Universität Hamburg

Fachbereich Informatik

Arbeisbereiche Verteilte Systeme und
Informationssysteme (VSIS)

STUDIENARBEIT

# Entwicklung eines generischen Robots für das Scone-Framework

Frank Wollenweber

im Juni 2002

Betreuer: Prof. Dr. W. Lamersdorf

Studienarbeit:
Entwicklung eines generischen Robots für das Scone-Framework

Autor:
Frank Wollenweber
Harnacksweg 44
22417 Hamburg
E-Mail: mail@frank-wollenweber.de

Betreuer:
Prof. Dr. Winfried Lamersdorf
Arbeitsgruppe Verteilte Systeme und Informationssysteme (VSIS)

Dipl. Inform. Harald Weinreich
Arbeitsgruppe Verteilte Systeme und Informationssysteme (VSIS)

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

# Erklärung

Hiermit erkläre ich an Eides statt, daß ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hamburg, am 28. Juni 2002

Frank Wollenweber

# Danksagung

Mein Besonderer Dank für die vielen Anregungen und Hinweise während der Erstellung dieser Arbeit gilt den Mitarbeitern unseres Fachbereichs Herrn Harald Weinreich und Herrn Matthias Mayer.

Für die Erstellung des Textes wurde LaTeX verwandt. Als Grundlage diente dabei die Formatvorlage `hagenberg.sty`, bei deren Autor, Herrn Dr. Wilhelm Burger von der Fachhochschule Hagenberg in Österreich, ich mich ebenfalls bedanken möchte.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

> „Where humans cannot cope with the amount of information it is attractive to let the computer do the work.“(Zitat: [Kos95])

Die Menge der im World Wide Web angebotenen Inhalte ist in den letzten Jahren extrem gewachsen. Dementsprechend ist die Suche nach relevanten Inhalten immer schwieriger geworden.

Ein Ansatz die Navigation im WWW effizienter zu gestalten sind Suchmaschinen. Basierend auf einem von Web-Robots erstellten Index schlagen sie Benutzern Seiten vor, die zu der jeweiligen Suchanfrage passen. Die Ergebnisse der Suchanfragen sind jedoch in vielen Fällen nur sehr eingeschränkt hilfreich, da vielfach mehrere Tausend Dokumente aufgelistet werden.

Ein weiterer Ansatz sind Werkzeuge, die auf dem Rechner des Benutzers neben dem Web-Browser laufen oder die Funktionalität des Web-Browsers erweitern und so den Nutzer bei der Informationssuche im Web unterstützen. Beispiele sind die History-Visualisierung Browsing-Icons und die Erweiterung der Darstellung von Hyperlinks durch das Programm HyperScout. Interessante Anwendungen ergeben sich aus der Kombination von client-seitig ablaufenden Programmen mit dem Leistungsangebot von Suchmaschinen. Ein Beispiel ist der Google-Rückspiegel, auf den in Kapitel 7.2 eingegangen wird.

Derartige Anwendungen zur Unterstützung bei der Navigation im Web lassen sich mit dem Scone-Framework prototypisch implementieren und evaluieren. Basierend auf WBI von IBM realisiert Scone einen Proxy-Server, der zudem die Benutzeraktionen überwacht, die geladenen HTML-Dokumente parst und in einer Datenbank persistent macht. Des weiteren bietet Scone die Möglichkeit den Datenstrom zwischen Client und Server zu manipulieren und mittels eines Web-Robots automatisch Objekte aus dem Web zu laden. Die Konstruktion dieses Robots ist das zentrale Thema dieser Arbeit.

In einem Grundlagen-Kapitel stelle ich die unter dem Aspekt der Konstruktion eines Robots relevanten Merkmale des WWW vor, gebe einen Überblick über Anwendung und Funktion von Web-Robots und stelle das Scone-Framework vor, in das der beschriebene Robot integriert wurde. Anschließend gebe ich einen Überblick über verschiedene Projekte, die sich mit Konstruktion, Anwendung und Optimierung von Web-Robots beschäftigen. Die Probleme bei der Navigation im Web und die daraus resultierende Motivation für die Integration eines Robots in das Scone-Framework werden in Kapitel 4 beschrieben. Die Anforderungen an den Robot und dessen softwaretechnische Realisierung werden in den Kapiteln 5 und 6 dargestellt. Den Abschluß der Arbeit bildet die Vorstellung zweier unter Benutzung des Scone-Robots realisierter Applikationen (Kapitel 7) und ein Fazit (Kapitel 8).

# Kapitel 2

# Grundlagen

In diesem Kapitel werden die der Studienarbeit zugrunde liegenden Themen vorgestellt. Abschnitt 2.1 gibt einen Überblick über das WWW. Es folgt ein Abschnitt über Web-Robots 2.2 und die Darstellung des Scone-Frameworks 2.3.

## 2.1 Architektur des World Wide Web

Das *World Wide Web*, im Folgenden kurz Web oder *WWW* genannt, spezifiziert ein verteiltes, multimediales Hypertext System, das auf der Internet-Infrastrsuktur aufbaut. Die Client-Server-Architektur des Webs umfaßt ein auf TCP/IP basierendes Übertragungsprotokoll *HTTP (Hypertext Transport Protocol)*, ein Adress-Schema *URI (Uniform Ressource Identifier)* und eine Dokumenten Auszeichnungssprache *HTML (Hypertext Markup Language)*. Die Funktion beschreibt Tim Barners Lee, der als Erfinder des Webs gilt, in seinem Artikel „The World Wide Web “ [BLCL$^+$94]

Die Interaktion der Komponenten im WWW verdeutliche Abbildung 2.1. Die verlinkten Dokumente können auf verschiedenen Servern liegen. Clickt der Benutzer auf einen Link, dann wird das jeweilige Dokument wie nachfolgend beschrieben über das Internet vom jeweiligen Server geladen. Die Kommunikation mit den verschiedenen Servern erfolgt für den Benutzer transparent, so daß auch in Hypertexten, die auf externen Servern liegende Ressourcen einbeziehen, relativ bequem navigiert werden kann.

Wird im Webbrowser eine neue Seite aufgerufen, dann wird im ersten Schritt durch eine DNS-Abfrage die durch die URL angegebene symbolische Adresse in eine IP-Adresse aufgelöst. Wurde kein anderer Port angegeben, dann wird zu Port 80 dieser Adresse eine TCP-Verbindung aufgebaut und per HTTP (siehe 2.1.2) ein Request geschickt. Der Server antwortet dann mit

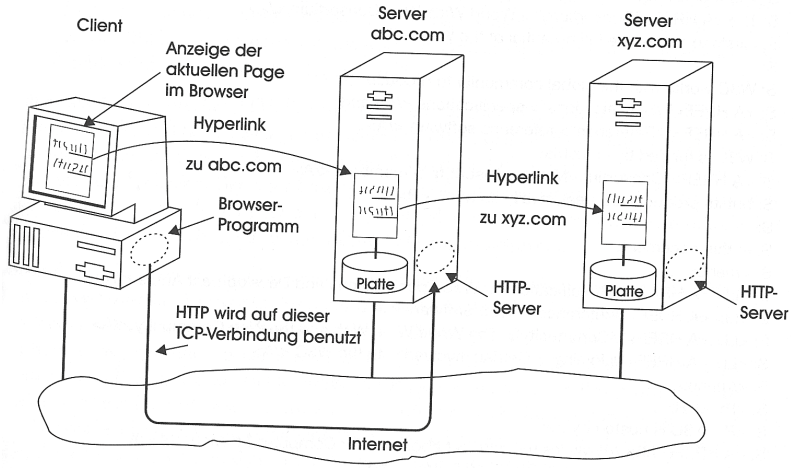


**Abbildung 2.1:** Funktionsskizze des WWW. Quelle: [Tan97].

einer HTTP-Response, die im MIME-Format (siehe 2.1.5) das angeforderte Dokument enthält. Bei Verwendung von HTTP/1.0 wird nun die TCP-Verbindung wieder geschlossen. Handelt es sich dabei um ein in HTML (siehe 2.1.3) geschriebenes Dokument, dann wird es vom Browser geparst und alle in die Seite eingebetteten Dateien, wie z.B. Bilder, werden ebenfalls per HTTP vom Server geladen. Das Dokument wird nun entsprechend den HTML Anweisungen formatiert dargestellt. Klickt der Benutzer auf einen Link, dann wiederholt sich der beschriebene Vorgang mit der im Link angegebenen URL.

Das Internet und das darauf aufbauende World Wide Web bildet den technologischen Rahmen sowohl für Web-Robots, als auch für das Scone-Framework. Angesichts der Verbreitung des Internets wird davon ausgegangen, daß der Leser dieser Arbeit grundsätzlich mit dieser Technologie vertraut ist. Dennoch werden im Folgenden kurz die wesentlichen Merkmale des World Wide Web dargestellt, um eine gesicherte Grundlage für die folgende Darstellung zu schaffen. Der folgende Überblick über die Architektur des WWW basiert im wesentlichen auf den kompakten Darstellungen in [Wil99] und [Tan97], Details finden sich auf der Web-Site des World Wide Web Konsortiums (www.w3c.org) und in den entsprechenden RFCs.

### 2.1.1 Uniform Ressource Identifier

Für die Funktion des Webs als verteiltest Hypermedia System ist es unerläßlich Informationsressourcen mit netzweit eindeutigen Identifieren zu benennen. Dies ermöglicht einen Zugriff auf Dokumente unabhängig davon,

auf welchem Server sich diese befinden und die Verknüpfung von auf unterschiedlichen Servern liegenden Dokumenten mittels Links.

Die im Web verwendeten *Universal Ressource Identifier (URI)* bieten ein allgemeines Schema zur Identifizierung von Ressourcen im Web. Man unterscheidet dabei zwischen *Uniform Ressource Names (URN)*, die eine Ressource benennen und *Uniform Ressource Locators (URL)*, die deren Adresse angeben. URNs bieten die Möglichkeit Ressourcen unabhängig von deren konkreter Adresse zu bennen, was jedoch den Mehraufwand eines Namensdienstes mit sich bringt, der dann URNs auf URLs abbildet. In der Praxis spielten URNs bislang kaum eine Rolle, wenngleich ihre Bedeutung im Zusammenhang mit der XML-Technologie zugenommen hat. Als Teil der Spezifikation von URIs werden URNs in den folgenden Ausführungen mit berücksichtigt und es wird im Regellfall der allgemeinere Begriff der URI verwendet.

In RFC 1630 [BL94] wird die Syntax von URIs angegeben. Die einfache Syntax basiert auf der Idee die URI in zwei Teile zu unterteilen, wobei ein Teil das Identifikationsschema angibt und der Andere in einer vom Schema abhängigen Weise das zu identifizierende Objekt benennt.

```
Uri = Schema ":" schemaspezifischer Teil
```

Nach den RFCs 1738 [BLMM94] und 1808 [Fie95] hat der schemaspezifische Teil einer URL die folgende Syntax:

```
schemaspezifischer Teil =
"//" [ benutzer [ ":" kennwort ] "@" ] rechner [ ":" port ]
[pfad] [datei] [ "?" abfrage ] [ "\#" fragment ]
```

Das gebräuchlichste Schema ist `http` für den Zugriff auf Dokumente über das *Hypertext Transport Protocol*. Die beschriebene Syntax des schemaspezifischen Teils ist aber auch für andere Schemata, wie z.B. `ftp` *(File Transfer Protocol)* gültig, wobei die einzelnen Komponenten eventuell unterschiedlich interpretiert werden.

Der Parameter `rechner` bezeichnet einen Rechner im Internet. Dies kann entweder über die Angabe einer IP-Adresse oder über einen Domainnamen geschehen. Der Wert `port` kennzeichnet den Prozess auf diesem Rechner, mit dem kommununiziert werden soll. Verzichtet man auf die Angabe eines Ports, so wird bei Verwendung des HTTP-Protokolls automatisch Port 80 angenommen. `pfad` und `datei` benennen einen Ressource auf dem Rechner, wobei meistens der Pfad zu einer Datei im Dateisystem des Webservers genannt wird. Mit dem Parameter `abfrage` können Parameter an die auf dem Server angesrochene Ressource übermittelt werden. `fragment` dient dazu geziehlt Positionen in großen Dokumenten zu adressieren.

URNs und URIs lassen sich syntaxtisch nicht unterscheiden. Es ist also notwendig den Schemateil der URI zu interpretieren und dann zu entscheiden, ob es sich um ein URN- oder URL-Schema handelt.

Die URL `http://www.informatik.uni-hamburg.de/aktuelles/index.html` kennzeichnet also eine URL des http-Schemas. Hier wird auf dem Rechner `www.informatik.uni-hamburg.de` die Datei `index.html` im Verzeichnis `/aktuelles` adressiert.

Neben den beschriebenen absoluten URLs gibt es auch noch relative URLs. Die in relativen URLs gemachten Angaben werden in Abhängigkeit von der URL des Dokuments in dem sie definiert wurden interpretiert. Wenn z.B. das unter der Adresse `http://www.xyz.de/a/a.htm` einen Link zur URL `../b/b.htm` enthält, dann wird diese URL vom Web-Browser zur absoluten URL `http://www.xyz.com/b/b.htm` umgesetzt.

Prinzipiell kann es für ein Dokument eine Vielzahl von URLs geben, was im Zusammenhang mit dem Betrieb von Web-Robots zu Problemen führen kann. Die URL `http://www.informatik.uni-hamburg.de` entspricht z.B. den URLs `http://www.informatik.uni-hamburg.de:80`, `http://www.informatik.uni-hamburg.de/index.html` und `http://rzdspc77.informatik.uni-hamburg.de`.

### 2.1.2 Hypertext Transport Protocol

Die Kommunikation zwischen Web-Client und Server wird auf der Ebene der Applikations-Schicht des ISO/OSI Modells über das Hypertext Transport Protocol abgewickelt. Http bietet einen einfachen Mechanismus um Dateien von Servern zu laden, aber auch weitergehende Funktionalität, die das Publizieren von Informationen auf Web-Servern vereinfacht. Basierend auf dem Internet-Protokoll TCP, das wiederrum auf dem Protoll IP aufsetzt, definiert HTTP ein textbasiertes Request-Response Protokoll. Details über TCP und IP finden sich in [Ste94]. Die folgenden Ausführungen beziehen sich auf HTTP/1.1, dessen genaue Spezifikation in [FGM$^+$99] nachzulesen ist. Die Unterschiede zur vorherigen Version 1.0 dieses Protokolls, die in [BLFF96] spezifiziert wurde, werden am Ende dieses Abschnitts aufgezeigt.

Abbildung 2.2 zeigt den grundlegenden Abblauf der HTTP-Kommunikation. Der Client schickt einen Request an den Server und dieser Antwortet mit einer entsprechenden Response. Es können aber auch noch weitere Stationen, wie z.B. ein Proxy-Server dazwischengeschaltet sein. Für einen Proxy-Server gibt es sehr unterschiedliche Anwendungen, so kann man damit einen Dokumenten-Cache realisieren oder die Kommunikation zwischen Client und

Server filtern und manipulieren. Bei Verwendung eines Proxy-Server kommunizieren Client und Server nicht direkt miteinander, sondern jeweils mit dem Proxy-Server als Vermittler. Abbildung 2.3 verdeutlicht diesen Vorgang.

Das zustandslose Protokoll zeichnet sich durch sehr geringen Overhead aus. Request und Response bestehen jeweils aus einem Header in ASCII Darstellung, an den sich eventuell in einem MIME codierten Body weitere Informationen, wie z.B. ein HTML Text oder ein Bild, anschließen. Das in Request oder Response als Nutzlast übertragene Objekt wird in Anlehung an die Darstelltung in [Wil99] als Entity bezeichnet.

Das erste Element eines Requests ist die Bezeichnung der Methode, die ausgeführt werden soll. Es folgen optionale Felder, die die auszuführende Operation näher beschreiben, wie z.B. die Angabe der URI, auf die sich die Operation bezieht.

Ein HTTP-Request hat folgende EBNF-Syntax:

```
<Request>          := <SimpleRequest> | <FullRequest>
<SimpleRequest>    := GET <URI> CrLf
<FullRequest>      := <Method> <URI> <ProtocolVersion> CrLf
                      [<HTRQ Header>]
                      [CrLf <data>]
<Method>           := GET | POST | some others
<ProtocolVersion>  := HTTP/1.x
<URI>              := as defined in URL spec
<HTRQ Header>      := Fieldname : Value CrLf
<data>             := MIME-conforming-message
```

Für die verschiedenen Operationen sind im HTTP-Protokoll verschiedene Methoden vorgesehen. Tabelle 2.1 listet die Gebräuchlichsten auf.

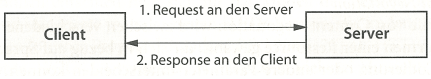


**Abbildung 2.2:** Grundlegende Funktionsweise von HTTP. Quelle: [Wil99]

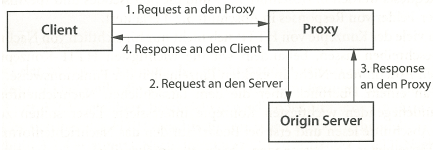


**Abbildung 2.3:** Kommunikation zwischen Client und Server mit einem zwischengeschalteten Proxy-Server. Quelle: [Wil99]

**Tabelle 2.1:** Übersicht über die für HTTP definierten Methoden

| METHODE | BEDEUTUNG |
|---|---|
| GET | Fordert ein durch die angegebene URI identifiziertes Dokument vom Server an. |
| HEAD | Entspricht der Methode GET, mit der Ausnahme, daß die Response nur einen Header enthält. Diese Methode ist z.B. geeignet, um eine URI auf Gültigkeit zu überprüfen. |
| PUT | Die im Body des Requests befindlichen Daten werden im unter der angegebene URI liegenden Dokument gespeichert. |
| POST | Die im Body des Requests befindlichen Daten werden an das unter der angegebenen Uri liegende Dokument angehängt. Dies war ursprünglich mal für Newsgroups gedacht,wird aber in der Praxis meistens zum Übertragen umfangreicher Formulardaten genutzt. |
| DELETE | Entfernt das unter der angegebenen URI liegende Dokument |

Auf eine Anfrage antwortet der Server mit einer HTTP-Response, die die folgende Syntax hat:

```
<status line>   :=  <http version>  <status code>  <reason line> CrLf
<http version>  :=  3*<digit>
<status code>   :=  3*<digit>
<digit>         :=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<reason line>   :=  {printable}
```

In Request und Response können eine Reihe optionaler Parameter übermittelt werden. Tabelle 2.2 listet einige der optionalen Parameter und ihre Bedeutung auf. Viele dieser Felder sind bei der Programmierung des Robots zu beachten. In Übereinstimmung mit der Darstellung in [Wil99] sind die Parameter in vier unterschiedliche Gruppen unterteilt:

- *General Header*
  Derartigen Header-Felder finden sowohl in Request-, als auch in Response-Nachrichten Verwendung. Die Werte haben keinerlei Auswirkung auf die übermittelte Entity.

- *Entity Header*
  In einem Entity Header werden Meta-Informationen zu der übermittelten Entity dargestellt. Wird keine Entity übertragen, dann beschreiben diese Parameter die im Request angegebene Ressource. Dies ist zum Beispiel der Fall, wenn bei einem Request die Methode HEAD verwendet wird.

- *Request Header*
  Informationen über den Request, die nicht in Zusammenhang mit den im Nachrichtenkörper übermittelten Daten stehen.

- *Response Header*
  Außer den Statuscodes können in diesen Feldern noch weitere Informationen zu der Response übertragen werden, die sich nicht auf die im Nachrichtenkörper übermittelten Daten beziehen.

In einem Response wird eine dreistellige Zahl als Statuscode übertragen, der angibt, ob die Übertragung erfolgreich war, bzw. welche Art von Fehler vorlag. Die Statuscodes werden in fünf Gruppen unterteilt:

- `Informational (1xx)`
  Der Request wurde vom HTTP-Server verstanden und wird jetzt bearbeitet. Ein Reponse dieser Klasse hat nur vorläufigen Charakter, da der Response mit den angeforderten Daten noch gesendet wird.

- `Successful (2xx)`
  Wenn der Server den Request empfangen, verstanden und akzeptiert hat, sendet er einen Statuscode dieser Klasse zurück. Der Statuscode `200 (ok)` wird bei einem fehlerlos verarbeiteten Request gesendet.

- `Redirection (3xx)`
  Statuscodes dieser Klasse geben an, daß der Client zum Vervollständigen des Requests weitere Maßnahemen ergreifen muß. Diese können darin bestehen, daß er einen Request an einen anderen, in der Response genannten Server schickt. Für den Einsatz von Web-Robots ist aus dieser Klasse insbesondere der Code `304 (not modified)` von Bedeutung. Wenn der Client im Header eines Requests das Feld `if-modified-since` mit einem Datum sendet und eine Response mit Statuscode `304` zurückkommt, dann wurde das Dokument seit dem im Header genannten Datum nicht mehr modiefiziert und muß nicht erneut vom Server geladen werden.

- `Client Error (4xx)`
  Sollte der Request nicht bearbeitet werden können,weil der Client einen Fehler gemacht hat, wie z.B. einen Syntaxfehler im Requst, dann sendet der Server einen Statuscode dieser Klasse. Statuscode `400 (bad request)` bedeutet, daß der Client einen syntaktisch ungültigen Request gesendet hat. Beim unauthorisierten Zugriff auf geschützte Dokumente wird Code `401` zurückgegeben.

- `Server Error (5xx)`
  Sollte der Server nach Empfang eines korrekten Requests nicht in der Lage sein diesen auszuführen, dann Antwortet er mit einem Statuscode dieser Klasse.

