Fallstudie: der Haskell Web Server

Unser Tagesmenü:

- Der Haskell Web Server — eine Fallstudie in
  - reaktiver,
  - nebenläufiger,
  - hochverfügbarer Programmierung.
- Simon Marlow: Writing High-Performance Server Applications, Haskell Workshop 2000.
- Disclaimer: übersetzt nicht, da für `ghc-5.x`
  - $\Longrightarrow$ paradigmatisch.

Warum ein Web Server in Haskell?

- Einfaches, aber nicht triviales Protokoll — `HTTP`
- Nebenläufigkeit
  - viele Requests gleichzeitig
- Geringe Latenz
  - schnell Antworten
- Fehlertoleranz und Verfügbarkeit
  - Fehler im Server, Fehler auf Clientseite, Konfigurierbarkeit

Nebenläufigkeitsmodell

- Betriebssystemprozesse
  - Einfach, mehrprozessorfähig; aber: schwergewichtig
- Betriebssystemthreads
  - Einface IPC, mehrprozessorfähig; aber: schwierig;
- Ein Prozess (`select()` oder asynchrone I/O)
  - schnell; aber: mangelnde Abstraktion
- Benutzerthreads (Concurrent Haskell)
  - Einfach, leichtgewichtig, schnell; aber: nicht mehrprozessorfähig.

Struktur

Top-Level Schleife:

```
acceptConnections conf ps sock = do
(h, SockAddrInet port haddr) <- accept sock
forkIO ( (talk conf ps h haddr  `finally`  (hClose h))
          `catch`
         (\e -> trace ("servlet died: "  ++ show e) (return  ()))
       )
acceptConnections conf ps sock
```

- Aufräumen mit `finally`

Serving Requests

1. Request lesen
2. Request parsieren
3. Antwort generieren
4. Antwort senden
5. Wenn Verbindung aufrecht erhalten ($keep\ alive$), dann zurück zu 1.

Request lesen

```
getRequest :: Handle -> IO [String]
getRequest h = do
  l <- hGetLine h
  if (emptyLine l) then getRequest h else getRequest' l h

getRequest' l h = do
  if (emptyLine l) then return []
     else do l' <- hGetLine h; ls <- getRequest' l' h
             return (l:ls)
```

Request parsieren

- Konkrete Syntax:

```
GET /~cxl/ HTTP/1.1
Host: kierkegaard.informatik.uni-bremen.de
Date: Wed Jan 12 16:20:11 CET 2005
```

- Abstrakte Repräsentation von Requests:

```
data Request = Request {
   reqCmd      :: RequestCmd,
   reqURI      :: ReqURI,
   reqHTTPVer :: HTTPVersion,
   reqHeaders :: [RequestHeader],
   reqFilename:: FilePath }
data RequestCmd  = OptionsReq | GetReq | HeadReq ...
```

Repräsentation der Antworten

```
data ResponseBody
  = NoBody | FileBody Integer{-size-} FilePath | HereItIs String

data Response = Response {
  respCode        :: ResponseCode,
  respHeaders     :: [String],
  respCoding      :: [TransferCoding],
  respBody        :: ResponseBody,
  respSendBody    :: Bool,
  respContentType :: Maybe String  }
```

Antworten generieren

- Der Normallfall: eine Datei senden ("servieren")
  - Neuere Version hat plug-in Architektur $\Longrightarrow$ "Servlets"
- Achtung: Datei nicht erst in `String` konvertieren.

- Datei senden:

```
bufsize = 4 * 1024 :: Int
squirt :: Fd-> Fd-> IO ()
squirt rd wr = do
  arr <- newArray (0, bufsize-1)
  let loop = do r <- hGetArray rd arr bufsize
                if (r == 0) then return ()
                   else if (r < bufsize)
                           then hPutArray wr arr r
                           else hPutArray wr arr bufsize >> loop
  loop
```

Timeouts

- Wichtig: wenn keine Antwort kommt, Verbindung abbrechen.
- Generischer Timeout-Kombinator:

```
timeout :: Int       -- timeout in Sekunden
        -> IO a      -- normale Aktion
        -> IO a      -- timeout-Aktion
        -> IO a
```

Rekonfigurierbarkeit

- Wichtig: zur Laufzeit rekonfigurierbar
- Dafür: asynchrone Ausnahmen

```
topServer conf ps = do
    catchError (server conf ps)
        (\e -> case e of
                   ErrorCall "**restart**" -> do readConfig
server = ...
```

- ```
  readConfig = do
      blockSignals sigsToBlock   --- Signale blockieren!
      r <- parseConfig configPath
      case r of
        Right b  -> do ...
          topServer conf' plugins
  ```
  ```

Server `main`

```
main :: IO ()
main = do
  main_thread <- myThreadId
  installHandler sigPIPE Ignore Nothing
  installHandler sigHUP (Catch (hupHandler main_thread)) Nothing
  block $ readConfig

hupHandler :: ThreadId -> IO ()
hupHandler main_thread
  = throwTo main_thread (ErrorCall "**restart**")
```

Performance

- Kleinere Verbesserungen an `ghc` RTS und Büchereien
  - Binär-IO
  - Context-Switches
  - Verbesserung an `hGetLine`
- Latenz: linear ca. 710 Req/s, danach Einbruch.
- Zusammenfassend:
  - Zufriedenstellende Performance
  - . . . in nur ca. 3000 Zeilen Haskell (ursprüngl. 1500)
  - Plugin-Architektur — zur Laufzeit neue Handler hinzufügen.

Mehr Haskell im Web

- `HaskellCGI` — CGI-Skripte in Haskell
  - Direkte Repräsentation von `HTML` — bleibt im WebServer undurchsichtig (bis auf Fehlermeldungen)
  - Nur sehr einfaches Request — Response
- `WASH` — Server-Side Scripting in Haskell
  - Setzt auf `CGI` auf
  - Session-Konzept

Zusammenfassung

- Implementierung eines standardkonformen, stabilen HTTP-Servers in Haskell in ca. 1500 loc möglich;
- Wichtig:
  - Nebenläufigkeit,
  - Stabilität,
  - Verfügbarkeit.
- Darauf aufsetzend Plug-In Rahmenwerk (Servlets)
- Problem: Dokumentation, mangelnde Wartung
- . . . und nächste Woche ?