Casty: una red de streaming de media*

Federico C. Repond
frepond@unq.edu.ar

Esteban Dimitroff Hódi
esteban.dimitroff@unq.edu.ar

16 de junio de 2016

1. Introducción

En este ejercicio vamos a construir una red de streaming de media. Vamos a jugar con streams de shoutcast y construir proxies, distributors y clientes peer-to-peer. Vamos a usar la sintaxis de bits de Erlang para implementar un protocolo de comunicación sobre HTTP. Vamos a implementar el parser usando funciones de alto orden para esconder la interface de sockets. Vamos a aprender como decodificar streams de audio mp3 y ponerlos disponibles para que se conecten reproductores de audio. ¿Suena divertido, no? ¡Empecemos!

2. ICY – Shoutcast

Lo primero que tenemos que hacer es entender como funciona shoutcast. Es un protocolo muy simple para hacer streaming de audio usando HTTP. Inicialmente se llamó "I Can Yell" y por eso vemos muchos tags `icy` en el header de HTTP; nos vamos a referir al mismo con protocolo ICY.

2.1. Request – Reply

Un cliente de media como Amarok o VLC se conecta a un servidor enviando un request HTTP GET. En este request el cliente pide por un feed específico de la misma forma que uno podría pedir una página web. En el header del request también va anunciar si puede manejar metadata, el nombre del reproductor, etc. Un request podría verse de la siguiente forma:

```
GET / HTTP/1.0<cr><lf>
Host: 192.99.62.212:9904<cr><lf>
User-Agent: Casty<cr><lf>
Icy-MetaData: 1<cr><lf>
<cr><lf>
```

*Adaptado al español del material original de Johan Montelius (`https://people.kth.se/~johanmon/dse.html`)

El header `Icy-Metadata` es importante ya que señala que nuestro cliente es capaz de recibir metadata en streams de audio. Para ver que esto efectivamente funciona podemos usar `wget` y mirar por ejemplo que devuelve cuando intentamos conectarnos a un servidor. Probar los siguiente en un shell pero terminarlo con `ctrl-c` o se va a mantener leyendo el stream de audio.

```
wget --header="Icy-MetaData:1" -S -O reply.txt\
                              http://192.99.62.212:9904
```

Mirar el contenido del file `reply.txt` (recordamos apretar ctrl-c, no!). Decodificar la respuesta y tratar de entender que significan los distintos tags. Uno de los headers de respuesta que es muy importante para nosotros es `icy-metaint`. Este header tiene el número de bytes en cada bloque de mp3 enviado, en nuestro caso 32768 bytes. Dado que este stream de audio tiene un bitrate de 128Kb/s un bloque de 32768 bytes va a contener 2 segundos de música.

2.2. Metadata

La metadata viene como una secuencia de caracteres después de cada bloque de audio. La cantidad de bloques de la secuencia es codificada en un entero `k` en el primer byte del bloque de metadata. La longitud de la secuencia es de múltiplos de bloques de 16 bytes (no incluyendo el byte `k`); siendo el bloque de metadata más chico simplemente el byte `k` en 0. Si el mensaje de texto no es múltiplo de 16 el contenido se completa (padding) con 0 al final. En sintaxis de bit de Erlang un segmento de metadata podría ser escrito de la siguiente forma:

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

El primer byte es la cantidad de bloques. Los siguientes 5 bytes representan el string `"hello"` y el padding consiste en 11 ceros. Esto puede ser escrito:

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

Cuando nos unimos a un estación de radio shoutcast vamos a ver que la mayoría de los bloques de metadata son vacíos. Cuando una nueva canción comienza se usa la metadata para enviar el nombre del artista, título, etc.

2.3. Codificación de Mensajes

Un módulo, `icy`, va a implementar detalles de como el protocolo ICY se codifica y decodifica. Para hacerlo más interesante vamos a usar funciones de alto orden y hacer que el módulo maneje secuencias incompletas y desconocer como las secuencias de bytes se leen o escriben.

Por supuesto vamos a usar sockets para comunicarnos, pero por qué vamos a meter esto en el módulo `icy`; vamos a pasar un función como un argumento a las funciones de `icy` que luego van a usar para enviar segmentos codificados cuando están listos.

