Seminar zum Thema Messenger

Oliver Schramm

2. Januar 1997

Inhaltsverzeichnis

1 Einleitung

Die heutige Kommunikation zwischen Computern basiert auf dem TCP/IP- und OSI Referenzmodell. Diese Modelle sind semi-statisch, das bedeutet, daß Computer nur über bestimmte definierte Protokolle miteinander kommunizieren können. Die Tatsache, daß die Entwicklung von Standards für Protokolle sich als recht langsam erwiesen hat und semi-statische Protokoll-Suites ungeeignet sind für Hochgeschwindigkeitsnetzen, führte zur Entwicklung eines neuen Kommunikationsmodells *Messenger* .

Messenger wird an der Universtät von Genf, im Forschungsbereich für *Teleinformatics and Operating System Groups* entwickelt. Die Entwicklung wird von Dr. Christian F. Tschudin `<tschudin@cui.unige.ch>` geleitet.

In dieser seminaristischen Ausarbeitung wird das *Messenger Paradigma* genauer analysiert. Weiter soll die Möglichkeit eines Einsatzes eines solchen Systems als Agentensystem untersucht werden. Abschließend wird auf weitere Forschungen im Gebiet *Messenger* eingegangen.

2 Messenger

Im Folgenden wird das *Messenger Paradigma* im Vergleich zum herkömmlichen *Modell von Shannon* vorgestellt.

2.1 Modell von Shannon

Das bisherige Kommunikationsmodell basiert auf dem Modell von Shannon. In diesem Modell steht die Übertragung von Nachrichten als passive Elemente im Vordergrund. Die Interpretation einer Nachricht wird im OSI Referenzmodell durch das Schichtenkonzept ermöglicht. Hierbei führt jede Schicht abhängig von der PDU [1] eine bestimmte definierte Aktion aus. In der Literatur wird daher das *Modell von Shannon* auch *PDU-Modell* genannt.

[1] Protokoll Data Unit

Das *Modell von Shannon* bringt einige Nachteile mit sich:

- Protokolle sind statisch gebunden, d.h. der Kernel muß die entsprechende Protokoll-Suite beim Booten zur Verfügung stellen.
- Kommunizierende Rechner benötigen die gleiche Protokoll-Suite.
- Weniger geeignet für Hochgeschwindigkeitsnetzwerke [8].

2.2 Messenger Modell

Das *Messenger Modell* sieht Nachrichten nicht als passive Elemente sondern als aktive Elemente an, die ausgeführt werden können. Eine Nachricht besteht dabei aus Instruktionen und Daten. Dieses Modell benötigt keinen Protokoll-Stack wie beim *Modell von Shannon* . Jeder Rechner muß aber über eine *Ausführungsplattform* verfügen, die es ermöglicht, Nachrichten auszuführen.

Messenger definiert ein neues Referenzmodell für Computerkommunikation. Es kann gezeigt werden, daß das *Messenger Modell* mächtiger ist als das *Modell von Shannon* [1]. Dies führt dazu, daß das bisherige Referenzmodell in diesem neuen Referenzmodell integriert werden kann.

2.3 Messenger Paradigma in anderen Wissenschaften

Das Messenger Paradigma ist auch Forschungsgebiet anderer, der Informatik entfernten, Wissenschaften. Abbildung 1 zeigt weitere Wissenschaften, die am *Messenger Paradigma* forschen.

2.4 Ziele von Messenger

Es ist zu erwähnen, daß bei der Entwicklung von Messenger das Kommunikationsparadigma im Vordergrund stand. Man entdeckte erst später das Potential des *Messenger Paradigma* im Bereich von *Mobilen Agenten* .

Ansatzweise wurde das *Messenger Paradigma* schon recht früh vorgeschlagen. Schon im Jahre 1983 [3] kam man auf die Idee, electronic Mail Meldungen mit Instruktionen zu versehen, so daß sie als `"intelligentes Dokument"` gleich selbst über ihre Behandlung entscheiden können. Produkte wie TELESCRIPTAgenten verwenden diese Idee.

