Namy: un name server distribuido*

Federico C. Repond
frepond@unq.edu.ar

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

15 de marzo de 2016

Introducción

La tarea será implementar un *name server* distribuido similar a *DNS*. En lugar de direcciones vamos a almacenar identificadores de procesos a hosts. No va a poder inter-operar con servidores de DNS reales pero nos mostrará los principios de caching en una estructura de árbol.

1. La Arquitectura

Nuestra arquitectura tendrá 4 tipos de nodos:

- **servers**: son responsables de un dominio y mantienen un conjunto de hosts registrados y servidores de sub-dominios. Los servers forman una estructura de árbol.
- **resolvers**: son responsables de ayudar a un cliente a encontrar una dirección de un host. Van a consultar a los servers en forma iterativa y mantener un cache de las respuestas.
- **hosts**: son nodos que tienen un nombre y están registrados en un server. Los hosts responden mensajes *ping*.
- **clients**: solo conocen la dirección a resolver y la usan para encontrar la dirección de un host. Sólo enviaran mensajes *ping* y van a esperar por la respuesta.

Separando las tareas de los servers y resolvers va a hacer la implementación más clara y fácil de entender. En la un servidor de DNS real, estos también tienen la responsabilidad de un resolver.

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

1.1. El Server

Acá mostramos como implementamos el server. Esta es una versión simple donde hacemos spwan de un proceso y lo registramos en el name server. Esto significa que solo vamos tener un server corriendo en cada nodo Erlang. Se puede modificar para que tome un argumento extra con el nombre a registrar el server.

```
-module(server).
-export([start/0, start/2, stop/0, init/0, init/2]).

start() ->
    register(server, spawn(server, init, [])).

start(Domain, DNS) ->
    register(server, spawn(server, init, [Domain, DNS])).

stop() ->
    server ! stop,
    unregister(server).

init() ->
    server(entry:new(), 0).

init(Domain, Parent) ->
    Parent ! {register, Domain, {dns, self()}},
    server(entry:new(), 0).
```

Notar que hay 2 formas de iniciar el server. Puede ser el server root en nuestra red o un servidor responsable de un sub-dominio. Si es responsable de un sub-dominio el nombre tiene que estar registrado en el servidor padre. Los nombres de dominio están representados como átomos: `se`, `kth`, etc. Entonces el server kth estará registrado en el servidor se bajo el nombre `kth` pero no mantiene ninguna información que es responsable por el sub-dominio `[kth, se]`; esto queda implícito en la estructura de árbol.

El proceso server mantiene una lista de entradas *key-value*. Los hosts se registran deben registrar una tupla `{host, Pid}` y los servers una tupla `{dns, Pid}`. La diferencia va a evitar que el resolver envíe pedido a nodos hosts.

El servidor también mantiene un *ttl* (time-to-live) que será enviado con cada respuesta. El valor es el número de segundos por los cuales la respuesta es válida. En la realidad esto es establecido usualmente a 24h pero para experimentar con caching usaremos segundos. El valor por defecto es 0s, esto es no permite cachear.

```
server(Entries, TTL) ->
    receive
        {request, From, Req}->
            io:format("request ~w~n", [Req]),
```

```
            Reply = entry:lookup(Req, Entries),
            From ! {reply, Reply, TTL},
            server(Entries, TTL);
        {register, Name, Entry} ->
            io:format("register ~w~n", [Name]),
            Updated = entry:add(Name, Entry, Entries),
            server(Updated, TTL);
        {deregister, Name} ->
            io:format("deregister ~w~n", [Name]),
            Updated = entry:remove(Name, Entries),
            server(Updated, TTL);
        {ttl, Sec} ->
            server(Entries, Sec);
        status ->
            io:format("cache ~w~n", [Entries]),
            server(Entries, TTL);
        stop ->
            io:format("closing down~n", []),
            ok;
        Error ->
            io:format("strange message ~w~n", [Error]),
            server(Entries, TTL)
    end.
```

Cuando un server recibe un request va a tratar de buscar en su lista de entradas. La función `lookup/2` retornará `unknown` si no lo encuentra. **Esto es algo que tenemos que implementar**. No importa que resultado es, el servidor no va a tratar de encontrar una respuesta mejor al request. Es responsabilidad del resolver hacer los request iterativos.

En esta implementación hay un solo tipo de request. Podríamos haber dividido los hosts registrados de los sub-dominios y explícitamente pedir uno o el otro, o incluso un diseño más claro, priorizamos mantener las cosas simples.

