Namy: a distributed name server

Johan Montelius

October 2, 2016

Introduction

Your task will be to implement a distributed name server similar to DNS. Instead of addresses we will store process identifiers to *hosts*. It will not be able to inter-operate with regular DNS servers but it will show you the principles of caching data in a tree structure.

1 The architecture

Our architecture will have four kind of nodes:

- **servers**: are responsible for a domain and holds a set of registered hosts and sub-domain servers. Servers form a tree structure.
- **resolver**: are responsible for helping client find addresses to hosts. Will query servers in a iterative way and keep a cache of answers.
- **hosts**: are nodes that have a name and are registered in one server. Hosts will reply to ping messages.
- **clients**: know only the address of a resolver and uses it to find addresses of hosts. Will only send a ping messages to a host and wait for a reply.

Separating the tasks of the server and the resolver will make the implementation cleaner and easier to understand. In real life DNS servers also take on the responsibility of a resolver.

1.1 a server

This is how we implement the server. This is a vanilla set-up where we spawn a process and register it under the name `server`. This means that we will only have one server running in each Erlang node. If you want to you can modify the code to take an extra argument with the name to register the server under.

```
-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([], 0).

init(Domain, Parent) ->
    Parent ! {register, Domain, {dns, self()}},
    server([],0).
```

Note that there are two ways to start a server. Either it will be the root server in our network or a server responsible for a sub-domain. If it's responsible for a sub-domain this name has to be registered in the parent server. Domain names are represented with atoms such as: `se`, `com`, `kth` etc. Note that the *kth-server* will register with the *se-server* under the name `kth` but it does not hold any information that it is responsible for the `[kth, se]` sub-domain; this is implicit in the tree structure.

The server process itself, will keep a list of key-value entries. Hosts that register will register a tuple `{host, Pid}` and servers a tuple `{dns, Pid}`. The difference will be used by the resolver to prevent it from sending request to host nodes.

The server also keeps a time-to-live value that will be sent with each reply. The value is the number of seconds that the answer will be valid. In real life this is normally set to 24h but to experiment with caching we use seconds instead. The default value is zero second, that is no caching allowed.

```
server(Entries, TTL) ->
    receive
        {request, From, Req}->
            io:format("request ~w ", [Req]),
            Reply = entry:lookup(Req, Entries),
            From ! {reply, Reply, TTL},
            server(Entries, TTL);
        {register, Name, Entry} ->
            Updated = entry:add(Name, Entry, Entries),
            server(Updated, TTL);
        {deregister, 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.
```

Note that when the server receives a request it will try to look it up in its list of entries. The `lookup/2` function will return `unknown` if not found. This is something that you will have to implement. It does not matter what the result is, the server will not try to find a better answer to the request or a best match. It is up to the resolver to make iterative requests.

Also note that in this implementation there is only one kind of request. We could have divided the registered hosts and sub-domains and explicitly requested either or, perhaps a cleaner design, but we'll keep things simple.

1.2 a resolver

The resolver is more complex since we will now have a cache to consider and since we will do a iterative lookup procedures to find the final answer. We will use a `time` module (we'll have to implement it) that will help us to determine if a cache entry is valid or not. We will also use a trick and enter a permanent entry in the cache that refers to the root server.

```
-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.
```

Note that the resolver only knows the root server. It does know in which domain it is working. If it can not find a better entry in the cache it will send a request to the root. The requests are on the form `[www, kth, se]` and if we do not find a match of the whole name in the cache we will try with `[kth, se]`. If there is no entry for `[kth, se]` nor for `[se]` we will find the entry for `[]` which will give us a address of the root server.

When we contact the root server we ask for a entry for the `se` domain. We save the answer in the cache and then send a request to the se-server asking for the `kth` domain etc. When we have the address of the `www` host we send the reply back to the client.

The implementation of the resolve function is quite intricate and it takes a while to understand why and how it works. Since the resolving of a name can change the cache the procedure returns both the reply and an updated cache. The idea is now as follows: `lookup/2` will look in the cache and return either `unknown`, `invalid` in case a old value was found or, a valid entry, `{ok, Reply}`. If the domain name was unknown or invalid a recursive procedure takes over, if a entry is found this can be returned directly.

```
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, remove(Name, Cache));
        {ok, Reply} ->
            io:format("found ~w ~n ", [Reply]),
            {Reply, Cache}
    end.
```

The recursive procedure will divide the domain name into two parts. If we are looking for `[www, kth, se]` we should first look for `[kth, se]` and then use this value to request an address for `www`. The best way to find an address for `[kth, se]` is to use the resolve procedure.