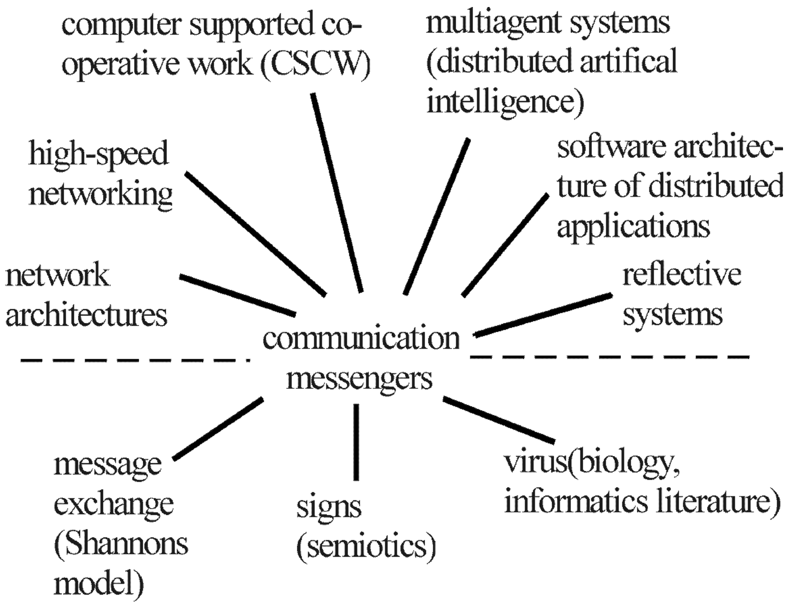

Abbildung 1: Forschungsbereich Messenger

3 Computer Kommunikation mit Messenger

Ziel dieses Abschnittes ist es die Kommunikation zwischen Rechner mit Hilfe des Messenger Paradigmas darzustellen. Das *Messenger Paradigma* basiert auf der Tatsache, daß die übertragenen Nachrichten aktiv sind, also Instruktionen, die auf dem entfernten Rechnern ausgeführt werden.

Das *Messenger Paradigma* wird durch sogenannte *Kommunikations Messenger* und *Messenger Plattform* realisiert. Der *Kommunikations Messenger* kann als Objekt angesehen werden. Er besteht aus Instruktionen und optional aus Daten.

Das nächste Kapitel stellt eine hypothetische Plattform vor, auf welcher aktive Nachrichten [2] ausgeführt werden können. Im Kapitel 5.2 wird eine reale Implementierung einer *Messenger Paradigma* gezeigt.

Nachdem solch eine hypothetische Plattform vorgestellt wurde, soll anhand eines Beispieles der praktische Ablauf von *Messenger* gezeigt werden.

Abgeschlossen wird diese Kapitel mit einem Vergleich der Mächtigkeit der beiden Paradigmas.

[2] *Kommunikations Messenger*

Zur Notation: anstatt *Kommunikations Messenger* wird im Folgendenen nur noch der Begriff *Messenger* verwendet.

3.1 Messenger Plattform

Die hier angenommene *Messenger Plattform* besteht im Wesentlichen aus:

- einem *Messenger* , welcher aus sequentiellen Instruktionen besteht.
- einer *Ausführungsplattform* , mit den folgenden Eigenschaften:

  Menge von Prozessen
  Aus jeder bei der Plattform eintreffenden Nachricht wird ein Prozess generiert. Sämtliche Prozesse laufen unabhängig und parallel voneinander ab. Prozesse können nur über das Verzeichnis miteinander kommunizieren.

