Erlang DNS Server

Simon Fowler

December 13, 2015

1 Introduction

In this report, we describe the adaptation of a freely-available DNS server, `erlang-dns`[1] to use multiparty session types, within the `monitored-session-erlang` [6] framework for monitoring session-typed Erlang applications.

The actor model [1, 7], while originally developed in the context of a model for artificial intelligence, has found much use in developing scalable and fault-tolerant distributed systems. Erlang [2] is a programming language, which, following the actor model, disallows communication via shared memory, instead requiring all co-ordination to happen via explicit message passing.

The `erlang-dns` application a server for the Domain Name System (DNS) [9, 4] which allows domain names to be resolved to IP addresses. Unlike the Simple Mail Transfer Protocol (SMTP) use case, the DNS protocol is a simple request-response protocol: instead, the request is encoded using a sophisticated packet structure with many inter-dependencies: as an aside, these are possible to encode using dependent types [5]. The purpose of this report, however, is to show how multiparty session types can be used to describe nontrivial communication patterns *within the DNS server itself*.

2 Description

The `erlang-dns` project is an open-source Erlang DNS server. The Erlang supervision tree for `erlang-dns` is shown in Figure 2.

In particular, there are three types of actors that are of interest:

- `ed_udp_handler` (denoted by $\texttt{Req}_0$ ... $\texttt{Req}_n$): these actors handle incoming UDP requests.
- `ed_zone_data_server` (denoted by `.com`, `.net`): these actors contain mappings from domain names to IP addresses, for each type of
- `ed_zone_registry_server`: maps domain names to the actor `ed_zone_data_server` PIDs.

At a high level, the steps taken upon receiving an incoming DNS request are as follows:

1. A `ed_udp_handler` actor is created to handle the DNS request
2. The `ed_udp_handler` decodes the request, and contacts the `ed_zone_registry_server` with the requested domain name
3. The `ed_zone_registry_server` attempts to find the actor responsible for the given domain name: if the domain name exists, then the PID of the `ed_zone_data_server` is returned, whereas if the domain name does not exist, then an error is returned.
4. If the zone does not exist, and error packet response is sent to the client. If the zone *does* exist, then the request is sent to the `ed_zone_data_server`.
5. The `ed_zone_data_server` responds with the relevant mapping entries.

[1] `http://www.github.com/hcvst/erlang-dns`

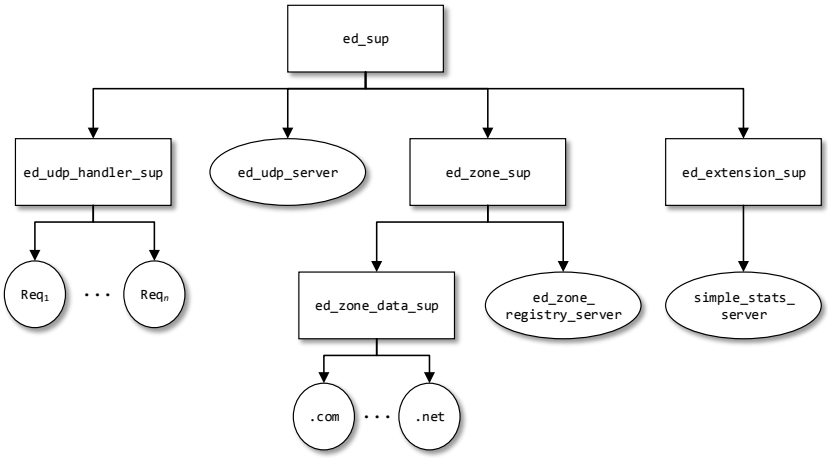


Figure 1: Supervision Tree for `erlang-dns`

6. The `ed_udp_handler` uses the response to either fulfil the request, sending a successful response to the client, or uses the response to make recursive queries.

3 Scribble Protocol

We describe the multiparty session type using an extension of the Scribble [8, 11] protocol description language. Note that in order to encode the dynamic introduction of a new party into the session, we make use of *subsessions* [3] to invite external participants. It is not possible to encode the protocol without subsessions, since the inhabitant of the `DNSZoneDataServer` role (the actor containing the mapping table for a particular DNS zone) is not known at the start of the protocol. We also make use of a new construct `call` for encoding synchronous calls.

```
module src.com.simonjf.ScribbleExamples.DNSServer.DNSServer;

type <erlang> "atom" from "" as EncodedRequest;
type <erlang> "pid" from "" as ZonePID;
type <erlang> "atom" from "" as RRTree;
type <erlang> "string" from "" as DomainName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as Zone;


global protocol HandleDNSRequest(role UDPHandlerServer, role DNSZoneRegServer) {
  rec QueryResolution {
    // Request the nearest zone
    FindNearestZone(DomainName) from UDPHandlerServer to DNSZoneRegServer;

    // Either we have it, or we don't...
    choice at DNSZoneRegServer {
        // If we do, then get the PID for the zone data server
        ZoneResponse(ZonePID) from DNSZoneRegServer to UDPHandlerServer;
        // Introduce the zone data server using a subsession
        UDPHandlerServer initiates GetZoneData(UDPHandlerServer, new DNSZoneDataServer) {
          // Now we've done that, we can do possible recursive lookups
          continue QueryResolution;
        }
    } or {
      InvalidZone() from DNSZoneRegServer to UDPHandlerServer;
    }
  }
}

global protocol GetZoneData(role UDPHandlerServer, role DNSZoneDataServer) {
    call ZoneDataRequest() returning RRTree from UDPHandlerServer to DNSZoneDataServer {}
}
```

The Scribble protocol consists of two separate protocols: `HandleDNSRequest`, which is the top-level protocol which co-ordinates how a request is handled, and `GetZoneData`, which requests the concrete zone data from the external participant.