2.3.1. Un Request

Enviar un request es simple y vamos a hacerlo mayormente hardcoded. El `Host` es el nombre del servidor el que estamos contactando y el `Feed` es el recurso, típicamente `"/"`.

```
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)).
```

El tercer argumento es la función que aplicamos al binario final. Es responsabilidad de quien llama a `send_request/3` proveer una función que haga algo útil con el binario. Cuando usamos sockets vamos a llamar la función de la siguiente forma.

```
Sender = fun(Bin) -> gen_tcp:send(Socket, Bin) end,
icy:send_request("192.99.62.212:9904", "/", Sender)
```

Podemos sin embargo usarlo para usarlo para hacer debugging de nuestro código:

```
Sender = fun(Bin) -> Bin end,
icy:send_request("192.99.62.212:9904", "/", Sender)
```

O por qué no algo como esto:

```
Sender = fun(Bin) -> io:format("Request:~n~s~n", [Bin]) end,
icy:send_request("192.99.62.212:9904", "/", Sender)
```

2.3.2. La Respuesta

Una respuesta completa ICY consiste de una línea de estado, una secuencia de headers, y un cuerpo. Dividimos la codificación en una función que codifica la línea de estado y headers, y otra función que codifica un segmento del cuerpo. La codificación de la línea de estado y los headers es bastante simple.

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

Vamos a representar los headers con una lista de tuplas conteniendo un header (un atom) y un valor (un string). Un header ICY puede verse de la siguiente forma:

```
[{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"}]
```

El encoding de headers, implementado en `header_to_list/1`, se hace simplemente transformando la lista de tuplas en una secuencia de bytes con el nombre y valor de cada header separados por `:` y terminado por un `<cr><lf>` (en Erlang `"\r \n "`). Podemos asumir que los headers son válidos así no tenemos que chequear cada header. La implementación se deja como ejercicio. La función built-in `atom_to_list/1` va a ser de utilidad.

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

2.3.3. Los Segmentos de Datos

Un segmento de datos va a ser representado como una tupla con la información de audio y un (posiblemente vacío) string que debería ser nuestra metadata. La información de audio es, pronto va a ser más claro, codificada como una lista de binarios. La longitud de todos los binarios deben ser igual a la información del header `metaint`. Asumimos que esto es así y no chequeamos esto en cada segmento.

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

La implementación de `send_audio/2` se deja como ejercicio. Enviar la metadata es un poco más complicado. Necesitamos agregar padding al string y calcular el valor de `k`.

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

La implementación de `padding/1` se deja como ejercicio. Vamos a necesitar las construcciones aritméticas `N rem 16` que nos va a dar el resto de dividir con 16 y `N div 16` que es la división entera por 16. Calcular cuan largo es el padding necesario, y agregar ese número de 0s al final de la secuencia y finalmente convertirla en binario.

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

Esto completa la codificación de mensajes, ahora vamos a empezar con la tarea de decodificar que es un poco más complicada.

2.4. Sockets y Continuaciones

Para entender la estructura de nuestro parser debemos entender el problema de leer de un socket stream. Leer de un socket es muy distinto a leer de un archivo. Cuando leemos de un socket podemos quedar a mita de camino de una estructura dado que el resto de los mensajes no han llegado aún. En un sistema concurrente no queremos bloquear los procesos suspendiendo en un socket.

En Erlang hay una solución a este problema. En lugar de "leer" de un socket hacemos que el socket vaya enviando los segmentos a medida que llegan. Podemos de esta forma or a una sentencia `receive` y esperar ya sea por más segmentos o cualquier otro mensaje. Un proceso que está recibiendo segmentos de un socket puede ser programado de la siguiente forma:

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

El proceso que llama a `reader/2` puede ser abortado enviando un mensaje `stop`. ¿Una solución flexible, pero tenemos que hacer esto en el parser? ¿No podemos simplemente leer todo lo que hay que leer del socket y entonces pasar todo al parser? El problema es que no siempre sabemos (especialmente con HTTP) cuan largo el mensaje es a menos que empecemos a parsear; solo parseando podemos determinar si el mensaje está completo.

Si tratamos de construir esta estrategia de lectura en el parser debemos ser consientes que estamos leyendo de un socket y que debe estar abierto para recibir otros mensajes y no solo mensajes TCP. Esto puede hacer el parser complicado

y menos flexible. Una alternativa es el uso de "continuaciones"; démosle una mirada.

