt wurde. So kann es beim Neustart unterdrückt werden.

TACOMA hat somit leider nur eine *at-least-one* Semantik. D.h. ein Agent sollte idempotent sein, es darf also keinen Unterschied ausmachen, ob der Agent einmal oder mehrmals ausgeführt wird.

[11] Achtung, stürzt auch sie ab$\rightarrow$ rekursiver Aufruf möglich

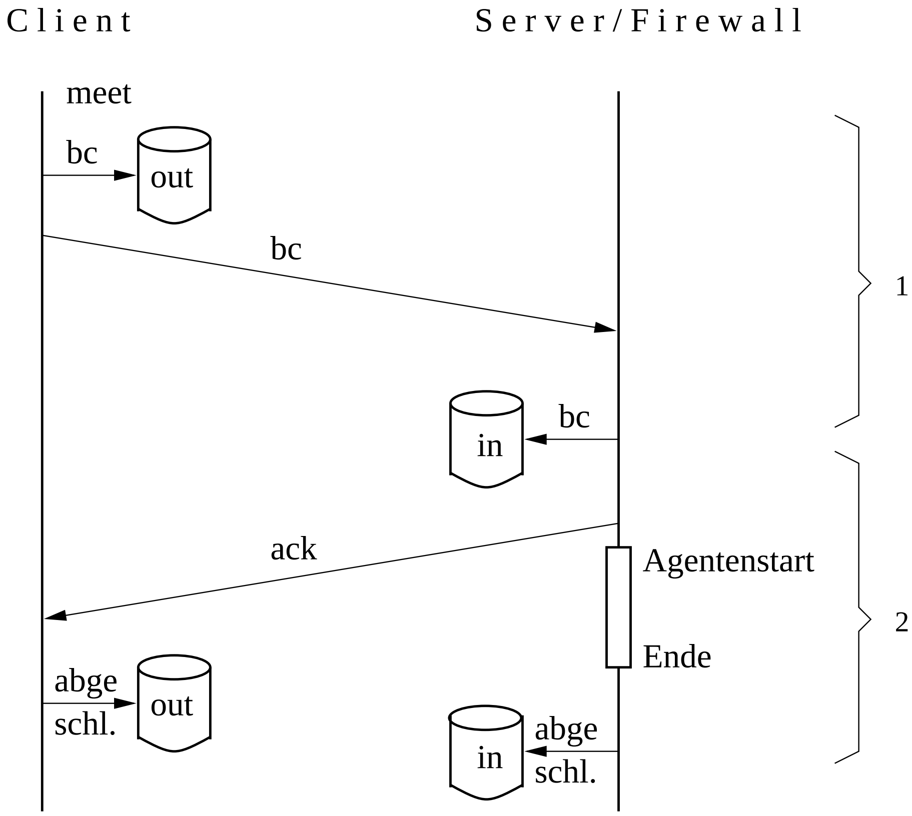


Abbildung 6: Recovery

At-most-once Semantik würde man erreichen, indem man die Lebensdauer eines Agenten als *eine* Transaktion auffaßt. Ein Problem wäre dann aber immer noch, wenn ein Agent weitere Agenten erzeugt. Das läßt sich zwar mit verteilten Transaktionen in den Griff bekommen, da aber *Tacoma* keine Transaktionen unterstützt (und aus Effizienzgründen wohl nie unterstützen wird), gehe ich darauf auch nicht weiter ein.

5 Unterstützung von Programmiersprachen

Das Agentensystem ist wie alles andere auf der Welt in *C* geschrieben. Ältere Versionen basierten noch auf *Tcl*.

Folder sind nichts weiteres als ein Relikt aus alten *Tcl*zeiten, nämlich Strings. Da es in *Tcl* nur Strings gibt war das damals keine Einschränkung. Um das nochmals zu betonen: man kann nur Strings in Folder ablegen, keine anders typisierten Werte!

Will man eine weitere Programmiersprache hinzufügen, so muß man nur `meet` und Folderfunktionen wie `fetch` und `store` implementieren. Schaut man sich nochmals Abb. 4 an, so sieht man das `meet` relativ einfach zu implementieren ist. Sehr trivial sollte die Implentierung der Folderfunktionen sein, die ja im Prinzip nichts weiteres als UNIX Dateiaufrufe sind. Die (für Agentensysteme) große Anzahl von unterstützen Sprachen spricht hierfür. Im übrigen haben viele Programmiersprachen ein *C* Interace, über das auf die schon vorhandenen *C* Routinen zurückgegriffen werden kann.

Dadurch das *Tacoma* ein noch relativ einfaches Agentensystem ist, lassen sich einfach neuen Programmiersprachen hinzufügen.