We now make the assumption that `resolve/2` actually does return something (remember that the cache holds the permanent entry for the root domain `[]`) and that it's either `unknown` or a server entry `dns, Srv`. We could have a situation where it returns a host entry `host, Hst` but then our setup would be faulty.

```
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, TLL} ->
    Expire = time:add(time:now(), TTL),
                    {Reply, add([Name|Domain], Expire, Reply, Updated)}
            end
    end.
```

If the domain `[kth, se]` turns out to be unknown then there is no way that `[www, kth, se]` could be known so an unknown value can be returned directly. If however, we have a domain name server for `[kth, se]` we should of course ask this for the address to `www`. We send a request and wait for a reply, whatever we get is the final answer. We return the reply but also update the cache with a new entry for the full name `[www, kth, se]`.

Left to implement are the functions to handle the cache and the time. This is one way of solving the time module. We will of course have a millennium bug if we expect this to work over midnight but for our purposes it will be ok. We take advantage of the fact that any atom is greater then any integer so `inf` will always be greater than any time.

```
-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(S, T) ->
    S + T.

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

Left to implement is the lookup procedure which will be almost identical to the lookup procedure of the server. We must however store a time-to-live value which each entry and check if the entry is still valid when performing the lookup.

1.3 a host

We create some host only in order to have something to register and something to communicate with. The only thing our hosts will do is reply to `ping` messages. The only thing we have to remember is to register with a DNS server.

```
-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.
```

Note that a host is started by giving it a name and a DNS server. The name is only the name of the host, for example `www`, the location of the name server in the tree decides the full domain name.

1.4 some test sequences

We will not implement any clients but could need some functions to test our system. Given that we have a hierarchy of name servers with registered hosts we can use a resolver to find a host and then ping it. We wait for 1000ms for a reply from the resolver and 1000 ms for a ping reply.

```
-module(client).
-export([test/3]).

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.
```

Before the seminar you should have implemented the missing pieces so that you can take part in a larger name server network. You will either be a server, resolver or managing a set of hosts and clients.

2 A first try

Now let's set up a network of name severs. Start some Erlang shells on different computers. Let's have name servers one dedicated computers and have several hosts and clients on others.

Remember to start Erlang using the `-name` and `-setcookie` parameters. A root server on 130.237.250.69 could be started like this:

```
erl -name root@130.237.250.69 -setcookie dns
Eshell V5.4.13  (abort with ^G)
(root@130.237.250.69)1> server:start().
true
```

We then start servers for the top-level domains. Notice how they register with their local name only, not the full domain name. This is what it would look like on two machines: 130.237.250.123, 130.237.250.145.

```
erl -name se@130.237.250.123 -setcookie dns
Eshell V5.4.13  (abort with ^G)
(se@130.237.250.69)1> server:start(se, {server, 'root@130.237.250.69'}).
true

erl -name kth@130.237.250.145 -setcookie dns
Eshell V5.4.13  (abort with ^G)
(kth@130.237.250.145)1> server:start(kth, {server, 'se@130.237.250.123'}).
true
```

Now set up more servers and register some hosts. Start a resolver and experiment.

```
erl -name hosts@130.237.250.152 -setcookie dns
Eshell V5.4.13  (abort with ^G)
(hosts@130.237.250.152)1> host:start(www, www, {server, 'kth@130.237.250.145'}).
true
(hosts@130.237.250.152)1> host:start(ftp, ftp, {server, 'kth@130.237.250.145'}).
true
```

3 Using the cache

In the vanilla set-up the time-to-live is zero seconds. What happens if we extend this to two or four seconds. How much is traffic reduced? Extend it to a minute and then move hosts, that is close them down and start them up registered under a new name. When is the new server found, how many nodes needed to know about the change?

Our cache also suffers from old entries that are never removed. Invalid entries are removed and updated but if we never search for the entry we will not remove it. How can the cache be better organized? How would we do to reduce search time? Could we use a hash table or a tree?