Casty: a streaming media network

Johan Montelius

October 2, 2016

Introduction

In this assignment you will build a streaming media network. We will play around with shoutcast streams and build proxies, distributors and peer-to-peer clients. You will use the Erlang bit-syntax to implement a communication protocol over HTTP. The parser will be implemented using higher order functions to hide the socket interface. You will learn how to decode a mp3 audio stream and make it available for connecting media players. Sounds fun? - Let's go!

1 Shoutcast - ICY

You first need to understand how shoutcast works. It's very simple protocol for streaming audio content built using HTTP. It was at first called "I Can Yell" and you therefore see a lot of `icy` tags in the HTTP header; we will also refer to it as the ICY protocol.

1.1 request - reply

A media client such as Amarok (or VLC) connects to a server by sending a HTTP GET request. In this request the client asks for a specific feed in the same way as it would ask for a web page. In the request header it also announce if it can handle meta-data, the name of the player etc. A request could look like this:

```
GET / HTTP/1.0<cr><lf>
Host: mp3-vr-128.smgradio.com<cr><lf>
User-Agent: Casty<cr><lf>
Icy-MetaData: 1<cr><lf>
<cr><lf>
```

The `Icy-Metadata` header is important since it signals that our client is capable of receiving meta data in an audio stream. To see that this actually works you could use `wget` and look at what for example Virgin Radio returns when you try to connect. Try the following command in a shell but terminate it with ctrl-c or it will keep reading the audio stream.

```
wget --header="Icy-MetaData:1" -S -O reply.txt http://mp3-vr-128.smgradio.com
```

Now look at the file (you did terminate the loading right) reply.txt. Decode the reply and try to figure out what the different header tag means. One reply header that is very important for us is the `icy-metaint` header. It typically has the value 8192 which is the number of bytes in each mp3-block that is sent. Since the bit rate for this audio stream is 128Kb/s a 8192 byte block will contain half a second of music. Look at position 8192 in the body, do you find the meta data?

1.2 meta-data

The meta data comes as a sequence of characters after each audio block. The length of the sequence is coded in one integer k in the first byte of the meta data block. The length of the sequence is $16k$ bytes (not including the k-byte); the smallest meta data block is thus simply one k-byte of zero. If the text message is not an even multiple of 16 it is padded with trailing zeros. In Erlang bit-syntax a meta data segment could be written as follows:

```
<<1,104,
  101,108,
  108,111,
  0,0,0,0,
  0,0,0,0,
  0,0,0>>
```

The first byte is a the length of the following padded string. The next five bytes spell out "hello" and the padding consist of 11 zeros. This could also be written:

```
<<1,"hello", 0:(11*8)>>
```

When you attach to a shoutcast radio station you will see that most meta data blocks are empty. When a new song i played they use the meta data to send the name of the artist, title etc.

1.3 encoding of messages

One module, `icy`, will implement all details of how the ICY protocol is encoded and decoded. To make it more interesting we will use higher order functions and make the module both able to handle incomplete sequences and being unaware of how byte sequences are actually read or written.

We will of course communicate using sockets but why build this into the icy module; we pass a function along as an argument that the icy procedures can use to send a binary encoded segment when it is ready.

1.3.1 a request

Sending a request is simple an we will do with this almost hard coded version. The host is the name of the server we're contacting and the feed is the resource, typically "/".

```
send_request(Host, Feed, Sender) ->
    Request = "GET " ++ Feed ++" HTTP/1.0\r\n" ++
    "Host: " ++ Host ++ "\r\n" ++
    "User-Agent: Ecast\r\n" ++
    "Icy-MetaData: 1\r\n" ++ "\r\n",
    Sender(list_to_binary(Request)).
```

The third argument is the function that we apply to the final binary. It is up to the caller of `send_request/3` to provide a function that does something useful with the binary. When using socket communication we will call this procedure as follows:

```
Sender = fun(Bin) -> gen_tcp:send(Socket, Bin) end,
icy:send_request("mp3-vr-128.smgradio.com", "/", Sender)
```

We could however use it like this if we're debugging our code:

```
Sender = fun(Bin) -> Bin end,
icy:send_request("mp3-vr-128.smgradio.com", "/", Sender)
```

Or why not like this:

```
Sender = fun(Bin) -> io:format("Request:~n~s~n", [Bin]) end,
icy:send_request("mp3-vr-128.smgradio.com", "/", Sender)
```