**Tabelle 2.2:** Übersicht über einige Header-Felder von HTTP/1.1

| Header-Feld | Typ | Bedeutung |
|---|---|---|
| `Accept` | Request | Legt fest, welche Medientypen für den Response akzeptiert werden. So kann ein Client z.B. die Angabe machen, daß er nur HTML-Dokumente akzeptiert. Nähere Informationen zu den Medientypen finden sich in Abschnitt 2.1.5. |
| `Accept-Language` | Request | Mit diesem Feld kann der Client Präferenzen für die Sprache des angeforderten Dokuments ausdrücken. Bei nicht sprachspezifischen Inhalten wie Bildern ist dieses Feld bedeutungslos. Die Bestimmung der Sprache eines Dokuments erfolgt mit Hilfe von Sprach-Tags, wie sie in [Alv95] definiert sind. |
| `Accept-Ranges` | Response | Mit diesem Feld kann der Server anzeigen, daß er Bereichsrequests für eine Ressource akzeptiert, ein Client also nur Teile einer Datei anfordern kann. |
| `Authorization` | Request | Clients verwenden dieses Feld um sich beim Server zu authentifizieren. |
| `Connection` | General | Mit diesem Parameter können sowohl Client als auch Server Optionen festlegen, die für die Verbindung gelten sollen. Mit dem Wert `close` kann der Abbruch der Verbindung nach erfolgter Übertragung angeforder werden. |
| `Content-Encoding` | Entity | Dieses Feld gibt an, welches Codierungsverfahren auf den Message Body angewendet wurde. |
| `Content-Language` | Entity | Macht Angaben über die Sprache der im Message-Body gesendeten Entity. |

| Header-Feld | Typ | Bedeutung |
|---|---|---|
| `Content-Lenght` | Entity | Die Länge des Message Bodys in Byte. Bei Verwendung von persistenten Verbindungen ist diese Angabe obligatorisch, da nicht am Verbindungsabbruch, da Ende des Message Bodys erkannt werden kann. |
| `Content-Type` | Request | Medientyp des im Message Body enthaltenen Entity. |
| `Date` | General | Entstehungsdatum der im Message Body enthaltenen Entity. |
| `Expire` | Entity | Angabe von Datum und Uhrzeit, nach der ein Entity als veraltet betrachtet werden kann. Nach diesem Zeitpunkt sollte kein Cache mehr eine Kopie dieses Dokuments zurückgeben. |
| `If-Modified-Since` | Request | Mit diesem Feld lassen sich bedingte Requests erstellen, die den Server nur dann veranlassen ein Dokument zu übertragen, wenn es seit dem angegebenen Datum modifiziert wurde. In diesem Fall entspricht ein derartiger Request einem gewöhnlichen `GET` Request. Wurde das Dokument nicht geändert, dann wird in der Response der Statuscode `304 (not-modified)` übertragen. |
| `Last-Modified` | Entity | Der Zeitpunkt der letzten Änderung der im Message Body enthaltenen Entity. |
| `User-Agent` | Request | Hier werden Informationen über die den Request aussendende Software übertragen. Meistens wird hier der Typ und die Version des Web-Browsers übertragen. Mit diesem Feld kann sich aber auch z.B. ein Robot identifizieren. Dieser Parameter ermöglicht es auf die spezifischen Fähigkeiten bestimmter Client-Software abgestimmte Inhalte zu übertragen. |

```
C: GET /hypertext/WWW/TheProject.html HTTP/1.0
C:
S: HTTP/1.0 200 Document follows
S: MIME-Version: 1.0
S: Server: CERN/3.0
S: Content-Type: text/html
S: Content-Lenght: 8247
S:
S: <HEAD><TITLE>The World Wide Web Consortium (W3C)</TITLE></HEAD>
S: <BODY>
S: <H1>The World Wide Web Consortium</H1>
S: ...
S: </BODY>
```

**Abbildung 2.4:** Exemplarischer Ablauf eines Dateitransfers per HTTP.

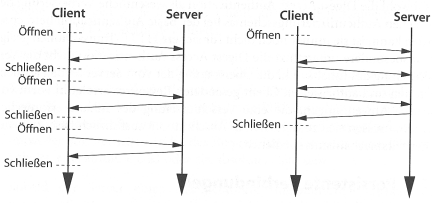


**Abbildung 2.5:** Persistente HTTP-Verbindungen. Quelle: [Tan97]

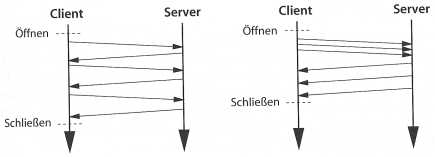


**Abbildung 2.6:** HTTP Request Pipelining. Quelle: [Tan97]

Abbildung 2.4 zeigt ein Beispiel für einen Dateiabruf per `GET` von einem Webserver. Es wurde in gekürzter Form [Tan97] entnommen und bezieht sich noch auf Version 1.0 des HTTP-Protokolls. Details zum HTML-Dokument folgen in Abschnitt 2.1.3.

**Neuerungen in HTTP/1.1** Gegenüber der Vorgängerversion HTTP/1.0 bietet die Nachfolgeversion eine Reihe von neuen Funktionen. Insbesondere ist die Unterstützung des Header Felds `Host` zu nennen, mit dem sich unter einer IP-Adresse mehrere virtuelle Server installieren lassen.

Die Darstellung eines HTML-Dokuments macht in vielen Fällen mehrere Interaktionen mit dem Server erforderlich, da neben der HTML-Datei vielfach auch noch eingebettete Bilder, Scripte und Style-Sheets nachgeladen werden müssen. HTTP/1.0 erlaubt nur eine Request-Response Interaktion pro TCP-Connection. Dies führt zu sehr viel Overhead, da ständig neue Verbindungen aufgebaut werden müssen. Der langsame Startmechanismus von TCP/IP wirkt sich in diesem Fall nachteilig auf die Performance aus. Zur

Effizienzsteigerung haben einige Browser parallel mehrere TCP-Connections geöffnet, aber dies ist mit dem Nachteil Verbunden, daß sowohl im Client, als auch im Server viele Betriebssystem-Ressourcen gebunden werden.

Die neue Funktion der persistenten TCP-Verbindungen macht die Kommunikation zwischen Client und Server deutlich effizienter, als das bei HTTP/1.0 der Fall war. Nun wird nicht mehr für jede Request/Response-Interaktion eine neue Verbindung aufgebaut, sondern die bestehende Verbindung offen gehalten, so daß nachfolgende Interaktionen mit dem selben Server, z.B. zum Laden eingebetteter Grafiken, ohne zusätzlichen Overhead über die bestehende Verbindung erfolgen können (Siehe Abbildung 2.5). Verbunden mit den persistenten Verbindungen ist die Möglichkeit des Request-Pipelinings. So können über eine offene Verbindung mehrere Requests geschickt werden, ohne das jeweils die Responses abgewartet werden. Abbildung 2.6 zeigt die Request-Response Interaktion mit Request-Pipelining. Es ist dabei wichtig, daß der Server die Requests in der Reihenfolge ihres Eintreffens bearbeitet, da die Zuordnung zwischen Request und Response im Client allein durch die Reihenfolge der Aussendung erfolgt.

### 2.1.3 Hypertext Markup Language

Für die Web-Seiten wird die *Hypertext Markup Language (HTML)* verwendet. Diese Sprache wurde basierend auf der *Standard Generalized Markup Language (SGML)* mittels einer *Document Type Definition (DTD)* definiert. Die Grundidee ist die Trennung von Inhalt und Darstellung eines Dokuments. Markup Sprachen, wie SGML und die SGML konforme Hypertext Markup Language verwenden zur expliziten Darstellung der Dokumentenstruktur *Elemente*, die als *Tags* in das Dokument eingebettet werden. HTML-Tags bestehen jeweils aus einem Start-Tag, der mit dem Zeichen `<` eingeleitet wird und einem End-Tag, der mit der Zeichenfolge `</` beginnt. Tags haben eindeutige Bezeichner und können teilweise parametrisiert werden. Zwischen Start- und End-Tag befindet sich der Teil des Dokuments, auf den sich die durch das Tag gemachte Angabe bezieht. Einige Tags dürfen auch geschachtelt werden. Entsprechend der durch die Tags gemachten Vorgaben wird dann das Dokument vom Browser formatiert und dargestellt.

Die erste offizielle Version war HTML 2.0 [BLC95]. In dieser frühen Version umfaßte die Sprache neben den bereits in den ersten Prototypen realisierten Tags zur Auszeichnung von Überschriften, Listen, Einbettung von Bildern und Links eine Reihe von Erweiterungen, die durch die Browser Mosaic und Arena gemacht wurden. HTML 2.0 blieb lange Zeit die einzig gültige HTML Spezifikation, denn der als eine der ersten Empfehlungen des World Wide Web Konsortiums 1994 veröffentlichte Entwurf für HTML 3.0 wurde niemals verabschiedet. Neue Browser-Funktionen beinhalteten in der Folge

Features von HTML 3.0 und eine Reihe proprietärer Erweiterungen. 1997 wurde HTML 3.2 freigegenen. Diese Sprachversion umfaßte nun Tabellen, Applets, Textfluß um Bilder und eine Reihe weiterer Funktionen, die von den damals aktuellen Browser-Versionen bereits implementiert wurden. HTML 3.2 entsprach bereits zum Zeitpunkt der Verabschiedung nicht mehr dem aktuellen Stand der Technik, so daß bereits im Dezember 1997 die erste Version der Recommendations für HTML 4.0 [RHJ98] veröffentlicht wurden. Zu den Neuerungen von HTML 4.0 zählen Funktionen zur Internationalisierung, Unterstützung von Style Sheets, Frames, ein verbesssertes Tabellenmodell, Unterstützung für die Einbindung allgemeiner Multimedia-Objekte und einer Erweiterung der Formular-Funktionen.

Die Grundidee von HTML durch die Auszeichnung der Dokumentenstruktur ohne explizite Layoutangaben die Darstellung auf verschiedenen Endgeräten, vom ASCII-Terminal bis hin zur Grafik-Workstation, möglich zu machen, steht in Konflikt mit den Interessen vieler Web-Designer, eine vollständige, pixelgenaue Kontrolle über das Layout der von ihnen gestalteten Seiten zu erlangen. Durch die Erweiterung des Sprachumfangs um Tags zur Layout-Kontrolle und Gestaltungstechniken, wie sie unter anderem in [Sie96] beschrieben werden, ist die Geräteunabhängigkeit der Darstellung vieler Web-Sites nicht mehr gewährleistet. Die mobile Internetnutzung mit Personal Digital Assitants (PDAs) und Smartphones, deren grafische Fähigkeiten deutlich unter denen aktueller stationärer Computer liegen, hat in den letzten Jahren stark zugenommen. Durch diesen Trend kommt einer von den Eigenschaften der Endgeräte unabhängigen Darstellung der Inhalte eine größere Bedeutung zu. Langfristig könnten in der *Extensible Markup Language (XML)* geschriebene Dokumente, die dann mittels der Style-Sheet Sprache *Extensible Style Language (XSL)* für verschiedene Klassen von Endgeräten formatiert werden, an Akzeptanz gewinnen.

Ein Html Dokument ist in einen Head und einen Body Bereich aufgeteilt. Im Head eines Dokuments, also zwischen den Tags `<HEAD>` und `</HEAD>` werden Angaben über das Dokument, wie z.B. der Titel gemacht. Im Body, der von den Tags `<BODY>` und `</BODY>` eingeschlossen wird, befindet sich der Inhalt des Dokuments. Unter dem Aspekt der automatischen Verarbeitung von HTML Dokumenten sind insbesondere der Tag `<TITLE>` mit Angaben zum Seitentitel, aber auch die Meta-Tags mit Meta-Informationen zum Dokument, wie Stichworte, Beschreibung und Autor von Interesse. Die verschiedenen Arten in einem HTML Dokument Hyperlinks zu anderen Dokumenten zu plazieren werden im folgenden Abschnitt 2.1.4 vorgestellt. Auf weitere Details von HTML wird im Folgenden nicht eingegangen, diese sind entweder in den Spezifikationen des World Wide Web Konsortiums [RHJ98] oder in dem Online-Tutorial SelfHtml [Mün01] nachzulesen.

Abbildung 2.7 zeigt ein HTML-Dokument, in das mittels `<IMG>` Tag ein Bild

```
<HTML>

     <HEAD>
          <TITLE>Homepage von Frank Wollenweber</TITLE>
     </HEAD

     <BODY>
          <H1>Frank Wollenweber</H1>
          <IMG SRC="/bilder/foto.jpg">
          <P>
          Hier könnte jetzt eine ganze Menge Text stehen.<BR>
          <A HREF="http://www.informatik.uni-hamburg.de">
               Fachbereich Informatik
          </A>
          <A HREF="http://vsis-www.informatik.uni-hamburg.de">
               Arbeitsbereich VSIS
          </A>
          </P>
     </BODY>
<HTML>
```

**Abbildung 2.7:** Beispiel einer HTML-Datei.

integriert wurde und das Links zu anderen Seiten enthält.

### 2.1.4 Arten der Realisierung von Links

Wie in Kapitel 2.2 ausführlich beschrieben wird, verfolgen Web-Robots automatisch die in Hypertexten vorhandenen Links. Links stellen eine Verknüpfung zwischen zwei Dokumenten her. Im WWW sind dabei nur unidirektionale Links vorgesehen. Sie sind in ein Dokukment eingebettet und verweisen auf genau ein anderes Dokument, das durch seine URL adressiert wird. Es ist nicht ersichtlich, welche Dokumente Links zu einem bestimmten anderen Dokument enthalten. Befindet sich das verlinkte Dokument auf dem gleichen Server, so spricht man von einem *internen Link*, anderenfalls von einem *externen Link*.

Der Sprachumfang von HTML, sowie die offene Architektur des WWW, haben dazu geführt, daß es eine Vielzahl von Möglichkeiten gibt, einen Hyperlink zu realisieren. Für das Verständnis der Funktion eines Robots ist es deshalb notwendig sich zunächst einmal einen Überblick über die verschiedenen Typen von Links zu verschaffen, die im Kontext des World Wide Webs zu finden sind.

**Standard-Link** Im Normalfall werden Links in HTML Dokumenten über das `<A>`-Tag realisiert. Die zwischen `<A>` und `</A>` eingeschlossenen Elemente können vom Benutzer angeklickt werden. Der Browser lädt dann die im Parameter `href` angegebene URL.

Im Beispiel `<A HREF="http://www.informatik.uni-hamburg.de/aktuelles/index.htm">FB Informatik</A>` wird ein Link zu einer absoluten URL angegeben. `<A HREF="../lehre/uebersicht.htm">...` verlinkt eine Seite auf dem selben Server mittels einer relativen URL.

**Imagemap** Teile einer Grafik können mittels *Imagemaps* als Hyperlinks gekennzeichnet werden. Mittles `AREA`-Tags werden Polygonflächen einer Grafik verlinkt. Die mittels einer *Client-Side Imagemap*, wie sie im nachfolgenden Beispiel dargestellt ist, realisierten Links können prinzipiell auch durch einen Robot verfolgt werden. *Server-Side Imagemaps* entziehen sich jedoch der Auswertung durch Robots, da in diesem Fall die eigentlichen Links mit einem CGI-Skript realisierten werden, das die Koordinaten der Clicks auf die Grafik übermittelt bekommt und dann das entsprechende Dokument sendet.

Beispiel einer Client-Side Imagemap:

```
<IMG src="map.gif" ismap usemap="#map">
<MAP name="map">
     <AREA shape="RECT" coords="10,15,50,7"
           href="seite1.htm">
     <AREA shape="RECT" coords="80,100,200,140"
           href="seite2.htm">
</MAP>
```

**Formular** HTML Dokumente können Formulare enthalten. Im Attribut `action` des `Form`-Tags wird eine URL spezifiziert, an die die Formulardaten gesendet werden. Bei Verwendung der Formularmethode `GET` werden die Parameter in der URL codiert. Dies läßt sich auch mit einem mit einem `<A>`-Tag simulieren.

**Frameset** Framesets teilen das Browserfenster in mehrere Bereiche auf, in denen dann separate Dokumente geladen werden. Framesets können beliebig verschachtelt werden. Aus Sicht des Anwenders stellt die Gesamtheit der in einem Frameset angezeigten Dateien ein Dokument dar. Für einen Robot, der eine derartige Struktur besucht, handelt es sich jedoch um eine Menge separater Dokumente, die jeweils eigenständig erfaßt werden. So kann es dazu kommen, daß Anfragen an Suchmaschinen Links zu Dokumenten liefern, die deren Autoren eigentlich nur für die Anzeige in einem bestimmten Frameset vorgesehen hatten.

**Weiterleitung** Für Weiterleitungen gibt es prinzipiell zwei Möglichkeiten. Das HTTP-Protokoll sieht eine Möglichkeit vor, auf Requests mit einem Redirect zu antworten und dabei eine neue URL zu übermitteln, von der das angeforderte Dokument dann geladen werden kann. Ein Besipiel ist der HTTP-Statuscode 302. In diesem Fall spricht man auch von *Server Redirection*. Eine Alternative dazu ist die Verwendung des Meta-Tags `http-equiv=Refresh` in einem HTML-Dokument. Nach dem Laden einer Seite mit einem derartigen Element lädt der Browser die in diesem Meta-Tag spezifizierte URL. Beide Formen von Weiterleitungen lassen sich prinzipiell auch von Robots benutzen, haben aber den Nachteil, daß der Robot möglicherweise URLs abspeichert, an denen sich keine Inhalte mehr befinden.

```
<META http-equiv=Refresh content=10; URL=target.html>
```

**`LINK`-Tag** Mit dem `link`-Tag lassen sich Relationen zwischen Dokumenten ausdrücken. Die Art der Darstellung ist browserabhängig. In der Praxis wird dieses Element oftmals zur Verknüpfung eines Dokuments mit einem Style-Sheet verwandt. Interessant erscheint die Möglichkeit mittels typisierter Links explizit Angaben über Abhängigkeiten zu anderen Dokumenten eines Hypertextes, z. B. zu Folgeseiten, zu machen. Obwohl der `link`-Tag bereits seit HTML 2.0 Teil der Spezifikation ist, werden die weitergehenden Anwendungsmöglichkeiten momentan von keinem Browser unterstützt. Im aus [Mün01] entnommenen Beispiel ist ersichtlich wie ein Dokument mittels `link`-Tags in den Site-Kontext integriert werden kann.

```
<head>
<link rel="contents" href="inhalt.htm" title="Inhaltsverzeichnis">
<link rel="index" href="stichwort.htm" title="Stichwortverzeichnis">
<link rel="glossary" href="glossar.htm" title="Begriffs-Glossar">
<link rel="next" href="augsburg.htm" title="nächste Seite">
<link rel="previous" href="aachen.htm" title="vorherige Seite">
  <!-- ... andere Angaben im Dateikopf ... -->
</head>
```

**JavaScript** Mit in HTML-Dokumenten enthaltenen JavaScript Programmen können ebenfalls Links realisiert werden. JavaScript Programme können belibig komplex sein, so daß es sich nicht automatisch ermitteln läßt, ob ein Link ausgeführt wird und zu welche Adresse gesprungen wird.

**Java** Mit Java-Applets lassen sich ebenfalls Links realisieren. Auch für ein Java-Applet läßt sich nicht automatisch analysieren, welche Links angesprungen werden.

**ActiveX** Über die ActiveX Schnittstelle läßt sich der Microsoft Internet Explorer mit Funktionen des Windows Betriebssystems verbinden. Ebenso wie Java-Applets entziehen sich auch die ActiveX-Controls der automatischen Verarbeitung durch Robots.

**Plugins** Für Web-Browswer gibt es eine Vielzahl von Plugins, die die Darstellungsmöglichkeiten der Browser um verschiedene Medientypen erweitern. Einige Plugins, wie z.B. das Flash-Plugin, mit dem sich interaktive multimediale Darstellungen, realisieren lassen, enthalten ebenfalls die Möglichkeit auf andere Dokumente zu verweisen.

### 2.1.5 Mime-Types

Die Architektur des World Wide Webs ermöglicht es verschiedene Datei-Typen in Hypertextdokumente einzubetten und mittels des Hypertext Transfer Protocols zu übertragen (siehe 2.1.2). Ein Programm, das Dokumente aus dem Web lädt und diese verarbeitet, muß deshalb erkennen können, welchen Typ ein geladenes Dokument hat, um dann zu entscheiden, wie mit dem geladenen Dokument zu verfahren ist. Die Erkennung des Datei-Typs ist insbesondere für einen Web-Robot unerläßlich, da dessen Parser in der Regel nur bestimmte Dateitypen verarbeiten kann. Die ursprünglich für E-Mails konzipierten *Multipurpose Internet Mail Extensions (MIME)* werden auch zur Codierung und Kennzeichnung der per HTTP übertragenen Entitäten genutzt. Mit dem im HTTP-Header übertragenen Feld `Content-Type` läßt sich der Dateityp einer übertragenen Entität bestimmen. Die Angabe erfolgt aufgeteilt in Medientyp und Subtyp. Der Medientyp gibt den generellen Datentyp an, während der Subtyp das Format für diesen Datentyp angibt. Ein Beispiel ist der Mime-Type `image/gif`, der ein Bild (Image) kennzeichnet, welches im *Graphics Interchange Format (GIF)* codiert wurde. Details finden sich in den RFCs [FB96a, FB96b, Moo96, FKP96, FB96c].

## 2.2 Robots

Im Zentrum dieser Studienarbeit steht die Entwicklung eines Web-Robots. Die nachfolgenden Abschnitte vermitteln Grundlagen zu diesem Thema, auf denen dann in den weiteren Kaptiteln aufgebaut wird. Einen guten Überblick über die Technologie der Web-Robots liefert [Che96].

### 2.2.1 Definition

In dieser Arbeit werde ich die Definition von *Web-Robot*, die Martijn Koster in seinem Aufsatz „Robots in the Web: threat or treat?“ [Kos95] gegeben hat, verwenden.

> A Web robot is a program that traverses the Web's hypertext structure by retrieving a document, and recursively retrieving all documents that are referenced. These programs are sometimes called „spiders“, „web wanderers“, or „web worms“. These names, while perhaps more appealing, may be misleading, as the term „spider“ and „wanderer“ give the false impression that the robot itself moves, and the term „worm“ might imply that the robot multiplies itself, like the infamous Internet worm. In reality robots are implemented as a single software system that retrieves information from remote sites using standard Web protocols. (Zitat: [Kos95])

Robots sind also Programme, die automatisch, d. h. ohne menschliche Steuerung, alle von einem Start-Dokument aus durch rekursive Verfolgung der Hyperlinks erreichbaren Dokumente laden. Dazu schicken sie per HTTP (vgl. 2.1.2) Anfragen an Web-Server und extrahieren aus den empfangenen Dokumenten alle Links, die dann wiederum geladen werden. Web-Roboter automatisieren den manuellen Umgang eines Menschen mit einem Web-Browser. Derartige Programme werden in verschiedenen Quellen auch vielfach mit anschaulichen Begriffen, wie *Spider*, *Wanderer*, *Crawler*, *Worm* oder *Bots* bezeichnet. Diese Bezeichnungen suggerieren fälschlicherweise, daß sich der Robot selbst durch das Netz bewegt oder vervielfältigt. Ein Robot bewegt sich nicht zwischen den Sites, wie es ein in bösartiger Absicht programmierter Virus tun würde. Er besucht Server, in dem er Dokumente von ihm lädt. Koster nimmt eine weitere Ausdifferenzierung der Begriffe für automatisch das Web durchwandernde Programme vor, in dem er bei Worm betont, daß sie sich im Gegendsatz zu Robots replizieren und WebAnts verteilte, kooperative Robots bezeichnen.

