WASH

Evgeni Genev

Universität Freiburg

18.02.2008

Motivation

- Früher – Statische Webinhalte
- Heute – Dynamische Webinhalte
  - Clientseitige Anwendungen
    - JavaScript, VBScript...
  - Serverseitige Anwendungen
    - PHP, mod_perl, JSP...
    - CGI – Common Gateway Interface

Der Wash / CGI Approach

- EDSL für serverseitiges Webscripting
  - mit Sitzungen (Sessions)
    *(später in Session-Sprache)
  - und Formulakombinatoren
    *(später in Widget-Sprache)

Exkurs: DSL und EDSL

- Domänenspezifische Programmiersprache
  - formale Sprache
  - für ein bestimmtes Problemfeld

- Domänenspezifische **eingebettete** Programmiersprache
  - eine echte Untermenge der Wirtssprache
  - gesenkter Implementierungsaufwand
  - gesenkte Fehleranfälligkeit

Was sind Sitzungen?

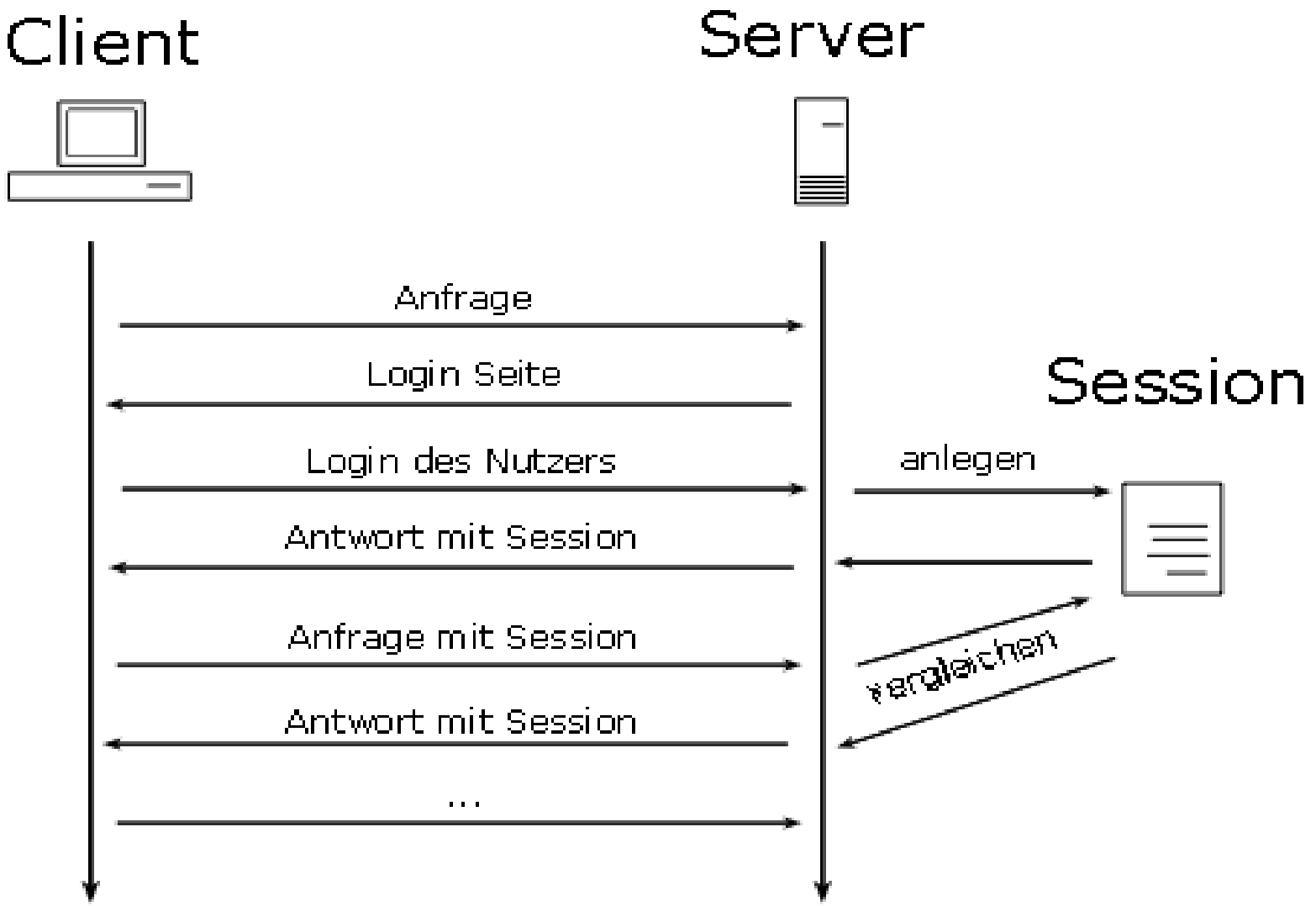
- Eine alternierende Reihe von Webformularen und generierten Antwortdokumenten
  - von dem gleichen Benutzer
  - von dem gleichen Browser

