Namy: a distributed naming system

Jordi Guitart

Adapted with permission from Johan Montelius (KTH)

February 10, 2025

Introduction

Your task will be to implement a distributed naming system similar to DNS. Instead of addresses we will store process identifiers of hosts. Our system will not be able to interoperate with regular DNS servers but it will show you the principles of caching data in a tree structure.

This assignment includes a number of coding and experimental milestones. On accomplishing each of them, the team must show their outcome to the professor during the practical sessions, so that he can track the progress and evaluate the team accordingly. The team can validate their coding milestones beforehand through the review questionnaires at ATENEA. This assignment also includes a number of open questions. The team must answer each of them through the questionnaire at ATENEA before the deadline (one week after the last session of the seminar).

1 Architecture

Our architecture will have four types of nodes:

- **Servers** are responsible for a domain and hold a set of registered hosts and sub-domain servers. Servers form a tree structure.
- **Resolvers** are responsible for helping clients to find addresses to hosts. Will query servers in an **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 use it to find addresses of hosts. Will only send a ping message 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 a 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 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]).

start() ->
    register(server, spawn(fun()-> init() end)).

start(Domain, Parent) ->
    register(server, spawn(fun()-> init(Domain, Parent) end)).

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

init() ->
    io:format("Server: create root domain~n"),
    server([], 0).

init(Domain, Parent) ->
    io:format("Server: create domain ~w at ~w~n", [Domain, Parent]),
    Parent ! {register, Domain, {domain, self()}},
    server([], 0).
```

Note that there are two ways to start a server. Either it will be the root server in our network (use `start/0`) or a server responsible for a sub-domain (use `start/2`). If it is responsible for a sub-domain, the domain name has to be registered in the parent server. Domain names are represented with atoms such as: `edu`, `com`, `upc`, etc. Note that the 'upc server' will register in the 'edu server' under the name `upc` but it does not hold any information that it is responsible for the `[upc,edu]` sub-domain; this is implicit in the tree structure.

The server process will keep a list of key-value entries (`Entries`). The key will be the domain name and the value will be the identifier of the process responsible for that domain. In particular, hosts will register an identifier formatted as a tuple {`host, Pid`} and name servers as a tuple {`domain, Pid`}. The difference will be used by the resolver to prevent it from sending resolution requests to host nodes. The list of entries can be queried by sending a `status` message to the server.

The server also keeps a time-to-live value (`TTL`) 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 seconds, that is, no caching allowed, and it can be changed by sending the {`ttl, Sec`} message to the server.

```
server(Entries, TTL) ->
    receive
        {request, From, Req}->
            io:format("Server: received request to solve [~w]~n", [Req]),
            Reply = entry:lookup(Req, Entries),
            From ! {reply, Reply, TTL},
            server(Entries, TTL);
        {register, Name, Entry} ->
```

```
            io:format("Server: registered subdomain ~w~n", [Name]),
            NewEntries = entry:add(Name, Entry, Entries),
            server(NewEntries, TTL);
        {deregister, Name} ->
            io:format("Server: deregistered subdomain ~w~n", [Name]),
            NewEntries = entry:remove(Name, Entries),
            server(NewEntries, TTL);
        {ttl, Sec} ->
            io:format("Server: updated TTL to ~w~n", [Sec]),
            server(Entries, Sec);
        status ->
            io:format("Server: List of DNS entries: ~w~n", [Entries]),
            server(Entries, TTL);
        stop ->
            io:format("Server: closing down~n", []),
            ok;
        Error ->
            io:format("Server: reception of 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. 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.

You must implement the `lookup/2` function in an `entry` module, together with the `add/3` and `remove/2` procedures (hint: you can use `lists:keyfind/3`, `lists:keystore/4`, and `lists:keydelete/3` functions).

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 an **iterative** lookup procedure to find the final answer. Each cache entry will have an expiration time that we will use to determine if it is valid or not. We will also use a trick and enter a permanent entry in the cache that refers to the root server. Note that we take advantage of the fact that any atom is greater than any integer so `inf` will always be greater than any time.

```
-module(resolver).
-export([start/1, stop/0]).
-define(timeout, 1000).

start(Root) ->
    register(resolver, spawn(fun()-> init(Root) end)).

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

```
    unregister(resolver).

init(Root) ->
    Cache = [],
    NewCache = cache:add([], inf, {domain, Root}, Cache),
    resolver(NewCache).