Begrifflich nicht eindeutig ist die Abgrenzung zum Begriff der *Agenten*. Unter Agenten versteht man gemeinhin Programme, die selbstständig im Auftrag ihrer Benutzer klar definierte Aufgaben ausführen. In der Literatur werden Web-Robots teilweise der Klasse der *intelligenten Agenten* zugeordnet [Tur98b], was damit begründet wird, daß es sich um autonom agierende Softwareinheiten handelt, die Informationen aufnehmen und verarbeiten, wobei sie standardisiert mit ihrer Umgebung einschließlich anderer Agenten interagieren. Diese Charakterisierung mag für im unmittelbaren Auftrag des Benutzers agierende Robots treffend sein, für Robots die für Suchmaschinen das Web indizieren scheint diese Zuordnung jedoch sehr fraglich, da derartige Robots keine „intelligenten" Entscheidungen treffen, und eine Einordnung in die Kategorie der *primitiven Agenten* angemessen.

### 2.2.2 Anwendungen

Robots werden in einem breiten Spektrum von Anwendungen eingesetzt. Es bietet sich dabei eine Unterteilung danach an, ob die eingesetzten Robots serverseitig laufen, in diesem Fall spricht man von *server-side Robots* bzw. *Server-Agents* oder ob die Robots direkt auf dem Rechner des Benutzers ablaufen. Derartige Robots werden als *client-side Robots* oder *User-Agents* bezeichnet (vgl. [Eic95]).

Ein weiteres Unterscheidungsmerkmal ist die Art der Analyse der gefundenen HTML-Dokumente. Einfache Robots analysieren den HTML-Text nur auf einer syntaxtischen Ebene, während einige Agenten auch semantische Analysen der gefundenen Dokumente anstellen. Exemplarisch ist hier der in [DEW97] beschriebene Preisvergleichs-Agent „ShopBot" zu nennen.

Ihre wichtigste serverseitige Verwendung finden Robots im Bereich *Ressource Discovery*, also dem Auffinden von Informationsquellen im WWW. Sie bilden dabei die Kernkomponente von *Suchmaschinen*, da sie durch kontinuierliches Crawlen des Webs den Index einer Suchmaschine aufbauen und aktualisieren. Im Index werden die gefundenen Wörter zusammen mit Verweisen auf die jeweiligen Dokumente abgespeichert, so daß dann Suchanfragen effizient bearbeitet werden können. Desweiteren führen Suchmaschinen oft auch andere Datenstrukturen in denen beispielsweise Informationen über die Linkstruktur der gefundenen Dokumente gespeichert werden. Suchmaschinen sind von größter kommerzieller Bedeutung, denn angesichts der Fülle im Web angebotener Dokumente sind sinnvolle Recherchen vielfach nur noch mit ihrer Hilfe möglich, so daß Suchmaschinen zu den wichtigsten Einstiegspunkten ins Web geworden sind.

Die Leistung von Suchmaschinen, und damit ihre Attraktivität für den Benutzer, läßt sich nach Kriterien des *Information Retrival* bewerten. *Recall* und *Presision* sind maßgeblich von der Qualität des verwendeten Robots und

der Indexierungs-Algorithmen abhängig [KT00]. Da der Webgraph nicht zusammenhängend ist, man denke dabei nur an die vielen privaten Homepages, die zwar viele Links enthalten, die aber von niemandem verlinkt werden, bleiben viele Seiten im Web für die Suchmaschinen im verborgenen. Der Index einer Suchmaschine kann immer nur einen kleinen Teil des Webs abbilden, da die Skalierbarkeit der Suchmaschinen nicht mit dem Wachstum und der Dynamik des Webs schritthalten kann. Martijn Koster fast diesen Sachverhalt in [Kos95] mit den Worten „There is too much material and it's too dynamic" zusammen.

Ebenfals dem Aufbau eines Indexes dient der Einsatz eines Robots für *lokale Suchmaschinen*, die eine Suchfunktion in den Dokumenten einer Web-Site bereitstellen. Als Beispiel sind hier die Programm „ht://Dig" [1] und „Harvest" [2] zu nennen, welche auf vielen Web-Sites zur Realisierung einer lokalen Suchfunktion benutzt werden.

Zu den klassischen Robot-Anwendungen zählen die Spiegelung einer Web-Site, das *Mirroring* und die automatische Web-Site Wartung, also die automatsche Validierung aller HTML-Dokumente und der enthaltenen Links.

Interessant ist auch die Möglichkeit mittels Robots statistische und strukturelle Informationen über das World Wide Web zu gewinnen. Derartige Untersuchungen wurden u. a. in [Tur98a] und [BKM+00] dokumentiert.

Auch client-seitig gibt es eine große Bandbreite verschiedener Robot-Anwendungen. Robots ermöglichen es den Nutzern des WWW Änderungen auf sie interssierenden Seiten zu überwachen, und komplette Hypertexte abzuspeichern. Navigationstools, die dem Benutzer bei der Suche nach Informationen einen Überblick über die Hypertextstruktur geben sollen, verwenden vielfach Web-Roboter, um die Link-Umgebung eines Dokuments zu ermitteln.

Starken Bezug zur KI und Sprachverarbeitung haben Agenten, die Ergebnisseiten von Suchmaschinen weiterverarbeiten, im Internet automatisch Preisvergleiche [DEW97] anstellen oder wissenschaftliche Publikationen recherchieren [ME96].

### 2.2.3 Techniken zum Durchlaufen des WWW

Für das systematische Vorgehen der Robots beim Durchwandern des Webs werden Algorithmen der Graphentheorie verwandt. In diesem Zusammenhang wird das Web als ein gerichteter Graph modelliert, wobei die Knoten die Dokumente und die gerichteten Kanten die Links repräsentieren.

[1] www.htdig.org
[2] http://harvest.sourceforge.net/harvest/doc/

Der einer Hypertext-Struktur entsprechende Graph wird oftmals auch als Web-Graph bezeichnet. Die nachfolgende Darstellung setzt die Kenntnis der elementaren Begriffe und Definitionen der Graphentheorie voraus, welche ausführlich in [Tur96] beschrieben werden.

Für das Absuchen des Webs finden die Algorithmen *Tiefen-* und *Breitensuche* Verwendung. Da der Web-Graph Zyklen enthält ist es dabei unumgänglich eine Liste der bereits besuchten URLs zu verwalten.

Bei der *Breitensuche* werden zunächst alle Links der Startseite verfolgt und dann jeweils alle Links der gefundenen Seiten besucht. Dieser Vorgang wird bis zu einer bestimmten Link-Tiefe wiederholt. Es wird also erst die unmittelbare Nachbarschaft der Startseite vollständig abgesucht, bevor die Links der gefundenen Seiten bearbeitet werden. Dieser Algorithmus läßt sich recht einfach implementieren, wenn man für die Liste der noch zu besuchenden URLs einen FIFO-Warteschlange verwendet. Ein schematischer Algorithmus findet sich in Abbildung 2.8. Abbildung 2.9 verdeutlicht das Vorgehen anhand einer Grafik. Dabei sei darauf hingewiesen, daß es Dokumente gibt, zu denen es von der Startseite keinen Pfad gibt, und die somit auch nicht durch den Robot gefunden werden können. Breitensuche verteilt tendenziell die Last gleichmäßiger auf die verschiedenen besuchten Server als die nachfolgend beschriebene Tiefensuche. Die Auswirkungen auf Recall und Precision werden maßgeblich durch die Struktur der gefundenen Seiten bestimmt. Findet der Robot im Verlauf der Suche sehr schnell Seiten mit vielen Links, z. B. zu einem Thema zusammengestellte Link-Listen, dann tendiert die Breitensuche dazu viele Top-Level Seiten, beispielsweise nur die Inhaltsverzeichnisse mehrseitiger Dokumente, zu erfassen. Im allgemeinen wird dies die Recall Rate erhöhen und die Precision Rate senken. Kleinere Dokumentensammlungen mit geringer Tiefe werden durch die Breitensuche vollständig erschlossen und dabei die Precision Rate auf Kosten der Recall Rate erhöht (vgl. [Wic99]).

Aus der rekursiven Implementierung der Suche in einem Graphen ergibt sich der Algorithmus der *Tiefensuche*. Auf jeden gefundenen Link wendet man die Tiefensuche an, wobei der gefundene Link die neue Startseite darstellt und die maximale Link-Tiefe bei jedem rekursiven Aufruf um 1 verringert wird. Es wird also der gesamte Graph, auf den der erste Link der ersten Seite zeigt exploriert, bevor der nächste Link verfolgt wird (siehe Abbildung 2.10). Die direkte Nachbarschaft der Startseite wird dabei schnell verlassen und erst wieder indiziert, wenn der zuerst beschrittene Teilgraph vollständig durchsucht wurde. Neben der beschriebenen rekursiven Implementierung, ist es eine Realisierung mit einer LIFO-Warteschlange möglich. Dokumente mit großer Tiefe werden durch die Tiefensuche unter Umständen erst vollständig erschlossen, bevor Links nach außen verfolgt werden, so daß potentiell die Precision auf Kosten der Recall Rate erhöht wird (vgl. [Wic99]).

```
var niveau : array[1..max] of integer;

procedure breitensuche(G : Graph; startecke : Integer);
var
        i,j : Integer;
        W   : warteschlange of Integer;
begin
        Initialisiere niveau mit -1;
        niveau[startecke] := 0;
        W.einfügen(startecke);
        while W <> 0 do begin
                i := W.entfernen;
                for jeden Nachbar j von i do
                        if niveau[j] = -1 then begin
                                niveau[j] := niveau[i] +1;
                                W.einfügen(j);
                        end
        end
end
```

**Abbildung 2.8:** Schematischer Algorithmus *Breitensuche* (Quelle: [Tur96])

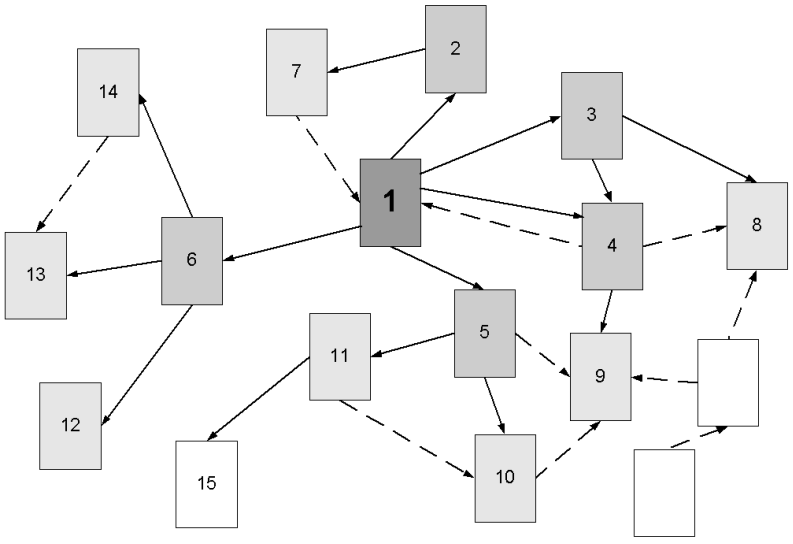


**Abbildung 2.9:** Breitensuche

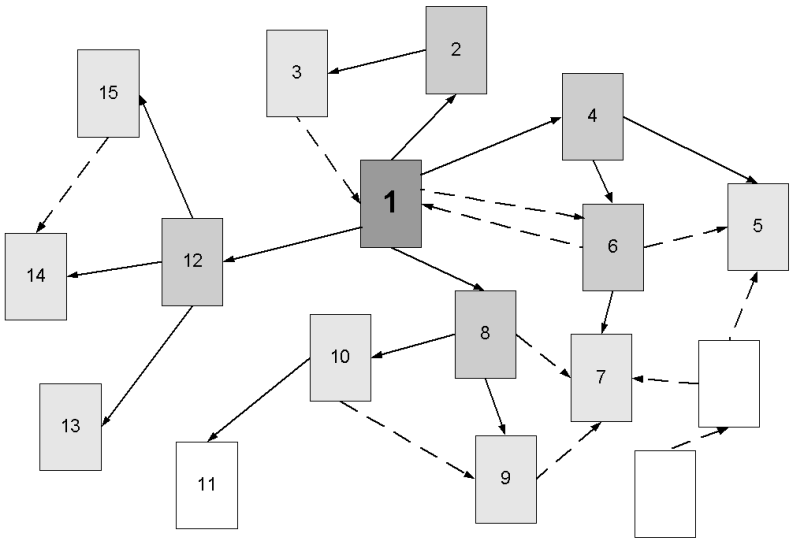


**Abbildung 2.10:** Tiefensuche

Zur Steigerung der Performanz laden einige Robots Dokumente nebenläufig aus dem Netz, dabei kann es natürlich sein, daß Dokumente unterschiedlich lange laden, und ein Teilgraph, der aus vielen kleinen Dokumenten besteht deutlich schneller und mit einer größeren Tiefe abgesucht wird, als ein anderer Teilgraph mit vielen großen Dokumenten. Von der durch den verwendeten Suchalgorithmus vorgegebenen Reihenfolge wird demnach vielfach abgewichen.

In der Praxis werden oftmals Mischformen beider Algorithmen verwandt, da das Web aufgrund seiner Größe nur unvollständig indexiert werden kann und die im Suchmaschinen-Kontext eingesetzten Robots deshalb vielfach nur selektiv Links verfolgen. Die Optimierung der Vorgehensstrategie von Web-Robots ist Gegenstand der Forschung.

In [CGMP98] werden verschiedene Techniken vorgestellt um durch geschickte Anordnung der Einträge der URL-Warteschlange möglichst schnell zu den „wichtigen“ Seiten zu gelangen. Unter anderem wird dort der *Backward-Count*, die Anzahl der auf eine Seite verweisenden Dokumente, verwendet um die Relevanz von Dokumenten zu bewerten. Diesem Ansatz liegt die Annahme zu Grunde, daß Dokumente, auf die viele Links verweisen, wichtiger sind, als Dokumente, die nur von wenigen anderen Seiten referenziert werden. Ähnliche Zusammenhänge lassen sich im Bereich wissenschaftlicher Publikationen herstellen, da in diesem Fall viel zitierte Beiträge als besonders relevant eingestufft werden. Eine Verfeinerung des Backward-Count stellt der *Page-Rank* dar. Ein Algorithmus, der von der Suchmaschine Google

zum Ranking der Ergebnisse einer Suchanfrage verwendet wird [PBMW98]. Einzelheiten zu Google finden sich in Abschnitt 3.2.

Ein weiterer Ansatz zur Optimierung des Crawl-Vorgangs wird in [CvdBD99] beschrieben.

### 2.2.4 Funktionsweise und typische Systemkomponenten

Die genaue Architektur der verschiedenen Robots ist natürlich sehr unterschiedlich, was sich bereits aus den sehr unterschiedlichen Anforderungen an Performanz und Skalierbarkeit der verschiedenen Systeme ergibt. Dennoch lassen sich einige grundlegende Komponenten bennen und ein prinzipieller Crawl-Algorithmus beschreiben. Der schematische Aufbau eines Robots findet sich in Abbildung 2.11.

Typische Systemkomponenten eines Robots sind:

- *URL-Warteschlange*
  In dieser Datenstruktur wird die Liste der noch zu besuchenden URLs gespeichert. Bei Verwendung der Breitensuche wird hier eine FIFO-Warteschlange verwendet. Tiefensuche wird mit einem Stack realisiert. Durch Modifikation der Zugriffsoperationen der URL-Warteschlange lassen sich verschiedene Vorgehens-Strategien implementieren.

- *DNS-Abfrage*
  Der Robot muß den *Domain Name Service* kontaktieren um die zu den Host-Namen zugehörige IP-Adresse zu bestimmen.

- *Komponente zum Download der Dokumente per HTTP*
  Ein Web-Robot enthält in der Regel mehrere Einheiten, die parallel per HTTP mit den Web-Servern kommunizieren und die Dokumente laden.

- *Parser*
  Um die in einem HTML-Dokument enthaltenen Links zu extrahieren, Seitentitel und Meta-Tags zu bestimmen, benötigt ein Robot einen Parser. Dieser erstellt aus jeder geholten HTML-Seite einen Ableitungsbau. Er muß alle HTML-Versionen und -Dialekte kennen. Da viele Dokumente nicht den HTML-Spezifikationen entsprechen, ist hohe Fehlertolleranz eine Anforderung an den Parser.

- *Liste der besuchten URLs*
  Da der Web-Graph Zyklen enthalten kann, ist es unumgänglich die besuchten URLs zu protokollieren um das mehrfache Laden eines Dokuments zu vermeiden.

- *Komponente zum Testen, welche Inhalte schon gesehen wurden*
  Server können unter verschiedenen Domain-Namen erreichbar sein, oder die gleichen Inhalte können an verschiedenen Stellen im Netz gespiegelt sein. Will man sicherstellen, daß Inhalte nur einmal indexiert werden, dann ist ein einfacher Vergleich der URLs nicht ausreichend. Für die Dokumente können sogenannte *Fingerprints* berechnet werden, die einen effizienten Vergleich zweier Texte auf Gleichheit ermöglichen.

- *Komponente zum Verarbeiten und Speichern der Dokumente*
  Die als Bestandteil einer Suchmaschine eingesetzten Robots verwalten einen Index, der zu jedem im Verlauf des Crawl-Vorgangs in einem Dokument gefundenen Wort einen Verweis auf alle das jeweilige Wort enthaltenden Dokumente beinhaltet. Abhängig von den spezifischen Aufgaben eines Robots können jedoch auch andere Datenstrukturen und Verarbeitungs-Algorithmen verwendet werden.

Der vereinfachte von einem Robot ausgeführte Algorithmus sieht folgendermaßen aus:

```
Eingabe: Liste von Start-URLs

1. nimm eine URL aus der URL-Warteschlange

2. bestimme die IP des Host

3. lade das Dokument

4. extrahiere die Links

5. normalisiere die Links (z.B. Umwandlung in absolute URLs

6. füge alle noch nicht geladenen URLs in die URL-Warteschlange ein
   und gehe zu 1.
```

### 2.2.5 Etiquette für Roboter

Robots mögen für so manchen Benutzer von großem Nutzen sein, für die Betreiber von Web-Servern kann ihre Benutzung jedoch auch negative Konsequenzen haben. Robots verursachen Traffic, der in der Regel von den Server-Betreibern zu bezahlen ist. Sie benutzen Bandbreite im Internet, die dann menschlichen Benutzern nicht mehr zur Verfügung steht und unzulänglich programmierte Robots können eine übermäßig hohe Serverlast hervorrufen.

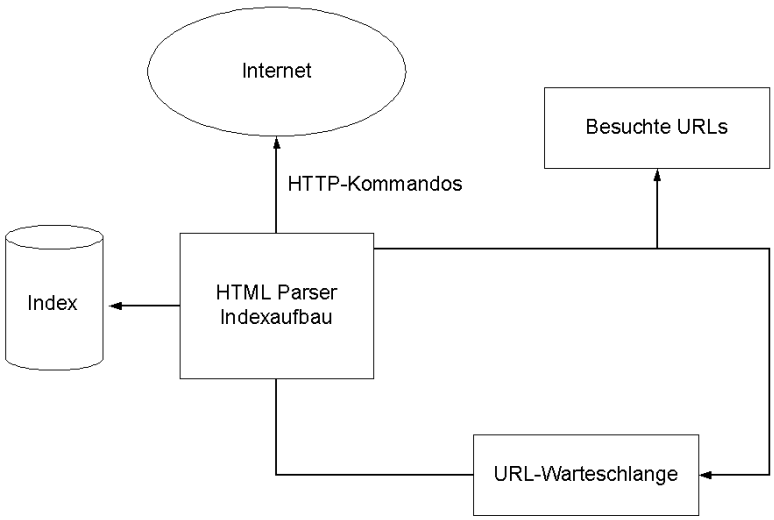


**Abbildung 2.11:** Schematischer Aufbau eines Web-Robots

Die Verfälschung der Serverstatistik stellt für Web-Site Betreiber und Werbetreibende ein Problem dar, da Werbung üblicherweise nach der Anzahl der Seitenzugriffe bezahlt wird. Beim Betrieb von Robots ist immer der Nutzen gegenüber den nachteiligen Auswirkungen abzuwägen. Es ist die Entscheidung zu treffen, ob die Kosten des Robot-Betriebs, die auf andere zukommen durch das Ergebnis gerechtfertigt werden. Mit dieser Problematik befaßt sich Martijn Koster in seinem Papier „Guidelines for Robot Writers“ [Kos93]. Er bennent dort folgende Regeln für Entwicklung und Betrieb von Web-Robots, die zwar keinen verbindlichen Charakter haben, in der „Internet-Community“ aber allgemein akzeptiert werden. Zu bedenken ist jedoch, daß in den letzten Jahren die im Internet zur Verfügung stehende Bandbreite stark ausgebaut wurde und sich Anwendungen etabliert haben, bei denen einzelne Benutzer sehr viel Bandbreite belegen. In Peer-to-Peer Filesharing-Diensten werden z. B. große Dateien mit MP3 Musikstücken oder Videos über das Internet ausgetauscht. Auch durch das Streaming von Videos und Tönen beanspruchen einzelne Benutzer viel Bandbreite. Das Argument der übermäßigen Inanspruchnahme von Netzwerkressourcen ist also durch die Entwicklung der letzten Jahre zu relativieren.

- Überdenke
  Man sollte grundlegend überdenken, ob man wirklich einen Robot gebraucht. Vielfach ist die Projekt-Idee gar nicht so neu und es besteht die Option die Ergebnisse anderer Projekte zu verwenden. Vor dem Einsatz eines Robots sollte man sich auch darüber bewußt werden, welche Datenmengen anfallen werden und welche Rechenleistung zur

Verarbeitung dieser Daten benötigt wird. Vielfach wird man zu der Erkenntnis kommen, daß es unrealistisch ist auch nur einen größeren Ausschnitt des Webs zu crawlen.

- Zeige dich verantwortlich
  Sollte der Robot Probleme verursachen, dann ist es notwendig, daß die Betreiber von Web-Servern den Robot-Betreiber kontaktieren können. Es ist deshalb anzuraten den Robot über das Feld `User-agent` des HTTP-Headers zu identifizieren. Im `From` Feld kann eine E-Mail-Adresse zur Kontaktaufnahme übergeben werden. Es ist sinnvoll den Robot in einschlägigen Newsgroups anzukündigen. Betreiber von Robots sollten während des Crawl-Vorgangs erreichbar sein, um bei eventuellen Beschwerden schnell reagieren zu können.