There are three types of role:

- `UDPHandlerServer`: an actor spawned to handle a DNS request.
- `DNSZoneRegServer`: an actor which acts as a registry for each individual DNS zone, mapping top-level DNS zone names to actor PIDs
- `DNSZoneDataServer`: an actor which contains the zone data for each individual DNS zone

The first message is sent from the UDPHandlerServer to the DNSZoneRegServer to find the PID of the nearest zone. If this is not found, then the DNSZoneRegServer responds with an `InvalidZone()` message. If it is found, however, then the `DNSZoneRegServer` sends a `ZoneResponse` message, along with the PID of the actor holding the zone data. The `UDPHandlerServer` then initiates the `GetZoneData` subsession, directly inviting the returned PID to fulfil the `DNSZoneDataServer` role. Finally, the zone data is returned: the `UDPHandlerServer` is then free to make a recursive lookup if necessary.

3.1 Local Projections

3.1.1 UDPHandlerServer

```
module src.com.simonjf.ScribbleExamples.DNSServer.DNSServer_UDPHandlerServer;

type <erlang> "atom" from "" as EncodedRequest;
type <erlang> "pid" from "" as ZonePID;
type <erlang> "atom" from "" as RRTree;
type <erlang> "string" from "" as DomainName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as Zone;

local protocol HandleDNSRequest at UDPHandlerServer(role UDPHandlerServer,role DNSZoneRegServer) {
  rec QueryResolution {
    FindNearestZone(DomainName) to DNSZoneRegServer;
    choice at DNSZoneRegServer {
      ZoneResponse(ZonePID) from DNSZoneRegServer;
      UDPHandlerServer initiates GetZoneData( UDPHandlerServer , new DNSZoneDataServer ) {
        continue QueryResolution;
      }
    } or {
      InvalidZone() from DNSZoneRegServer;
    }
  }
}

local protocol GetZoneData at UDPHandlerServer(role UDPHandlerServer,role DNSZoneDataServer) {
  send_call_request ZoneDataRequest() to DNSZoneDataServer;
  receive_call_response ZoneDataRequest(RRTree) from DNSZoneDataServer;
}
```

3.1.2 DNSZoneRegServer

```
module src.com.simonjf.ScribbleExamples.DNSServer.DNSServer_DNSZoneRegServer;

type <erlang> "atom" from "" as EncodedRequest;
type <erlang> "pid" from "" as ZonePID;
type <erlang> "atom" from "" as RRTree;
type <erlang> "string" from "" as DomainName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as Zone;

local protocol HandleDNSRequest at DNSZoneRegServer(role UDPHandlerServer,role DNSZoneRegServer) {
  rec QueryResolution {
    FindNearestZone(DomainName) from UDPHandlerServer;
    choice at DNSZoneRegServer {
      ZoneResponse(ZonePID) to UDPHandlerServer;
    } or {
      InvalidZone() to UDPHandlerServer;
    }
  }
}
```

3.1.3 DNSZoneDataServer

```
module src.com.simonjf.ScribbleExamples.DNSServer.DNSServer_DNSZoneDataServer;

type <erlang> "atom" from "" as EncodedRequest;
type <erlang> "pid" from "" as ZonePID;
type <erlang> "atom" from "" as RRTree;
type <erlang> "string" from "" as DomainName;
type <erlang> "list" from "" as StringList;
type <erlang> "string" from "" as Zone;

local protocol GetZoneData at DNSZoneDataServer(role UDPHandlerServer,role DNSZoneDataServer) {
  receive_call_request ZoneDataRequest() from UDPHandlerServer;
  send_call_response ZoneDataRequest(RRTree) to UDPHandlerServer;
}
```

4 Implementation using `monitored-session-erlang`

The `monitored-session-erlang` framework is a framework for allowing actors in Erlang programs to be verified using multiparty session types. The system is described in depth elsewhere [6], but for the purposes of this document it sufficies to know that `monitored-session-erlang` provides a mechanism, based heavily on the work of [10], by which actors are *invited* into a session.

The first step is to define a configuration file, which describes the roles that actors may play in each protocol.

```
-module(edns_conversation_conf).
-export([config/0]).

config() ->
  [{ed_zone_data_server, [{"GetZoneData", ["DNSZoneDataServer"]}]},
   {ed_zone_registry_server, [{"HandleDNSRequest", ["DNSZoneRegServer"]}]},
   {ed_udp_handler_server,
    [{"HandleDNSRequest", ["UDPHandlerServer"]},
     {"GetZoneData", ["UDPHandlerServer"]}]
   }].
```

This configuration file states, for example, that the `ed_udp_handler_server` actor can play the "`UDPHandlerServer`" role in the "`HandleDNSRequest`" and "`GetZoneData`" protocols.