resolver(Cache) ->
    receive
        {request, From, Req}->
            io:format("Resolver: request from ~w to solve ~w~n", [From, Req]),
            {Reply, NewCache} = resolve(Req, Cache),
            From ! {reply, Reply},
            resolver(NewCache);
        status ->
            io:format("Resolver: cache content: ~w~n", [Cache]),
            resolver(Cache);
        stop ->
            io:format("Resolver: closing down~n", []),
            ok;
        Error ->
            io:format("Resolver: reception of 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 cannot find a better entry in the cache it will send a request to the root server. The requests are of the form `[www, upc, edu]`. If we do not find a match of the whole name in the cache we will try with `[upc, edu]`. If there is no entry for `[upc, edu]` nor for `[edu]` we will find the entry for `[]`, which will give us the address of the root server.

When we contact the root server we ask for an entry for the `edu` domain. We save the answer in the cache and then send a request to the 'edu server' asking for the `upc` domain, and so on. 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 an old value was found, or a valid entry (`Reply`). If the domain name was `unknown` or `invalid`, a recursive procedure takes over, if an 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", []),
            NewCache = cache:remove(Name, Cache),
            recursive(Name, NewCache);
        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, upc, edu]` we should first look for `[upc, edu]` and then use this value to request an address for `www`. The best way to find an address for `[upc, edu]` 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 is either `unknown` or a server entry {`domain, 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, NewCache} ->
            {unknown, NewCache};
        {{domain, Srv}, NewCache} ->
            Srv ! {request, self(), Name},
            io:format("Resolver: sent request to solve [~w] to ~w: ", [Name, Srv]),
            receive
                {reply, unknown, _} ->
                    io:format("unknown ~n", []),
                    {unknown, NewCache};
                {reply, Reply, TTL} ->
                    io:format("reply ~w~n", [Reply]),
                    Now = erlang:monotonic_time(),
                    Expire = erlang:convert_time_unit(Now, native, second) + TTL,
                    NewerCache = cache:add([Name|Domain], Expire, Reply, NewCache),
                    {Reply, NewerCache}
            after ?timeout ->
                io:format("timeout~n", []),
                {unknown, NewCache}
            end
    end.
```

If the domain `[upc, edu]` turns out to be unknown then there is no way that `[www, upc, edu]` could be known so an `unknown` value can be returned directly. If however, we have a domain name server for `[upc, edu]` 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, upc, edu]`.

Left to implement is the lookup procedure in the cache which will be almost identical to the lookup procedure of the server. We must however store the

expiration time of each entry and check if the entry is still valid when performing the lookup. **You must implement the `lookup/2` function in a `cache` module, together with the `add/4` and `remove/2` procedures.**

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 to reply to ping messages. We only have to remember to register the host with a name server.

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

start(Name, Domain, Parent) ->
    register(Name, spawn(fun()-> init(Domain, Parent) end)).

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

init(Domain, Parent) ->
    io:format("Host: create domain ~w at ~w~n", [Domain, Parent]),
    Parent ! {register, Domain, {host, self()}},
    host().

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

Note that a host is started by giving it a name (to register the process ID), a domain name, and a name server. The domain 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 of the host.

1.4 Client

We will implement a simple client 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 2000 ms for a reply from the resolver and for a ping reply.

```
-module(namy).
-export([ping/2]).
-define(timeout, 2000).

ping(Host, Resolver) ->
    io:format("Client: looking up ~w~n", [Host]),
    Resolver ! {request, self(), Host},
    receive
        {reply, {host, Pid}} ->
            io:format("Client: sending ping to host ~w ... ", [Host]),
            Pid ! {ping, self()},
            receive
                pong ->
                    io:format("Client: pong reply~n")
            after ?timeout ->
                io:format("Client: no reply from host~n")
            end;
        {reply, unknown} ->
            io:format("Client: unknown host~n", []),
            ok;
        Strange ->
            io:format("Client: strange reply from resolver: ~w~n", [Strange]),
            ok
    after ?timeout ->
        io:format("Client: no reply from resolver~n", []),
        ok
    end.
```

2 Testing

Now let's set up a network of name servers and do some experiments. Following the idea of the example below, build a name space having several top-level domains, intermediate domains and hosts.

You need to start some Erlang instances. Let's have name servers on dedicated instances and have several hosts and clients on others.

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

```
erl -name root@127.0.0.1 -setcookie dns

(root@127.0.0.1)1> server:start().
true
```

We can then start servers for the top-level domains. Notice how they register with their local name only, not the full domain name.

