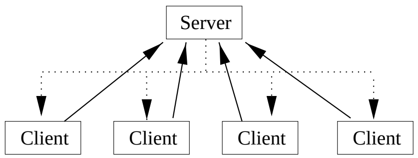
Längeres Beispiel: It's good to talk

- Ziel: ein Programm, um sich zu mehreren über das Internetz zu unterhalten (`talk`, `IRC`, etc.)
- Dazu: ein Server mit netzweit bekannter Adresse, zu dem Verbindung aufgenommen wird, und der Nachrichten an alle angeschlossenen Clients weiterverteilt.



Socketprogrammierung

- Serverseite:
  - Socket erzeugen, mit `bind` an Namen binden, mit `listen` Verbindungsbereitschaft anzeigen
  - Mit `accept` auf eingehende Verbindungen warten und annehmen.
  - Jede Verbindung erzeugt neuen Filedescriptor
    $\Rightarrow$ inhärent nebenläufiges Problem!
- Clientseite: Socket erzeugen, mit `connect` Verbindung aufnehmen
- GHC-Modul `Socket`

`Socket.hs`

```
Module Socket

data PortID = Service String -- z.B. "ftp"
            | PortNumber PortNumber
            | UnixSocket String -- socket in file system

type Hostname = String
mkPortNumber :: Int-> PortNumber

listenOn   :: PortID-> IO Socket                 -- serverside
accept     :: Socket-> IO (Handle, Hostname)

connectTo  :: Hostname -> PortID -> IO Handle    -- clientside

socketPort :: Socket-> IO PortID
```


Serverarchitektur

- Ein Kanal zur Nachrichtenverbreitung:
  eine Nachricht, viele Empfänger (*broadcast*)
- Realisierung des broadcast mittels `dupChan`
- Zentraler Scheduler kreiiert socket und Kanal
- Für jede ankommende Verbindung wird Prozess gestartet, der Nachrichten vom Socket auf den Kanal schreibt, und Nachrichten aus dem Kanal liest, und in den Socket schreibt
- Problem: Wie aus Socket *oder* Kanal lesen wenn beide blockieren?
- Lösung 1: `select` (BSD) oder `poll` (SysV)
  nicht in Haskell Standard-Bücherei — daher blockierend
- Lösung 2: Zwei Prozesse

Talk V0.1: simple-talk.hs

```
main :: IO ()
main =
  do port_num <- getArgs >>= return . mkPortNumber . read . head
     s <- listenOn (PortNumber port_num)
     ch<- newChan
     loop s ch
loop :: Socket -> Chan String -> IO ()
loop s ch =
  do (handle, wh, p) <- accept s
     putStrLn ("New connection from "++ wh ++ " on " ++ show p)
     ch2    <- dupChan ch
     forkIO (newUser handle ch)
     loop s ch2
```


Talk V0.1: simple-talk.hs

```
newUser :: Handle-> Chan String -> IO ()
newUser socket msgch =
  forkIO read >> write
    where read :: IO ()
          read  = hGetLine socket >>= writeChan msgch >> read
          write :: IO ()
          write = readChan msgch >>= hPutStrLn socket >> write
```


Talk V0.1: simple-talk.hs

Nachteile:

- Nachrichten stauen sich im Kanal
- Serverprozess fällt um, wenn Benutzer Verbindung beendet
- Keine Fehlerbehandlung
- Benutzer anonym

Talk V0.2: talk.hs

```
loop :: Socket -> Maybe (Chan String) -> IO ()
loop s ch =
  do (handle, wh, p) <- accept s
     putStrLn ("New connection from "++ wh ++ " on port " ++ show p)
     ch2    <- case ch of Just c  -> dupChan c
                          Nothing -> newChan
     forkIO (catch (newUser handle wh ch2) (\_ -> hClose handle))
     loop s (Just ch2)
main :: IO ()
main =
  do port_num <- getArgs >>= return . mkPortNumber . read . head
     s <- listenOn (PortNumber port_num)
     loop s Nothing
```

Talk V0.2: `talk.hs`

```
newUser :: Handle-> String-> Chan String -> IO ()
newUser socket wh msgch =
  do hPutStrLn socket "Hello there. Please send your nickname."
     nick <- hGetLine socket >>= return . filter (not . isControl)
     hPutStrLn socket ("Nice to meet you!")
     writeChan msgch (nick ++ "@" ++ wh ++ " has joined.")
     wp <- forkIO write
     catch (read ((++) (nick ++ ": ")))
         (\e-> do killThread wp
                  if isEOFError e then
                     writeChan msgch (nick ++ "@" ++ wh ++ " has left.")
                     else writeChan msgch (nick ++ "@" ++ wh ++
                           " left hastily ("++ ioeGetErrorString e++ ")")
                  hClose socket)
    where read :: (String-> String)-> IO ()
          read f = hGetLine socket >>= writeChan msgch . f >> read f
```


Sockets: the inside Story

Sockets werden erzeugt mit:

```
socket          :: Family -> SocketType -> ProtocolNumber -> IO Socket

data Family = instance of: Eq, Ord, Ix, Show
     AF_802 | AF_APPLETALK | AF_ARP  | AF_AX25 | AF_CCITT | AF_DECnet |
     ... | AF_INET | AF_INET6 | ...
data SocketType =  Stream | Datagram |  Raw | RDM | SeqPacket
type ProtocolNumber = Int -- "normally only a signle protocol exists"
```

Normalerweise: `socket AF_INET Stream 6`

Sockets: the inside Story

Verschiedene *Adressen* — auch Unix-Filenamen:

```
data SockAddr = SockAddrUnix String
              | SockAddrInet PortNumber HostAddress
```

Auf Serverseite mit `bindSocket` an Addresse binden, `listen`, `accept` für jede neue Verbindung; auf Client-Seite: mit `connect` verbinden:

```
bindSocket      :: Socket -> SockAddr -> IO ()
listen          :: Socket -> Int -> IO ()
accept          :: Socket -> IO (Socket, SockAddr)

connect         :: Socket -> SockAddr -> IO ()
```

Lesen/Schreiben mit `writeSocket`, `readSocket` oder `socketToHandle`.