El parser recibe un segmento, posiblemente incompleto, a parsear y puede retornar:

- `{ok, Parsed, Rest}`: si el parseo fue exitoso, `Parsed` es el resultado del parser y `Rest` es lo que queda por parsear del segmento.
- `{more, Continuation}`: si se necesitan más segmentos, `Continuation` es una función que debe ser aplicada al siguiente segmento.
- `{error, Error}`: si el parseo falló.

Entonces definimos un función de propósito general del parser que use una función con 0 argumentos. La función es aplicada sin argumentos y puede devolver el resultado del parseo, un pedido de más datos o un 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.
```

Si necesita más información entramos en una sentencia `recieve` y esperamos por el siguiente segmento. Si recibimos un nuevo bloque TCP construimos una nueva función y aplicamos `reader`/2 recursivamente. Para leer y parsear un mensaje de un socket podemos llamar a `reader`/2 de la siguiente forma:

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

No vamos a usar el código de arriba, pero esta va a ser la estrategia que vamos a usar para implementar el parser ICY. El usuario del parser va a ser el cliente y los procesos proxy que vamos a definir más adelante.

2.5. El Parser

Veamos el parser. Vamos a necesitar parsear 2 tipos de mensajes: un request (pedido) y un reply (respuesta). El request va a ser enviado por un cliente de audio a uno de nuestros clientes proxies y el reply va a ser enviado desde el servidor a nuestro servidor proxy. El reply, que está formado de una línea de estatus, headers y un cuerpo, va a ser partido en 2 partes como hicimos en el encoding. Primero vamos a parsear la línea de estatus y los headers y luego vamos a parsear el cuerpo. El cuerpo va a ser entonces partido en una secuencia de segmentos de datos.

El parser va a trabajar sobre binarios; esto va a ser mucho más eficiente el manejo de datos de audio grandes.

2.5.1. Un Request

Parsear un request es bastante simple, leemos la primer línea (todas las líneas están terminadas por <cr><lf>) y chequeamos que es igual a "GET / HTTP/1.0". Esto es una simplificación; un request podría incluir algo más interesante que "/" pero servirá por ahora.

La función `line/2` va a buscar los caracteres de fin de línea y si no los encuentra va a devolver `more`. El parser del request va devolver entonces una continuación en forma de una función que debe ser aplicada al siguiente segmento. Podemos hacer esto más complicado pero por que no simplemente no devolver una función que añadas el segmento que tenemos al siguiente segmento e intentamos repetir el parseo del 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.
```

Tener en cuenta como funcionan las reglas de scope; Erlang usa alcance (scope) lexicográfico y la variable `Bin` va a ser ligada cuando la función `request/1` sea llamada. La función retornada puede ahora ser llamada en cualquier entorno y la variable Bin va a mantener su binding. Esto se denomina alcance estático o lexicográfico.

Algunos lenguajes de programación funcionales usan alcance dinámico donde el valor de `Bin` va a depender del entorno donde es usada.

Una vez que tenemos la línea de request continuamos parseando los headers. El parseo de los headers va a exitoso o un pedido de más información. Si es exitoso devolvemos los `Headers` y también lo que queda parsear del segmento. En la práctica le resto va a ser un segmento vacío y posiblemente descartado por quien invoca.

Notar que hemos hecho una simplificación en la implementación del parser. Si la función `line/1` o `header/2` pasaron cientos de caracteres solo para encontrar que se necesitan más, simplemente retornan el átomo `more`. La próxima vez que sean invocadas van a tener que pasar por los mismos cientos de caracteres de nuevo. Sería bueno si pudiésemos guardar la posición exacta donde estamos y continuar solo con el nuevo segmento. Podríamos mantener registro de si deberíamos seguir leyendo una línea o un header. En la práctica, el binario original va a contener siempre ambos, la línea de request y todos los headers así que la pregunta se vuelve académica.

Leer una línea es un ejercicio de pattern matching simple y se deja para completar. Este es el esqueleto para partir: (<cr>puede ser escrito como `13` o `$\r` y <lf>como `10` o `$\n`)

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

2.5.2. Un Reply

Parsear una respuesta es muy similar. Aplicamos la misma estrategia que antes, intentamos parsear tanto como sea posible y empezamos desde el principio si necesitamos más. Lo que es levemente diferente es cuando hemos parseado exitosamente los headers. Ahora no solo devolvemos los headers sino una continuación que cuando la aplicamos genera el primer segmento de datos. Dado que la decodificación de los segmentos de datos necesitan información adicional de la longitud de la parte de audio primero extraemos esta información de los headers. La implementación de `metaint` se deja como ejercicio.

```
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.
```

Parsear un segmento consiste de 2 partes. Primero leemos `M` bytes del input stream y entonces decodificamos la sección de metadata. Vamos a adherirnos a nuestra estrategia de continuaciones y retornamos ya sea:

- `{more, Continuation}`: donde `Continuation` es una función que debe ser aplicada al siguiente segmento si se necesitan más segmentos o
- `{ok, {Audio, Meta}, Continuation}`: una vez que un segmento de datos completo fue leído. La continuación nos va a dar el siguiente segmento de datos cuando se aplica sin argumentos.