Die momentan unterstützen Sprachen sind:

- Tcl
- C
- Python
- Scheme (ELK[12])
- Perl

6 Rückblick & Kritik

Das Agentensystem ist in der aktuellen Version bei weitem noch nicht ausgereift. Zuallererst fällt auf, daß die lokale Kommunikation fehlt. Die Agenten die es im Umfeld von TACOMA gibt, sind meistens einfache Agenten die nur von einem Rechner zum nächsten Rechner migrieren und dort schon vorbereitete Daten abholen um sie zu einem Server zu bringen der sie auswertet. In diesen Beispielen ist keine lokale Kommunikation notwendig, wohl aber in komplexeren Anwendungen!

Bei der Authentifikation des Absenderechners verläßt man sich auf die Angabe von *IP*, was i.a. ausreichend ist. Mit entsprechendem Aufwand könnte man aber auch einen falschen Absenderechner angeben.

Bezüglich der Sicherheit des Agenten vor dem Agentensystem gibt es auch noch Mängel: man stelle sich nur vor, daß der Agent vor dem Start von der Firewall modifiziert werden kann. Da mittlerweile die Agenten unter einer anderen *USER*-Id ausgeführt werden als die Firewall braucht im Gegenzug das Agentensystem keine Angst vor den Agenten zu haben.

In TACOMA werden einige ganz interessante Ideen verwirklicht wie das versenden von Briefcases über alternative Protokolle (http, smtp) [JvRS96].

TACOMA soll einmal ein universal einsetzbares Agentensystem werden. Es muß da wohl noch eine Menge getan werden! In Zukunft soll es Unterstützung für folgende Gebiete geben [JvRS95b] & [S96]:

- **Electronic Cash:** Einführung eines Agenten der das im Umlauf befindliche Geld verwaltet.
- **Dienstevermittlung:** Ein *Broker*agent vermittelt Dienste. Er sucht einen billigen/ nicht-ausgelasteten/ naheliegenden Rechner der den entsprechenden Service anbietet.
- **Zugriffskontrolle für Agenten:** Authentifizierte Briefcases dürfen alle Agenten aufrufen, nichtauthentifizierte Briefcases dürfen nur „sichere“ Agenten aufrufen.

Da viel Dokumentation nur auf norwegisch vorlag, gibt es vielleicht noch weitere interessante Gebiete ...

[12] ein Scheme für eingebettete Systeme

Anhang

A Beispielagent

Aufgabenstellung:

Es soll ein Agent *Client* geschrieben werden, dem ein Datei- und ein Rechnername übergeben werden. Der Agent migriert auf den Rechner und fragt *Agent 1*, ob die Datei auf dem Rechner vorhanden ist. Falls das der Fall ist, wird *Agent 2* um die Aushändigung der Datei gebeten. Mit der Datei migriert *Client* wieder zum Ausgangsrechner zurück, um sie anzuzeigen.

Bemerkung

Da, wie bereits schon besprochen, das Agentensystem keine lokale Kommunikation zuläßt, muß die Kommunikation über das Filesystem stattfinden. Die Primitive, die TACOMA dazu bereitstellt (Cabinets & Folder), sind syntaktisch geringfügig einfacher als die entsprechenden UNIX Dateimanipulationsaufrufe. Daher werden sie verwendet. Die Synchronisation muß zwangsläufig über den bekannten Algorithmus von *Dykstra-Dekker-Peterson* zum wechselseitigen Ausschluß realisiert werden. Welch üble Vorstellung! Als Alternative benutze ich dafür *Zufallszahlen* und *busy wait*. Man beachte weiter, die Dateioperationen sind nicht atomar!

Das Grobmodell des Beispielagenten

Der Klient migriert (via `ag_tcl`) auf den entsprechenden Rechner. Er frägt dort Agent 1 über Kabinetfolder, ob die Datei X vorhanden ist. Agent 1 schreibt die Antwort (entweder Ja oder Nein) in ein weiteres Kabinetfolder. Fällt die Antwort positiv aus, bittet der Klient den Agent 2 über Kabinetfolder um die Aushändigung dieser Datei. Agent 2 schreibt die Datei in ein weiteres Kabinetfolder, aus dem dann der Klient die Datei auslesen kann. Der Klient schickt dann die Datei und einen kleinen Anzeigeagent zum Ausgangsrechner zurück. Der Anzeigeagent dumpt schliesslich die Datei nach `/dev/console`.

Da die Details der Kommunikation schwierig und trickreich — und meiner Meinung nach überhaupt nicht lehrreich — sind, wird auf die weitere Besprechung verzichtet.

Der geneigte Leser möge sich durch den Source Code quälen.