- Teste lokal
  Selbstverständlich sollte der Robot zuerst ausführlich lokal getestet werden, bevor er Anfragen an externe Web-Server schickt.

- Verschwende keine Ressourcen
  Ein Robot sollte Server nicht mit einer schnellen Abfolge von Anfragen überlasten. Koster empfielt maximal ein Dokument in der Minute von einem Server zu laden. Dieser Wert ist angesichts der Leistungsfähigkeit heutiger Web-Server sehr niedrig. Ein Robot sollte dennoch nicht mit maximaler Geschwindigkeit Dokumente von einem Server laden. Dies ist auch in vielen Anwendungen gar nicht notwendig, denn ein effizientes Vorgehen läßt sich auch erreichen, wenn der Robot so programmiert wird, daß er abwechselnd verschiedene Server anfragt. Sicherzustellen ist, daß ein Robot keine Ressourcen abfragt, die er nicht verarbeiten kann. Ein Robot, der nur mit HTML-Dokumenten umgehen kann, sollte dies durch einen entsprechenden Wert des Feldes `Accept` im HTTP-Header ausdrücken und URLs, die offensichtich nicht auf ein HTML-Dokument verweisen, da die Dateiendung z. B. „.gif“ oder „.zip“ ist, ignorieren. Es ist unerläßlich zu protokollieren, welche URLs schon besucht wurden um Schleifen zu vermeiden.

- Überwache den Robot
  Es ist unerläßlich den Betrieb des Robots kontinuierlich zu überwachen um Fehlfunktionen frühzeitig zu erkennen und entsprechende Korrekturmaßnahmen treffen zu können. Protokolle sollten die Aktionen des Robots dokumentieren.

- Lasse andere an den Resultaten teilhaben
  Sowohl die Rohdaten, als auch die aufbereiteten Daten eines Crawls sollten öffentlich zugänglich sein.

Den in dieser Arbeit thematisierten client-seitig arbeitenden Robots stünde Koster aus der Sicht von 1995 äußerst kritisch gegenüber, da aus ihrer Funktion nur ein einzelner Anwender Nutzen ziehe (vgl. [Kos95]). Dem wäre entgegenzusetzen, daß sich, wie vorstehend angemerkt wurde, Dienste etabliert haben, bei denen ebenfalls zum Nutzen einzelner Benutzer große Datenmengen übertragen werden. Scone-Anwendungen, die den in Kapitel 6 beschriebenen Robot benutzen, ließen sich auch auf Arbeitsgruppen-Servern installieren oder als Dienst im Web anbieten, so daß von den Aktivitäten des Scone-Robots mehrere Anwender profitieren würden.

> Where the use of a robot may be acceptable to the community if its data is then made available to the community, client-side robots may not be acceptable as they operate only for the benefit a single user. (Zitat: [Kos95])

Differenzierter beurteilt Eichmann den Einsatz von client-seitigen Robots in [Eic95], für deren Betrieb er folgende Richtlinien vorschlägt, die er als „User Agent Ethic“ bezeichnet:

- Identifikation
  Die Identität des Benutzers eines User-Agents sollte nachvollziehbar sein.
- Maßhalten
  Die Geschwindigkeit, mit der ein User-Agent Dokumente abfragt, sollte der Kapazität von Server und Netzerk-Anbindung angemessen sein.
- Angemessenheit
  Es sollten jeweils nur die für eine Aufgabe auch wirklich relevanten Server besucht werden. Ein Agent, der beispielsweise Suchanfragen an verschiedene Suchmaschinen richtet und die Ergebnisse aggregiert, sollte also anhand der Suchanfrage entscheiden, von welchen Suchmaschinen auch wirklich sinnvolle Ergebnisse zu erwarten sind.
- Wachsamkeit
  Ein User-Agent ist so zu programmieren, daß es dem Benutzer nicht möglich ist Anfragen zu stellen, die unabsehbare Konsequenzen haben.

Der Scone-Robot wurde erstellt, um eine Umgebung zu schaffen, in der Tools zur Unterstützung der Navigation im Web prototypisch implementiert und evaluiert werden können. Navigationstools bieten nur dann eine Unterstützung für den Benutzer, wenn sie ohne große zeitliche Verzögerungen beim „Surfen“ im Web eingesetzt werden können. Basieren solche Tools

auf einem Robot, dann ist es unumgänglich, daß dieser mit maximal möglicher Geschwindigkeit die Dokumente lädt, in Kauf nehmend, daß es dabei zu hohen Serverlasten kommt. Für den prototypischen Einsatz erscheint ein derartiges Vorgehen vertretbar. Einige der so erprobten Funktionen könnten später mit Hilfe von Server- und Browsererweiterungen implementiert werden, so daß der Einsatz eines Robots verzichtbar wird.

### 2.2.6 Beeinflußung von Robots

Auch Robots, die sich konform zu den in [Kos93] beschriebenen ethischen Richtlinien verhalten, können die Betreiber von Web-Sites verärgern. Ein Robot könnte z.B. ein CGI-Skript aufrufen, das die Stimmen einer online Abstimmung zählt oder Seiten laden und indexieren, die nach Meinung der Betreiber nicht durch Suchmaschinen zugreifbar sein sollten. Im „Standard for Robot Exclusion“ [Kos94] beschreibt Koster ein Verfahren, daß es Web-Site-Betreibern ohne technische Erweiterungen des Servers ermöglicht, Robots Richtlinien für den Besuch der Site zu übermitteln.

Die im Root-Verzeichnis des Servers liegende Textdatei `robots.txt` sollte von jedem Robot vor dem Besuch geladen werden. Die in dieser Datei formulierten Richtlinien schließen selektiv Robots vom Besuch der ganzen Site oder von bestimmten Verzeichnissen aus. Die Einträge dieser Datei sind jeweils in zwei Teile unterteilt. Im ersten Teil, der mit dem Schlüsselwort `User-agent` eingeleitet wird, wird angegeben, für welche Robots die nachfolgenden Anweisungen Gültigkeit haben. Im zweiten Teil wird dann mit den Anweisungen `Disallow` und `Allow` der Zugriff auf ganze Verzeichnisse oder auch nur einzelne Dateien verboten bzw. explizit erlaubt. Da der Zugriff auf alle Verzeichnisse, für die es keine `Disallow`-Anweisungen gibt implizit erlaubt ist, machen `Allow`-Angaben nur Sinn um Ausnahmen zu formulieren. Abbildung 2.12 zeigt das Beispiel einer derartigen Datei.

Da ein Robot genau wie ein normaler Web-Browser mit dem Server kommuniziert, gibt es jedoch keine weiteren Möglichkeiten auch technisch den Zugriff durch Robots zu unterbinden.

Der Standard for Robot Exclusion hat keinen offiziellen Charakter, ist aber als de facto Standard gemeinhin akzeptiert.

Eine weiter Möglichkeit auf das Verhalten von Robots Einfluß zu nehmen stellen die Meta-Tags dar. Soll eine Web-Seite nicht durch einen Robot indexiert werden, dann kann dies durch einen Eintrag `<meta name="robots"content="noindex">` im Kopf der HTML-Datei kenntlich gemacht werden. Desweiteren ist es möglich mit `<meta name="robots"content="index">` explizit die Aufnahme in den Suchindex zu erlauben. Soll nur die aktuelle Seite indexiert werden, jedoch keine Links verfolgt werden, dann läßt sich dies duch `<meta`

```
# robots.txt für http://www.site.com

# Kein Robot darf Dateien aus dem Verzeichnis /cgi-bin/ laden
User-agent: *
Disallow: /cgi-bin/

# Der Scone-Robot darf außerdem keine Dateien aus dem
# Verzeichnis /privat/ laden
User-agent: Scone-Robot
Disallow: /privat/
```

**Abbildung 2.12:** Beispiel einer `robots.txt`-Datei.

`name="robots "content="nofollow">` ausdrücken. Die Angabe `<meta name="robots"content="follow">` erlaubt es dem Robot die Seite in den Index aufzunehmen und die Links zu verfolgen. Die Frequenz, in der Robots eine Seite wiederholt besuchen sollten, läßt sich durch den Meta-Tag `<meta name="revisit-after"content="20 days">` spezifizieren.

### 2.2.7 Probleme beim Betrieb von Robots

Der Betrieb eines Robots bringt eine Reihe von Problemen mit sich, die bei dessen Konstruktion berücksichtigt werden müssen. Neben den negativen Auswirkungen auf die besuchten Sites, im wesentlichen Traffic, Serverlast und Verfälschung der Zugriffsstatistiken, gibt es eine Reihe von Problemen, die den Betrieb des Robots selbst betreffen.

Der Aufgabe Teile des Webs automatisch zu durchwandern inhärent ist es, daß dabei enorme Datenmengen anfallen, die zu übertragen, verarbeiten und speichern einiges an Bandbreite, Rechenleistung und Speicherkapazität verlangt. Die Anzahl der gefundenen Objekte steigt beim Crawlen durchschnittlicher Hypertext-Strukturen exponentiell zur maximalen Tiefe, mit der ausgehend von der Startseite Links verfolgt werden.

Die Umgebung, in der Robots operieren, ändert sich ständig. Neue Dokumente kommen hinzu, andere werden geändert, gelöscht oder sind temporär nicht erreichbar, wenn die Kommunikation zwischen Robot und Server gestört wird. Viele HTML-Dokumente enthalten syntaktische Fehler, durch die die Funktion des Parsers nicht gestört werden darf. Die Architektur des Webs stellt keinerlei Funktionen bereit, die sicherstellen, daß verlinkte Dokumente auch existieren. Ein Robot wird also immer wieder auf ungültige Links treffen.

Die vorstehend beschriebenen Probleme machen deutlich, daß Robots sehr robuste Systeme sein müssen.

Besondere Aufmerksamkeit bei der Implementierung eines Robots ist dem Umgang mit Links und URLs zu widmen. Wie in Abschnitt 2.1.4 beschrieben können Verweise zwischen Dokumenten auf viele verschiedene Arten realisiert werden. Die Gebräuchlichsten sollte ein Robot beherrschen, die Anderen sollten zumindest nicht zu Fehlerzuständen führen.

Um zu Verhindern, daß Seiten mehrfach geladen werden, ist vor jedem Eintrag einer neuen URL in die Warteschlange zu überprüfen, ob das unter dieser URL befindliche Dokument bereits geladen wurde. Dieser Test ist nicht-trivial, denn die Umwandlung in absolute URLs und anschließende Prüfung auf Gleichheit der neuen URL mit den bereits verarbeiteten URLs ist nicht ausreichend, denn für das gleiche Dokument kann es eine Vielzahl von URLs geben.

Port 80 wird standardmäßig für die HTTP-Kommunikation verwandt. Die Angabe einer Port-Nummer ist optional, so daß die URLs http://www.informatik.uni-hamburg.de und http://www.informatik.uni-hamburg.de:80 identisch sind. Viele Web-Server sind so konfiguriert, daß sie auf Requests nach einer URL, in der keine Datei angegeben wurde, eine im jeweiligen Verzeichnis liegende Index-Seite, meistens `index.html`, laden. http://www.informatik.uni-hamburg.de/index.html zeigt also genau wie die beiden vorstehend genannten URLs auf die Homepage des Fachbereichs Informatik. Manche auf Windows-Rechnern laufende Web-Server unterscheiden nicht zwischen Groß- und Kleinschreibung. Anfragen nach http://www.server.de/pfad/datei.html und http://www.server.de/Pfad/Datei.HTML würden also mit dem gleichen Dokument beantwortet werden. Gängige Praxis ist es durch Verweise im Dateisystem Shortcuts für lange URLs zu erzeugen, so daß die URLs http://www.server.de/personen/meier und www.server.de/~meier auf das selbe Dokument verweisen. Ein Web-Server kann unter mehreren Host-Namen erreichbar sein. Sogenannte DNS-Alias Namen führen dazu, daß z. B. www.cocacola.com und www.coke.com durch das DNS zur selben IP-Adresse aufgelöst werden. Es ist aber natürlich auch der Fall möglich, daß Inhalte zur Lastverteilung gespiegelt wurden und sich absolut identische Inhalte unter verschiedenen URLs auf verschiedenen Servern befinden.

Letztendlich ist es nicht entscheidbar, ob zwei verschiedene URLs auf das selbe Dokument verweisen, so daß an dieser Stelle Heuristiken Verwendung finden, bzw. erst nach dem Laden der Seite entschieden werden kann, ob identische Inhalte bereits von einer anderen URL geladen wurden.

Immer mehr Inhalte im Web werden dynamisch durch Skriptsprachen, wie PHP[3] Pearl, ASP oder JSP erzeugt. Wenn diese Skripte HTML-Dokumente generieren, dann sind derartige Dokumente durch Robots prinzipiell verarbeitbar. Durch Skripte generierte Inhalte sind teilweise sehr dy-

[3] www.php.net

namisch, da sie z. B. Inhalte, die mittels eines Content Management Systems in einer Datenbank eingepflegt wurden, in HTML-Templates integrieren. Suchmaschinen-Robots verzichten deshalb in vielen Fällen generell darauf URLs mit Parametern zu besuchen, da die dort plazierten Inhalte sehr schnell veralten, so daß es nich sinnvoll erscheint diese zu indexieren. Im Zusammenhang mit Skript-Sprachen steht auch das Problem, daß es nicht eindeutig möglich ist, aus der Dateiendung einer URL auch auf den MIME-Type der dort liegenden Ressource zu schließen. Bei gebräuchlichen Dateiendungen wie `.htm`, `.html`, `.gif` und `.jpg` mag die Zuordnung zu den Mime-Types `text/html` bzw. `image/gif` und `image/jpeg` eindeutig erscheinen. Anders sieht es jedoch bei den für Skripte verwendeten Dateiendungen, wie `.php` und `.pl` aus, denn es ist gängige Praxis dynamisch Bilder zu generieren, um z. B. den Zwischenstand einer Umfrage zu visualisieren [Kun02]. Die Verwendung des Feldes `Accept` im HTTP-Request und die Prüfung des Mime-Types der im Response enthaltenen Daten ist unerläßlich.

HTTP ist ein zustandsloses Protokoll, was bei E-Commerce Applikationen vielfach durch die Verwendung von an die URL angehängten Session-Ids kompensiert wird. Diese eindeutigen Zeichenketten ermöglichen es mehrere Requests dem selben Benutzer zuzuordnen. Durch Session-Ids wird einem Dokument eine potentiell unendlich große Menge von URLs zugeordnet.

Sogenannte `Black Holes` oder `Crawler Traps` können dazu führen, daß sich ein Robot in einem Server „verfängt“, indem er ein Skript lädt, das eine unendliche Folge von Seiten generiert. Man denke nur an einen Online-Kalender, dessen Darstellung einen Link „nächstes Jahr“ enthält.

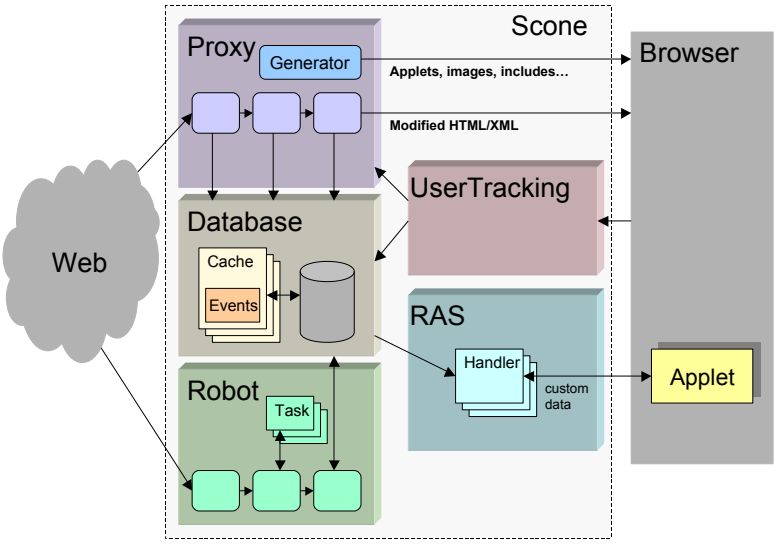


**Abbildung 2.13:** Architektur des Scone-Frameworks

## 2.3 Scone

Der im Rahmen dieser Studienarbeit implementierte Robot ist Teil des Java-Frameworks Scone [4], das am Fachbereich Informatik der Universität Hamburg als technische Grundlage der Dissertationen von Matthias Mayer [5] und Harald Weinreich [6] entwickelt wurde. Scone bietet eine Umgebung, die es ermöglicht prototypisch Programme zur Unterstützung der Navigation im Web zu implementieren und zu evaluieren. Scone-Plugins können die Funktionalität des Web-Browsers erweitern und vollkommen neue Sichten auf das Web generieren. Einen Überblick über die Scone-Architektur zeigt Grafik 2.13

Scone bietet folgende Features:

- *Proxy*
  Ein Proxy-Server ist Bestandteil von Scone. Über diesen wird die Kommunikation zwischen Browser und Server abgewickelt, dabei wird der Datenstrom analysiert und gegebenenfalls manipuliert. Diese erweiterten Proxy-Funktionalitäten basieren auf IBM's WBI (Web Based Intermediary), das zur Realisierung des Scone-Proxy verwendet wurde [BM99].

[4] www.scone.de
[5] asi-www.informatik.uni-hamburg.de/personen/mayer
[6] vsis-www.informatik.uni-hamburg.de/members/info.phtml/20

- *Dokumenten-Parser*
  Analysiert die Dokumente und sammelt Meta-Informationen über Inhalt, Topologie und Sprache. Der Parser erzeugt einen Strom mit den einzelnen Tokens der HTML-Dokumente. Dieser durchläuft dann die verschiedenen Scone-Plugins und kann von diesen verändert werden.

- *Persistenz Package*
  Die den Proxy durchlaufenden bzw. vom Robot geladenen Dokumente werden von Scone persistent gemacht. Dabei werden die gefundenen Dokumente und Links auf Java-Objekte abgebildet, die dann in Objekt-Caches abgelegt werden. Diese wiederum kapseln den Zugriff auf eine Datenbank, wobei Anbindungen für die relationale Datenbank „MySql“ [7] und die objektorientierte Datenbank „Poet OSS“ [8] existieren. Für jedes neue Objekt erzeugen die Caches Events, auf die die Scone-Plugins dann reagieren können.

- *Robot*
  Ein im Verlauf dieser Studienarbeit entwickelter sehr flexibel konfigurierbarer Robot erlaubt es Scone-Plugins automatisch Dokumente aus dem Web zu laden, die dann in den Scone-Caches abgelegt werden.

- *User Tracking*
  Alle Aktionen des Benutzers werden durch Scone überwacht. Durch Events werden Scone-Plugins über jede durch den Benutzer besuchte Seite informiert.

- *Remote Access Server*
  Ein Remote Access Server bietet den Zugriff auf alle Scone-Klassen und Plugins über eine Socket-Schnittstelle. Dieses Feature läßt sich z. B. für die Interaktion von Applets mit Scone einsetzen.

Scone-Anwendungen werden mittels eines Plugin-Konzepts realisiert. Die Plugins haben dabei über eine Java-API Zugriff auf alle Funktionen von Scone. Ein Plugin kann den geparsten Strom der HTML-Dokumente manipulieren und als Server für dynamische und statische Dokumente agieren. Es hat vollen Zugriff auf die Scone-Caches und kann auf die verschiedenen von Scone erzeugten Events reagieren, um z. B. beim Laden einer neuen Seite durch den Benutzer eine Visualisierung zu aktualisieren.

[7] www.mysql.com
[8] www.fastobjects.de

Der Robot wurde optimal in Scone integriert. Die Schnittstellen zu den übrigen Teilen des Frameworks wurden dabei schlank gestaltet. Der Robot benutzt im wesentlichen die von Scone bereitgestellten Klassen zur Modellierung von URIs, Links und Web-Dokumenten, den über das Package `scone.netobjects` gekapselten Zugriff auf die Datenbank und den DocumentParser.

# Kapitel 3

# Verwandte Arbeiten

In diesem Kapitel werden exemplarisch einige Projekte, in denen Web-Roboter entwickelt wurden, vorgestellt, um so den im Verlauf dieser Arbeit entwickelten Robot in den Gesamtkontext dieser Technologie einordnen zu können. Die Darstellung der serverseitig arbeitenden Robots „Google“ (Abschnitt 3.2) und „Mercator“ (Abschnitt 3.1) wurde einbezogen, um die spezifischen Anforderungen an derartige Robots aufzuzeigen und die Anforderungen an persönliche Robots davon abzugrenzen. Das Projekt „eRace“ (Abschnitt 3.3) wird vorgestellt, da hier ein Web-Robot konstruiert wurde, der ähnlich dem Scone-Robot in Verbindung mit einer Proxy-Server-Architektur zum Einsatz kommt.

Der Robot wurde entwickelt, um als Bestandteil des Scone-Frameworks optimale Voraussetzungen für die prototypische Entwicklung von Navigationstools zu schaffen. Die Entwicklung derartiger Tools stand jedoch nicht im Zentrum dieser Arbeit. Bei der Entwicklung des Robots wurden die Anforderungen der auf Scone basierenden Projekte HyperScout [WL00] und BrowsingIcons [MB01] berücksichtigt. Weitere Arbeiten zur Navigation im Web, wie [YN02] und [GBKS00] lieferten Anregungen zum möglichen Einsatz eines Robots in diesem Kontext.

Der Scone-Robots erlaubt es den Vorgang des Crawlens sehr flexibel zu steuern. Dazu sieht die API des Robots Schnittstellen zu sogenannten *Classifiern* und *Filtern* vor, die Dokumente und Links klassifizieren und anschließend entscheiden, wie der Crawl-Vorgang weiter fortgesetzt werden soll. Dieses Konzept wurde von dem Projekt „SPHINX“ [MB98] übernommen, das in Abschnitt 3.4 vorgestellt wird. Classifier und Filter können beliebig komplex sein, so daß der Scone-Robot auch als Grundlage für „intelligente“ Internet-Agenten, wie *ShopBot* [DEW97] oder *WebFind* [ME96], dienen könnte. Die Programmierung derartiger Agenten hat starke Bezüge zur KI und war nicht Gegenstand dieser Arbeit. Auch fortschrittliche Suchverfahren, wie

der in [BP94] vorgestellte *Fish-Search*-Algorithmus oder die in [CCRY98] beschriebenen Verfahren ließen sich basierend auf dem Konzept der Classifier und Filter realisieren. Es wurden entsprechende Schnittstellen für Classifier und Filter entworfen, ohne jedoch konkret derartige Komponenten zu entwickeln.