No es muy común que seamos capaces de leer todo el segmento de audio en una pasada. Un segmento de audio es generalmente al menos de 8192 bytes de tamaño y los paquetes de TCP tienen un tamaño de 1460 bytes en una Ethernet común. Cada segmento audio va a consistir de 6 chuncks. Dado que no queremos decodificarlos no tiene sentido concatenarlos en un binario. En cambio los mantenemos en una lista. Eventualmente es tarea de `send_audio/2` enviar los chunks uno por uno.

```
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.
```

Es importante entender como evitamos parsear los mismo binarios cada vez que pedimos más información. Retornamos una función que debemos aplicar al siguiente segmento pero recordamos lo que hemos visto hasta el momento y cuanto más nos queda por ver. Esto es diferente de la implementación de `request/1` donde simplemente empezamos desde el principio.

Parsear el segmento de metadata es un poco más complicado. Tenemos que primero leer el byte `k` así sabemos la longitud del siguiente string. El string

consiste de texto completado con 0s (padding) a un múltiplo de 16. Cuando leemos el byte `k` (puede ser 0), el siguiente string y removemos el padding. Podemos mantener el padding dado que solo vamos a tener un problema cuando debemos enviar el segmento a un cliente pero sería bueno tener un string correcto como metadata.

```
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.
```

Podríamos no haber visto la construcción que usamos para eliminar el padding. Se llama *listas por comprensión* y puede ser interpretado – "dame una lista de C's donde C es tomado de `binary_to_list(Padded)` y C es mayor que 0".

2.6. ¿Funciona?

Completar el módulo `icy` que exporta las funciones descriptas anteriormente: `request/1`, `reply/1`, `send_reply/1` y `send_data/1`. Luego podemos probar nuestra la implementación con los siguientes test de ejemplo:

```
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).
```

Experimentar con el parser es igualmente simple. Opera sobre binarios pero la sintaxis de binarios hace muy fácil construir los segmentos que necesitamos.

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

Probemos con algo incompleto.

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

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

Un reply debe siempre devolvernos una función que debería ser aplicada sin argumentos para devolvernos los segmentos de datos.

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

Dado que el cuerpo no contiene 5 bytes de audio ni sección de metadata deberíamos obtener un pedido de más si aplicamos la continuación.

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

Ahora apliquemos esta continuación al resto de la sección. El "1" indica un total de 16 bytes. El mensaje `"hello"` es entonces completado con 11 bytes en 0 (un entero 0 codificado en 11*8 bits).

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

3. La Arquitectura

Nuestro objetivo es construir un proxy (server proxy), que se conecta a un servidor shoutcast, y un cliente (client proxy) que acepta conexiones de un media player. El cliente debe conocer acerca del proxy y comunicarse con el usando mensajes Erlang. Vamos a usar los siguientes mensajes entre el cliente y el proxy.

- `{request, Client}`: donde el `Client` es un proceso Erlang que quiere conectarse.
- `{reply, N, Context}`: donde `Context` es el la información de header recibida desde la fuente. `N` es número de segmentos de datos que deberían llegar a continuación.
- `{data, N, Data}`: donde `N` es n entero con el número de todos los segmentos y `Data` es el audio y metadata que hemos recibido.

La numeración de los segmentos de datos es para mantener registro de que segmentos necesitamos cuando empezamos a construir redes de distribución más complicadas. En principio el orden FIFO de los mensajes de Erlang va a darnos los mensajes en el orden apropiado.