Der Beispielagent en detail

Server 1 — Anfrageserver

```
#!/usr/local/bin/tclsh
set auto_path [linsert $auto_path 0 $env(TACOMAPATH)/cabinets/Tcl_files]

cabinet_create Anmeldung1
cabinet_create Kommunikation1

# vorhandene Dateien einlesen
set dateien [exec find Dateien]
puts stdout $dateien
puts stdout "Dateien eingelesen"

#############################################################################

while 1 {
    while {[cabinet_list Anmeldung1] == ""} {} ;# busy wait
    # Kunde ist jetzt da
```

```
    set token [lindex [cabinet_list Anmeldung1] 0]
    # Warten, bis Kunde in $token+1 den Filenamen hingeschrieben hat
    while {[cabinet_fetch Anmeldung1 $token] == ""} {}
    # Jetzt ist der Filename in $token+1
    set filename [cabinet_fetch Anmeldung1 [expr $token + 1]]
    # Anfrage auswerten, d.h. gibt es die Datei?
    # lsearch liefert den index, oder -1 bei Misserfolg
    set nichtgefunden [expr [lsearch -exact $dateien $filename] == -1]
    if {$nichtgefunden} {
        puts stdout nichtgefunden
        cabinet_store Kommunikation1 [expr $token + 1] nichtgefunden
    } {
        puts stdout gefunden
        cabinet_store Kommunikation1 [expr $token +1] gefunden
    }
    cabinet_store Kommunikation1 $token "antwort abgelegt"

    cabinet_delete Anmeldung1       $token
    cabinet_delete Anmeldung1 [expr $token + 1]
}
```

Server 2 — Dateiübergabeserver

```
#!/usr/local/bin/tclsh

set auto_path [linsert $auto_path 0 $env(TACOMAPATH)/cabinets/Tcl_files]


cabinet_create Anmeldung2
cabinet_create Kommunikation2

while 1 {
    while { [cabinet_list Anmeldung2] == "" } {} ;# busy wait
    # Kunde ist da -- vielleicht auch mehrere. wir nehmen den ersten.
    set token [lindex [cabinet_list Anmeldung2] 0]
    # Warten, bis Kunde in $token+1 den Filenamen hingeschrieben hat
    while {[cabinet_fetch Anmeldung2 $token] == ""} {}
    # Jetzt ist der Filename in $token+1
    set filename [cabinet_fetch Anmeldung2 [expr $token + 1]]
    cabinet_store Kommunikation2 [expr $token + 1] [exec cat $filename]
    cabinet_store Kommunikation2       $token      "file abgeliefert"
    cabinet_delete Anmeldung2       $token
    cabinet_delete Anmeldung2 [expr $token + 1]
}
```

Client

```
#!/usr/local/bin/tclsh

# Aufruf: client <host> <dateiname>

set auto_path [linsert $auto_path 0 $env(TACOMAPATH)/cabinets/Tcl_files]
```

```
set movingcode {
    proc main {bc} {
        upvar $bc loc_bc
        set filename [folder_fetch loc_bc FILENAME]
        set ausgangshost [folder_fetch loc_bc AUSGANGSHOST]
        set ran 555 ;# Zufallszahlengenerator einbauen ...
        # Kommunikation mit Server 1
        cabinet_store Anmeldung1       $ran      ""
        cabinet_store Anmeldung1 [expr $ran + 1] $filename
        cabinet_store Anmeldung1       $ran      "nicht leer"
        while {[cabinet_fetch Kommunikation1 $ran] == ""} {} ;# busy wait
        # Antwort ist jetzt in $ran+1
        if {[cabinet_fetch Kommunikation1 [expr $ran + 1]] == "gefunden"} then {
            # Kommunikation mit Server 2
            cabinet_store Anmeldung2       $ran      ""
            cabinet_store Anmeldung2 [expr $ran + 1] $filename
            cabinet_store Anmeldung2       $ran      "nicht leer"
            while {[cabinet_fetch Kommunikation2 $ran] == ""} {} ;# busy wait
            # Fileinhalt ist jetzt in $ran+1
            set fileinhalt [cabinet_fetch Kommunikation2 [expr $ran + 1]]
            cabinet_delete Kommunikation2 $ran
            cabinet_delete Kommunikation2 [expr $ran + 1]
        } else {
            # dann eben nicht ...
            set fileinhalt "file nicht gefunden"
        }
        set show {
            proc main2 {bc} {
                upvar $bc loc_bc
                set out [open /dev/console w]
                puts $out [folder_fetch loc_bc FILE]
                close $out
            }
        }
        # clean up the mess
        cabinet_delete Kommunikation1 $ran
        cabinet_delete Kommunikation1 [expr $ran + 1]
        # Reisevorbereitung: briefcase packen
        bc_create resultbc
        folder_store resultbc HOST     $ausgangshost ""
        folder_store resultbc TCLCODE  SHOWCODE ""
        folder_store resultbc TCLMAIN  main2 ""
        folder_store resultbc SHOWCODE $show
        folder_store resultbc FILE     $fileinhalt
        # und ab nach Hause
        meet ag_tcl resultbc
    }
}

############################################################################

# Achtung
# argv0 ist das 1. Kommandozeilenargument, und nich die ganze Zeile
```

```
if {$argc != 2} {
    # falsche Anzahl von Argumente
    puts "\nAufruf: client <host> <dateiname>"
    puts "     Client migririert nach <host>. Er fragt ob dort die Datei"
    puts "     <dateiname>  vorhanden ist  um sie dann zurueck zum  Aus-"
    puts "     gangshost mitzunehmen und anzuzeigen."
} {
    # sanity check passed
    bc_create clientbc
       # fuer ag_tcl
       folder_store clientbc HOST          [lindex $argv 0] ""
       folder_store clientbc TCLCODE       AGENTCODE ""
       folder_store clientbc AGENTCODE     $movingcode
       # Parameter fuer den migrierten Agenten
       folder_store clientbc FILENAME      [lindex $argv 1]
       folder_store clientbc AUSGANGSHOST $env(HOST)

    # let´s go
    meet ag_tcl clientbc
}
```