  Prozesswarteschlangen
  Zur Synchronisation von Prozessen werden Prozesswarteschlangen benötigt. In diese können sich Prozesse selbst einreihen und auch wieder selbst verlassen. Prozesswarteschlangen sind ein sehr einfaches Synchronisationmittel, sie dienen dazu, den Zugriff auf das Verzeichnis zu regeln. Es wird darauf hingewiesen, daß in zukünftigen Versionen auch höhere Synchronisationsmittel wie Monitore zur Verfügung stehen werden.

  Verzeichnis
  Das Verzeichnis dient als globaler Speicher auf einem Ort. Hierbei versteht man unter einem Ort einen einzigen physikalischen Rechner.

  Kanäle
  Kanäle dienen der Kommuikation mit anderen Rechner.

Abbildung 2 skizziert eine hypothetische *Ausführungsplattform* .

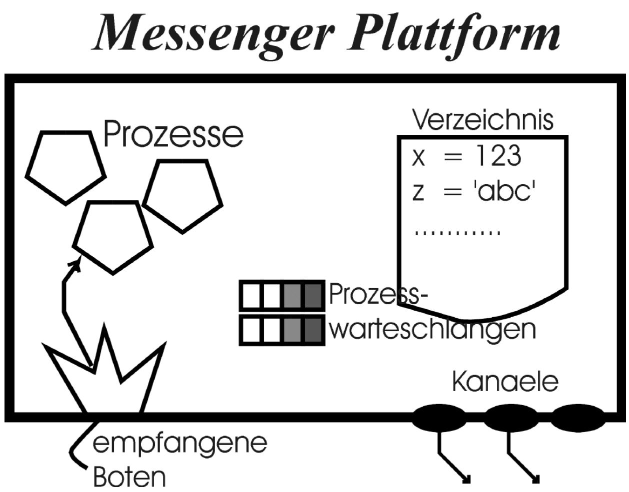

Abbildung 2: Messenger Plattform

Innerhalb einer Ausführungsumgebung wird aus jeder ankommenden Nachricht ein Prozess generiert. Dieser kann wiederum lokale Prozesse erzeugen oder *Messenger* an andere Rechner senden. Daten die globalen Zugriff benötigen werden im Verzeichnis abgelegt.

3.2 Kommunikation Messenger

Um den Leser nicht unnötig zu verwirren, werden die Instruktionen in einem *Kommunikations Messenger* in PASCAL Notation dargestellt.

Im Folgenden werden Funktion definiert, die den Zugriff auf die einzelnen Komponenten einer *Messenger Plattform* regeln.

ACTIVATE('done') DEACTIVATE('done')
: ermöglichen eine Prozesswarteschlange zu aktivieren bzw. zu deaktivieren. Ist eine Prozesswarteschlange deaktiviert, so sind alle Prozesse in dieser Prozesswarteschlange blockiert.

SET('a',5) GET(a)
: ermöglichen den Zugriff auf das Verzeichnis. SET(a,5) setzt die Variable a im Verzeichnis auf 5.

SEND(channel, messenger)

generiert einen weitereren *Messenger* im Kanal channel. Ist channel lokal, so wird ein weiterer Prozess auf der lokalen Plattform gestartet.

Abbildung 3 zeigt einen in Pascal-Notation definierten *Messenger* .

```
MSGTYPE ack(); /* entspricht PROGRAM ack; in PASCAL

BEGIN
  SET('RESPONSE', TRUE);
END;
```

Abbildung 3: Acknowledge Messenger

Dieser Acknowledgement *Messenger* setzt die Variable RESPONSE im Verzeichnis des Rechners Host A auf TRUE.

3.3 Realisierung des Ping Protokoll anhand des Messenger Paradigma

Nachdem nun dargestellt wurde wie eine *Messenger Plattform* und ein *Messenger* aussehen können, wird an einem praktischen Beispiel das TCP/IP Protokoll Ping durch *Messenger* realisiert.

Wir nehmen an, daß die Kommunikation zwischen 2 Rechnern, Host A und Host B, stattfindet. Jeder dieser Rechner verfügt über eine *Messenger Plattform* .