3.1. El Cliente

El cliente va a escuchar en un puerto TCP y esperar por las conexiones entrantes. Una vez que una conexión es aceptada un request es leído del socket. El contenido del request no es importante para nuestras necesidades, vamos a conectar alegremente cualquier media client a un proxy predefinido.

```
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.
```

Cuando se crea un listen socket especificamos las propiedades en una lista de opciones. Las opciones que usamos son:

- `binary`: es más eficiente dado que no tiene sentido manejar audio mp3 como una lista de enteros.
- `{packet, 0}`: poner la longitud del mensaje es muy útil pero estamos tratando con el protocolo ICY y no uno propio.
- `{active, true}`: el proceso del socket va a enviarnos los segmentos a medida que llegan, no tenemos que suspender leyendo el socket.
- `{nodelay, true}`: enviar los segmentos lo más rápido posible.

Notar como elegimos tener un timeout cuando esperamos por mensajes TCP. Podríamos haber elegido aceptar un mensaje `stop` o similar; el cliente es el módulo que tiene el control.

Una vez que leímos el request correcto (en realidad no nos preocupamos que es el request) tratamos de conectar al proxy. El proxy va a responder con un contexto (la información del header enviada por el server) que nosotros podemos enviar al cliente y luego entrar en un loop que continuamente entregue segmentos de datos de un proxy al media player.

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

El loop es simple y va a continuar enviando segmentos de datos hasta que la conexión TCP del media player se cierre o de un timeout. En este punto no chequeamos que recibimos todos los segmentos de datos pero esto podría hacerse fácilmente.

```
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.
```

Esta es la interfase que usamos al módulo `icy`. Enviar es bastante simple, usamos las funciones exportadas y proveemos una función de envío tcp que debería ser usada para enviar los binarios construidos por la interface de socket.

```
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).
```

La interface del parser usa un reader de propósito general que va a retornar `{ok, Parsed, Rest}` o `{error, Error}`. Recibe como argumento una función sin argumentos y un socket. La función es aplicada y el resultado o bien es parseo exitoso, un pedido de más datos o un mensaje de error. El pedido de más datos puede ser manejado fuera del módulo del parser. El reader va a entrar en una sentencia `receive` y esperar por más mensajes TCP.

Vamos a tener un timeout así no nos quedamos bloqueados esperando por segmentos que nunca van a llegar.

```
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.
```

Leer un request se puede definir de la siguiente manera:

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

Implementar el módulo `client` y exportar la función `init/2`. Esto es el el proceso cliente completo. Ahora veamos el proxy.

3.2. El Proxy

El proxy es aún más simple de implementar. Cuando arrancamos el proxy vamos a darle un stream shoutcast al cual conectarse. El stream está definido como una tupla:

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

Una fuente de shoutcast podría ser:

- `{cast, "192.99.62.212", 9904, "/"}`.

Primero vamos a esperar que un cliente pida una conexión antes de conectarnos al servidor shoutcast. Dado que no vamos a escuchar nada del cliente vamos a agregar un monitor. Si el cliente termina vamos a recibir un mensaje y podemos decidir que hacer. Esto no es estrictamente necesario pero va a reducir el número de proxies zombie cuando hagamos puebas.

```
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.
```

Una conexión al server va a consistir de: un stream en la forma de un socket abierto, una continuación, de la cual vamos a recibir segmentos de datos, y un contexto (la información del header). Una vez conectados enviamos una respuesta y empezamos transmitir los segmentos de datos.

```
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.
```

Conectarnos al servidor requiere algo de programación de sockets. Nos conectamos al servidor y enviamos un ICY request. Si el envío es exitoso continuamos leyendo la respuesta que va a resultar en una conexión válida o un mensaje de error.

```
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.
```

Cuando enviamos el request le pasamos a la función `gen_tcp:server/2` a la función `icy:send_request/3`.

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

La función `reader/3` es levemente diferente de la usada para el cliente. Ahora aprovechamos el hecho de que podemos actuar con otros mensajes además de los del proceso TCP. Un mensaje `'DOWN'` es enviado si el proceso del cliente monitoreado muere y queda no disponible. Podemos por supuesto suspender y esperar por nuevas conexiones, pero también podría morir.

```
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.
```

La lectura de una respuesta puede ser codificada de la siguiente mejoras.

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

Implementar el módulo `proxy` y exportar la función `init/1`. Si todo anduvo bien deberíamos ser capaces de conectar un media player a un cliente, un cliente a un proxy y un proxy a un server.

3.3. Streaming de Audio

Ahora tenemos las piezas del rompecabezas para empezar a hacer streaming de audio en nuestra red. Crear un módulo `test` y escribir algunas funciones que vamos a usar en nuestros experimentos. Primero vamos a crear un proxy y procesos cliente en el mismo nodo Erlang y ver que las cosas funcionan, luego vamos a hacer que los procesos ejecuten en 2 computadoras.