Personalisierung der Benutzer

POST url0; FORM f1; POST url1; FORM f1;...

- Voraussetzung - ein Sitzungszustand sichtbar für den Klienten und für den Server

Session Beispiel



Sitzungszustand

- Das zugrunde liegende Protokol HTTP ist zustandslos und unterstützt keine Sitzungen
- Gebräuchliche Technik – speichere ein Token in jedes FORM und übergebe es mit jedem POST
  - der Zustand mit der URL übergeben
  - der Zustand auf dem Klient als Textdatei speichern (Cookies)
  - die Zustandsdaten in versteckte Eingabe-Felder speichern und übergeben

Sitzungen

- Probleme
  - Der Zurück-Knopf
  - Das Kopieren von Sitzungen
  - Das Lesezeichen-Problem

Wie geht WASH/CGI mit diesen Problemen um?

* später in „*Session-Sprache*“

WASH / CGI Teilprachen

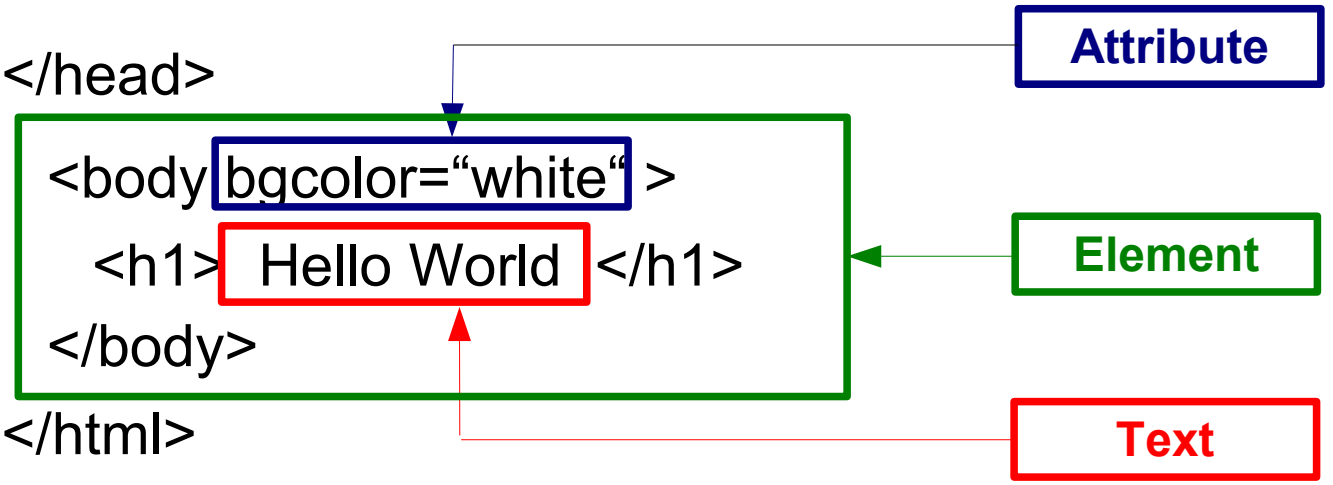
- **Dokument-Sprache**
- Widget-Sprache
- Session-Sprache