1.3.2 a reply

A complete ICY reply consist of a status line, a sequence of headers and, a body. We divide the encoding into one procedure that encodes the status line and headers, and one procedure that encodes a segment of the body. Encoding the status line and headers is quite simple.

```
send_reply(Header, Sender) ->
    Status = "ICY 200 OK\r\n",
    Reply = Status ++ header_to_list(Header),
    Sender(list_to_binary(Reply)).
```

We will represent headers by a list of tuples containing a header (an atom) and a value (a string). A ICY header could look as follows:

```
[{'icy-notice', "This stream requires Winamp .."},
 {'icy-name', "Virgin Radio ..."},
 {'icy-genre', "Adult Pop Rock"},
 {'icy-url', "http://www.virginradio.co.uk/"},
 {'content-type', "audio/mpeg"},
 {'icy-pub', "1"},
 {'icy-metaint' "8192"},
 {'icy-br', "128"}]
```

The encoding of the headers, implemented in `header_to_list/1`, is done simply by turning the list of tuples into a byte sequence with the name and value of each header separated by a colon and terminated by a <cr><lf> (in Erlang written as "\r\n" . The whole header sequence is terminated by an additional <cr><lf>. We can assume that the headers are valid headers so we don't have to check every header. The implementation is left as an exercise. The built-in function `atom_to_list/1` will come at handy.

```
header_to_list([]) ->
     :
header_to_list([{Name, Arg}|Rest]) ->
      :
   ... ++ header_to_list(Rest).
```

1.3.3 the data segments

A data segment will be represented by a tuple with audio data and a (possibly empty) string that should be our meta data. The audio data is, as will soon be clear, coded as a list of binaries. The total length of all binaries must be equal to the `metaint` header information. We assume that this is the case and do not check this for every segment.

```
send_data({Audio, Meta}, Sender) ->
    send_audio(Audio, Sender),
    send_meta(Meta, Sender).
```

The implementation of `send_audio/2` is left as an exercise. Sending the meta data is slightly more complicated. We need to add the padding to the text string and calculate the k-value.

```
send_meta(Meta, Sender) ->
    {K, Padded} = padding(Meta),
    Sender(<<K/integer, Padded/binary>>).
```

Implementing the `padding/1` function is left as an exercise. You will need the arithmetic constructs `N rem 16` which will give you the reminder

when dividing with 16, and `N div 16` that is the integer division with 16. Calculate how large padding is needed, and that number of zeros to the end of the sequence and turn it into a binary.

```
padding(Meta) ->
    N = length(Meta),
     :
     :
    end.
```

This completes the encoding of messages, now for the slightly more complex task of decoding.

1.4 sockets and continuations

To understand the structure of our parser one must understand the problem with reading from a stream socket. Reading from a socket is very different from reading from a file. When reading from a socket we could be stuck halfway through a structure since the rest of the message has not yet arrived. In a concurrent system we do not want to block a process suspending on a socket.

In Erlang there is a solution to this problem. Instead of "reading" from the socket we have the socket send us segments as they arrive. We can thus go into a receive statement and wait for either more segments or any other message. A process that is receiving segments from socket could then be programmed as follows:

```
reader(Socket, Sofar) ->
   receive
       {tcp, Socket, Next} ->
           reader(Socket, [Next|Sofar]);
       {tcp_closed, Socket} ->
           {done, lists:reverse(Sofar)};
       stop ->
           {aborted, Sofar}
   end.
```

The process that calls `reader/2` can now be aborted by sending a stop message. A flexible solution, but do we have to build this into the parser? Can we not simply read everything there is to read from the socket and then pass everything to the parser? The problem is that we do not always know (especially true for HTTP) how long the message is unless we start to parse it; only by parsing can we determine if the message is complete.

If we try to build this reading strategy into the parse the parser would have to be aware of that it is reading from a socket and that it must be open

to receive other messages and not just the TCP related messages. This would of course make the parser complicated and less flexible. An alternative is to use "continuations"; take a look at this.

The parser is given a, possibly not complete, segment to parse and could return either:

- `{ok, Parsed, Rest}` : if the parsing was successful, Parsed is the result of the parser and Rest is what is left of the segment.
- `{more, Continuation}` : if more segments are needed, Continuation is a function that should be applied to the next segment.
- `{error, Error}` : if the parsing failed.