## 3.1 Mercator

In [HN99] wird der komplett in Java geschriebene Web-Robot „Mercator“[1] beschrieben. Skalierbarkeit und Erweiterbarkeit waren die zentralen Anforderungen bei der Konstruktion des Robots. Unter Skalierbarkeit wurde in diesem Zusammenhang die Fähigkeit verstanden, auf einer geeigneten Hardwareplattform große Teile des Webs crawlen zu können. Erweiterbarkeit wurde durch einen modularen Aufbau des Robots erreicht. Alle Systemkomponenten können durch an spezielle Anforderungen angepaßte Versionen ausgetauscht werden. Es wurden die Schnittstellen aller Komponenten als abstrakte Klassen spezifiziert. In einem Konfigurationsfile kann dann angegeben werden, welche Klassen für die einzelnen Funktionen verwendet werden sollten. Diese werden dann zur Laufzeit instanziert und in das Robot-System integriert. Eine große Klassenbibliothek bietet eine gute Umgebung, um angepaßte Robot-Komponenten zu entwickeln. Ursprünglich wurde das System konstruiert, um statistische Daten über das Web zu sammmeln. Durch verhältnismäßig geringe Anpassungen konnte der Robot aber auch für eine Reihe anderer Aufgaben verwendet werden.

Das Crawlen wird durch mehrere *Worker Threads*, die jeweils zyklisch alle Schritte zum Download und zur Verarbeitung eines Dokuments durchlaufen, nebenläufig ausgeführt. Sie entfernen eine URL aus der Warteschlange, anhand des URL-Schemas wird dann entschieden, welches *Protokoll-Modul* zum Download verwendet wird. Neue Protokoll-Module können hinzugefügt werden, um das Anwendungsspektrum des Robots zu vergrößern. Die DNS-Abfrage ist häufig ein Engpaß in Robot-Systemen, deshalb wurde ein performanter *DNS-Resolver* in das System integriert. Die vom Protokoll-Modul geladenen Daten stehen dann über einen *RewindInputStream* allen *Processing Modules* zur Verfügung. Der RewindInputStream erlaubt dabei das wiederholte Lesen der Daten, ohne das diese mehrfach aus dem Netz geladen werden müssen. Die Dokumente werden dann einem *Content-Seen-Test* unterzogen. Durch diesen Test wird sichergestellt, daß unter unterschiedlichen URLs abrufbare Dokumente identischen Inhalts nur einmal bearbeitet werden. Dazu wird eine Checksumme des Dokuments, der sogenannte *Document Fingerprint* mit denen der vorher besuchten Dokumente verglichen. Dieses

[1] research.compaq.com/SRC/mercator

Verfahren ist eine effiziente Möglichkeit mit einigen der in Abschnitt 2.2.7 beschriebenen Probleme umzugehen. Die Berechnung von Fingerprints wurde in das Scone-Framework übernommen, wenngleich dort ein anderer Algorithmus verwendet wurde. Wurde vorher noch keine Dokumente identischen Inhalts vom Robot geladen, dann werden nun die Daten an die dem jeweiligen MIME-Type zugeordneten *Processing Modules* übergeben. Dieses allgemeine Konzept erlaubt es belibige Verfahren zur Verarbeitung der Dokumente in das System zu integrieren. So könnten z. B. Statistiken aktualisiert oder Inhalte bewertet und abgespeichert werden. Auch das extrahieren der Links wird von einem Processing Module, dem *Link Extractor* übernommen. Die gefundenen Links werden anschließend den von den Benutzern angegebenen *Filtern* übergeben. Diese entscheiden, ob eine URL besucht werden soll. Passiert eine URL die Filter, dann wird geprüft, ob diese bereits geladen wurde. Fällt der *URL-seen test* negativ aus, dann wird die URL in die Warteschlange eingetragen. Die Vorgehens-Strategie des Robots kann durch Angabe einer modifizierten Version der Warteschlange beeinflußt werden. Die Warteschlange könnte z. B. sicherstellen, daß immer nur eine Seite zur Zeit von einem Server geladen wird.

Alle durch den Robot verwendeten Datenstrukturen wurden hinsichtlich Speicherbedarf und Zugriffsgeschwindigkeit optimiert. Caching-Konzepte stellen dabei sicher, daß relevante Teile der Daten im Speicher gehalten werden können und der Robot dennoch mit mehreren Millionen Seiten umgehen kann.

Die Überlegungen zur Erweiterbarkeit wurde im Scone-Robot übernommen und zum Teil noch weitergeführt. Das Kriterium der Skalierbarkeit war jedoch bei diesem Projekt von nachrangiger Bedeutung.

## 3.2 Google

Die Suchmaschine „Google“ [2] verdeutlicht, welches Potential in der Robot-Technologie steckt. Google erfreut sich bei den Internetnutzern großer Beliebtheit, was sicher auf das hohe Maß an Vollständigkeit des Suchindexes und die guten Ranking-Algorithmen zurückzuführen ist. Google speichert die Dokumente im Volltext ab und erfaßt bei der Indexierung die Position der Worte im Text, so daß ein Wortvorkommen im Titel oder in einer Überschrift bei der Beantwortung von Suchanfragen höher gewertet wird, als ein im Fließtext vorkommendes Wort. Zur Bewertung der Relevanz von Dokumenten benutzt Google die Link-Struktur des WWW. Der *PageRank* (siehe [PBMW98]) genannte Algorithmus basiert auf der Annahme, daß ein vielfach verlinktes Dokument von größerer Relevanz ist, als ein Dokument,

[2] www.google.com

auf das nur wenige Seiten verweisen. Der PageRank Algorithmus gewichtet nun zusätzlich die einzelnen Links, so daß ein Link von einer Seite, die ihrerseits einen hohen PageRank-Wert besitzt mehr zählt, als der Link von einer unbedeutenden Seite.

Der zum Aufbau und die kontinuierliche Aktualisierung des Indexes verwendete Robot ist eine der Schlüsselkomponenten der Suchmaschine. Bei der Konzeption des Systems war Skalierbarkeit die wichtigste Anforderung, da das Web möglichst vollständig erfaßt werden soll. Die Suchmaschine wurde als verteiltes System entworfen, bei dem mehrere Rechner für das Crawlen verwendet werden. Der Robot besteht aus fünf Komponenten, die jeweils als eigenständige Prozesse laufen. Der *URL-Server* liest URLs aus einer Datei und leitet diese an einen der *Crawler*[3] weiter. Diese laden parallel Dateien von jeweils bis zu 300 Servern und leiten diese an einen *Store Server* weiter, der die Seiten komprimiert und abspeichert. Die gespeicherten Dateien werden dann von einem *Indexer* geparst und indexiert. Bei diesem Vorgang werden dann auch die Links extrahiert und abgespeichert. Der *URL Resolver* liest die Links, wandelt die enthaltenen URLs in absolute URLs um schreibt diese in die Datei, aus der sie später von dem URL-Server gelesen werden. Es werden bis zu vier Maschinen für den Betrieb der Crawler verwendet, so daß für den Betrieb der kompletten Suchmaschine bis zu acht Rechner verwendet werden.

Details zu Google können in [BP98] nachgelesen werden.

## 3.3 WebRACE

Direkt mit dem Scone-Framework vergleichbar ist das System „WebRACE“ das in [Dem, ZYD] beschrieben wird. WebRace ist der web-spezifische Teile der „eXtensible Retrival Annotation Caching Engine“ (eRace), einem System, das automatisch in Bezug auf bestimmte Benutzerprofile interessante Informationen von verschiedenen heterogenen Internet-Quellen, wie Web, E-Mail und Newsgroups sammelt, annotiert und über verschiedene Kommunikationsprotokolle, wie HTTP, GSM/WAP und SMS weiterverbreitet.

Zentrale Komponente der WebRACE Architektur ist ein Proxy-Server. Dieser beinhaltet einen Web-Robot, der automatisch Dokumente lädt, die dann in einem *Object Cache* abgelegt werden. Eine *Annotation Engine* indexiert die gefundenen Ressourcen und klassifiziert diese hinsichtlich der Benutzerprofile. Desweiteren bietet WebRACE Schnittstellen zu dem *dispatcher*, der aus den Profilen Aufträge für den Proxy-Server generiert und deren Ausführung terminiert. Ein *Alerting-Server* benachrichtigt den Benutzer

[3] Crawler meint hier nicht den kompletten Robot, sondern nur die Komponente, die per HTTP mit den Servern kommuniziert.

über neue Informationen und verteilt diese über verschiedene Kommunikationskanäle.

Scone und WebRace haben unterschiedliche Zielsetzungen. Scone wurde als Grundlage von Navigationstools in erster Linie für den Client-seitigen Betrieb konzeptioniert. WebRACE ist eine serverseitige-Anwendung, die Benutzer automatisch mit Informationen versorgen soll, deren Präsentation den Charakteristika der verschiedenen Kommunikationskanäle und Endgeräte angepaßt wird. Dennoch haben beide Systeme große Ähnlichkeit. Sie erweitern beide den Funktionsumfang eines Web-Proxies um Funktionen zum Zwischenspeichern, Transformieren und Personalisieren von Dokumenten. Beide können mittels eines flexibel konfigurierbaren Robots automatisch Dokumente laden und in beide System sind gut dafür geeignet um unter Benutzung vorhandener Web-Dienste neuartige Dienste anzubieten bzw. Dienste an die Erfordernisse spezieller Endgeräte anzupassen.

Die Entwickler von WebRace charakterisieren ihr System als *Proxy-Agent*. Diese Bezeichnung würde auch Scone gut charakterisieren, da das Scone-Framework mittels des in dieser Studienarbeit entwickelten Robots die Möglichkeit der eigenständigen Informationssuche bietet.

## 3.4 SPHINX

Viele Anregungen für die Entwicklung des Scone-Robots haben sich aus dem Projekt *SPHINX* (Specific Processors for HTML INnformation eXtraction), einem Java-Framework zum Erstellen von persönlichen, Web-Site spezifischen Crawlern, ergeben. SPHINX wurde von Robert C. Miller und Krishna Bharat an der Carnegie Mellon University entwickelt und in [MB98] dokumentiert. Ein vorgestelltes Beispiel für den Einsatz eines persönliches Robots ist der automatische Download von Fotos aller Mitarbeiter eines Instituts. Unter Web-Site spezifischen Crawlern, werden hier Crawler verstanden, die eine Kenntnis über die Struktur der Site die sie besuchen besitzen und so in der Lage sind selektiv bestimmte Links zu verfolgen und andere zu ignorieren. Bestandteil von SPHINX ist eine grafisches Benutzungsinterface, das *Crawler Workbench*. Dieses Tool bietet Funktionen zur interaktiven Konfiguration von Robots, zur Überwachung der Ausführung von Crawl-Aufträgen und zur Visualisierung der Ergebnisse.

Die API von SPHINX ermöglicht es durch erweitern der Klasse `Crawler` und überschreiben der Methoden `boolean shouldVisit(Link link)` und `void visit(Page page)` spezifische Crawler zu implementieren. `shouldVisit` wird für jeden gefundenen Link aufgerufen und entscheidet, ob dieser in die Warteschlange aufgenommen wird. Ferner werden die gefundenen Links von der `shouldVisit`-Methode mit einer Priorität versehen, so daß sich die

Suche im Web durch heuristische Verfahren steuern läßt. `visit` wird mit jeder geladenen Seite ausgeführt und kann beliebige Funktionen zur Verarbeitung der gefundenen Dokumente enthalten.

Die Regeln für das Crawlen kapselt SPHINX in den *classifiern*, welche die Inhalte der gefundenen Dokumente, die in SPHINX durch `Page`- und `Link`-Objekte repräsentiert werden, analysieren und mit Attributen versehen. Alle registrierten Classifier werden mit jeder gefundenen Seite aufgerufen bevor die Seite an die `visit`-Methode übergeben wird. In Abhängigkeit von den, durch die Classifier gesetzten Attributen, kann dann die weitere Bearbeitung des gefundenen Dokuments und das weitere Vorgehen beim Crawlen entschieden werden. Die von den Classifiern gesetzten Attribute können auch auf bestimmte Merkmale der Visualisierung von Objekten abgebildet werden. Mittels eines Classifiers ließe sich z. B. eine Web-Seite als persönliche Homepage kennzeichnen oder ein Link in der Ergebnis-Seite einer Suchmaschinenen-Anfrage als Teil des Suchergebnisses kennzeichnen und damit von anderen externen Links unterscheiden. Ein großer Vorteil von Classifiern ist die Wiederverwendbarkeit und die Kapselung der Web-Site spezifischen Programmteile gegenüber dem übrigen Robot. Eine Anwendung, die z. B. Anfragen an die Suchmaschinen *AltaVista* [4] stellt und die Ergebnisse der Suchanfragen interpretiert, ist abhängig von der syntaxtischen Struktur der Suchmaschinen Ergebnis-Seiten. Ändert die Suchmaschine ihr Layout, dann muß nur der entsprechende Classifier geändert werden.

SPHINX bietet ein Event-Modell, das z. B. Visualisierungen über neue Dokumente informiert. Eine derartige Architektur macht es möglich Visualisierungen unabhängig von dem verwendeten Robot zu entwickeln.

Der Scone-Robot weist große Ähnlichkeit zu dem Robot des SPHINX-Systems auf, was die Art der Steuerung des Crawlens anbelangt. Das in SPHINX eingeführte Konzept der Classifier wurde übernommen, wobei das Zusammenspiel der Classifiern mit den Methoden `visit` und `shouldVisit` des Robots durch ein allgemeineres Konzepts ersetzt wurde, bei dem das Crawlen durch *PageClassifier*, *LinkClassifier*, *PageFilter* und *LinkFilter* gesteuert wird und sogenannte *RobotUser* über jedes gefundene Dokument informiert werden. Diese führen dann spezifische Verbeitungsschritte, wie etwa die Aktualisierung einer Visualisierung durch. In SPHINX muß für jeden Auftrag ein neuer Robot instanziiert werden. Scone sieht im Gegensatz dazu nur genau einen Robot vor, der Aufträge in Form von sehr komplexen *RobotTask*-Objekten entgegennimmt.

[4] www.altavista.com

# Kapitel 4

# Lost in Hyperspace – Motivation für die Integration eines Robots in das Scone-Framework

Viele Benutzer haben bei der Suche nach Informationen im World Wide Web die frustrierende Erfahrung gemacht, in komplexen Hypertexten den Überblick verloren zu haben. Hat ein Benutzer ein interessantes Dokument gefunden und „surft“ dann ein paar Seiten weiter, dann kann es leicht passieren, daß er diese Seite nicht mehr wiederfindet. Derartige Phänomene werden in der Literatur mit der Phrase „Lost in Hyperspace“ bezeichnet.

Andere Informationsquellen, wie die Printmedien, sind aus Benutzersicht um einiges einfacher zu handhaben, da die Informationen in einer linearen Folge von Seiten angeordnet sind, Seitenzahlen und Kopfzeilen Überblick über die Position geben und sich der Gesamtumfang des vor einem liegenden Werks abschätzen läßt.

Kennzeichnendes Merkmal des Webs ist die Verknüpfung von Dokumenten mittels Hyperlinks. Durch diese stehen die Hypertextdokumente zueinander in einer n:m Relation, denn eine Seite enthält Links zu m anderen Dokumenten und wird wiederum von n Seiten verlinkt. Der Benutzer sieht jedoch immer nur die von einem Dokument ausgehenden Links und nimmt somit nur einen kleinen Teil der Struktur war, in die ein Dokument eingebettet ist. Dem Benutzer fehlt es also an Überblick, zumal er immer nur die direkte Linkumgebung einer Seite wahrnimmt, ohne z. B. zu sehen, ob ein Link in eine Sackgasse führt, oder ob er den Startpunkt einer linear angeordneten Folge mehrere thematisch zusammenhängender Dokumente markiert. Die grafische Darstellung von Links beschränkt sich auch bei den heutigen Browsern

in der Regel auf die Kennzeichnung durch Unterstreichung und die Markierung von bereits besuchten Links. Es gibt keine Informationen darüber, ob ein verlinktes Dokument überhaupt erreichbar ist, ob es auf einen externen Server verweist oder eine große Datei verlinkt wurde. Zur Unterstützung der Navigation bieten die Browser meist nur einen als History-Mechanismus und eine Bookmark-Funktion, die es ermöglicht Links zu bestimmten Seiten zu hinterlegen. Ein gängiges Benutzverhalten bei der Suche im Netz ist es von einer Seite ausgehend jeweils ein paar Links tief in die Hypertextstruktur hinabzusteigen, dann mittels des Back-Buttons wieder zur Ursprungsseite zurückzukehren und die nächsten Links zu besuchen. Da die History gemeinhin als Stack implementiert wird, verschwinden alle vorher besuchten Seite aus der History, wenn der Benutzer wieder zur Ursprungsseite gelangt.

Zusammenfassend ist festzuhalten, daß die Einfachheit, mit der im Web Informationen publiziert und verlinkt werden können, sicher entscheidend zum Erfolg des Webs beigetragen hat. Das Fehlen einer zentralen Instanz, in der alle Dokumente und Verknüpfungen registiert sind, aber einige konzeptionelle Nachteile bezüglich der Navigation im Hypertext mit sich bringt, die auch von den heutigen Browsern nicht kompensiert werden.

Browser können immer nur ein Dokument zur Zeit darstellen, während Benutzer aber oftmals Aktionen durchführen wollen, die sich auf Mengen von Dokumenten beziehen. So kann es z. B. bei der Suche nach Informationen zu einem bestimmten Projekt sinnvoller sein die komplette Web-Site als eine Einheit anzusehen, auf der eine Suche durchgeführt wird, anstatt manuell die Seiten zu laden und zu durchsuchen. Tutorials werden oftmals aufgeteilt in mehrere Dokumente präsentiert. Will ein Benutzer das gesamte Tutorial drucken, oder speichern, so muß er manuell alle Seiten laden. Praktisch wäre es auch, wenn man direkt beim Aufruf der Homepage eines Servers einen Überblick über alle Dokumente der Site erhalten würde, die sich seit dem letzten Besuch geändert haben.

Viele der oben beschriebenen Probleme ließen sich verringern, in dem man den Benutzern beim „Surfen“ im Web Programme zur Verfügung stellt, die unter Benutzung eines Web-Robots neue Sichten auf das Web generieren, bzw. Dienste zur Vefügung stellen, die die automatische Verarbeitung von Dokumenten-Mengen gestatten. Das Scone-Framework bietet eine gute Basis für die Entwicklung von Tools zur Navigationsunterstützung. Durch die Integration eines Browsers können Scone-Plugins neben den durch den Benutzer geladenen Dokumenten auch selbstständig Dokumente aus dem Netz anfordern. Applikationen die beispielsweise die Hypertextstruktur visualisieren oder das Link-Interface des Browsers um weitergehende Informationen zu dem verlinkten Dokument erweitern, lassen sich nur unter Einbeziehung eines Robots verwirklichen.

Das Spektrum der mit Scone zu realisierenden Anwendungen wird also durch die Integration eines Robots erheblich vergrößert.

# Kapitel 5

# Anforderungen an den Scone-Robot

Das Einsatzgebiet des Scone-Robots sind Tools, die den Benutzer bei der Navigation im Web unterstützen. Im Gegensatz zu den im Bereich der Suchmaschinen eingesetzten Robots kommt es hier nicht darauf an das Web möglichst vollständig zu crawlen, sondern einen sehr kleinen Ausschnitt sehr schnell. Der Robot wird im Kontext interaktiv zu bedienender Programme eingesetzt. Für Benutzer ist der Umgang mit solchen Systemen nur dann akzeptabel, wenn diese schnell auf Aktionen reagieren. Der Robot muß also sehr schnell die ersten gefundenen Objekte zurückliefern, in der Lage sein Aufträge zu priorisieren und hinfällig gewordene Aufträge schnell abzubrechen. Dabei ist zu beachten, daß einem client-seitig ablaufenden Robot nur sehr begrenzte Ressourcen hinsichtlich Rechenleistung und Speicherkapazität zur Verfügung stehen. Vielen Internetbenutzer steht nur eine schmalbandige Internetnetanbindung zur Verfügung. Ein auf dem Rechner des Benutzers betriebener Robot solltes deshalb sehr ökonomisch mit der Bandbreite umgehen. Mehrfaches Laden eines Dokuments sollte deshalb unbedingt vermieden werden.

Das Spektrum der mit Scone realisierbaren Anwendungen ist sehr breit, so daß mit sehr unterschiedlichen Anforderungen an den Robot zu rechnen ist. Der Robot sollte deshalb generisch und modular wie möglich realisiert werden, so daß er sich durch Austausch von Komponenten und Modifikation von Konfigurationseinstellungen an sehr unterschiedliche Anwendungen anpassen läßt. Der Robot sollte sich nahtlos in das Scone-Framework integrieren, so daß es Scone-Plugins bei minimalem Zusatzuaufwand möglich ist automtisch Dokumente aus dem Web zu laden, die dann in den Scone-Datenstrukturen abgelegt werden. Bei der Gestaltung des Application Programming Interface (API) war die Maßgabe diese so einfach wie möglich zu gestalten. Die Scone-Architektur gestattet es parallel mehrer Plugins zu

betreiben. Damit unterschiedliche Plugins nebenläufig den Robot benutzen können, ist es notwendig, daß die relevanten Konfigurationseinstellugen für jeden Auftrag unabhängig voneinander spezifiziert werden können. Der Robot sollte überdies in der Lage sein viele konkurrierende Anfragen schnell zu bearbeiten, und bei der Bedienung der verschiedenen Aufträge eine faire Bedienstrategie anzuwenden.

# Kapitel 6

# Der Scone-Robot

In diesem Kapitel wird der entwickelte Web-Robot präsentiert. Zunächst werden die Leistungsmerkmale des Systems vorgestellt (Abschnitt 6.1). Es folgt die Erleuterung von Architektur und Funktionsweise des Robots (Abschnitt 6.2). Details zu den einzelnen Systemkomponenten finden sich in Abschnitt 6.3. Die Erstellung von Crawl-Aufträgen und die API des Robots werden in den Abschnitten 6.4 bzw. 6.5 beschrieben. Den Abschluß dieses Kapitels bildet die Darstellung der Testumgebung (Abschnitt 6.6).