Weiter werden 4 *Messenger* definiert:

Source
: Der Source *Messenger* verweilt auf Host A und generiert solange *Messenger* vom Typ SEND auf Host A bis die Variable RESPONSE im Verzeichnis auf True gesetzt wird. Nachdem Source ein *Messenger* vom Typ SEND generiert hat, reiht er sich selbst in die Prozesswarteschlange 'DONE' ein, die im Zustand Stop ist. (D.h. der Source *Messenger* ist blockiert).

Send
: Send generiert ein *Messenger* vom Typ Transit. Dieser wird zu Host B übersand, der daraus einen Prozess generiert.

Transit

Transit erzeugt auf dem Rechner Host B einen weiteren *Messenger* vom Typ Acknowledge, der zurück nach Host A geschickt wird.

Ack

Ack setzt in das Verzeichnis von Rechner Host A die Variable Response auf TRUE. Dadurch wird bei der nächsten Abfrage des Source *Messenger* der Variablenwert erkannt und der Applikation die entsprechende Information gegeben.

In Abbildung 4 wird der Ablauf der Kommunikation durch *Messengers* dargestellt. Der verwendet Quellcode ist in Abbildung 5 zusammengefaßt.

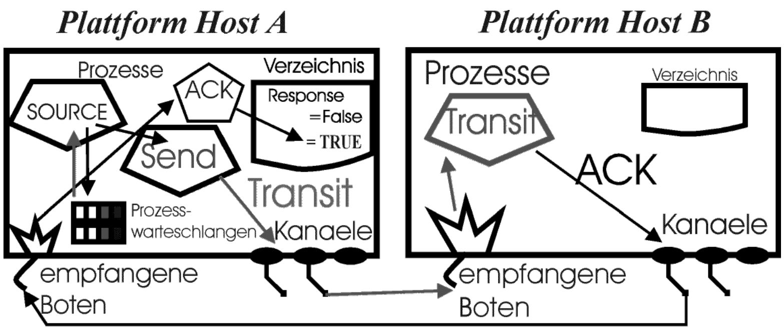

Abbildung 4: Kommunikation zwischen zwei Messenger Plattformen

3.4 Mächtigkeit der beiden Paradigmas

Um ein Vergleich der beiden Paradigmas machen zu können, wird an dieser Stelle tiefer auf die Konzepte eingegangen. Das OSI Referenzmodell basiert auf 2 Strukturen.

- den Protokollen (horizontale Struktur)
- den Service Access Points (vertikale Struktur).

Dabei wird über die Implementierung der Strukturen nichts ausgesagt. Das *Messenger Paradigma* dagegen definiert, wie eine *Ausführungsplattform* auszusehen hat, nicht aber die Struktur von Protokollen.

```
MSGTYP source();

  MSGTYPE send(orig,dest,information);

    MSGTYPE transit(orig,information,ack);

      MSGTYPE ack();

      BEGIN
        SET('RESPONSE', TRUE);
      END;

    BEGIN
      /* Deliver Information */
      SUBMIT(orig, ack);
    END;

  BEGIN
    SUBMIT(dest,transit(orig,information,ack));
    AKTIVATE('done');
  END;



 VAR information:STRING[255];
     tmp    :INTEGER;
 BEGIN
   LOOP
     DEAKTIVATE('done');
     tmp=GET('RESPONSE');

     IF (TMP <> TRUE)
        SUBMIT(HostA, send(HostA, HostB,information);
     ELSE
        EXIT(); /* Information an aufrufende Applikation das entfernter
                   Rechner existiert */

     ENTER('done')
   ENDLOOP;
 END;

```


Abbildung 5: Source Messenger Quellcode

Es konnte gezeigt werden [1], daß das *Messenger Paradigma* mächtiger ist als das *Modell von Shannon* . Abbildung 6 stellt die beiden Paradigmen in einem Mengendiagramm dar.