RPC und RMI: Entfernte Methoden

- Erst lokal, dann auf anderen Rechnern
- Entwurfsziele:
  - Ein Request soll *asynchron* auswertet werden
  - Fehlerbehandlung
  - Typisierung
- Kanal zum Verschicken der Eingabedaten
- `MVar` zur Synchronisation des Rückgabewertes

Wesentliche Datenstrukturen: Eingabetyp `a`, Rückgabetyp `b`

```
data Service a b = Service (Chan (Req a b))
data Req a b     = Req a (MVar (Answer b))
```

Hilfsdatenstrukturen und Funktionen:

```
type Answer b = Either IOError b

propagate :: Answer b-> IO b
propagate (Left e)  = ioError e
propagate (Right v) = return v

try :: (a-> IO b)-> a-> IO (Answer b)
try f a = catch (f a >>= return . Right) (return . Left)
```


Neuen Server für Aktion f erzeugen:

```
newServer :: (a-> IO b)-> IO (Service a b)
newServer f =
  do c<- newChan
     forkIO (serve c)
     return (Service c)
  where serve c = do (Req val s) <- readChan c
                     res <- try f val
                     catch (putMVar s res) (\_ -> return ())
                     serve c
```

– neuen Thread erzeugen

– `MVar` in Eingabe sollte leer sein, aber trotzdem Fehler behandeln

Service vom Typ `Service a b` benutzen:
ergibt für Eingabewert eine Aktion, die mit der Rückgabe synchronisiert

```
call :: Service a b-> a -> IO (IO b)
call (Service ch) val =
  do v<- newEmptyMVar
     writeChan ch (Req val v)
     return (takeMVar v >>= propagate)
```

Benutzung (beispielhaft):

```
do s<- newServer (\x-> return (x*(x::Int)))
   c<- call s 10
   putStrLn "called, waiting for result..."
   r<- c -- hier wird erst auf Rueckgabe gewartet
   putStrLn ("Result: "++ show r)
```


RPCs auf entfernten Rechnern:

- Typ muß festgelegt sein
- Daten müssen übertragbar sein:
  Konversion in Binärdaten (effizient aber aufwendig),
  oder Typklassen `Read` und `Show` (ineffizient, aber einfach)

Modulstruktur:

`Client` – Benutzendes, aufrufendes Modul

`Server` – Server, exportiert Datenstrukturen und Verbindungsparameter

`Main` – Sowohl für den Client als auch für den Server, da zwei Executables

RPC: Modul Server

```
module Server(Answer, propagate, try, rpcPort, main) where
-- Answer, propagate, try (fast!) wie vorher:
type Answer b = Either String b

-- Diese Funktion ist der Service:
service :: Integer-> IO Integer
service n = return . fac where fac n = if n <= 0 then 1 else n*(fac (n-1))

-- der Port, unter dem wir addressierbar sind
rpcPort :: PortID
rpcPort = PortNumber (mkPortNumber 2405)
```


RPC: Modul Server

```
rpc :: (Read a, Show b)=> (a-> IO b)-> Handle-> IO ()
rpc f fd = do r<- hGetContents fd
              v<- catch (readIO r >>= try f)
                        (\_ -> return (Left "Cannot read request"))
              hPutStr fd (show v)
              rpc f fd

main :: IO ()
main =
  do s <- listenOn rpcPort
     loop s where loop s = do (fd, wh, _)<- accept s
                              putStrLn ("Rec’d request from "++ wh)
                              forkIO (rpc service fd)
                              loop s
```


RPC: Modul Client

`call` behält das gleiche Interface, nur `newServer` erhält andere Funktionalität:

```
import Server hiding (main)

data (Read a, Show a, Read b, Show b)=> Service a b = Service (Chan (Req a b
data (Read a, Show a, Read b, Show b)=> Req a b     = Req a (MVar (Answer b)

call :: (Read a, Show a, Read b, Show b)=> Service a b-> a -> IO (IO b)
call (Service ch) val =
  do v<- newEmptyMVar
     writeChan ch (Req val v)
     return (takeMVar v >>= propagate)

newRemoteServer :: String-> IO (Service Integer Integer)
```


RPC: Modul Client

```
newRemoteServer host =
  do c<- newChan
     h<- connectTo host rpcPort
     forkIO (serve h c)
     return (Service c)
  where serve :: Handle-> Chan (Req Integer Integer)-> IO ()
        serve h c = do (Req val s) <- readChan c
                       hPutStrLn h (show val)
                       res <- catch (hGetContents h >>= readIO)
                                    (return . Left . ioeGetErrorString)
                       catch (putMVar s res) (\_ -> return ())
                       serve h c
```

RPC: Benutzung

- Auf dem Serverrechner (z.B. `gemini`):

```
module Main where
import qualified Server(main)
main = Server.main
```

  Kompilieren (z.B. `server-main`), starten.

- Auf dem Clientrechner:

```
do s<- newRemoteServer "gemini"
    c<- call s 10
    putStrLn "called, waiting for result..."
    r<- c
    putStrLn ("Result: "++ show r)
```


Zusammenfassung

- Socketprogrammierung in Haskell schmerzfrei und flexibel
- . . . trotzdem Zugriff auf untere Schichten.
- RPCs können lokal flexibel nachgebildet werden, entfernte (auf anderen Rechnern) Prozeduren auch möglich