We then define a general purpose user of the parser that uses a zero argument function. The function is applied without arguments and could then result in a parsed result, a request for more or an error.

```
reader(Parser, Socket) ->
    case Parser() of
        {ok, Parsed, Rest} ->
            {ok, Parsed, Rest};
        {more, Cont} ->
            receive
                {tcp, Socket, More} ->
                    reader(fun() -> Cont(More) end,  Socket);
                {tcp_closed, Socket} ->
                    {error, "server closed connection"};
                stop ->
                    aborted
            end;
        {error, Error} ->
            {error, Error}
    end.
```

If more information is needed we go into the receive statement and wait for the next segment. If we receive a new TCP block we construct a new function and apply `reader/2` recursively. To read and parse a message from a socket we could call the reader as follows:

```
reader(Socket) ->
    reader(fun() -> parser(<<>>) end, Socket).
```

We will not use the code above but this is the strategy that we will use when implementing our ICY parser. The user of the parser will be the client and proxy processes that we will define later.

1.5 the parser

So now let's look at the parser. We will need to parse two types of messages: a request and a reply. The request will be sent by a media client to one of our client proxies and the reply will be sent from the server to our server proxy. The reply, that consist of a status line, headers and a body will, in the same way as the the encoding procedures, be broken up into two parts. First we will parse the the status line and headers and then we will parse the body. The body will then be broken up into a sequence of data segments.

The parser will work on binaries; this will make it more efficient when handling the larger audio data.

1.5.1 a request

Parsing a request is quite simple, we read the first line (all lines are terminated by <cr><lf>) and check that it is equal to "GET / HTTP/1.0". Now this is of course a simplification; a request could of course include something more interesting than "/" but it will do for now.

The `line/2` function will look for the end-of-line characters but if these are not found it will return `more`. The request parser will then return a continuation in the form of a function that should be applied to the next segment. We could make this more complicated but why not simply return a function that appends the segment that we have to the next segment and tries to redo the parsing of the request.

```
request(Bin) ->
    case line(Bin) of
        {ok, "GET / HTTP/1.0", R1} ->
            case header(R1, []) of
                {ok, Header, R2} ->
                    {ok, Header, R2};
                more ->
                    {more, fun(More) -> request(<<Bin/binary, More/binary>>) end}
            end;
        {ok, Req, _} ->
            {error, "invalid request: " ++ Req};
        more ->
            {more, fun(More) -> request(<<Bin/binary, More/binary>>) end}
    end.
```

Note how the scoping rules work; Erlang uses lexical scoping and the variable `Bin` will have it's binding when the function `request/1` is called. The returned function could now be called in any environment and the `Bin` variable will maintain it's binding. This is called static or lexical scoping.

Some functional programming languages use dynamic scoping where the value of `Bin` would depend on the environment in which it is used.

Once we have seen the request line we continue to parse the headers. The parsing of the headers will either succeed or result in a request of more information. If we succeed we return the Headers and also what is left of the segment that we parsed. In practice the rest will be an empty segment and probably ignored by the caller but why not be polite and return what is left.

Note that we have made a simplification in the implementation of the parse. If the function `line/1` or `header/2` has stepped through a hundred characters only to find out that more characters are needed, they simply return the atom `more`. The next time they are called they will have to run through the same hundred characters again. It would of course be nice if we could save exactly the position that we're in and continue only with the new segment. This would however make the function `request/1` more complicated. We would have to keep track of if we should continue reading a line or a header. In practice, the original binary will always contain both the request line and all headers so the question becomes academic.

Reading a line is a simple pattern matching exercise that is left as an exercise. This is skeleton to work on: (<cr >could be written as `13` or `$\r` and <lf >as `10` or `$\n`)

```
line(Bin) ->
    line(Bin, ...).

line(..., _) ->
    ...;
line(<<..., ..., Rest/binary>>, Sofar) ->
    {ok, ..., ...};
line(<<..., Rest/binary>>, Sofar) ->
    line(..., ...).
```