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

Arrancamos el proceso y conectamos nuestro cliente de audio (Totem, Amarok o VLC deberían funcionar) al stream `http://localhost:8080` (o el port que estemos usando). Si hemos agregado algunas sentencias print vamos a ver como el cliente acepta el request de player, lo envía al proxy que se conecta al server de read. Cada segmento de datos va a contener 2s de audio. Notar como el media player decodifica la metadata y la usa para describir el canal.

Tenemos 2 loops uno del lado del proxy y el otro en del lado del cliente. La llamada recursiva en el proxy es una llamada a `stream/5`, ahora si esto es una llamada a `proxy:stream/5` podemos usar la última versión cargada de la función en la recursión. Probar esto – cambiar el loop para que use `proxy:stream/5`, compilar y empezar a jugar, editar el código e incluir un print de cada loop, compilar y cargar el módulo mientras sigue reproduciendo. Observar que podemos actualizar el código sin perder un solo paquete de audio.

Si todo funciona correctamente es tiempo de arrancar el proxy en una computadora y el cliente en otra. Podemos incluso ejecutar el media player en una tercer computadora. Cuando corremos varios nodos Erlang asegurarnos de arrancarlos con un nombre apropiado y la misma cookie.

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

Chequear la carga del procesador en todas las máquinas. Estamos manejando un stream de audio de 128Kbps ¿se nota? Que pasa si desconectamos los cables, ¿podemos sobrevivir 3s o 4s de falla de red? Descartar 1 de cada 10 paquetes y ver si sigue funcionando. ¿Los procesos son terminados como se espera cuando el media player deja de reproducir? Hacer algunas pruebas antes de avanzar.

4. Un Servidor de Distribución

Ahora que hemos resuelto las partes complicadas de la comunicación con un media player y un servidor Shoutcast. Manejar los mensajes en Erlang es mucho más simple, los únicos mensajes que necesitamos atender son:

- `{request, Client}`: un request de un cliente
- `{reply, N, Context}`: la respuesta de un request, espera que `N` sea el siguiente paquete de datos
- `{data, N, Data}`: solo tenemos que ver `N` si lo deseamos

Nuestro sistema actual es limitado dado qeu solo permite la conexión de un cliente a cada proxy. Podemos extender el proxy para que maneje más clientes pero también podemos implementarlo como un proceso separado. Este es el esqueleto para el módulo `dist` que va a hacer justamente esto.

```
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.
```

Completar las partes faltantes para conectar un proceso de distribución a un proxy. Entonces arrancar los clientes uno por uno y conectarlos al distribuidor. Dado que el distribuidor necesita un media player antes de que se conecte puede ser útil contar con un cliente dummy.

```
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.
```

¿Cuántos clientes dummy podemos correr antes de que nuestra máquina empiece a tener problemas? No perdamos tiempo empezando clientes dummy 1 por 1. Escribir una función que arranca N clientes y entonces ver cuantos podemos arrancar. También tratar correr el distribuidor en una máquina en y los clientes dummy en otras – ¿es la capacidad de procesamiento de la red lo que limita el factor?

5. Construyendo un Árbol

Intentemos construir un árbol de distribución dinámico a medida que los clientes quieren conectarse. Vamos a usar 2 nuevos procesos que son similares y los vamos a implementar en el mismo módulo. Uno es el proceso root que se va a conectar a un proxy y esperar por hijos que se conecten. Solo va a permitir la conexión de 2 hijos y va a redireccionar a alguno de sus 2 hijos.

El otro proceso es el proceso hijo. Va a esperar la conexión de clientes y entonces conectarse al root. Debe estar preparado para redireccionar a otro hijo al cual intentará conectarse. Una vez conectado también debe abrirse para servir otros 2 hijos y redireccionar a estos.

Ahora vamos a tener 6 elementos en nuestra arquitectura de shoutcast. Un media player se conecta a un `porxy`. El proxy cliente piensa que se conecta a un proxy pero en realidad se conecta a un proceso hijo. El proceso hijo conoce que se está conectado a un root u otro hijo y el root se conecta a un proxy server. El proxy server se conecta entonces con el servidor shoutcast real.