Dokument-Sprache

- Erzwingt die Generierung von wohlgeformten XHTML Dokumente
- XHTML Dokumente bestehen aus:
  - Element-Knoten
  - Attribut-Knoten
  - Text-Knoten

Wohlgeformtes XHTML-Dokument

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Hello World</title>
</head>
<body bgcolor="white" >
  <h1> Hello World </h1>
</body>
</html>
```



Dokument-Sprache von XHTML nach WASH

- Konstruktoren für die Element-Knoten
  - sind für jedes Element vordefiniert

z.B.:

<html>...</html>	wird durch die *html*-Funktion implementiert	
<head>...</head>	*head*-Funktion	
<title>...</title>	*title*-Funktion	
<body>...</body>	*body*-Funktion	
...	*h1*-Funktion	usw.

Dokument-Sprache - Funktionen

- Konstruktoren für die Element-Knoten
  - Nehmen als Argumente Sequenzen von:
    - Kind-Elementen
    - Text-Knoten
    - Attributen
  - Sequenzen werden mit der für Monaden üblichen do-Notation, oder den Operatoren '>>' und '##' verknüpft

Dokument-Sprache - Funktionen

- z.B. die Funktion *body*:

```
body    :: (Monad m, AdmitChildBODY e)
                => WithHTML BODY m a -> WithHTML e m a
```

Beispiel:

```
body (do attr "bgcolor" "white"
         h1 (text "Hello World"))
```

- Wofür stehen 'AdmitChildBODY e' und 'WithHTML BODY m a'?

„AdmitChildBODY“ und „WithHTML BODY“

- Der Parametertyp '*e'* in *'WithHTML e m a'* steht hier für den Name des Eltern-Knotens
- *'AdmitChildBODY e'* muss ein BODY Kind erlauben
- *'WithHTML BODY m a'* – nur gültige Kinder-Elemente von BODY können in diesem Kontext benutzt werden*
- Im Weiteren sehen wir, dass die anderen Funktionen ähnlich definiert werden

* Dadurch wird die (quasi)-XHTML-Gültigkeit erzwungen

Textknoten
und die leere Sequenz

- *text*

```
text:: (Monad m, AdmitChildCDATA e)
      => String -> WithHTML e m ()
```

- *empty*

```
empty   :: Monad m
          => WithHTML x m ()
```

- Wie wird der Beispielkode auf der Seite 11 generiert?

WASH / CGI - einfache Beispielanwendung

- Teil 1

```
module Main where

import Prelude hiding (head, span, div, map)
import WASH.CGI.CGI

main =
  run mainCGI

...
```

WASH / CGI - einfache Beispielanwendung

- Teil 2

```
...

mainCGI =
   ask (html
          (do head (title (text "Hello World"))
              body (do attr "bgcolor" "white"
                       h1 (text "Hello World"))))
```

standardPage

- Der Standard Dokument Wrapper

```
standardPage ttl nodes =
    html (do head (title (text ttl))
             body (h1 (text ttl) >> nodes))
```

  - Definiert ein parametrisiertes Dokument
  - Illustriert die Verwendung von Haskell-Funktionen zur Erstellung eigener Abstraktionen in einem WASH-Programm

WASH / CGI - einfache Beispielanwendung v2

```
module Main where

import Prelude hiding (head, span, div, map)
import HTMLMonad
import CGI

main =
  run mainCGI

mainCGI =
  ask (standardPage "Hello World" empty)
```

WASH / CGI Teilprachen

- Dokument-Sprache
- **Widget-Sprache**
- Session-Sprache

Widget-Sprache

- Widgets – Komponente einer grafischen Benutzeroberfläche
- Ein Widget-Konstruktor
  - erstellt ein XHTML-Element mit zugehörigen Attributen
  - und zusätzlich gibt noch ein Eingabe-Handle zurück
    - Das Handle wird meistens an eine Variable gebunden - dadurch wird die weitere Interaktion zwischen Klient und Server ermöglicht

Widgets

- Wir werden die folgenden Eingabeelemente betrachten
  - Formulare
  - Eingabeknöpfe
  - und Eingabefelder für Texteingabe
- Wir betrachten die Folgenden aus Zeitmangel nicht:
  - Radiobutton
  - Checkbutton
  - usw.

Widget-Sprache - Funktionen

- Formular