1.5.2 a reply

Parsing a reply is very similar. We apply the same strategy as before, try to parse as much as possible and start from the beginning if you need more. The thing that is slightly different is when we have successfully parsed the headers. Now we return not only the headers but also a continuation that when applied will generate the first data segment. Since the decoding of the data segments needs information of the length of the audio part we first extract this information from the headers. The implementation of `metaint` is left as an exercise.

```
reply(Bin) ->
```

```
    case line(Bin) of
        {ok, "ICY 200 OK", R1} ->
            case header(R1, []) of
                {ok, Header, R2} ->
                    MetaInt = metaint(Header),
                    {ok, fun() -> data(R2, MetaInt) end, Header};
                 more ->
                    {more, fun(More) -> reply(<<Bin/binary, More/binary>>) end}
            end;
        {ok, Resp, _} ->
            {error, "invalid reply: " ++ Resp};
        more ->
            {more, fun(More) -> reply(<<Bin/binary, More/binary>>) end }
    end.
```

Parsing a data segment consists of two parts. First we read M number of bytes from the input stream and then we decode a meta data section. We will stick to our continuation strategy and return either:

- {`more, Continuation`}: where Continuation is a function that should be applied to the next segment if more segments are needed or
- {`ok, {Audio, Meta}, Continuation`}: once a complete data segment has been read. The continuation will give us the next data segment when applied without arguments.

It is not very likely that we will be able to read a whole audio segment in one go. An audio segment is typically 8192 bytes long and TCP packets have a maximum size of 1460 bytes over regular Ethernet. Each audio segment will typically consist of six chunks. Since we do not want to decode those there is no point in appending them into one binary. Instead we keep them in a list. Eventually it is up to the `send_audio/2` procedure to send these chunks one by one.

```
data(Bin, M) ->
    audio(Bin, [], M, M).

audio(Bin, Sofar, N, M) ->
    Size = size(Bin),
    if
        Size >= N ->
            {Chunk, Rest} = split_binary(Bin,N),
            meta(Rest, lists:reverse([Chunk|Sofar]), M);
        true ->
            {more, fun(More) -> audio(More, [Bin|Sofar], N-Size, M) end}
    end.
```

It's important to understand how we avoid parsing the same binaries every time we request more information. We return a function that should be applied to the next segment but we remember what we have seen so far and how much more we need to see. This is different from the implementation of `request/1` where we simply started from the beginning.

Parsing the meta data segment is slightly more tricky. We first have to read the k-byte so that we know the length of the following string. The string consist of a text padded with zeros to a multiple of 16. We read the k-byte (could be zero), the following string and remove the padding. We could have kept the padding since we will only have a problem when we should send the segment to a proper client but it could be nice to have a proper string as the meta data.

```
meta(<<>>, Audio, M) ->
    {more, fun(More) -> meta(More, Audio, M) end};
meta(Bin, Audio, M) ->
    <<K/integer, R0/binary>> = Bin,
    Size = size(R0),
    H = K*16,
    if
        Size >= H ->
            {Padded, R2} = split_binary(R0,H),
            Meta = [C || C <- binary_to_list(Padded), C > 0],
            {ok, {Audio, Meta},  fun() -> data(R2, M) end};
        true ->
            {more, fun(More) -> meta(<<Bin/binary, More/binary>>, Audio, M) end}
    end.
```

You might not have seen the construct used to remove the padding. It's called *list comprehension* and could be read - "give me the list of C's where C is taken from `binary_to_list(Padded)` and C is greater than 0".

1.6 does it work

Complete a module `icy` that exports the above described functions: `request/1`, `reply/1`, `send_request/1`, `send_reply/1` and `send_data/1`. You can then test your implementation with the following test examples:

```
icy:send_request("www.host.com", "/",
       fun(Bin) -> io:format("~s~n", [Bin]) end).

icy:send_reply([{key, "value"}],
       fun(Bin) -> io:format("~s~n", [Bin]) end).

icy:send_data({[<<"hello">>], "Message"},
       fun(Bin) -> io:format("~w~n", [Bin]) end).
```

Experimenting with the parser is equally simple. It's operating on binaries but the binary syntax makes it very easy to construct the segments that we need.

```
icy:request(<<"GET / HTTP/1.0\r\nkey:value\r\n\r\n">>).
```

Let's try parsing something incomplete.

```
{more, F} = icy:request(<<"GET / HTTP/1.0\r\nkey:value\r\n">>).
```

```
F(<<"\r\n">>).
```