## 6.1 Leistungsmerkmale

Der Scone-Robot zeichnet sich durch einen modularen Aufbau aus. Wichtige Systemkomponenten, wie die URL-Warteschlange wurden duch Interfaces spezifiziert und können durch angepaßte Versionen ersetzt werden. Der Robot arbeitet multi-threaded, was den nebenläufigen Download mehrerer Dateien ermöglicht und die Fähigkeit parallel mehrere Aufträge zu bearbeiten verbessert. Um den durch ständiges Neuerzeugen von Threads entstehenden Overhead zu minimieren wurde ein Thread-Pool in das System integriert. Dieser verwaltet eine Menge von Threads, die dann immer wieder neue bei der Abarbeitung eines Crawl-Auftrags entstehende Teilaufträge ausführen. Robots laden oftmals mehrere Dokumente in folge von einem Server. HTTP/1.0 erlaubte nur die Übertragung einer Datei pro geöffneter Verbindung, was zu einem großen Mehraufwand führte, da viele Verbindungsauf- und -abbau-Operationen notwendig waren. Der Scone-Robot unterstützt das HTTP/1.1 Protokoll und profitiert insbesondere von den mit diesem Protokoll möglichen persistenten Verbindungen. Ein HTTP-Connection-Pool verwaltet die offenen Verbindungen, so daß auch dann eine bestehende Verbindung zu einem Server genutzt werden kann, wenn zwischenzeitlich Dateien von einem anderen Server geladen werden. Diesem

Aspekt der Optimierung ist besondere Aufmerksamkeit gewidmet worden, da es durch den gleichzeitigen Einsatz mehrerer Plugins schnell zu Situationen kommen kann, in denen mehrere konkurrierende Crawl-Aufträge bedient werden, die jeweils viele Dateien von einem Server laden. Zur weiteren Minimierung der Netzlast trägt der Zugriff auf die Scone-Caches bei, in denen aus dem Web geladende Objekte persistent gespeichert werden. Der Robot nutzt das Feld `if-modified-since` des HTTP-Headers, um im Cache bereits vorhandene Objekte nur dann erneut zu laden, wenn diese seit dem letzten Zugriff modifiziert wurden.

Der Scone-Robot ist sehr flexibel konfigurierbar. Grundlegende Parameter lassen sich über die Konfigurationsdatei `robot.xml` einstellen. Diese Datei kann direkt editiert werden, es existiert aber auch eine grafische Benutzungsoberfläche, in der sich alle Parameter des Scone-Systems konfigurieren lassen. Dort ist die maximale Anzahl verwendeter Threads ebenso einstellbar wie die Zeitspanne, nach der ein unbeschäftigter Thread beendet wird. Desweiteren kann der Name, mit dem sich der Scone-Robot im `User-Agent` Feld bei der HTTP-Kommunikation identifiziert angegeben werden. Ebenfalls einzustellen ist die Menge der Datei-Extensions, die vom Robot beachtet werden. Die vom Robot verwendete Warteschlange wird ebenfalls in der Konfigurationsdatei benannt. Die hier angegebene Klasse wird dann zur Laufzeit instanziiert. Alle auf einen Crawl-Auftrag bezogenen Parameter können als Attribute eines Auftrags-Objekts gesetzt werden. Das Konzept der *Filter* und *Classifier* gestattet es relativ komplexe Regeln für das Crawlen zu definieren und die gefundenen Objekte mit Attributen zu versehen. Die aus dem Netz geladenen Seiten werden unter zuhilfenahme des des in Scone vorhandenen *DocumentParsers* geparst, so daß Scone-Plugins, die den Robot benutzen direkten Zugriff auf alle Attribute einer Seite haben. In Abschnitt 2.2.7 wurden die Probleme erleutert, die sich aus der Situation ergeben, daß oftmals ein Dokument unter mehreren URLs erreichbar ist, bzw. Dateien identischen Inhalts auf verschiedenen Servern liegen. Scone bildet über den Inhalt der im Cache abgelegten Dokumente eine Checksumme, die vom Robot zur Identifikation von Dokumenten identischen Inhalts benutzt werden. Der Scone-Robot bietet optionale Unterstützung füt den *Standard for Robot Exclusion*.

Der Zustand des Robot kann über ein Web-Interface, den Robot-Monitor (siehe Abbildung 6.2), eingesehen werden. Dieses bietet Informationen über den Bearbeitunszustand der aktuellen Aufträge und die Länge der Warteschlange. Laufende Aufträge können im Robot-Monitor gelöscht werden.

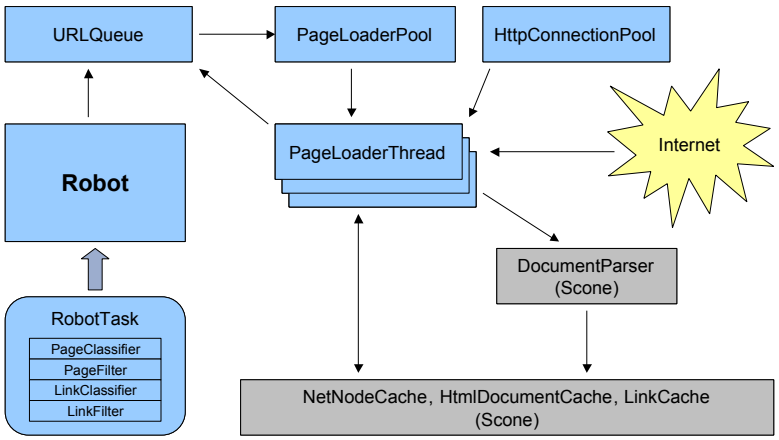


**Abbildung 6.1:** Architektur des Scone-Robots

## 6.2 Architektur und Funktionsweise

In diesem Abschnitt wird ein Gesamtüberblick über Architektur und Funktion des Robots gegeben. Details zu den einzelnen Komponenten werden dann im folgenden Abschnitt aufgeführt.

Aus der Vorgabe den Robot in das in Java geschriebene Scone-Framework zu integrieren, resultierte die Entscheidung den Crawler objektorientiert in Java zu entwickeln. Der Scone-Robot wurde als Singleton (vgl. [GHJ95]) implementiert. Die Beschränkung auf genau einen Robot hat den Vorteil, daß sich die Vergabe von Ressourcen an den Robot zentral kontrollieren läßt. Insbesondere ist sichergestellt, daß nicht mehrere Scone-Plugins gleichzeitig eigene Instanzen eines Robots starten, die dann jeweils mehrere TCP-Verbindungen aufbauen. Um dennoch Plugins mit unterschiedlichen Anforderungen an den Robot, dessen gleichzeitigen Einsatz zu ermöglichen wurden alle für einen Auftrag an den Robot relevanten Parameter in einer Klasse `RobotTask` modelliert. Abbildung 6.1 zeigt einen Überblick über die Architektur des Robots.

Wird ein `RobotTask` an den Robot zur Bearbeitung übergeben, dann erzeugt dieser einen ersten Eintrag in der Warteschlange, ausgedrückt durch ein Objekt der Klasse `QueueEntry`, welches eine Referenz auf den zugehörigen Auftrag beinhaltet. Die Warteschlange implementiert das Interface `URLQueue` und ist nicht auf eine bestimmte Bedienstrategie festgelegt. Der Download der Seiten und deren weitere Verarbeitung erfolgt durch die *PageLoaderThreads*, deren Einsatz durch den `PageLoaderPool` gesteuert wird, der kontinuierlich Sub-Aufträge aus der Warteschlange entfernt und der Bearbeitung durch die `PageLoaderThreads` zuführt. `PageLoaderThreads`

sind die Komponenten, die mit den Web-Servern kommunizieren. Dabei greifen sie auf den `HTTPConnectionPool` zu, der die offenen TCP-Verbindungen und deren Benutzung durch die `PageLoaderThreads` verwaltet. Ein `InputStream` mit aus dem Netz geladenen Seiten wird an den *DocumentParser* übergeben. Diese im Package `scone.util` bereitgestellte Komponente parst den HTML-Datenstrom, erzeugt für die aus dem Netz geladene Seite neue `NetNode`-, `HtmlNode`- und `Link`-Objekte und setzt deren Attribute. Die geladenen Objekte werden in den Scone-Datenstrukturen `NetNodeCache`, `HtmlDocumentCache` und `LinkCache` abgelegt. Sie sind damit für alle Scone-Plugins zugreifbar. Für alle neu gefundenen URLs werden dann von dem `PageLoaderThread` Sub-Aufträge erstellt und in die Warteschlange des `PageLoaderPools` eingetragen.

## 6.3 Systemkomponenten

Im folgenden werden einige Implementierungsdetails des Robots beschrieben. Aufgeführt wurden die konzeptionell wichtigsten Klassen.

### 6.3.1 `Robot`

Dies ist die zentrale Klasse des Scone-Robots. Sie implementiert das Singleton-Pattern, so daß es nur eine Instanz des Robots pro Virtual Machine geben kann. Alle Scone-Plugins, die den Robot benutzen, interagieren ausschließlich mit dieser Klasse. Nach der Instanziierung werden aus der Datei `robot.xml` die Konfigurationsparameter geladen und die anderen Systemkomponenten `PageLoaderPool`, `HttpConnectionPool`, `UrlQueue` erzeugt. Intern kapselt der Robot über eine Reihe von `private` Methoden den Zugriff auf die Warteschlange, den `HttpConnectionPool` und den `NoRobotsTester`, der ein zum *Standard for Robot Exclusion* konformes Verhalten sicherstellt.

### 6.3.2 `RobotHtmlNode`

Zur internen Repräsentation eines aus dem Netz geladenen HTML-Dokuments werden Objekte der Klasse `RobotHtmlNode` verwandt. `RobotHtmlNodes` beeinhalten Referenzen zu korrespondierenden `NetNodes` und `HtmlNodes`, die als Teil des Frameworks im Package `scone.netobjects` deklariert wurden. Desweiteren werden Angaben über die Link-Tiefe gemacht, mit der der Robot die Umgebung des zugehörigen Web-Dokuments gescannt hat. `Classifier` können `RobotHtmlNodes` Attribute zuweisen, die über ein Hashtable abfragbar sind. Zu einer `NetNode` bzw. `HtmlNode` kann es mehrer `RobotHtmlNodes` geben. Dies trägt der Tatsache Rechnung,

daß der Robot bei der Abarbeitung unterschiedlicher Aufträge mehrmals auf das selbe Dokument treffen kann, diesem aber unterschiedliche Attribute zuzuweisen sind. Im Normalfall verarbeitet der Robot nur HTML-Dokumente, so daß sich im Scone-Cache zu den `RobotHtmlNodes` zugehörige `HtmlNodes` befinden. Der Robot läßt jedoch so konfigurieren, daß er nur per HTTP-Befehl `HEAD` den HTTP-Header eines Dokuments lädt. In diesem Fall kann es sein, daß es lediglich eine zur `RobotHtmlNode` korrespondierende `NetNode` gibt, was beim Zugriff bei Aufruf der Methode `getHtmlNode()` eine `NoHtmlNodeException` auslöst.

### 6.3.3 `HttpConnectionPool` und `RobotHttpConnection`

Die Kommunikation des Robots mit den Webservern wird über das Package `HTTPClient`[1] abgewickelt. Diese Lösung wurde gegenüber dem `java.net` Package präferiert, da `HTTPClient` im Gegensatz zu `java.net` den Standard HTTP/1.1 inklusiv persistenter Verbindungen und Request-Pipelining implementiert. Desweiteren unterstützt HTTPClient Timeouts für die Socket-Verbindungen, über die die HTTP-Kommunikation abgewickelt wird. Für Robots, die sich einigermaßen robust gegenüber Netzwerkproblemen verhalten sollten, ist dies ein unerläßliches Feature.

Die `HTTPConnection`-Klasse des HTTPClient realisiert eine persistente Verbindung. Nach 60 Sekunden Inaktivität schließt `HTTPConnection` die TCP-Verbindung. Kommen dann weitere Requests, so wird transparent eine neue Verbindung aufgebaut. Die Möglichkeit über eine Verbindung mehrere Dateien zu übertragen bringt für einen client-seitig ablaufenden Robot, der vielfach innerhalb kurzer Zeit einige Dateien von einem Server lädt, einen großen Leistungsgewinn. Um die Wiederverwendung offener Verbindungen zu optimieren wurde ein `HTTPConnectionPool` in den Robot integriert, der alle offenen Netzwerkverbindungen verwaltet. Wird über die Methode `getHttpConnection(URL url)` eine Connection aus dem Pool angefordert, dann wird zunächst überprüft, ob eine inaktive Verbindung zum entsprechenden Host vorhanden ist. Sollte dies nicht der Fall sein, dann werden gegebenfalls inaktive Verbindungen aus dem Pool verdrängt und eine neue `HTTPConnection` erzeugt. Da jeder `PageLoaderThread` maximal eine Verbindung benutzt, wird die Zahl der aktiven Verbindungen durch die maximale Anzahl Threads begrenzt, so daß es bei entsprechender Dimensionierung des `HTTPConnectionPools` immer möglich ist Anforderungen nach Verbindungen zu bedienen. Es hat sich als sinnvoll erwiesen die Anzahl offener Verbindungen auf das Doppelte der maximalen Anzahl Threads zu begrenzen. So können auch bei gleichzeitiger Bendienung mehrerer Aufträge, oder bei

[1] www.innovation.ch/java/HTTPClient

zwischenzeitlichem Verfolgen eines externen Links die offenen Verbindungen im Pool gehalten werden.

### 6.3.4 `PageLoaderPool`

Der Crawl-Vorgang wird durch den `PageLoaderPool` gesteuert. Dieser regelt die Zuordnung der in der Warteschlange befindlichen Sub-Aufträge zu den `PageLoaderThreads`, welche die Dokumente laden, verarbeiten und dann wiederum für die neu gefundenen URLs neue Sub-Aufträge in die Warteschlange schreiben. Der `PageLoaderPool` ist eine an die Anforderugen des Robots angepaßte Umsetzung des Konzepts eines *Thread-Pools*, wie es u. a. in [OW97] beschrieben wird.

Die Verwendung eines ThreadPools bieten den Vorteil, daß sich die Anzahl der `PageLoaderThreads` zentral kontrollieren läßt, desweiteren wird der Overhead minimiert, der durch das Starten neuer Threads entsteht.

Als Grundlage für die Implementierung diente der im Artikel [???] vorgestellte Thread-Pool, wobei die Warteschlange jetzt nicht mehr Objekte des Typs `Runnable` enthält, sondern dynamisch aus den in der Warteschlange abgelegten Sub-Aufträgen `PageLoaderThreads` erzeugt werden, die dann von den `PoolThreads` ausgeführt werden. Der `PageLoaderPool` enthält eine in der Konfiguration des Robots einstellbare Anzahl `PoolThreads`. Diese entfernen jeweils einen Eintrag aus der Warteschlange, erzeugen einen neuen `PageLoaderThread` und führen diesen aus. Nach jeder Ausführung eines `PageLoaderThreads` wird überprüft, ob sich noch weitere Einträge in der Warteschlange befinden. Ist dies der Fall, dann wird der nächste Sub-Auftrag ausgeführt, anderenfalls wartet der `PoolThread` auf neue Aufgaben, wobei er sich nach einer maximalen Wartezeit selbstständig beendet. Wird ein neuer Sub-Auftrag in die Warteschlange eingetragen, dann wird zunächst überprüft, ob sich ein `PoolThread` im Wartezustand befindet und reaktiviert werden kann. Wenn es keine wartenden Threads gibt und die Anzahl der aktiven Threads nicht die eingestellte Höchstgrenze übersteigt, dann wird ein neuer Thread gestartet, anderenfalls wird der Auftrag zur späteren Bearbeitung in die Warteschlange eingetragen.

### 6.3.5 `PageLoaderThread`

`PageLoaderThreads` führen die vollständige Verarbeitung des im Konstruktur angegebenen Sub-Auftrags aus. Sie implementieren das Interface `Runnable` und werden von den `PoolThreads` zur Ausführung gebracht.

Zunächst wird überprüft, ob sich die zu bearbeitende URL bereits im `NetNodeCache` bzw. `HtmlNodeCache` befindet. Ist dies nicht der Fall, dann

wird das unter der URL liegende Dokument aus dem Netz geladen. Anderenfalls wird überpüft, ob ein erneuter Download dieses Dokuments zu erfolgen hat. Dafür ist je nach Einstellungen des RobotTasks maßgeblich, ob beim vorherigen Zugriff das Dokument vollständig geladen wurde und ob die Links geparst wurden. Desweiteren wird überprüft, ob der Seiten-Quelltext als Attribut der `HtmlNode` abgespeichert wurde und ob die Berechnung eines Fingerprints erfolgte. Aus dem Netz geladene Dokumente werden durch den `DocumentParser` geparst. Dabei werden die Attribute von `NetNode` und `HtmlNode` gesetzt, für gefundene Hyperlinks neue `Link`-Objekte erstellt und `NetNodes` für die verlinkten Dokumente angelegt. Die folgenden Verarbeitungsschritte erfolgen unabhängig davon, ob das Dokument aus dem Netz geladen wurde oder sich bereits im Scone-Cache befand. Es wird eine `RobotHtmlNode` erzeugt. Diese wird dann von den `PageClassifiern` mit Attributen versehen, anhand derer dann die `PageFilter` entscheiden ob der Crawl-Vorgang an dieser Stelle fortgesetzt oder abgebrochen werden soll. Es kommt zum Abbruch, wenn einer der `PageFilter` entsprechend entscheidet, oder die maximale Link-Tiefe erreicht wurde. Soll das Crawlen fortgesetzt werden, so wird über alle gefundenen Links iteriert, wobei in jedem Iterationschrift ein `RobotLink`-Objekt erstellt wird. Das wird anschließend von den `LinkClassifiern` mit Attributen versehen. Nachfolgend wird dann durch die `LinkFilter` entschieden, welche der neu gefundenen URLs in die Warteschlange des `ThreadPools` eingetragen werden sollen. URLs, die während der Bearbeitung des aktuellen Auftrags bereits mit größerer Tiefe gescannt wurden, werden nicht erneut in die Warteschlange eingetragen. Abschließend wird das Objekt, welches den Crawl-Auftrag initiert hatte, über das neu gefundene Objekt informiert.

### 6.3.6 `QueueEntry`

Für bei der Bearbeitung eines Auftrags neu gefundenen URLs, die vom Robot verarbeitet werden sollen, werden Sub-Aufträge, die durch Objekte der Klasse `QueueEntry` repräsentiert werden, in die Warteschlange des ThreadPools eingetragen. Diese beinhalten Angaben über die zu ladende URL und die noch zu crawlende Link-Tiefe. Sie referenzieren den `RobotTask`, im Rahmen dessen Bearbeitung der Warteschlangeneintrag erzeugt wurde.

### 6.3.7 `RobotLink`

Das Scone-Framework bildet die Links zwischen Ressourcen im Web auf `Link`-Objekte ab, die jeweils eine Verbindung zweier `NetNodes` repräsentieren. Um Links mit Attributen versehen zu können, wurden `RobotLinks` implementiert, die ein korrespondierendes `Link`-Objekt referenzieren. Mehrere `RobotLinks` können dabei auf ein `Link`-Objekt verweisen und diesem

somit in verschiedenen Anwendungskontexten unterschiedlichen Attribute zuweisen.

### 6.3.8 `RobotMonitor`

Normalerweise arbeitet der Robot im Hintergrund und es findet keine Interaktion mit dem Benutzer statt. Insbesondere beim Auftreten von Fehlern ist es jedoch sinnvoll den aktuellen Status des Robots darzustellen. Zu diesem Zweck wurde eine Web-basierte Benutzungsschnittstelle, wie sie in Abbildung 6.2 zu sehen ist, für den Robot implementiert.

Dargestellt wird die Anzahl der momentan von Robot bearbeiteten Aufträge und die Gesamtzahl der im `PageLoaderPool` befindlichen Subaufträge. Dieser Wert ist die Summe aus der Länge der URL-Warteschlange und der Anzahl momentan durch die `PageLoaderThreads` bearbeiteten URLs. Zu jedem Auftrag ist die Start-URL und die Crawl-Tiefe ersichtlich. Es wird ausgegeben, wieviele Einträge in der URL-Warteschlange zu diesem Auftrag gehören, welche URLs gerade im Rahmen der Bearbeitung dieses Auftrags von den `PageLoaderThreads` geladen werden und wieviele Dokumente bereits gefunden wurden. Aufträge können über das Web-Interface abgebrochen werden.

Zum interaktive Testen des Robots und zum Füllen der Scone-Caches beinhaltet wurde die Möglichkeit interaktiv Aufträge zu erstellen und zu starten in die Bedienoberfläche integriert.

### 6.3.9 `URLQueue`

Da der `PageLoaderPool` nicht alle Anforderungen nach `PageLoaderThreads` unmittelbar bedienen kann, werden neue Sub-Aufträge als Objekte des Types `QueueEntry` in einer Warteschlange eingereiht. Die Anforderungen an die Warteschlange werden durch das Interface `URLQueue` (siehe Abbildung 6.7) spezifiziert. Die Operationen `queue()` bzw. `dequeue` legen die Warteschlangendisziplin und damit die Crawl-Strategie des Robots fest. Wird eine Warteschlange verwendet, die nach dem Prinzip *First In First Out (FIFO)* arbeitet, dann wird der Robot eine Breitensuche durchführen. Tiefensuche läßt sich mit einer *Last In First Out (LIFO)* Warteschlange realisieren. Robots, die parallel mehrere Seiten aus dem Netz laden, folgen nur näherungsweise den durch die Warteschlangendisziplin vorgegebenen Such-Strategien, da es durch die unterschiedliche Größe verschiedener Dokumente und den damit verbundenen Differenzen in der Bearbeitungszeit, vorkommen kann, daß ein `PageLoaderThread` mehrere kleine Dokumente lädt und die gefundenen Links in die Warteschlange einträgt, während ein anderer Thread eine sehr große Datei lädt.