```
makeForm :: (AdmitChildFORM context) =>
                    WithHTML FORM CGI a -> WithHTML context CGI ()
```

- Eingabeknopf

```
submit0 :: (CGIMonat cgi, AdmitChildINPUT context) =>
                CGI () -> HTMLField cgi INPUT context ()
```

Widget-Sprache – Beispiel 1

- Formular + Eingabeknopf Beispiel

```
main =
    run page1
page1 =
    ask (standardPage "Hello World!" (makeForm myinfo1))
myinfo1 =
    do p (text "This is my second CGI program!")
        submit0 page2 (attr "value" "Click for my hobbies")
...
```

Widget-Sprache – Beispiel 1

- Formular + Eingabeknopf Beispiel

```
...
page2 =
    ask (standardPage "My hobbies are" (makeForm myinfo2))
myinfo2 =
    ul (do li (text "swimming")
           li (text "music")
           li (text "skiing"))
```

Widget-Sprache

- Texteingabefeld

```
textInputField :: (AdmitChildINPUT context) =>
                  HTMLField context (InputField String INVALID)
```

- Eingabeknopf

```
submitButton :: (CGIMonad cgi, AdmitChildINPUT context,
                 InputHandle h) =>
                 h INVALID -> (h VALID -> cgi ()) ->
                 HTMLField cgi INPUT context ()
```

Widget-Sprache – Beispiel 2

- Teil 1

```
...
main =
    run page1
page1 =
   standardQuery "Hello World!" $
   do p (text "This is my third CGI program!")
            pname <- p (do text "Enter your name "
                                   textInputField (attr "size" "10"))
            submit pname page2 (attr "value" "Click for my hobbies")
...
```

Widget-Sprache – Beispiel 2

- Teil 2

```
...
page2 pname =
  standardQuery "My hobbies are" $
  do p (text "Hi, " ## text (value pname) ## text "!")
       ul (do li (text "swimming")
              li (text "music")
              li (text "skiing"))
```

WASH / CGI Teilsprachen

- Dokument-Sprache
- Widget-Sprache
- **Session-Sprache**

Session-Sprache der WASH Approach

- In den meisten Sprachen wird eine Anwendung durch mehrere Skripte implementiert
  - großer Synchronisierungsaufwand
    - die Skripten müssen den gleichen Konzept der Zustandsspeicherung folgen
    - die Skripte müssen gegenseitig ihre Namen kennen
  - Aus dem gleichen Grund wie oben ist das auch eine „gute“ Fehlerquelle

Session-Sprache der WASH Approach

- In WASH ist die ganze Anwendung ein zusammenhängendes Haskell-Programm. Dadurch:
  - entfällt der Synchronisierungsaufwand
  - der Session-Zustand wird transparent für den Programmierer
- Wie wird der Sessionzustand übergeben, sehen wir später

Session-Sprache - Funktionen

- Sitzungen werden durch 2 Operatoren konstruiert

```
ask :: WithHTML XHTML_DOCUMENT CGI a -> CGI ()
tell  :: (CGIOutput a) => a -> CGI ()
```

  - *ask* – stellt eine Frage in Form einer HTML-Seite
    - kann zur weiteren Interaktion führen
  - *tell* – nimmt einen beliebigen Wert, der zu einer HTTP-Antwort konvertiert werden kann, und liefert die entsprechende CGI-Aktion
    - beendet die Interaktion, d.h. erlaubt keine Fortsetzung der Sitzung

„ask“ und *„tell“*

- *ask*

```
mainCGI =  ask (standardPage "Hello World" empty)
```

- *tell*

```
mainCGI = tell (Location $ URL "http://whereveryouwantto.com/")
```

- **Wichtig!** Die CGI-Aktionen sind keine IO-Aktionen

Session-Sprache - Funktionen

- Um den Schritt von CGI- zu IO-Aktionen(und umgekehrt) zu machen, braucht man noch 2 Funktionen