A reply will always give us a function that should be applied without arguments to give us the data segments.

```
{ok, Data, H} = icy:reply(<<"ICY 200 OK\r\nicy-metaint: 5\r\n\r\n123">>).
```

Since the body did not contain 5 audio byte nor a meta data section we should get a request for more if we apply the continuation.

```
{more, More} = Data().
```

Now let's apply this continuation on the rest of the section. The "1" indicates a total of 16 bytes. The message "hello" is then padded with 11 bytes of zeros (an integer 0 encoded in 11*8 bits).

```
More(<<"45", 1, "hello", 0:(11*8)>>).
```

2 The architecture

Our goal is now to build a proxy (server proxy), that is connected to a shoutcast server, and a client (client proxy) that accepts connections from a media player. The client should know about the proxy and communicate with it using Erlang messages. We will use the following messages between the client and the proxy.

- {`request, Client`}: where the Client is the Erlang process that wants to connect.
- {`reply, N, Context`}: where Context is the header information received from the source. N is the number on the data segment that should arrive next.
- {`data, N, Data`}: where N is an integer to number all segments and Data is the audio and meta data that we have received.

The numbering of data segments is to keep track of which segments we need once we start to build more complicated distribution networks. In the beginning the FIFO-order of Erlang messaging will give us the messages in the right order.

2.1 the client

The client will listen on a TCP port and wait for incoming connection. Once a connection is accepted a request is read from the socket. The content of the request is not important for our needs, we will happily connect any media client to a predefined proxy.

```
init(Proxy, Port) ->
    {ok, Listen} = gen_tcp:listen(Port, ?Opt),
    {ok, Socket} = gen_tcp:accept(Listen),
    case read_request(Socket) of
        {ok, _, _} ->
            case connect(Proxy) of
                {ok, N, Context} ->
                    send_reply(Context, Socket),
                    {ok, Msg} = loop(N, Socket),
                    io:format("client: terminating ~s~n", [Msg]);
                {error, Error} ->
                    io:format("client: ~s~n", [Error])
            end;
        {error, Error} ->
            io:format("client: ~s~n", [Error])
    end.
```

When creating the listen socket we specify the properties in a options list. The options we will use are:

- `binary`: more efficient since there is no point in handling the mp3 audio as list of integers.
- {`packet, 0`}: framing messages with the length of the message is very useful but we're dealing with the ICY protocol and not our own.
- {`reuseaddr, true`}: allow us to reuse the port (and we will)
- {`active, true`}: the socket process will send us segments as they arrive, we do not have to suspend reading the socket.
- {`nodelay, true`}: send segments as soon as possible

Notice how we here choose to have a time-out when waiting for TCP messages. We could also have chosen to accept a `stop` message or similar; the client module is in control.

Once we have read a proper request (we don't really care what is requested) we try to connect to the proxy. The proxy will reply with a context

(the header information sent by the server) that we can send to the client and then go into a loop that continuously deliver data segments from the proxy to the media player.

```
connect(Proxy) ->
    Proxy ! {request, self()},
    receive
        {reply, N, Context} ->
            {ok, N, Context}
    after ?TimeOut ->
            {error, "time out"}
    end.
```

The loop is simple and will continue to deliver data segments as long as the TCP connection to the media player is not closed nor a time-out occurs. At this point we do not check that we actually receive all data segments but this could of course easily be done.

```
loop(_, Socket) ->
    receive
        {data, N, Data} ->
            send_data(Data, Socket),
            loop(N+1, Socket);
        {tcp_closed, Socket} ->
            {ok, "player closed connection"}
    after ?TimeOut ->
            {ok, "time out"}
    end.
```

This is the interface that we use to the icy module. Sending is quite simple, we use the exported procedures and supply a tcp send function that should be used to send the constructed binaries over the socket interface.

```
send_data(Data, Socket) ->
    icy:send_data(Data, fun(Bin)-> gen_tcp:send(Socket, Bin) end).

send_reply(Context, Socket) ->
    icy:send_reply(Context, fun(Bin)-> gen_tcp:send(Socket, Bin) end).
```

The interface to the parser uses a general purpose reader that will will return {`ok, Parsed, Rest`} or {`error, Error`}. It is given a zero argument function and a socket as input arguments. The function is applied and results either in a successful parsing, a request for more or an error message. The request for more input can now be handled outside of the parser module. The reader will go into a receive statement and wait for more TCP messages.