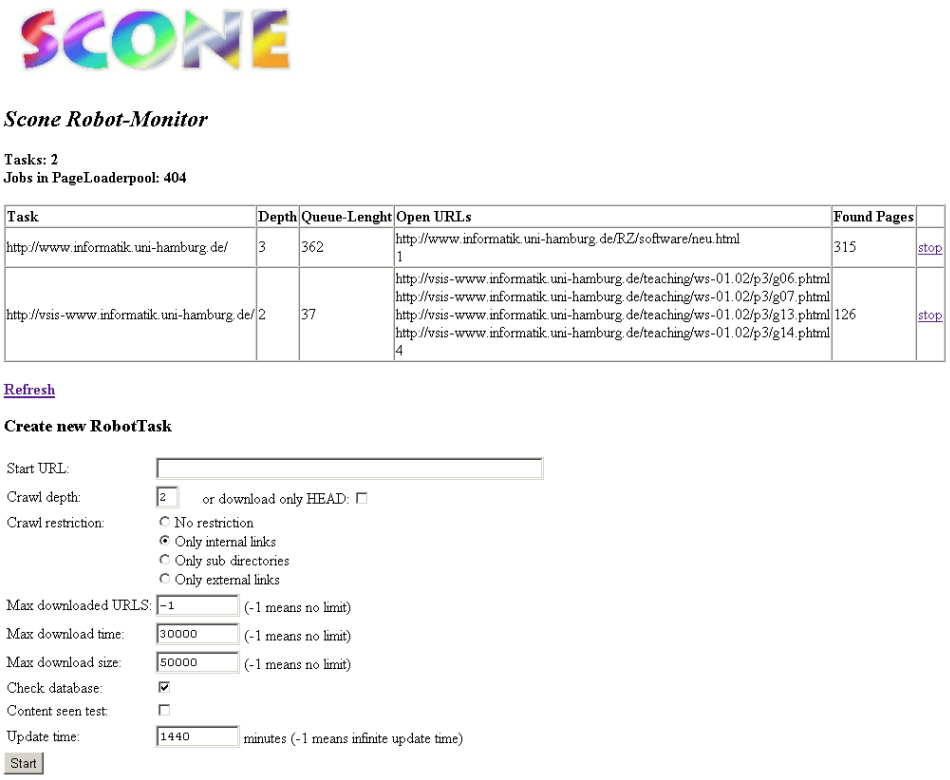


**Abbildung 6.2:** Screenshot des Robot-Monitors

Die abstrakte Klasse `GenericURLQueue` implementiert das Interface `URLQueue`. Sie realisiert eine einfache Warteschlange, wobei die Warteschlangendisziplin von der konkreten Implementierung der abstrakten Methode `dequeue()` durch Subklassen abhängig ist. Mit den Klassen `BreadthFirstQueue` bzw. `DepthFirstQueue` stehen FIFO- bzw. LIFO-Warteschlangen zur Verfügung. Die Vererbungsbeziehungen veranschaulicht dass in Abbildung 6.3 dargestellte UML-Diagramm.

Die im Package `scone.robot` vorhandenen Warteschlangen sind für viele Anwendungen des Scone-Robots ausreichend, sie wurde jedoch nicht hinsichtlich des Laufzeitverhaltens der oft benutzten `queue()` und `dequeue()` Funktionen optimiert. Wird eine eine normalen FIFO-Warteschlange verwendet, dann kann es zu dem Effekt kommen, daß bei der Bearbeitung eines Auftrags eine Seite mit sehr vielen Links gefunden wird, diese in einer Folge in die Warteschlange eingetragen werden und es dann zu einer sehr ungleichmäßigen Verteilung der Kapazitätem des Robots auf die gleichzeitig bearbeiteten Aufträge kommt. Besser wäre es für jeden Auftrag eine eigene URL-Warteschlange zu verwenden und abwechselnd Sub-

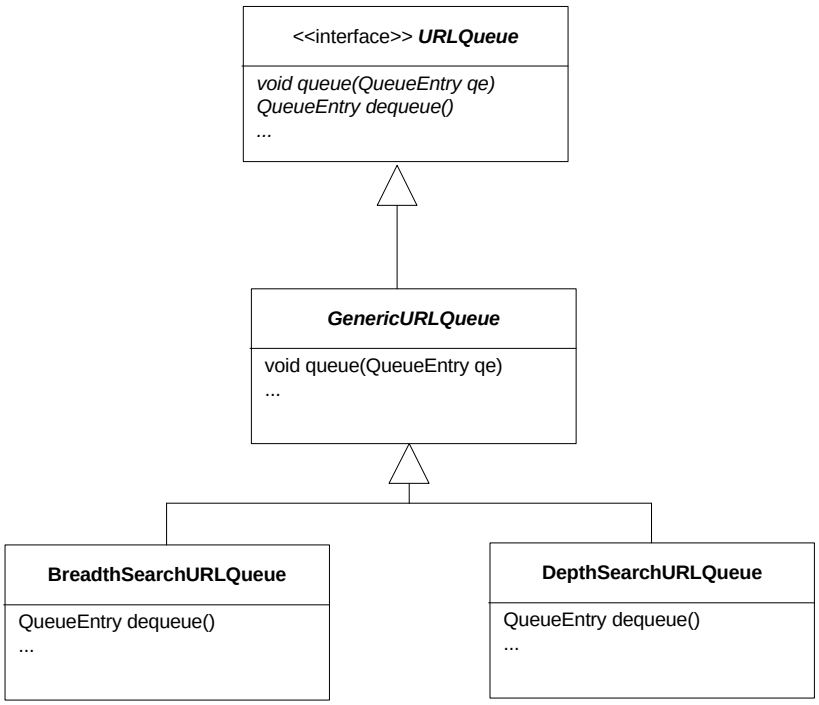


**Abbildung 6.3:** Klassendiagramm der Warteschlangen

Aufträge aus den verschiedenen Warteschlangen in Bedienung zu nehmen. Ein Warteschlange mit derartigem Verhalten wurde duch die Klasse `FairBreadthSearchURLQueue` implementiert, deren `dequeue()` Methode immer den nächsten Eintrag des aktuell zu bedienenden Auftrags zurückliefert.

Spezielle Crawl-Strategien können realisiert werden, in dem eine Warteschlange programmiert wird, die das Interface `URLQueue` (siehe Abbildung 6.7 implementiert. Diese Warteschlange wird dann in die Konfigurations-Datei `robot.xml` eingetragen. Die dort spezifizierte Klasse wird dann bei der Initialisierung des Robots instanziiert und als Warteschlange des Thread-Pools benutzt.

## 6.4 Crawl-Aufträge

In diesem Abschnitt werden die Möglichkeiten vorgestellt, wie Aufträge für der Robot erstellt und angepaßt werden können.

### 6.4.1 Die Klasse `RobotTask`

Neue Aufträge an den Robot werden als Objekte der Klasse `RobotTask` übergeben, diese können wie nachfolgend beschrieben parametrisiert werden, so daß sich die Regeln, nach denen der Robot das Web durchwandert in vielfältiger Weise steuern lassen. Während der Bearbeitung eines `RobotTasks` können eine Reihe statistischer Angaben zum Fortschritt des Crawl-Vorgangs abgefragt werden. In Abbildung 6.4 ist beispielhaft ein `RobotTask` dargestellt. Dieser wurde so konfiguriert, daß ausgehend von der Start-URL http://localhost/1.htm interne Links mit einer maximalen Tiefe von drei verfolgt werden. Dabei wird auf das erneute Laden von in der Datenbank gespeicherten Dokumenten verzichtet. Der Download einer Datei wird nach maximal 10kB abgebrochen. Der zum `RobotTask` hinzugefügte `LinkFilter` bewirkt, daß URLs, die eine "7ënthalten ignoriert werden. Details zum Konzept der *Filter* und *Classifier* werden nachfolgend beschrieben.

Folgende Parameter eines `RobotTasks` können konfiguriert werden:

- `StartURI`
  Mit der angegebenen URL wird der Crawl-Vorgang begonnen.

- `headOnly`
  Wird dieser Parameter auf `true` gesetzt, dann wird der Robot lediglich per HTTP-Kommando `HEAD` die `StartURI` laden.

- `depth`
  Spezifiziert die maximale Tiefe, mit der ausgehend von der `startURI` Links verfolgt werden.

- `restriction`
  Mittels vordefinierter Konstanten kann bestimmt werden, ob der Robot ausgehend von der `StartURI`nur internen Links, nur externe Links, nur Verweise auf Dateien in Unterverzeichnissen oder alle gefundenen Links besuchen soll.

- `obeyRobotExclusion`
  Der Robot beachtet das *Robot Exclusion Protocol*, wenn dieser Parameter den Wert `true` annimmt.

- `expiry`
  Hier kann der Zeitraum angeben werden, in dem der `RobotTask` Gültigkeit besitzt. Nach Ablauf des Zeitraums wird der Task als obsolet betrachtet und entweder gar nicht erst in Bedienung genommen oder abgebrochen.

```
private class TestLinkFilter implements LinkFilter {

   private String string;

   public TestLinkFilter(String string) {
      this.string = string;
   }

   public boolean filter(RobotLink robotLink, RobotHtmlNode robotHtmlNode,
                         QueueEntry qe) {
      if(robotLink.getLink().getToNode().getUri().indexOf(string) != -1) {
         return false;
      }
         else {
            return true;
         }
   }
}

...

rt = new RobotTask(new SimpleUri("http://localhost/breitensuche/1.htm"),
                   3, RobotTask.INTERNAL, this);
rt.setCheckDatabase(true);
rt.setMaxPageSize(10000);
rt.addLinkFilter(new TestLinkFilter("7"));
robot.scan(rt);

...
```

**Abbildung 6.4:** Beispiel eines Crawl-Auftrags

- `maxDownloadURIs`
  Der Robot lädt maximal die hier angebene Zahl von Dokumenten aus dem Web.

- `checkDatabase`
  Nimmt dieser Parameter den Wert `false` an, dann werden auch bereits im Scone-Cache vorhandene Dokumente aus dem Netz geladen.

- `updateDate`
  Alle Objekte im Scone-Cache, die älter sind, als das im Parameter `updateDate` angegebene Datum, werden erneut geladen.

- `maxPageSize`
  Die maximale Anzahl an Bytes, die pro Seite aus dem Web geladen werden.

- `maxDownloadTime`
  Dauert der Download eines Dokuments länger als `maxDownloadTime`, so wird die Verarbeitung des Dokuments abgebrochen.

- `doContenSeenTest`
  Nimmt der Robot einen *Content-Seen-Test* vor, so wird die weitere Untersuchung eines Abschnitts abgebrochen, wenn der Inhalt der aktuellen Seite bereits unter einer anderen URL gelesen wurde.

- `requireSourceCode`
  Hier kann spezifiziert werden, ob der Seiten-Quelltext benötigt wird. Dokumente, deren Quelltext sich nicht im Scone-Cache befindet werden erneut geladen.

- `Classifier und Filter`
  Zu einem `RobotTask` lassen sich mehrere Classifier und Filter hinzufügen. Details werden in Abschnitt 6.4.2 beschrieben.

Während des Ausführung eines `RobotTasks` werden die im folgenden aufgeführten Statistiken aktualisiert. Desweiteren können über den `RobotTask` alle gefundenen Objekte abgefragt werden. Tabelle 6.1 zeigt das Protokoll der Ausführung des in Abbildung 6.4 dargestellten Auftrags, wobei die in Abbildung 2.9 gezeigt Hypertext-Struktur verwendet wurde. Die Dokumente 6,12,13 und 14 befanden sich im Cache, so daß nur noch 10 Dateien geladen werden mußten. Dokument 7 wurde nicht geladen, da der registrierte `LinkFilter` verhindert, daß der Link von Dokument 2 zu Dokument 7 verfolgt wird.

- `arrivalTime`
  Der Zeitpunkt, an dem der Auftrag an den Robot übergeben wurde.

- `startTime`
  Zeitpunkt des Starts der Bearbeitung der `StartURI`.

- `endTime`
  Gibt an, wann der `RobotTask` beendet wurde.

- `checkedURIs`
  Die Anzahl an URIs, die während der Bearbeitung des Auftrags überprüft wurden. Es sei darauf hingewiesen, daß eine URI, die mehrmals gefunden wird auch mehrmals gezählt wird.

- `queuedURIs`
  Dieser Wert gibt an, wie viele Einträge in die Warteschlange geschrieben wurden.

**Tabelle 6.1:** Protokoll der Ausführung des in Abbildung 6.4 dargestellten Crawl-Auftrags. Verwendet wurde die Hypertext-Struktur aus Abbildung 2.9.

| checked URIs | queued URIs | filtered URIs | downloaded URIs | cache Hits |
|---|---|---|---|---|
| 6 | 6 | 0 | 1 | 0 |
| 7 | 6 | 1 | 2 | 0 |
| 9 | 7 | 1 | 3 | 0 |
| 12 | 8 | 1 | 4 | 0 |
| 15 | 10 | 1 | 5 | 0 |
| 18 | 13 | 1 | 5 | 1 |
| 18 | 13 | 1 | 6 | 1 |
| 18 | 13 | 1 | 7 | 1 |
| 19 | 13 | 1 | 8 | 1 |
| 21 | 14 | 1 | 9 | 1 |
| 21 | 14 | 1 | 9 | 2 |
| 21 | 14 | 1 | 9 | 3 |
| 22 | 14 | 1 | 9 | 4 |
| 22 | 14 | 1 | 10 | 4 |

- `filteredURIs`
  Anzahl der ausgefilterten URIs.

- `downloadedURIs`
  Die Zahl der aus dem Web geladenen Dokumente.

- `cacheHits`
  In dieser Variablen werden alle Dokumente gezählt, die im Scone-Cache gefunden wurden und somit nicht geladen werden mußten.

- `resultNodes`
  In dem `Vector` `resultNodes` werden alle gefundenen `RobotHtmlNodes` abgelegt.

- `openURIs`
  Die aktuell vom Robot bearbeiteten URIs.

- `openThreads`
  Anzahl der momentan für diesen `RobotTask` benutzten Threads.

- `stopped`
  Über diesen Parameter kann abgefragt werden, ob der Auftrag abgebrochen wurde.

- `finished`
  Wird der `RobotTask` komplett ausgeführt, dann wird dieser Variable der Wert `true` zugewiesen.

Es gelten die Beziehungen:

$$checkedURIs = queuedURIs + filteredURIs + vorherbereitsgeseheneURIs$$

$$queuedURIs = downloadedURIs + cacheHits$$

### 6.4.2 Classifier und Filter

Von dem Projekt *SPHINX* wurde das Konzept der Classifier übernommen (siehe Abschnitt 3.4) und zu einem allgemeineren Ansatz, der zwischen `PageClassifier`, `PageFiltern`, `LinkClassifier` und `LinkFiltern` unterscheidet, weiterentwickelt. Durch den Entwurf derartiger Objekte lassen sich Regeln definieren, nach denen der Robot beim Durchwandern der Hypertext-Struktur vorgeht. Die Kapselung der Regeln in seperaten Klassen bietet den Vorteil der Wiederverwertbarkeit und besseren Wartbarkeit. Bei der Erstellung von Robot-Anwendungen beschränkt sich der Web-Site spezifische Code auf wenige Klassen. Ändert z. B. eine Suchmaschine die Struktur ihrer Ausgabeseiten, dann kann durch Änderung der entsprechenden *Classifier* und *Filter* schnell eine Anpassung der diese Suchmaschine abfragenden Robot-Applikationen vorgenommen werden. *Classifier* und *Filter* lassen sich zu einem `RobotTask` hinzufügen. Sie werden im Verlauf der Verarbeitung jedes gefundenen Dokuments ausgeführt. Die Zweiteilung in *Classifier* und *Filter* soll eine strikte Trennung der Bewertung einer Seite bzw. eines Links und der Filter-Regeln ermöglichen. So ist es z. B. möglich mehrere Web-Site spezifische *Classifier* zu implementieren, die eine einheitliche Menge von Attributen setzen, so daß dann ein *Filter* geschrieben werden kann, der unabhängig von Implementierungs-Details der *Classifier* Regeln für das Crawlen definiert.

Die Interfaces, die ein *Classifier* bzw. *Filter* implementieren muß werden im Abschnitt 6.5 vorgestellt. Konkrete Beispiele werden in Abschnitt 7.2 präsentiert.

#### `PageClassifier`

`PageClassifier` analysieren die jeweilige `HtmlNode` und versehen diese mit einer Menge von Attributen. Ein Classifier könnte z. B. anhand einer Heuristik entscheiden, welche Sprache eine Seite hat, oder Seiten mit wenig

Fließtext und vielen externen Links als Link-Verzeichnis kennzeichnen. Die zu klassifizierende Seite ist als `HtmlNode` zugreifbar, so daß direkter Zugriff auf eine Vielzahl von Attributen des Dokuments, wie Seitentitel, Keywords und Description besteht. Ein `RobotTask` kann mehrere Classifier enthalten, die nacheinander ausgeführt werden.

`PageFilter`

Die Entscheidung, ob ausgehend von der aktuellen Seite weitere Links verfolgt werden sollen, wird durch den `PageLoaderThread` gesteuert. Dabei wird die bislang gescannte Tiefe und das Ergebnis der Ausführung der `PageFilter` berücksichtigt. Alle Filter werden nacheinander aufgerufen und die UND-Verknüpfung der Rückgabewerte gebildet. Die Filter können bei ihrer Entscheidung auf die von den vorher ausgeführten Classifiern gesetzten Attributwerte zugreifen.

`LinkClassifier`

Die `LinkClassifier` werden zu jedem gefunden Hyperlink aufgerufen, analysieren diesen anhand der Ziel-URL und des Link-Textes [2] und versehen ihn mit Attributen. Ein sehr einfacher `LinkClassifier` könnte z. B. externe Links kennzeichnen.

`LinkFilter`

Für jeden gefundenen Link wird anhand der eingestellten Parameter des `RobotTasks` und der UND-Verknüpfung der Resultate aller für einen `RobotTask` registrierten `LinkFilter` entschieden, ob das verlinkte Dokument geladen werden soll.

## 6.5 Application Programming Interface

Alle relevanten Klassen befinden sich im Package `scone.robot`. Der Robot wurde als Singleton realisiert. Mit `Robot.instance()` erhält man eine Referenz auf das Robot-Objekt, dem dann mit der Methode `scan(RobotTask robotTask)` ein neuer Auftrag zur Ausführung übergeben werden kann. Alle Objekte, die den Robot benutzen, müssen das Interface `RobotUser` (siehe Abbildung 6.5 implementieren. Die Methode `robotNewPage()` wird für

[2] Der Text zwischen `<A>` und `</A>`

```
public interface RobotUser {

public void robotNewPage(RobotHtmlNode robotHtmlNode, RobotTask robotTask);
public void robotTaskFinished(RobotTask robotTask);

}
```

**Abbildung 6.5:** Interface `scone.robot.RobotUser`

jedes während des Scan-Vorgangs gefundene Objekt aufgerufen. Der Parameter `robotTask` soll Objekten, die gleichzeitig mehrere Aufträge gestartet haben, eine Zuordnung der gefundenen Objekte zu den Aufträgen ermöglichen. Wurde ein Auftrag abgeschlossen, dann wird die Methode `robotTaskFinished` des `RobotUsers` aufgerufen.

Wie in Abschnitt 6.4.1 beschrieben wurde werden Crawl-Aufträge durch Objekte der Klasse `RobotTask` spezifiziert. Die elementaren Angabe, wie die Start-URI und die zu verfolgende Link-Tiefe werden im Konstruktor angegeben. Weitere Einstellungen lassen sich dann mittels der entsprechenden Methoden des `RobotTask` Objekts vornehmen. Spezifische Regeln für das Scannen werden durch Kombinationen aus *Classifiern* und *Filtern* beschreiben. Die zu implementierenden Interfaces wurden in Abbildung 6.6 dargestellt. Instanzen von Classifiern und Filtern können mit den Methoden `addPageClassifier`, `addLinkClassifier`, `addPageFilter` und `addLinkFilter` zu einem `RobotTask` hinzugefügt werden. Die Methoden zur Spezifikation der Eigenschaften eines Crawl-Auftrags lassen sich auch während der Bearbeitung des Auftrags aufrufen, so daß es möglich ist einen `RobotTask` während der Ausführug zu rekonfigurieren.

An den spezifischen Anforderungen einer Anwendung angepaßte Warteschlangen lassen sich entwickeln, in dem eine Unterklasse einer vorhandenen Warteschlange gebildet oder das Interface `URLQueue`, welches in Abbildung 6.7 abgedruckt wurde, implementiert wird.

## 6.6 Testumgebung

Der Test eines Robots ist mit einer Reihe von Schwierigkeiten verbunden. Tests sollten nicht im öffentlichen Internet erfolgen, denn ein fehlerhafter Robot kann Webserver-Betreibern einige Probleme bereiten. Der Robot arbeitet im Hintergrund, die aktuellen Aktivitäten können also nicht unmittelbar überwacht werden, insbesondere ist es nur schwer möglich die Kommunikation zwischen Robot und Webserver zu untersuchen. Ein weiterer Aspekt

```
public interface PageClassifier {
    public void classify(RobotHtmlNode robotHtmlNode, QueueEntry qe);
}

public interface LinkClassifier {
    public void classify(RobotLink robotLink, RobotHtmlNode robotHtmlNode,
                         QueueEntry qe);
}

public interface PageFilter {
    public boolean filter(RobotHtmlNode robotHtmlNode, QueueEntry qe);
}

public interface LinkFilter {
    public boolean filter(RobotLink robotLink, RobotHtmlNode robotHtmlNode,
                          QueueEntry qe);
}
```

**Abbildung 6.6:** Interfaces für Classifier und Filter

```
public interface URLQueue {

    void queue(QueueEntry qe);
    QueueEntry dequeue();
    int size();
    int getNumberOfPendingQueueEntries(RobotTask robotTask);
    Vector getPendingQueueEntries(RobotTask robotTask);
    void removeAllQueueEntries(RobotTask robotTask);
    int getNumberOfPendingQueueEntries(SimpleUri uri);
    Vector getPendingQueueEntries(SimpleUri uri);
    boolean isPendingURL(SimpleUri uri, RobotTask robotTask);
    QueueEntry getPendingURL(SimpleUri uri, RobotTask robotTask);
    void removeQueueEntry(QueueEntry qe);

}
```

**Abbildung 6.7:** Interface `scone.robot.URLQueue`

des Testens von Robots ist, daß es sich nur schwerlich nachweisen läßt, ob der Robot einen größeren Crawl-Auftrag korrekt ausgeführt hat, da die zu crawlende Dokumenten-Menge in der Regel unbekannt ist.

Der Test des Robots erfolgte auf einem Athlon PC mit 512 MB RAM unter dem Betriebssystem Microsoft Windows 2000. Für grundlegende Tests wurden Dokumente von einem lokal laufenden Sambar[3] Webserver abgefragt. Sambar bietet den Vorteil, daß die letzten Requests in einem GUI aufgelistet werden, so daß sich sehr einfach nachvollziehen läßt, in welches Reihenfolge der Robot Dokumente abgefragt hat. Mittels einer aus 39 Dateien bestehenden konstruierten Hypertextstruktur wurden der Crawl-Algorithmus überprüft. Die Testdokumente enthielten JavaScript Links, Frames und Imagemaps. Es wurde überprüft, wie der Robot sehr große HTML-Dokumente verarbeitet, und wie der Robot mit Links zu Dokumenten mit einem von `text/html` abweichenden Mime-Type umgeht.

Da Tests, bei denen sich Webserver und Robot auf einer Maschine befinden nur bedingt mit den realen Bedingungen des Internets vergleichbar sind, wurden weitergehende Tests mit den Webservern des Arbeitsbereichs VSIS durchgeführt. Insbesondere wurde der komplette Server print-www.informatik.uni-hamburg.de mit einem sehr umfangreichen Dokumentenbestand zu Testzwecken gecrawlt. Die geöffneten Netzwerkverbindungen wurden dabei mittels des Tools *netstat* überwacht.