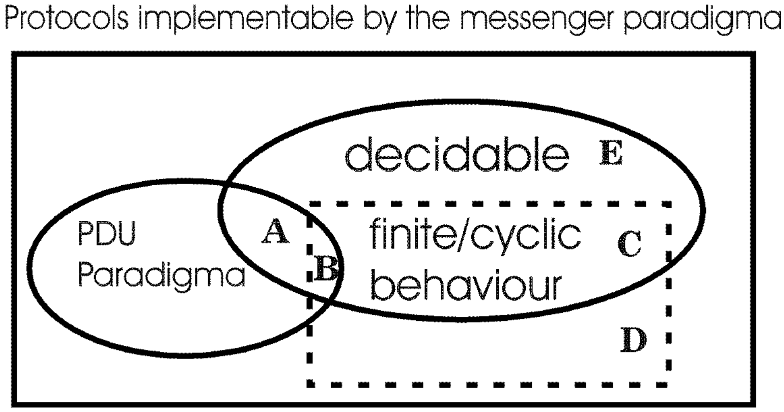

Abbildung 6: Mächtigkeit des *Messenger Paradigma*

Ein offener Punkt in der theoretischen Forschung dieser beiden Paradigmen ist die Transformation von PDU basierte Protokollen nach *Messenger* basierten Protokollen. Bisher gibt es noch kein automatisches Verfahren, das diese Transformation durchführt.

4 Messenger als Mobiler Agent

Die grundlegende Idee des *Messenger Paradigma* liegt in einer Verbesserung der Kommunikation zwischen Computern. Dies führte dazu, daß auf einem sehr niedrigen Niveau die Übertragung von aktiven Nachrichten möglicht wurde. Da *Messengers* vom Prinzip her nichts anderes als aktive Nachrichten sind, fand dieses Paradigma im Bereich der Mobilen Agenten Anklang. Man könnte auch statt *Messenger* den Begriff *Mobiler Agent* verwenden, doch möchte ich im Umfang dieser Arbeit den Begriff *Messenger* beibehalten. In Kapitel 5 werden dann die Eigenschaften von *Messenger* als *Agent / Agentensystem* analysiert.

5 M0

Im vorhergehenden Kapitel wurde die Grundlage des *Messenger Paradigma* erläutert. Folgend wird eine Implementierung dieses Paradigma - **M0** - vorgestellt. **M0** umfasst:

- eine Sprache in der Messengers programmiert werden können. (**M0** - Language)
- ein Interpreter für diese Sprache
- eine Ausführungsplattform, die sogenannte (**M0** -Plattform)

5.1 Das Sprachekonzept von M0

Die Programmiersprache **M0** ist eine Hochsprache mit starker Ähnlichkeit zu *Postscript* . Alle Datenwerte haben einen Typ und es stehen Mittel zur Verfügung um Datenelemente zu strukturieren (kein Goto). **M0** verfügt auch über das Konzept von Unterprogrammen.

Die Verwandschaft mit Postscript geht sehr weit - so wurde das Stack-orientierte Ausführungsmodell beibehalten, ebenso wie der in *Postscript* angewandte Mechanismus, an Namen gebundene Werte aufzufinden. **M0** verfügt nur über einen Stack als Laufzeitspeicher, nicht aber über dynamischen Speicher.

M0 besitzt einen Präprozessor, der es ermöglicht, Bibliotheken einzubinden.

Unter dem Gesichtspunkt heutiger Programmiersprachen ist **M0** mehr als eine Zwischensprache zu sehen. Wichtige Programmierkonzepte wie Lesbarkeit, Wartbarkeit aber auch Aspekte aus dem Gebiet der objektorientierten Programmierung sind in **M0** nicht integriert. Abbildung Kapitel 7 stellt die Funktion SUBMIT() in der Sprache **M0** .

