Client / Server Applications with Fudgets

Magnus Carlsson & Thomas Hallgren

{magnus,hallgren}@cs.chalmers.se

1 Abstract

The Fudget library, which is based on parallel stream processors, has been used to write Haskell [6] and LML programs with graphical user interfaces [1][2]. It has now been enhanced with capabilities for Internet socket communication, making it possible to write client / server applications with Fudgets. The sockets are introduced in a way that makes it possible for the compiler to catch errors that would occur if the client and server disagreed on the type of the messages that they exchange.

2 Organisation

Organisation of the paper: Internet sockets are presented in section 3. The Haskell interface for sockets is described in section 4. A fudget supporting the development of servers is given in section 5, and finally, we present the Calendar example in section 6.

3 Internet stream sockets

The type of sockets that we consider here are Internet stream sockets. They provide a reliable, two-way connection, similar to pipes, between any two computers on Internet. They are used in Unix tools like telnet, ftp, finger, mail, usenet and World Wide Web.

We will use them for two purposes: firstly, the server creates a *listener socket*. This socket is not used for communication, its purpose is to accept connections from clients. Secondly, when a connection is accepted, *communication sockets* are created in both the client and the server, and these are used for the actual communication.

Since there can be many listener sockets on a computer, they are distinguished by *port numbers*. When a server creates a listener socket, it must provide a port number. When a client wants to connect to a server, it must specify on what computer the server is running, and the port number.

4 Sockets in Haskell

In Haskell, we introduce two abstract types for listener sockets and communication sockets:

```
data LSocket -- Listener sockets
data Socket -- Communication sockets
```

The port number is just a type synonym:

```
type Port = Int
```

Of course, it is important that the client and server agree on what port number to use. Moreover, if we want to develop both the client and the server in Haskell, we can use the type system to ensure that both programs agree on the type of the messages that they exchange. This is done by associating ports with the types of the messages transmitted and received:

```
data TPort t r = TPort Port
```

t and r are the message types that the client transmit and receive, respectively.

We need sockets that carry this type information too:

```
data TLSocket t r = TLSocket LSocket
data TSocket t r = TSocket Socket
```

When a client wants to open a connection to the server, it uses openSocketF:

```
openSocketF :: HostName -> TPort t r ->(TSocket t r -> F c d) -> F c d
```

The first argument is the name of the computer where the server is running. openSocketF will return a typed communication socket to the continuing fudget. Note that the socket will get the same type as the port.

The communication is performed by the fudget transceiverF:

```
transceiverF :: (Text t, Text r) => TSocket t r -> F t (SocketMsg r)

data SocketMsg a = SocketMsg a | SocketEOS
```

The output from transceiverF is either a message from the socket, or SocketEOS, if the connection was closed by the server. Currently, transceiverF converts messages to and from strings by using the functions read and show, that are defined for types in the class Text. A more efficient version would use a binary protocol. Note that the class Binary is not good enough, since it does not take into account e.g. byte order.

5 Servers in Haskell

To write servers, we use socketServerF:

```
socketServerF :: TPort t r -> (TSocket r t -> F a (SocketMsg b)) -> F (Int,a) (Int,ClientMsg b)

data ClientMsg a = ClientMsg (SocketMsg a) | ClientNew
```

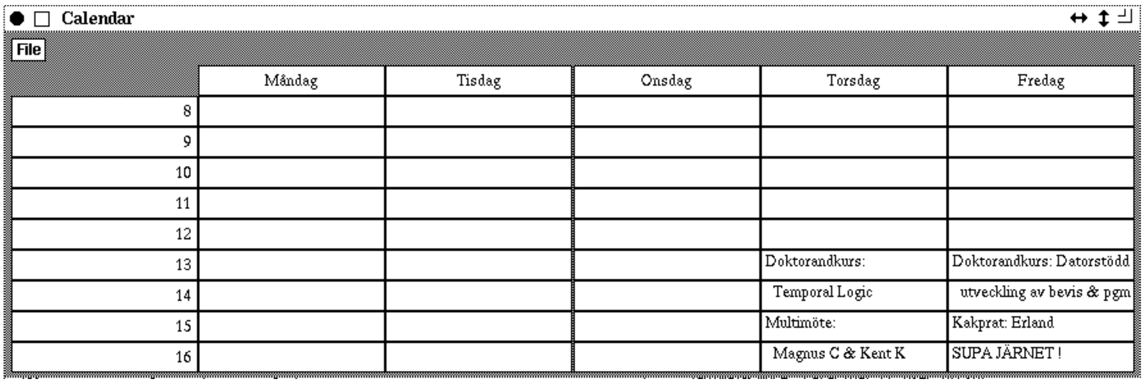
socketServerF will create a listener socket with the specified port number, and wait for connections. When a client is accepted a client handler fudget is spawned to serve the new connection (the second argument specifies the client handlers). In addition, the message ClientNew is emitted. The handler is expected to emit messages of type SocketMsg. If a handler emits SocketEOS, it will be killed, and socketServerF outputs ClientMsg SocketEOS. Since many handlers can be active, they are associated with unique integer tags.