Als unerläßliches Hilfsmittel bei den Tests hat sich der `RobotMonitor` erwiesen. Mittels der HTML Oberfläche wurde der Fortschritt der Probeläufe überwacht, so daß bei Fehlerfällen die verursachenden Dokumende ausfindig gemacht werden konnten.

[3] www.sambar.com

# Kapitel 7

# Beispielanwendungen

Nachfolgend werden zwei auf Scone basierende Anwendungen vorgestellt, die den Robot benutzen.

## 7.1 HyperScout

Das Scone-Plugin *HyperScout* (vgl. [WL00]) erweitert die Darstellung von Links in Web-Browsern um ein Popup-Fenster mit zusätzlichen Informationen zum verlinkten Dokument. Angezeigt werden eine Reihe von semantischen Informationen, wie Titel, Autor, Inhalt und Sprache des Dokuments. Überdies wurden Benutzungsinformationen, z.B. das Datum des letzten Besuchs, Hinweise auf durch den Link ausgelöste Aktionen, Informationen zum Link-Status und Infos zur Topologie des Hypertextes in die Darstellung einbezogen. Der Benutzer kann dann unter Einbeziehung der angezeigten Informationen die Entscheidung treffen einen Link zu besuchen und so, das ist die hinter dem Projekt HyperScout stehende Hypothese, schneller und zielgerichteter relevante Informationen finden. Das Beispiel einer solchen Visualisierung ist in 7.1 zu sehen.

Viele der durch den HyperScout dargestellten Informationen setzen voraus, daß das verlinkte Dokument bereits geladen wurde. Frühe Versionen dieses Scone-Plugins benutzen dazu ausschließlich die in den Scone-Caches vorhandenen Daten, die auf frühere Besuche der jeweiligen Sites zurückzuführen sind. Durch die Integration des Robots in das Scone-Framework kann nun der HyperScout auch aktiv im Hintergrund Dokumente aus dem Web laden. Anhand einer Heuristik wird entschieden, welche der verlinkten Dokumente durch den Robot geladen werdens sollen. Einige Dokumente werden per Http-Befehl `GET` angefordert, wobei nur der Anfang der Datei geladen wird. Von anderen URLs wird nur durch das Kommando `HEAD` der

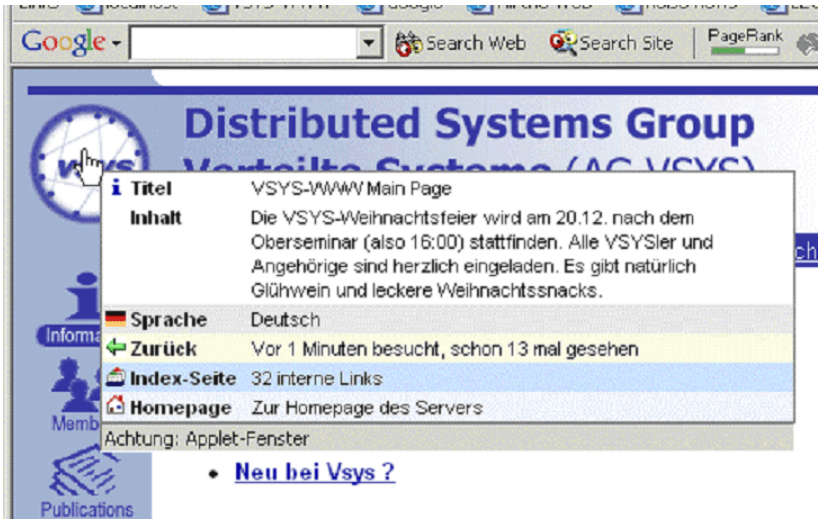


**Abbildung 7.1:** Beispiel einer HyperScout Einblendung

HTTP-Request-Header angefordert. Dadurch kann sehr schnell die Validität eines Links überprüft und Informationen zu Dateigröße und Mime-Type gewonnen werden.

## 7.2 Rückspiegel

Oftmals sind Seiten, die einen Link, auf ein gerade im Web-Browser angezeigtes Dokument enthalten thematisch dem aktuellen Text sehr ähnlich. Ein vergleichbarer Sachverhalt findet sich im wissenschaftlichen Publikationswesen. Wurde zu einem Thema ein Text als besonders relevant eingestufft, so erweisen sich vielfach Quellen, die diesen Text zitieren als ebenfalls interessant. Die Architektur des Webs sieht jedoch nur unidirektionale Links vor. Es ist also ohne Zuhilfenahme weiterer Datenbestände nicht möglich die auf ein Dokument verweisenden Links zu ermitteln.

Die Suchmaschine *Google* [1] (siehe Abschnitt 3.2) bietet die Möglichkeit nach Seiten zu suchen, die eine Link zu einer bestimmten URL enthalten. Möglich wird diese Funktion dadurch, daß die Strukturinformationen über die vom Google-Robot gefundenen Dokumente in der Datenbank der Suchmaschine abgelegt werden.

Im Verlauf dieser Studienarbeit wurde eine „Rückspiegel“ genannte Applikation entwickelt. Diese generiert automatisch Anfragen an Google zu den eingehenden Links der gerade angezeigten Seite. Der Robot lädt dann die

[1] www.google.com

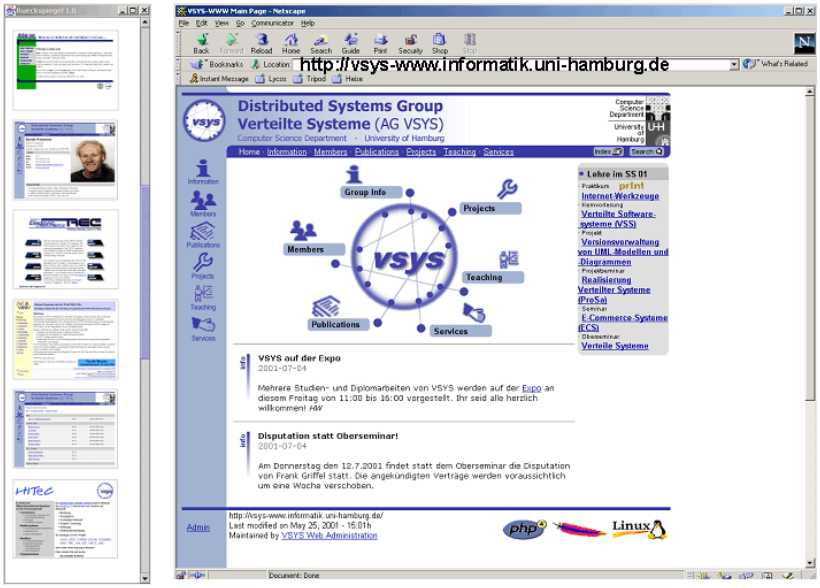


**Abbildung 7.2:** Screenshot des Programms „Rückspiegel“

Goolge Ergebnis-Seite, filtert die internen Google Links und lädt die von Google aufgelisteten Web-Seiten, von denen dann Thumbnails erstellt werden. Ein Screenshot des „Rückspiegels“ ist in Abbildung 7.2 dargestellt.

Nur ein Teil der Links der Antwortseite verweisen wirklich auf die Ergebnisse der Suchanfrage. Interne Links sollen nicht verfolgt werden. Abbildung 7.3 verdeutlicht, welche Links durch den Robot zu verfolgen sind. Das selektive Vorgehen des Robots wird durch einen einfachen `LinkClassifier` gesteuert, der in Abbildung 7.4 dargestellt wird. Alle nicht zum Ergebnis gehörenden Links werden daran erkannt, daß sie die Zeichenkette „google“ enthalten. Anhand des von diesem Classifier gesetzten Attributs „GoogleLink“ kann dann der entsprechende `LinkFilter` entscheiden, welche Links der Robot verfolgen soll.

Der Rückspiegel bietet unter Ausnutzung bestehender Technologien und Dienste eine neue Sicht auf das Web. Die Leistungsfähigkeit von Suchmaschinen-Robots wurde dabei mit der Flexibilität eines client-seitig arbeitenden Robots kombiniert.

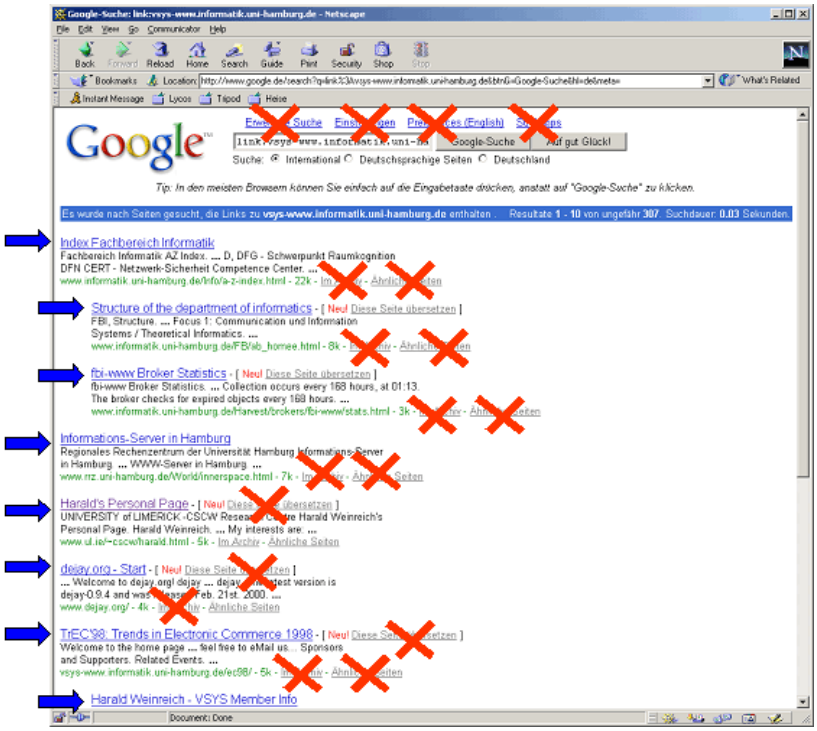


**Abbildung 7.3:** Der Robot muß die durchgestrichenen Links ignorieren und die anderen verlinkten Dokumente laden.

```
private class RueckspiegelLinkClassifier implements LinkClassifier {

    public void classify(RobotLink robotLink,
                         RobotHtmlNode robotHtmlNode,
                         QueueEntry qe) {
        String urlText = robotLink.getLink().getToNode().getUri();
        if(urlText.indexOf("google") != -1) {
            robotLink.setAttribute("GoogleLink", "Yes");
        }
    }
}
```

**Abbildung 7.4:** `scone.examples.Rueckspiegel.RueckspiegelPageClassifier`

# Kapitel 8

# Fazit

Im Verlauf dieser Studienarbeit wurde ein Web-Robot für das Scone-Framework entwickelt. Dabei konnte auf den Grundlagen Scones aufgebaut werden. So existierte bereits ein objektorientiertes Modell für das Web. Der nicht-triviale Vergleich von URIs mußte nicht programmiert werden, da die Klasse `scone.netobjects.SimpleUri` bereits über entsprechende Methoden verfügt. Mit den Scone-Caches waren leistungsfähige Datenstrukturen vorhanden, in denen die durch den Robot geladenen Dokumente abgelegt werden konnten. Elementar wichtig für die Entwicklung des Robots war der Scone `DocumentParser`. So mußte für den Robot kein eigenständiger Parser entwickelt werden und die geladenen Dokumente konnten in Form von entsprechenden Scone-Objekten verarbeitet werden.

Unterschätzt wurde die Komplexität der Aufgabe einen Web-Robot zu entwickeln. Das ursprünglich für die Netzwerkkommunikation vorgesehene Package `java.net` erwies sich als unzureichend, da es nicht möglich war `timeouts` anzugeben. Da sich eine Eigentwicklung als zu aufwendig erwies, wurde letztendlich das Package `HTTPClient` verwendet. Der Robot wurde als multi-threaded arbeitendes System konzipiert. Dabei war an vielen Stellen zu beachten, daß nur eine hinreichende Synchronisation der Threads einen fehlerfreien Betrieb des Robots gewährleistet. Zugriffe auf gemeinsam genutzte Datenstrukturen mußten synchronisiert werden, Operationen, die sich auf mehrere Datenstrukturen bezogen wurden durch Sperr-Objekte abgesichert. Ebenfalls unterschätzt wurden der hohe Ressourcenbedarf eines Robots. Die Zahl der gefundenen Objekte steigt typischerweise exponentiell zur Link-Tiefe. Für einen client-seitig arbeitenden Robot ist es deshalb schon aufgrund der begrenzten Speicherausstattung nur durchführbar mit einer auf zwei-drei Links begrenzten Tiefe zu scannen. Das Testen des Robots hat sich als äußerst schwierig erwiesen, da sich insbesondere Features, die sich auf die Netzwerkkommunikation beziehen, nur unter erheblichem Aufwand testen lassen. Um z. B. zu überprüfen, ob die Verbindug zwischen dem

Robot und einem Web-Server wirklich zwischen den Requests offen gehalten wird, müßte eine aufwendige Analyse der übertragenen Daten durchgeführt werden.

In dieser Arbeit wurde das Konzept der Classifier und Filter implementiert. Die Entwicklung entsprechender Objekte wurde nur am Rande, im Rahmen des Anwendungsbeispiels „Rückspiegel“, durchgeführt. Interessant wäre es anhand aufwendigerer Anwendungen festzustellen, ob sich dieses Konzept bewährt.

Mit Scone steht nun ein leistungsfähiges Framework zur Entwicklung von Tools zur Unterstützung bei der Navigation im Web zur Verfügung. Da Scone einen Proxy-Server bietet, ist die Anwendung von Scone nicht auf den Rechner des Benutzers beschränkt. Es wären auch Anwendungen realisierbar, bei denen Scone zentral auf einem Server läuft und Scone-Anwendungen ohne weitere Installationen auf den Rechnern der Benutzer nutzbar werden. Insbesondere die Entwicklung von Werkzeugen, welche Arbeitsgruppen bei der Informationssuche im Web unterstützen, erscheint dem Autor eine reizvolle zukünftige Anwendung von Scone zu sein.

# Literaturverzeichnis


[Alv95] Alvestrand, H.: *RFC 1766: Tags for the Identification of Languages*, März 1995.

[BKM+00] Broder, Andrei, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins und Janet Wiener: *Graph structure in the web: experiments and models.* In: *Proceedings of 9th International World Wide Web Conference*, 2000.

[BL94] Berners-Lee, T.: *RFC 1630: Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*, Juni 1994.

[BLC95] Berners-Lee, T. und D. Connolly: *RFC 1866: Hypertext Markup Language — 2.0*, November 1995.

[BLCL+94] Berners-Lee, Tim, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen und Arthur Secret: *The World-Wide Web.* Communications of the ACM, 37(8):76–82, 1994.

[BLFF96] Berners-Lee, T., R. Fielding und H. Frystyk: *RFC 1945: Hypertext Transfer Protocol — HTTP/1.0*, Mai 1996.

[BLMM94] Berners-Lee, T., L. Masinter und M. McCahill: *RFC 1738: Uniform Resource Locators (URL)*, Dezember 1994.

[BM99] Barrett, R. und P. P. Maglio: *Intermediaries: An approach to manipulating information streams.* IBM Systems Journal, 38(4):629–641, 1999.

[BP94] Bra, P. De und R. D. J. Post: *Information Retrieval in the World-Wide Web: Making Client-Based Searching Feasible.* Computer Networks and ISDN Systems, 27(2):183–192, 1994.

[BP98] Brin, Sergey und Lawrence Page: *The anatomy of a large-scale hypertextual Web search engine.* Computer Networks and ISDN Systems, 30(1–7):107–117, 1998.

[CCRY98] Chen, H., Y. Chung, M. Ramsey und C. Yang: *An intelligent personal spider (agent) for dynamic internet/intranet searching*, 1998.

[CGMP98] Cho, Junghoo, Hector García-Molina und Lawrence Page: *Efficient crawling through URL ordering.* Computer Networks and ISDN Systems, 30(1–7):161–172, 1998.

[Che96] Cheong, Fah-Chun: *Internet agents: spiders, wanderers, brokers, and 'bots.* New Riders, Indianapolis, Ind., 1996.

[CvdBD99] Chakrabarti, Soumen, Martin van den Berg und Byron Dom: *Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery.* In: *8th World Wide Web Conference*, Toronto, May 1999.

[Dem] Demetrios, Marios Dikaiakos: *WebRACE: A Distributed WWW Retrieval, Annotation, and Caching Engine.*

[DEW97] Doorenbos, Robert B., Oren Etzioni und Daniel S. Weld: *A Scalable Comparison-Shopping Agent for the World-Wide Web.* In: Johnson, W. Lewis und Barbara Hayes-Roth (Herausgeber): *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, Seiten 39–48, Marina del Rey, CA, USA, 1997. ACM Press.

[Eic95] Eichmann, Dave: *Ethical Web Agents.* In: Anonymous (Herausgeber): *Second International World-Wide Web Conference: Mosaic and the Web, Chicago, IL, October 17–20, 1994*, Urbana, IL 61801, USA, 1995. National Center for Supercomputer Applications, University of Illinois at Urbana-Champaign.

[FB96a] Freed, N. und N. Borenstein: *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, November 1996.

[FB96b] Freed, N. und N. Borenstein: *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, November 1996.

[FB96c] Freed, N. und N. Borenstein: *RFC 2049: Multipurpose Internet Mail Extension (MIME) Part Five: Conformance Criteria and Examples*, November 1996.

[FGM+99] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee: *RFC 2616: Hypertext transfer protocol – HTTP*, 1999.

[Fie95] Fielding, R.: *RFC 1808: Relative Uniform Resource Locators*, Juni 1995.

[FKP96] Freed, N., J. Klensin und J. Postel: *RFC 2048: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*, November 1996.

[GBKS00] Gandhi, R., Benjamin B. Bederson, G. Kumar und Ben Shneiderman: *Domain Name Based Visualization of Web Histories in a Zoomable User Interface.* In: *DEXA Workshop*, Seiten 591–600, 2000.

[GHJ95] Gamma, Erich, Richard Helm und Ralph Johnson: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[HN99] Heydon, Allan und Marc Najork: *Mercator: A Scalable, Extensible Web Crawler.* World Wide Web, 2(4):219–229, 1999.

[Kos93] Koster, Martijn: *Guidelines for Robot Writers.* URL,www.robotstxt.org/wc/guidelines.html, 1993.

[Kos94] Koster, Martijn: *A Standard for Robot Exclusion.* URL,www.robotstxt.org/wc/norobots.html, 1994.

[Kos95] Koster, Martijn: *Robots in the Web: threat or treat?* ConneXions, 4(4), April 1995.

[KT00] Kobayashi, Mei und Koichi Takeda: *Information retrieval on the web.* ACM Computing Surveys, 32(2):144–173, 2000.

[Kun02] Kunz, Christopher: *Webpalette.* iX Magazin für professionelle Informationstechnik, 5:78–81, 2002.

[MB98] Miller, R. und K. Bharat: *SPHINX: a framework for creating personal site-specific Web crawlers.* In: *Proceedings of the 7th World Wide Web Conference*, 1998.

[MB01] Mayer, Matthias und Benjamin B. Bederson: *Browsing Icons: A Task-Based Approach for a Visual Web History.* Technischer Bericht, University of Maryland, Human Computer Interaction Laboratory (HCIL), 2001.

[ME96] Monge, A. und C. Elkan: *The webfind tool for finding scientific papers over the Worldwide Web*, 1996.

[Mün01] Münz, Stefan: *SELFHTML*. URL,selfhtml.teamone.de, Okt. 2001.

[Moo96] Moore, K.: *RFC 2047: MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*, November 1996.

[OW97] Oaks, Scott und Henry Wong: *Java Threads*. O'Reilly, Cambridge;Paris;Tokyo, 1997.

[PBMW98] Page, Lawrence, Sergey Brin, Rajeev Motwani und Terry Winograd: *The PageRank Citation Ranking: Bringing Order to the Web*. Technischer Bericht, 1998.

[RHJ98] Raggett, D., A. Le Hors und I. Jacobs: *HTML 4.0 specification*, 1998.

[Sie96] Siegel, D.: *Creating Killer Web Sites*. Hayden Books, 1996.

[Ste94] Stevens, W. Richard: *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Professional Computing Series. Addison-Wesley, Sydney;Amsterdam;Tokyo, 1994.

[Tan97] Tanenbaum, Andrew S.: *Computernetzwerke*. Prentice Hall, München, et.al, 3 Auflage, 1997.

[Tur96] Turau, Volker: *Algorithmische Graphentheorie*. Addison Wesley, 1996.

[Tur98a] Turau, Volker: *Eine empirische Analyse von HTML-Dokumenten im WWW*. Technischer Bericht, FH Wiesbaden, Fachbereich Informatik, Jan. 1998.

[Tur98b] Turau, Volker: *Web-Roboter - Das Aktuelle Schlagwort*. Informatik Spektrum, 21(3):159–160, 1998.

[Wic99] Wichmann, A.: *Aufbau und Techniken von Suchmaschinen für das WWW*, 1999.

[Wil99] Wilde, Erik: *Wilde's WWW: technical foundations of the World Wide Web*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999.

[WL00] Weinreich, Harald und Winfried Lamersdorf: *Concepts for improved visualization of Web link attributes*. In: *Proceedings of the 9th International WWW Conference.*, 2000.

[YN02] Yamada, Seiji und Norikatsu Nagino: *Constructing a Personal Web Map with Anytime-Control of Web Robots*. International Journal of Cooperative Information Systems, 11(1-2):1–19, 2002.

[ZYD] Zeinalipour-Yazti, Demetris und Marios Dikaiakos: *High-Performance Crawling and Filtering in Java*.

# Anhang A

# Inhalt der CD-ROM

Die beiliegende CD-ROM enthält die das komplette Scone-System einschließlich aller Quelltexte.

**Pfad:**

| | |
|---|---|
| /readme.txt | Anmerkungen zu Installation und Betrieb von Scone |
| /build/ | Dateien zum Compilieren des Scone-Frameworks |
| /config/ | Konfigurations-Dateien. |
| /doc/scone/ | API Dokumentation von Scone |
| /doc/tutorial/ | Tutorial zum Erstellen von Scone-Plugins |
| /ressources/ | Ressourcen der verschiedenen Plugins |
| /setup/ | Für die Installation von Scone benötigte Dateien |
| /src/ | Quelltexte des Scone-Systems und einiger Plugins |
| /src/scone/examples/rueckspiegel/ | Quelltext des „Rückspiegels“ |
| /src/scone/robot/ | Quelltext des Robots |
| /src/scone/thumbnailgenerator/ | Quelltext des Thumbnail-Generators |