```
run :: CGI () -> IO ()
io   :: (Read a, Show a) => IO a -> CGI a
```

- Haskell erwartet eine IO-Aktion als Hauptprogramm – „*run*“ wandelt eine CGI-Aktion in eine IO-Aktion um
- „*io*“ wandelt eine IO-Aktion in eine CGI-Aktion

„run“ und *„io“*

- Wir haben schon gesehen wie *run* funktioniert

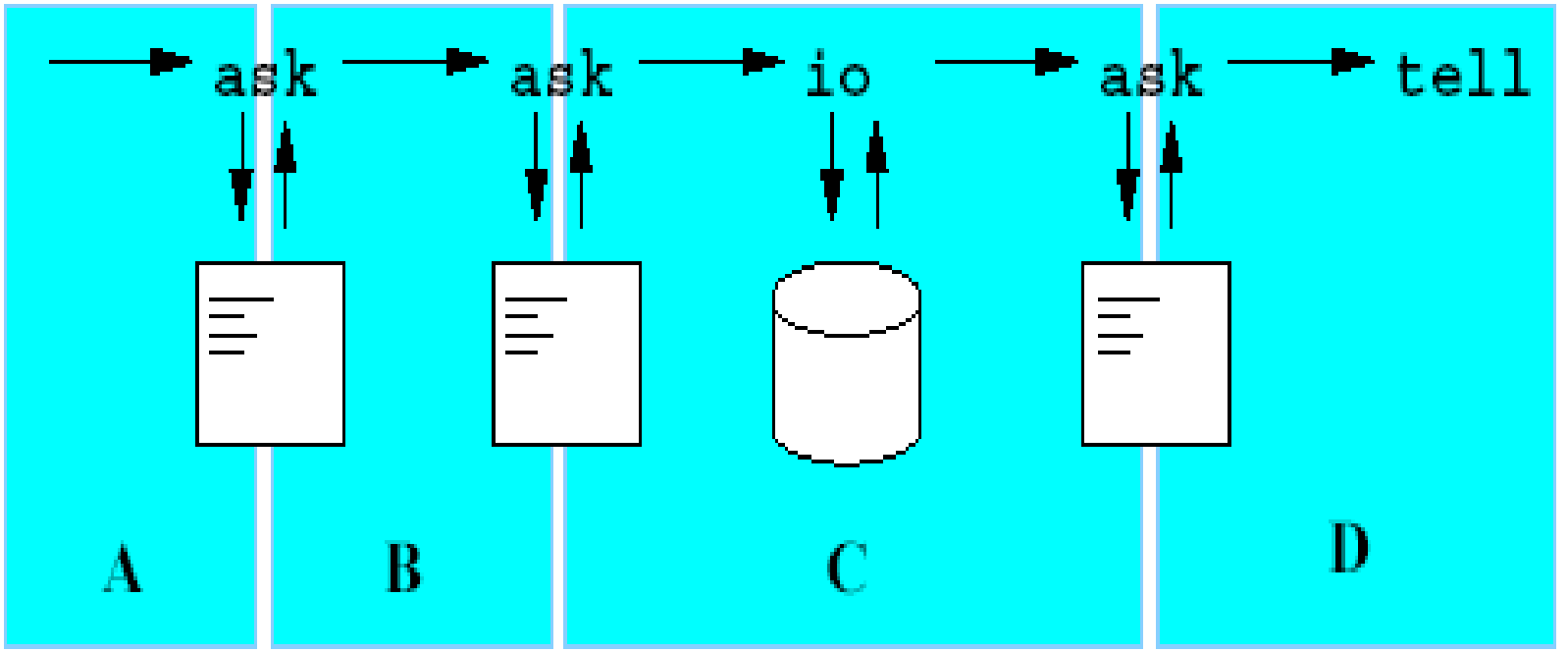
```
main = run mainCGI
```

- *io* (aus „*GuessNumber.hs*“ auf der WASH - Webseite)