```
erl -name edu@127.0.0.1 -setcookie dns

(edu@127.0.0.1)1> server:start(edu, {server, 'root@127.0.0.1'}).
true
```

```
erl -name upc@127.0.0.1 -setcookie dns

(upc@127.0.0.1)1> server:start(upc, {server, 'edu@127.0.0.1'}).
true
```

Now we can register some hosts per domain.

```
erl -name hosts@127.0.0.1 -setcookie dns

(hosts@127.0.0.1)1> host:start(www, www, {server, 'upc@127.0.0.1'}).
true
(hosts@127.0.0.1)2> host:start(ftp, ftp, {server, 'upc@127.0.0.1'}).
true
```

Finally, we can start a resolver and ping a host.

```
erl -name client@127.0.0.1 -setcookie dns

(client@127.0.0.1)1> resolver:start({server, 'root@127.0.0.1'}).
true
(client@127.0.0.1)2> namy:ping([www,upc,edu], resolver).
```

CODE REVIEW (2 points)
Functions on the list of entries in the server: Erlang file `entry.erl`.
Functions on the list of cached entries in the resolver: Erlang file `cache.erl`.

EXPERIMENTAL MILESTONES (1.25 points)	
MS1 (0.75 points)	Build a name space with servers and hosts and test with several clients asking for name resolution concurrently.
MS2 (0.5 points)	Solve the name of a host, shut it down (by sending a `stop` message), and try to solve its name again.

OPEN QUESTIONS (0.5 points)
a) What happens if a client repeats the same query on experiment `MS1`?
b) What is the observed behavior on experiment `MS2`? Justify why.

The behavior on experiment `MS2` is probably a bit awkward. If the host has been shut down, the naming system should not longer return its process identifier. **Modify the `server` and the `host` code so that when they are shut down, they unregister their entry in their corresponding parent domain** and repeat experiment `MS2`.

CODE REVIEW (1.5 points)
Unregister server/host entry on shutdown: Erlang files `server.erl` and `host.erl`.

OPEN QUESTIONS (0.25 points)
c) What is now the observed behavior on experiment `MS2`?

3 Using the cache

In our initial setup, the time-to-live (TTL) was zero seconds. Now, we will do some experiments with caching enabled by setting the corresponding TTL **on every server** (recall that the TTL can be changed by sending the {`ttl, Sec`} message to the servers).

EXPERIMENTAL MILESTONES (0.5 points)	
MS3 (0.5 points)	Set a long TTL (i.e., minutes) and 'move' a host (i.e., query the name of a host, shut it down, start it up registered under the same name, and finally query the host again).

OPEN QUESTIONS (1.25 points)
d) What is the observed behavior on experiment `MS3`? Justify why.
e) Which nodes have been informed about the host movement?
f) When will the client find the host correctly in the new location?
g) Derive a theoretical quantification of the amount of messages needed for name resolution without and with the cache (assume a resolver that repeats the same query about a host at depth D in the namespace every F seconds for a total duration of R seconds, being the TTL equal to T seconds)[1].

Our cache also suffers from stale entries that are never removed. Invalid entries are removed and updated but if we never search for the entry we will not remove it. **Modify the `resolver` code so that it purges the cache when it receives the message `purge`**. You must implement the `purge` function in the `cache` module, which will actively walk through the cache and **remove stale entries**, returning at the end the updated cache. Use the `status` message that prints the content of the cache in the resolver to facilitate the testing.

[1] Hint to check that your formulation is correct: if $D = 3$, $F = 4$, $R = 60$, and $T = 6$, 96 messages are needed without caching and 48 messages are needed with caching

CODE REVIEW (1 points)
Cache purge: Erlang file `cache.erl`.

EXPERIMENTAL MILESTONES (0.5 points)	
MS4 (0.5 points)	Enable caching and make tests to show that the purging procedure removes correctly the stale entries from the cache.

4 Recursive resolution

Set up the naming system by using the new versions of the `resolver` and the `server` that implement recursive resolution. Perform experiments to compare this version with the former one using iterative resolution. Focus especially on how caching performs on each one when using several clients.

EXPERIMENTAL MILESTONES (0.75 points)	
MS5 (0.75 points)	Repeat the experiments with several concurrent clients on `MS1` while using recursive resolution **with caching enabled**.

OPEN QUESTIONS (0.5 points)
i) What happens if two clients perform the same query on experiment `MS5`?
j) Which resolution can exploit caching better? Justify why.