1.2. El Resolver

El resolver es más complejo dado que debemos manejar el cache y debemos hacer una búsqueda iterativa para encontrar la respuesta final. Vamos a usar un módulo `time` (**que debemos implementar**) que nos ayudará a determinar si una entrada del cache es válida o no. Además usaremos un un truco y añadiremos una entrada permanente que en el cache que refiera al server root.

```
-module(resolver).
-export([start/1, stop/0, init/1]).

start(Root) ->
    register(resolver, spawn(resolver, init, [Root])).
```

```
stop() ->
    resolver ! stop,
    unregister(resolver).

init(Root) ->
    Empty = cache:new(),
    Inf = time:inf(),
    Cache = cache:add([], Inf, {dns,  Root}, Empty),
    resolver(Cache).

resolver(Cache) ->
    receive
        {request, From, Req} ->
            io:format("request ~w ~w~n", [From,Req]),
            {Reply, Updated} = resolve(Req, Cache),
            From ! {reply, Reply},
            resolver(Updated);
        status ->
            io:format("cache ~w~n", [Cache]),
            resolver(Cache);
        stop ->
            io:format("closing down~n", []),
            ok;
        Error ->
            io:format("strange message ~w~n", [Error]),
            resolver(Cache)
    end.
```

El resolver solo conoce el server root, no conoce en que dominio está trabajando. Si no puede encontrar una entrada en el cache va a enviar un request al server root. Los requests son de la forma `[www, kth, se]`, si no encuentra una entrada que corresponda al nombre completo en el cache intentará con `[kth, se]`. Si no encuentra una entrada para `[kth, se]` o `[se]`, vamos a encontrar una entrada para `[]` que nos devolverá la dirección del server root.

Cuando contactamos el server root consultamos por una entrada para el dominio `se`. Guardamos la respuesta en el cache y preguntamos al server `se` por el dominio `kth` y así sucesivamente. Cuando tenemos la dirección del host www enviamos la respuesta al cliente.

La implementación de la función resolve es algo intrincada y toma tiempo entender por qué y como funciona. Dado que la resolución de un nombre puede cambiar el cache la función retorna tanto la respuesta como el cache actualizado. La idea es la siguiente: `lookup`/2 busca en el cache y retorna o bien `unknown`, `invalid` en el caso de encontrar un valor expirado o, una entrada válida `{ok, Reply}`. Si el nombre de dominio fue `unknown` o `invalid` empieza un procedimiento recursivo, si se encuentra una entrada, esta puede ser retornada

directamente.

```
resolve(Name, Cache)->
    io:format("resolve ~w ", [Name]),
    case cache:lookup(Name, Cache) of
        unknown ->
            io:format("unknown ~n ", []),
            recursive(Name, Cache);
        invalid ->
            io:format("invalid ~n ", []),
            recursive(Name, cache:remove(Name, Cache));
        {ok, Reply} ->
            io:format("found ~w ~n ", [Reply]),
            {Reply, Cache}
    end.
```

El procedimiento recursivo va a dividir el nombre del dominio en 2 partes. So estamos buscando `[www, kth, se]` debemos buscar primero `[kth, se]` y usar este valor para buscar la dirección de `www`. La mejor forma de encontrar la dirección de `[kth, se]` es usar la función resolve.

Asumimos que `resolve/2` siempre devuelve un resultado (recordar que el cache mantiene una entrada permanente con el domain root `[]`) y esto es `unknown` o una entrada `{dns, Svr}`. Podríamos llegar a la situación que retorne un entrada `{host, Hst}` pero en tal caso la configuración es errónea.

```
recursive([Name|Domain], Cache) ->
    io:format("recursive ~w ", [Domain]),
    case resolve(Domain, Cache) of
        {unknown, Updated} ->
            io:format("unknown ~n", []),
            {unknow, Updated};
        {{dns, Srv}, Updated} ->
            Srv ! {request, self(), Name},
            io:format("sent ~w request to ~w ~n", [Name, Srv]),
            receive
                {reply, Reply, TTL} ->
                    Expire = time:add(time:now(), TTL),
                    {Reply, cache:add([Name|Domain], Expire, Reply, Updated)}
            end
    end.
```

Si el domino `[kth, se]` es desconocido entonces no hay forma de que `[www, kth, se]` pueda ser conocido, en tal caso se retorna el valor `unknown`. Si en caso contrario, tenemos un name server de dominio para `[kth, se]` podemos preguntar a este por la dirección de `www`. Podemos enviar un pedido y esperar por una respuesta, cualquier cosa que obtengamos va a ser la respuesta final. Retornamos la

respuesta pero también actualizamos el cache con la entrada para el nombre completo `[www, kth, se]`.

Se dejaron para implementar las funciones que manejan el cache y el tiempo de vida. Esta es una forma de resolver el módulo `time`. Vamos a tener el bug de milenio si esperamos que funcione después de medianoche, pero para nuestro propósito es más que adecuada. Nos valemos del hecho que cualquier atom es mayor que cualquier entero de forma tal que `inf` siempre va a ser más grande que cualquier tiempo.

```
-module(time).
-export([now/0, add/2, inf/0, valid/2]).

now() ->
    {H, M, S} = erlang:time(),
    H * 3600 + M * 60 + S.

inf() ->
    inf.

add(_, inf) ->
    inf;
add(S, T) ->
    S + T.

valid(C, T) ->
    C > T.
```

Se deja para implementar el procedimiento de lookup que es casi idéntico al lookup de una entry en el server. Debemos sin embargo guardar el valor de time-to-live con cada entrada y chequear si la entrada sigue siendo válida cuando se realiza el lookup.