Dieses Codebeispiel soll dem nicht *Postscript* Programmierer die Schwierigkeit klarmachen, in **M0** größere Softwareprojekte zu entwickeln.

5.2 Die Ausführungplattform M0

Die Ausführungsplattform **M0** stellt die Grundfunktionalität wie sie in Kapitel 3.1 beschrieben wurde zur Verfügung. Weiter definiert diese ein Protokoll wie die Übertragung von Messengers durchgeführt werden kann. In diesem

```
{\ffffffffffff\ 0 {}
 _cha 'ether G '_key G ! }
{ _orig2G _orig1G {}
 _cha _orig0G '_key G ! }
 neighbors
 ptop
```

Abbildung 7: Programm Beispiel in M0

Punkt unterscheidet sich die Kommunikation nicht von dem bisherigen *Modell von Shannon* .

In der vorliegenden Implementierung von **M0** basiert dieser Service auf dem UDP Protokoll. Die darauf aufbauende Plattform kann also davon ausgehen, daß die Pakete die eintreffen, korrekt sind. (Fehlerhafte Pakete werden an der Checksum erkannt und weggeworfen, verlorengegangene Packete treffen nicht ein.)

Die Repräsentation der Nachrichten erfolgen in einer Big-Endian Kodierung.

Im Weiteren verfügt die **M0** Plattform noch über einen Interpreter, der durch eine einfache Stackmaschine realisiert wird. Das Problem des Speicherzugriffes wurde durch einen lokalen virtuellen und globalen virtuellen Speicher gelöst. Jeder Prozess hat einen lokalen virtuellen Speicher. Das Verzeichnis wird durch den globalen virtuellen Speicher repräsentiert.

Objekte wie Daten, Prozesswarteschlangen werden über einen KEY referenziert. Der Key ist ein 4 Byte Objekt, der zufällig erzeugt wird. Die Wahrscheinlichkeit, daß dabei ein Konflikt auftritt, liegt bei `1/2^64` - also sehr gering.

5.2.1 Kommunikation, Synchronisation

Die einzigste Möglichkeit, daß Prozesse miteinander kommunizieren können bzw. sich synchronizieren können, besteht in der Verwendung von Prozesswarteschlangen und dem Verzeichnis. Es ist nicht möglich, daß Prozesse direkt bzw. über Rechnergrenzen hinweg miteinander kommunizieren können.

5.2.2 Service

Die *Ausführungsplattform* stellt keine Default Services zur Verfügung. Mechanismen zum Auffinden von anderen *Messenger* bzw. sogenannte Gateway Agenten werden konzeptionell nicht unterstützt. Dies liegt wie in Kapitel 2.4 angedeutet an der Intention des *Messenger Paradigma* .

5.2.3 Sicherheit

M0 ist ein offenes System. Dadurch, daß jeder Prozess unabhänig von jeder Kontrolle arbeitet, ist es sehr schwierig, wenn nicht sogar unmöglich, eine sichere *Ausführungsplattform* zur Verfügung zu stellen. Der einzigste Schutz, wenn man es so nennen kann, liegt in der Vergabe von Schlüsseln.

Sicherheit nach Innen
Unter Sicherheit nach Innen ist ein mögliches böswilliges Einwirken der *Ausführungsplattform* auf *Messenger* zu verstehen [6]. Da die die Plattform beliebig auf die Instruktionen und Daten eines Messengers zugreifen kann, besteht hierbei in keiner Form, irgendein Schutz.

Sicherheit nach außen
Unter Sicherheit nach außen ist zu verstehen, daß sich *Messenger* untereinander oder ein *Messenger* die Plattform schädigen kann [6]. *Messenger* können sich direkt nicht beeinflussen, da eine Kommunikation nur über das Verzeichnis stattfinden kann.