We have a time-out so that we do not get stuck waiting for segments that will never come.

```
reader(Cont, Socket) ->
    case Cont() of
        {ok, Parsed, Rest} ->
            {ok, Parsed, Rest};
        {more, Fun} ->
            receive
                {tcp, Socket, More} ->
                    reader(fun() -> Fun(More) end,  Socket);
                {tcp_closed, Socket} ->
                    {error, "server closed connection"}
            after ?TimeOut ->
                    {error, "time out"}
            end;
        {error, Error} ->
            {error, Error}
    end.
```

Reading a request can now be defines like this:

```
read_request(Socket) ->
    reader(fun()-> icy:request(<<>>) end, Socket).
```

Implement the module `client` and export the procedure `init/2`. That is the whole client process. Now let's turn to the proxy.

2.2 the proxy

The proxy is even simpler to implement. When we start the proxy we will give it a shoutcast stream to connect to. The stream is defined by a tuple:

- {`cast, Host, Port, Feed`}.

The source of Virgin Radio would be:

- {`cast, "mp3-vr-128.smgradio.com", 80, "/"`}.

We will first wait for a client to request a connection before attaching to the shoutcast server. Since we will not hear anything from the client we add a monitor. If the client terminates we will receive a message and can then decide what to do (die is one option). This is not strictly necessary but it will reduce the number of zombie proxies when experimenting.

```
init(Cast) ->
    receive
        {request, Client} ->
            io:format("proxy: received request ~w~n", [Client]),
            Ref = erlang:monitor(process, Client),
            case attach(Cast, Ref) of
                {ok, Stream, Cont, Context} ->
                    io:format("proxy: attached ~n", []),
                    Client ! {reply, 0, Context},
                    {ok, Msg} = loop(Cont, 0, Stream, Client, Ref),
                    io:format("proxy: terminating ~s~n", [Msg]);
                {error, Error} ->
                    io:format("proxy: error ~s~n", [Error])
            end
    end.
```

A connection to a server consist of: a stream in the form of a open socket, a continuation, from which we can receive data segments, and a context (the header information). Once attached we send a reply and start relaying data segments.

```
loop(Cont, N, Stream, Client, Ref) ->
    case reader(Cont, Stream, Ref) of
        {ok, Data, Rest} ->
            Client ! {data, N, Data},
            loop(Rest, N+1, Stream, Client, Ref);
        {error, Error} ->
            {ok, Error}
    end.
```

Attaching to a server requires some socket programming. We connect to the server and send an ICY request If the sending is successful we continue with reading the reply that either results in a valid connection or and error message.

```
attach({cast, Host, Port, Feed}, Ref) ->
    case gen_tcp:connect(Host, Port, [binary, {packet, 0}]) of
        {ok, Stream} ->
            case send_request(Host, Feed) of
                ok ->
                    case reply(Stream, Ref) of
                        {ok, Cont, Context} ->
                            {ok, Stream, Cont, Context};
                        {error, Error} ->
```

```
                            {error, Error}
                    end;
                _ ->
                    {error, "unable to send request"}
            end;
        _ ->
            {error, "unable to connect to server"}
    end.
```

When sending the request we supply the `gen_tcp:send/2` function to the `icy:send_request/3` function.

```
send_request(Host, Feed) ->
     icy:send_request(Host, Feed, fun(Bin) -> gen_tcp:send(Stream, Bin) end).
```

The `reader/3` function is slightly different from the one we used for the client. We now take advantage of the fact that we can act on other messages besides the once from the tcp-process. A `DOWN` message is sent if the monitored client process dies or becomes unavailable. We could of course suspend and wait for a new client connection but as you will we might as well die.

```
reader(Cont, Stream, Ref) ->
    case Cont() of
        {ok, Parsed, Rest} ->
            {ok, Parsed, Rest};
        {more, Fun} ->
            receive
                {tcp, Stream, More} ->
                    reader(fun() -> Fun(More) end,  Stream, Ref);
                {tcp_closed, Stream} ->
                    {error, "icy server closed connection"};
                {'DOWN', Ref, process, _, _}  ->
                    {error, "client died"}
            after ?TimeOut ->
                    {error, "time out"}
            end;
        {error, Error} ->
            {error, Error}
    end.
```

Reading a reply can now be coded like this.

```
reply(Stream, Ref) ->
    reader(fun()-> icy:reply(<<>>) end,  Stream, Ref).
```

Implement the `proxy` module and export the `init/1` function. If all is well we should now be able to connect a media player to a client, a client to a proxy and a proxy to a server.