```
mainCGI =
  io (randomRIO (1,100)) >>= \ aNumber ->
  standardQuery "Guess a number" $
  ...
```

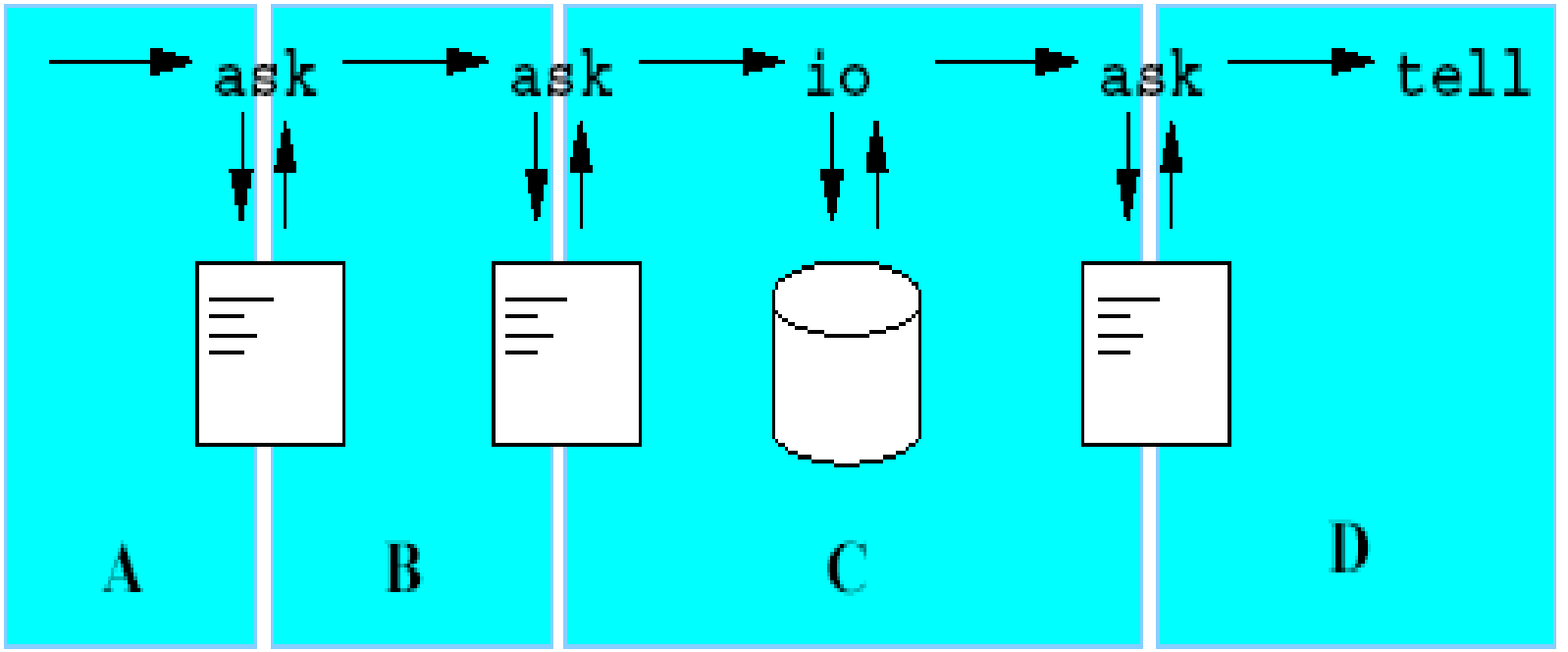
- Durch *„ask“* und *„io“* wird ein abstraktes Niveau aufgebaut

WASH/CGI Session



- „*ask*“ startet eine neue Session, die von „*tell*“ beendet wird
- „*io*“ wird innerhalb einer CGI-Aktion ausgeführt

WASH/CGI Session



- jeder Antwortdokument hat ein verstecktes Eingabefeld, das alle Antworte auf „*io*“ und „*ask*“ beinhaltet, die zu diesem Formular führen.

Session-Sprache - Beispiel

- Die Beispielanwendung GuessNumber von der WASH/CGI: Gallery

GuessNumber.hs

Zusammenfassung

- WASH bietet einige Vorteile
  - Gesenkte Fehleranfälligkeit
  - XHTML-Kompatibilität
  - Transparente Sitzungen