Es ist aber ohne weiteres denkbar, daß böswillige *Messengers* eine Plattform überfluten und die Resourcen belegen, d.h. einen sog. Denial of Service verursachen.

Auch kann ein böswilliger Messenger die Kommunikation zwischen *Messenger* beeinflussen, indem er ihre Daten im Verzeichnis, auf das jeder *Messenger* Zugriff hat, manipuliert.

5.3 Migrationskonzept

In **M0** findet eine Migration eines *Messenger* , egal in welchem Status (aktiv oder passiv) er sich befindet, nicht statt. Ein aktiver *Messenger* kann entweder eine Nachricht lokal oder entfernt verschicken. Dies führt dazu, daß entweder auf einem lokalen oder auf einem entfernten Rechner ein Prozess gestartet wird, der den entsprechenden *Messenger* ausführt. Der *Messenger* selbst verweilt aber auf seiner Plattform, bis er sich selbst beendet.

Es ist möglich, daß ein *Messenger* eine Übertragung von sich selbst ausführt und sich danach selbst beendet. Der durch die Übertragung erzeugte neue Prozess beginnt seine Ausführung aber wieder mit der 1.Instruktion.

6 Weitere Forschungen über Messenger

Wir haben bisher das zugrundeliegende Messenger Paradigma analysiert und eine Implementierung davon kennengelernt. Im folgenden sollen noch kurz *Messenger* unter dem Gesichtspunkt Objektorierter Programmierung vorgestellt werden.

6.1 OO-Betrachtung

Obwohl *Messenger* wie Objekte aufgebaut sind (Instruktionen und Daten) passen sie nicht in die sichtweise Objektorientierter Programmierung. *Messenger* Objekte kommunizieren nicht direkt miteinander, noch kann ein *Messenger* auf Instruktionen eines anderen *Messenger* zugreifen.

Es exsitieren jedoch konzeptionelle Ideen, daß *Messenger* System als Grundlage eines Objektorientierten Agentensystem zu verwenden [5].

A Anhang

Im Umfang diese Seminarsistischen Ausarbeitung wurde mit der **M0** Version m0-0.18 unter Linux gearbeitet. Diese verfügt über einen Eingabeconsole zur Programmierung eines *Messenger* . **M0** wird derzeit auf folgenden Plattformen unterstüetzt.

Hardware	Operating System
Dec Alpha	OSF/1 V1.3
Dec Risc	Ultrix V4.3
HP9000	HP-UX B.09.00
Intel 80386	Linux 1.0.9
Intel 80x86	MS/DOS 5.0
PowerPC601	AIX 3.2.5
Sun Sparc	SunOS 4.1.3, Solaris 2.3

Abbildung 8: Verfügbare Plattformen

Literatur

[1] Christian F. Tschudin. On the Structure of Computer Communications, Doktorarbeit, Universität Genf, 1993.

[2] Christian F. Tschudin, *M0 - a messenger execution environment. Usenet newsgroup comp.sources.unix, Vol 28,Issue 51-62, Juni 1994*

[3] Christian F.Tschudin, *Conceptual and Practical Problems of Validating Messenger-Based Protocols, Submitted for publication, Mai 1994*

[4] Christian F.Tschudin, *An Introduction to the M0 Messenger Language*

[5] Christian F.Tschudin, *Position Paper for the ECOOP'95 Workshop W10 on Objects and Agents, August 1995.*

[6] Fritz Hohl, *Konzeption eines einfachen Agentensystem und Implementierung eines Prototypes. Diplomarbeit Nr. 1267 Universität Stuttgart.*

[7] Genesereth, M.R., and Ketchpel, S.P., `"Software Agents"`*, available at URL http://logic.stanford.edu/sharing/papers/agents.ps*

[8] S.W.O'Malley and L.L.Peterson, *A highly layered architecture for high-speed networks. In Marjory J.Johnson, editor, Protocols for High-Speed Networks II, pages 141-156, Elsevier,1991.*