2.3 streaming audio

We now have all the pieces of the puzzle to start streaming audio across our network. Create a `test` module and write some functions that we will use in our experiments. First we will create a proxy and client process in the same Erlang node and see that things work, then we will have the two processes running on two computers.

```
direct() ->
    Proxy = spawn(proxy, init, [?Cast]),
    spawn(client, init, [Proxy, ?Port]).
```

Start the processes and then direct you media client (Amarok or VLC should work) to the stream `http://localhost:8080/` (or whatever port you're using). If you have added some trace print statement you will see how the client accepts the request from the player, sends it to the proxy that connects to the read server. Every data segment will contain half a second of audio information. Note how the media player decodes the meta data and uses it to describe the channel.

We have two loop constructs one on the proxy side and one on the client side. The recursive call on the proxy is a call to `stream/5`, now if this this was a call to `proxy:stream/5` we would use the latest loaded version of the function in every recursion. Try this - change the loop to using `proxy:stream/5`, compile load and start playing, edit the source and include a printout every loop, compile and load the module while still playing. See how we can update our code while not loosing a single audio packet.

If everything works it's time to start a proxy on one computer and let the client run on another. You can even let the media player run on a third computer. When running several Erlang node make sure that you start them with a proper name and the same cookie.

```
erl -name 'proxy@130.237.215.255' -setcookie C00l3r
```

Check the processor load on the machines. We're handling a 128Kbps audio stream, does it show? What happens of we remove the cables, can we survive one or two seconds of network failures? Drop every tenth packet and see if things still work. Are processes terminated as expected when the media player stops playing? Do some experiments before going further.

3 A distribution server

So now we have solved the tricky parts of communicating with a media player and a Shoutcast server. Handling the messages in Erlang is a lot simpler, the only messages that we need to keep track of are:

- {`request, Client`}: a request from a client
- {`reply, N, Context`}: the reply to a request, expect N to be the next data packet
- {`data, N, Data`}: only have to look at N if we want to

Our current system is limited since we only allow one client to connect each proxy. We could extend the proxy to handle more clients but we could also implement this as a separate process. This is the skeleton code for a a module `dist` that will do just this.

```
init(Proxy) ->
    Proxy ! ...
    receive
        ...  ->
            :

    after ?TimeOut ->
            ok
    end.


loop(Clients, N, Context) ->
    receive
        {data, N, Data} ->
             :
            loop(Clients, N+1, Context);
        {request, From} ->
            Ref = erlang:monitor(process, From),
            From ! ... ,
            loop(... , N, Context);
        {'DOWN', Ref, process, Pid, _} ->
            loop(... , N, Context);
        stop ->
            {ok, "stoped"};
        stat ->
            io:format("dist: serving  clients~n", [length(Clients)]),
            loop(Clients, N, Context)
    end.
```

Complete the missing parts connect a distribution process to a proxy. Then start clients one by one and make them connect to the distributor. Since the distributor needs a media player before it connects it could be useful with a dummy client.

```
init(Proxy) ->
    Proxy ! {request, self()},
    receive
        {reply, N, _Context} ->
            io:format("dummy: connected~n", []),
            {ok, Msg} = loop(N),
            io:format("dummy: ~s~n", [Msg]),
    after 5000 ->
            io:format("dummy: time-out~n", [])
    end.

loop(N) ->
    receive
        {data, N, _} ->
            loop(N+1);
        {data, E, _} ->
            io:format("dummy: received ~w, expected ~w~n", [E,N]),
            loop(E+1);
        stop ->
            {ok, "stoped"}
    after ?TimeOut ->
            {ok, "time out"}
    end.
```

How many dummy client can you run before your machine chokes? If you're sitting on anything close to what I'm sitting at don't spend to much time starting dummy clients one by one. Write a function that starts n clients and then see how many you can start. Also try to run the distributor on one machine and the dummy clients on other machines - is it the processing power or the network that is the limiting factor?