The client handler is given a communication socket for the accepted client. The message types of the socket are flipped compared to the port message types. This is because the messages that the client transmits are the messages that the server receives, and vice versa.

A simple client handler could be transceiverF. In the next section, we will see a server that uses transceiverF as the client handler.

6 Example: Calendar

Outside the lunch room in our department, there is a whiteboard where the week’s activities are registered. We will look at an electronic version of this calendar, where people can get a view like this on her workstation:



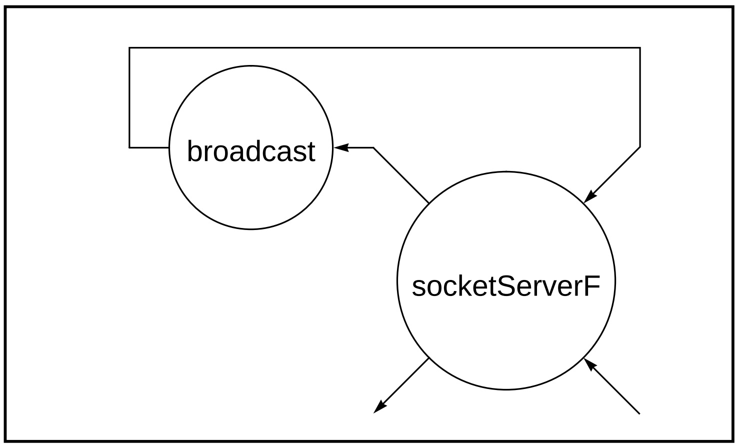
	Måndag	Tisdag	Onsdag	Torsdag	Fredag
8					
9					
10					
11					
12					
13				Doktorandkurs:	Doktorandkurs: Datorstödd
14				Temporal Logic	utveckling av bevis & pgm
15				Multimöte:	Kakprat: Erland
16				Magnus C & Kent K	SUPA JÄRNET !

The entries in the calendar can be edited by everyone. When that happens, all calendar clients should be updated immediately.

The calendar consists of a server maintaing a database, and the clients, running on the workstations. The server program is as follows:

```
module Main where -- Server

import Fudgets
import Socket
import Port

main = fudlogue (server port)

server port =
  let broadcast cl db =
      getSP $ \(i,e) ->
      let clbuti = filter (/= i) cl
      in case e of
            ClientNew ->
                     putSP [(i,d) | d <- db] $        -- Send the database to the new client
                     broadcast (i:cl) db
            ClientMsg (SocketMsg s)  ->
                     let db’ = replace s db in        -- Update database
                     putSP [(i,s) | i <- clbuti] $    -- Inform all the other clients of the new entry
                     broadcast cl db'
            ClientMsg SocketEOS  ->
                     broadcast clbuti db              -- Remove the client from the client list

  in loopF (broadcast [] [] >^^=< socketServerF port transceiverF)
```

The server consists of the stream processor broadcast, and a socketServerF, where the output from the stream processor goes to socketServerF, and vice versa:



broadcast maintains to values: cl, which is a list of the tags of the connected clients, and db, the database, organised as a list of (key,value) pairs. This database is sent to newly connected clients. When a user changes an entry in her client, it will send that entry to the server, which will update the database[1] and use the client list to broadcast the new entry to all the other connect clients. Finally, when a client disconnects, it is removed from the client list.

Now, what are the keys and the values in the database? Their types are determined by the port type, which is imported from Port:

```
module Port where

import Fudgets
import Socket

type SymTPort a = TPort a a

port :: SymTPort ((String,Int),String)
--        e.g. (("Torsdag",13),"Doktorandkurs:")
port = tPort 8888
```

Here, the messages are declared to be pairs of (string,int) pairs and strings. The idea is that the client program also should import this definition of the port, to ensure type correctness. The data types TPort and TSocket are abstract, and therefore it is impossible for the programmer to change the type of an already typed port or socket. The things to remember are: to use the same port declaration in the client and the server, and to give the port a monomorphic type signature.[2]

References

[1] M. Carlsson & T. Hallgren, Fudgets - A Graphical User Interface in a Lazy Functional Language, in *FPCA 93' - Conference on Functional Programming Languages and Computer Architecture*, pages 321--330, June 1993.

[2] M. Carlsson & T. Hallgren, The Fudget distribution, Available by anonymous ftp from ftp.chalmers.se:/pub/haskell/chalmers/lml-0.999.?.lmlx.tar.Z.

1. Unfortunately, the update will not take place until a new client connects, resulting in a space leak. It can be eliminated by inserting "seq (force db') $" after "let db' = replace s db in".

2. In this example, the server cannot be compiled if the type signature is omitted, since there will be an ambiguous overloading in transceiverF.

[3] Paul Hudak et al., *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.