En una implementación real probablemente podrían colapsar el proxy, el proceso root, el cliente y los procesos hijos pero notar como la separación de procesos hace que cada descripción sea más fácil de seguir. En un verdadero lenguaje concurrent la creación y ejecución de un proceso separado es tan normal como manejar la complejidad de manejar procedimientos y librerías. De la misma forma, cuando negociamos eficiencia por claridad cuando escondemos detalles de implementación en un módulo, usamos procesos para hacer nuestro sistema más fácil de implementar.

5.1. Los Mensajes

En el final del proyecto el objetivo es conectar todas las computadoras de la clase en un árbol de distribución. Vamos a tener un nodo dedicado que corre un proxy y un proceso root. Todos los otros nodos va a correr el media player, un proceso cliente y un proceso hijo. Dado que estamos implementando el módulo `tree` en forma independiente y un proceso hijo en un nodo no va a correr el mismo código que un proceso hijo en otro nodo, es importante especificar la interface de mensajes.

- `{request, Pid}`: un request a un proceso root o un hijo. El proceso (`Pid`) debe ser capaz de manejar mensajes de request una vez conectados.

- `{reply, N, Context}`: una respuesta de un proceso root o un hijo. El entero `N` es el número del siguiente segmento de dato, `Context` debe ser manejado por el módulo `icy`.
- `{redirect, Pid}`: este es el mensaje que se da como respuesta a un mensaje request cuando un root o un hijo no puede conectar más hijos. El `Pid` es un identificador de proceso a otro proceso hijo que debe tener lugar o nos va a redirigir de nuevo.
- `{data, N, Data}`: el n-esimo segmento de datos, los datos en si mismo deben ser manejados por el módulo `icy`.

Si nos quedamos con estos mensajes las cosas deberían funcionar en el primer intento.

5.2. El Root

Necesitamos nuestro propio root para hacer algunos experimentos iniciales. Este es el esqueleto que puede ser completado fácilmente. La primer función se va a conectar al proxy.

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

El loop va a aceptar mensajes de datos del proxy o requests de los procesos hijos que intentan conectarse. Vamos a aceptar los primeros 2 procesos hijos pero redirecciona el resto.

```
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.
```

Notar que nuestro árbol no se va a ver exactamente como un árbol si siempre redireccionamos al hijo izquierdo. Para mantener el árbol balanceado debemos redireccionar cada nuevo proceso a izquierda y derecha.

5.3. Los Hijos

Los procesos hijos se van a ver muy parecidos al proceso root. La diferencia es es que los procesos hijos deben primero esperar que se conecte un proceso cliente.

Cuando recibe un request de un cliente debe tratar de conectarse al root del árbol. Puede ser redireccionado varias veces pero al final debe terminar conectado.

Una vez conectado debe reenviar todos los paquetes de datos a su propio cliente. También debe estar abierto a pedidos de otros hijos. De forma parecida al proceso root debe aceptar los primeros dos hijos y redireccionar al resto.

Notar – un proceso hijo debe separar el proceso cliente de los hijos conectados. Debe recibir copias de los segmentos de datos pero no podemos redireccionar un nodo que se conecta a un cliente; el cliente no está preparado para manejar mensajes de request.

5.4. Manejo de Errores

¿Podemos hacer esta estructura más estable o auto reparable?. ¿Podemos detectar que nuestra fuente no está entregando como debería? Si el proceso root muere entonces no hay mucho por hacer pero si es un proceso hijo podríamos tratar de conectarnos de nuevo al root. ¿Tenemos tiempo de hacer esto antes de que algún otro hijo conectado nosotros tenga tiempo de descubrirlo?

Si estamos conectados al root podemos esperar tener los segmentos de datos entregados cada 500ms. ¿Deberíamos dar al root una ventana y definir un timeout después de 600ms? ¿Qué sucede si todos nuestros hijos tienen el mismo timeout?

¿Podemos usar monitores Erlang para detectar que que los nodos están caídos o tenemos que tener nuestra propia detección de fallas?

Asumamos que nuestra fuente falla la entrega y que manejamos la reconección. Si la fuente vieja ahora resume la transmisión y entrega los segmentos de datos vamos a tener 2 procesos entregando el mismo stream. ¿Cómo prevenimos esto? ¿Podemos introducir un mensaje de control para para la transmisión?

6. Una Arquitectura de Bittorrent

¿Cuán difícil sería implementar una red de distribución usando el protocolo *bittorrent*? ¿Qué problemas debemos resolver? Sería mejor que nuestro árbol de distribución? ¿Ventajas y desventajas?