1.3. El Host

Vamos a crear un host solo para tener algo para registrar y poder comunicarnos. Lo único que harán los hosts es responder a mensajes de ping. Lo único que necesitan recordar es registrarse con un servidor de DNS.

```
-module(host).
-export([start/3, stop/1, init/2]).

start(Name, Domain, DNS) ->
    register(Name, spawn(host, init, [Domain, DNS])).

stop(Name) ->
    Name ! stop,
    unregister(Name).
```

```
init(Domain, DNS) ->
    DNS ! {register, Domain, {host, self()}},
    host().

host() ->
    receive
        {ping, From} ->
            io:format("ping from ~w~n", [From]),
            From ! pong,
            host();
        stop ->
            io:format("closing down~n", []),
            ok;
        Error ->
            io:format("strange message ~w~n", [Error]),
            host()
        end.
```

Notar que el host se inicia dándole un nombre y un servidor de DNS. El nombre es solo el nombre del host, por ejemplo `www`, la ubicación del servidor en el árbol dicta el nombre completo de dominio.

1.4. Algunas Secuencias de Test

No implementaremos ningún cliente pero vamos a necesitar algunas funciones de ayuda para probar nuestro sistema. Dado que tenemos una jerarquía de name servers con hosts registrados podemos usar el resolver para encontrar un host y hacer un ping. Esperamos 1000ms por una respuesta del resolver y 1000ms por una respuesta del ping.

```
-module(client).
-export([test/2]).

test(Host, Res) ->
    io:format("looking up ~w~n", [Host]),
    Res ! {request, self(), Host},
    receive
        {reply, {host, Pid}} ->
            io:format("sending ping ...", []),
            Pid ! {ping, self()},
            receive
                pong ->
                    io:format("pong reply~n")
                after 1000 ->
                    io:format("no reply~n")
```

```
            end;
        {reply, unknown} ->
            io:format("unknown host~n", []),
            ok;
        Strange ->
            io:format("strange reply from resolver: ~w~n", [Strange]),
            ok
        after 1000 ->
            io:format("no reply from resolver~n", []),
            ok
    end.
```

Antes de terminar se deben implementar las piezas faltantes para poder formar parte de una red más grande de name servers. Cada uno o bien sera un server, un resolver o estará manejando un conjunto de hosts y clientes.

2. 1, 2, 3, Probando...

Ahora configuremos nuestra red de name servers. Para esto iniciaremos el shell de Erlang en diferente computadoras (podríamos hacerlo en la misma...). Dejemos los name servers en computadoras dedicadas y iniciemos varios hosts y clientes en otras.

Recordar iniciar el shell de Erlang usando `-name` y `-setcookie`.

Para iniciar un server root en 192.168.1.6, podemos iniciarlo de la siguiente forma:

```
$ rebar3 shell --name 'root@192.168.1.6' --setcookie dns

Eshell V7.2.1  (abort with ^G)
(root@192.168.1.6)1> server:start().
```

Ahora iniciamos los servidores de dominio top.level. Notar como se registran con su nombre local solamente y no el nombre de dominio completo. Así se vería en 2 máquinas diferentes: 192.168.1.7 y 192.168.1.8.

```
$ rebar3 shell --name 'se@192.168.1.7' --setcookie dns

Eshell V7.2.1  (abort with ^G)
(se@192.168.1.7)1> server:start(se, {server, 'root@192.168.1.7'}).

$ rebar3 shell --name 'kth@192.168.1.8' --setcookie dns

Eshell V7.2.1  (abort with ^G)
(kth@192.168.1.8)1> server:start(kth, {server, 'se@192.168.1.8'}).
```

Ahora configuremos más servers y registremos algunos hosts. Iniciar un resolver y empezar a experimentar.

```
$ rebar3 shell --name 'hosts@192.168.1.9' --setcookie dns

(hosts@192.168.1.9)1> host:start(www, www, {server, 'kth@192.168.1.8'}).
(hosts@192.168.1.6)1> host:start(ftp, ftp, {server, 'kth@192.168.1.8'}).
(hosts@192.168.1.9)3> resolver:start({server, 'root@192.168.1.6'}).
(hosts@192.168.1.9)4> client:test([www, kth, se], resolver).
```

3. Usando el Cache

En la configuración simple el ttl está puesto en 0. ¿Qué sucede si cambiamos esto a 2 o 4 segundos. ¿Cuánto se reduce el tráfico? Cambiarlo a un par de minutos y mover los hosts, esto es apagarlos e iniciarlos registrándolos bajo un nuevo nombre. ¿Cuándo se encuentra el nuevo server, cuantos nodos necesitan saber sobre el cambio?

Nuestra cache tambien tiene el problema de que hay entries que nunca son eliminadas. Las entries inválidas se eliminan y se actualizan pero si nunca se busca una entry nunca se eliminará. ¿Cómo puede la cache organizarse mejor? ¿Cómo podemos reducir el tiempo de búsqueda? ¿Podemos usar una tabla de hash o un árbol?