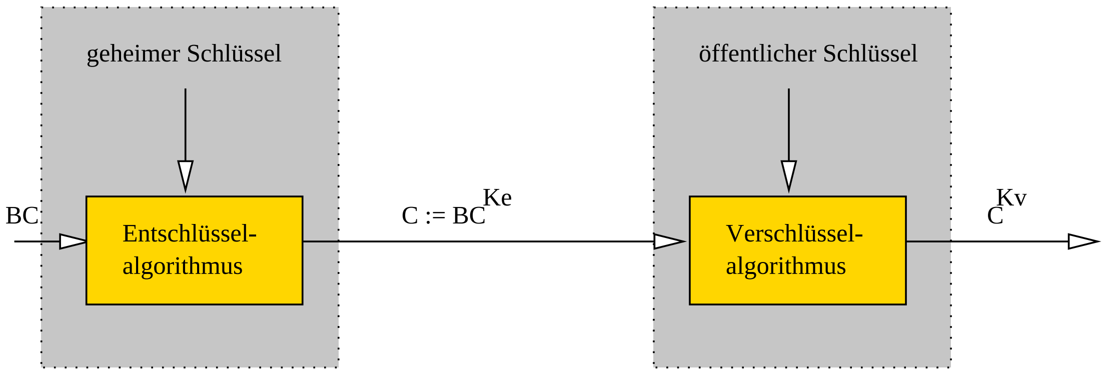
B Digitale Unterschrift

Digitale Unterschriften lassen sich gut über asymmetrische Verschlüsselungsverfahren realisieren. Wie bei asymmetrischen Verfahren üblich muß gelten:

- geheimer Schlüssel $\Rightarrow$ öffentlicher Schlüssel
- öffentlicher Schlüssel $\not\Rightarrow$ geheimer Schlüssel

Es muß weiter gelten:

- M = Entschlüsselung(Verschlüsselung(M))
- M = Verschlüsselung(Entschlüsselung(M))



Zu beachten ist, daß das Briefcase mit dem *Ent*schlüsselungsalgorithmus *ver*schlüsselt wird und mit dem *Ver*schlüsselungsalgorithmus *ent*schlüsselt wird [R96].

Literatur

[JvRS95a] Dag Johansen, Robbert van Renesse, Fred B. Schneider. An Introduction to the TACOMA Distributed System – Version 1. Technical report TR-95-23, Universität Tromsø, Norwegen, Juni 1995.

[JvRS95b] Dag Johansen, Robbert van Renesse, Fred B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HOTOS-V)*, Seite 42-45. IEEE Press, Mai 1995.

[JvRS96] Dag Johansen, Robbert van Renesse, Fred B. Schneider. Support Broad Internet Access to TACOMA. Eingereicht an *ACM SigOps Workshop, September 96*, Februar 1996.

[S96] Nils Peter Sudmann. TACOMA — fundamental abstractions supporting agent computing in a distributed environment. Diplomarbeit Universität Tromsø, Abteilung Informatik, August 1996.

[R96] Kurt Rothermel. Systemnahe Software II. Skript Universität Stuttgart, IPVR, Fakultät Informatik, Lehrstuhl „Veteilte Systeme“.

Die meisten Informationen wurden aus [S96] übernommen.