4 Build a tree

Let's try to build a distribution tree dynamically as clients want to connect. We will use two new processes that are so similar that we'll implement them in the same module. On is a `root` process that will connect to a proxy and then wait for branches to connect. It will only allow two branches to connect and will redirect other branches to the two that it has connected.

The other process is a `branch` process. It will wait for a client to connect and then connect to a `root`. It must be prepared to be redirected to another branch who it should then try to connect to. Once connected it must also be open to serve two other branches and redirect others to these two branches.

We will now have six elements in our shoutcast architecture. A media player is connected to a client proxy. The client proxy thinks it is connecting to a proxy but it is actually connecting to a branch process. The branch process know that it is connecting to a root (or another branch) and the root is connected to a server proxy. The server proxy is as before connected to the real shoutcast server.

In a real implementation one would probably collapse the proxy and root process and the client and branch process but note how the separation of the processes makes each description more easy to follow. In a true concurrent language creating and running a separate process is as normal as handling complexity with procedures and libraries. In the same way, as we trade efficiency for clarity when hiding implementation details in a module, we use processes to make our system easier to implement.

4.1 the messages

In the final project the goal is to connect all computers in the class in a distribution tree. We will have one dedicated node that runs the proxy and root process. All other nodes will run a media player, a client process and a branch process. Since we're now implementing the tree module independently and a branch process on one node will not run the same code as a branch process on another node, it is important to specify the message interface.

- {`request, Pid`}: a request sent to a root or a branch process. The process (Pid) must be able to handle request messages once connected.
- {`reply, N, Context`}: a reply from a root or a branch process. The integer `N` is the number of the next data segment, `Context` should be handled by the `icy` module.
- {`redirect, Pid`}: this is the message given as a reply to a request message when the root or branch can not connect more branches. The `Pid` is a process identifier to another branch process that might have a available slot or that will redirect us again.
- {`data, N, Data`}: the n'th data segment, the data itself should be handled by the `icy` module.

If we all stick to these messages things might actually work on the first try.

4.2 the root

You need your own root to do some initial experiments. This is a skeleton that you can easily complete. The first procedure will connect to the proxy.

```
root(Proxy) ->
    Proxy ! ..... ,
    receive
        ..... ->
            loop(... , ... , ...., Context)
    after ?TimeOut ->
            ok
    end.
```

The loop procedure will accept data messages from the proxy or request messages from branches processes that try to connect. We will accept the two first branch processes but redirect all other.

```
loop(Clients, N,  Context) ->
    receive
        {data, N, Data} ->
              :
            loop(Clients, N+1, Context);

        {request, From} ->
            L = length(Clients),
            if
                L < 2 ->
                    From ! ..... ,
                    loop([From|Clients], N+1, Context);
                true ->
                      :
                    From ! {redirect, ....} ,
                    loop(...,  .... , ...)
            end

    end.
```

Note that our tree will look less like a tree if we always redirect requesting processes down the left branch. To keep the tree balanced we should redirect every second process to the left and every second to the right.

4.3 the branch

The branch process will look very similar to the root process. The difference is that the branch process should first wait for a client process to connect.

When it receives a request from a client it should try to connect to the known root of the tree. It could be redirected several times but it should finally be connected.

Once connected it should forward all data packets to its own client. It should also be open for requests from other branches. Similar to the root process it should accept the two first branches and redirect the rest.

Note - a branch process must separate the client process from connected branches. Al should receive copies of the data segments but we can not redirect a connecting branch to the client; a client is not prepared to handle request messages.

4.4 error handling

Can we make this structure more stable or self repairing. Can we detect that our up-stream source is not delivering as it should? If the root process dies then there is not much to do but if it was a peer branch process we could try a new attempt at connecting to the root. Do we have time to do this before any of the branches connected to us will have time to find out?

If we are directly connected to the root we should expect to have data segments delivered every 500 ms. Should we give the root some slack and set a time out after 600ms? What happens if all branches below us has the same time out?

Can we use Erlang monitors to detect that nodes are down or do we have to do our own failure detection?

Assume our up-stream source fails to deliver and we manage to reconnect. If the old up-stream source now resumes transmission and delivers the data segments we will have two processes delivering the same stream. How do we prevent this? Can we introduce a control message to stop a transmission?

5 A BitTorrent architecture

How hard would it be to implement distribution network using a bitTorrent protocol? What are the problems that we need to solve? Would it be better than our tree distribution network? Pros and cons?