Next, each actor is implemented as an instance of the `ssa_gen_server` behaviour, which extends the Erlang `gen_server` behaviour with monitoring capabilities. Each instance of the `ssa_gen_server` behaviour must provide the following callbacks:

- `ssactor_init`: called when the actor is spawned
- `ssactor_join`: called when the actor is invited to fulfil a role in a protocol
- `ssactor_conversation_established`: called when a session is established
- `ssactor_conversation_error`: called if it is not possible to establish a session after inviting the actor

Each process is associated with an external monitoring process which maps the current protocol / role / session tuple with a finite state machine to monitor incoming and outgoing communication. A monitored message is only delivered if it is accepted by the FSM.

As an example, the `ed_zone_registry_server` provides the following callbacks:

```
ssactor_init([], _MonitorPID) ->
  {0, gb_trees:empty()}.
ssactor_join(_, _, _, State) -> {accept, State}.
ssactor_conversation_established(_PN, _RN, _CID, _ConvKey, State) -> {ok, State}.
ssactor_conversation_error(_, _, _, State) -> {ok, State}.

ssactor_handle_message("HandleDNSRequest", "DNSZoneRegServer", _CID, _Sender,
                       "FindNearestZone", [DomainName], State, ConvKey) ->
  {NumRequests, GBTree} = State,
  NameTails = ed_utils:tails(string:tokens(DomainName, ".")),
  Names = lists:map(fun(X) -> string:join(X, ".") end, NameTails),
  IsZoneNotDefined = fun(Z) -> not gb_trees:is_defined(Z, GBTree) end,
  case lists:dropwhile(IsZoneNotDefined, Names) of
    [] ->
      conversation:send(ConvKey, ["UDPHandlerServer"], "InvalidZone", [], []);
    [H|_] ->
      Pid = gb_trees:get(H, GBTree),
      conversation:send(ConvKey, ["UDPHandlerServer"], "ZoneResponse", ["ZonePID"], [Pid])
  end,
  {ok, {NumRequests + 1, GBTree}}.
```

Here, we see that the actor will always accept an invitation. The `ssactor_handle_message` is called upon receipt of a monitored asynchronous message, providing the current protocol name, role name, session ID, sender, message name, payload, actor state, and session key (`ConvKey`). The session key is an opaque value containing information about the current protocol, role, and session key: this is required to dispatch outgoing messages to the correct monitor. In order to send a monitored asynchronous message, the `conversation:send` function is used. This takes the session key, list of roles, message name, message payload types, and payload.

To start a new subsession, the `conversation:subsession` function is used. This takes the current session key; the name of the protocol that should be started; a list of *internal invitations* (that is, roles which should be populated using the inhabitants from the current session) and *external invitations*, which specify role-PID mappings for actors which are not part of the current session.

```
get_zone(Pid, ConvKey) ->
  conversation:start_subsession(ConvKey, "GetZoneData", ["UDPHandlerServer"],
                                [{"DNSZoneDataServer", Pid}]).
```

5 FSMs

5.1 UDPHandlerServer

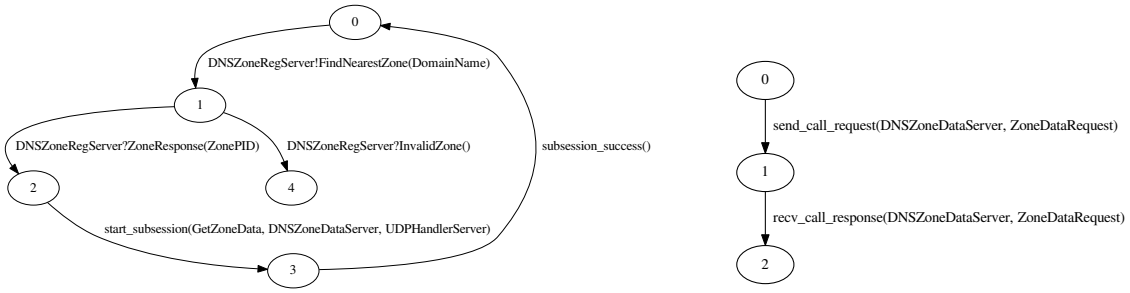


Figure 2: Monitor FSMs for `UDPHandlerServer` in `HandleDNSRequest` and `GetZoneData`

5.2 DNSZoneRegServer

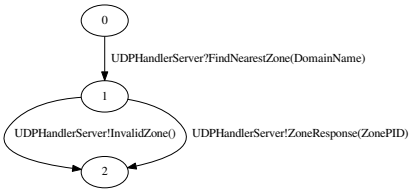


Figure 3: Monitor FSMs for `DNSZoneRegServer` in `HandleDNSRequest`

5.3 DNSZoneDataServer

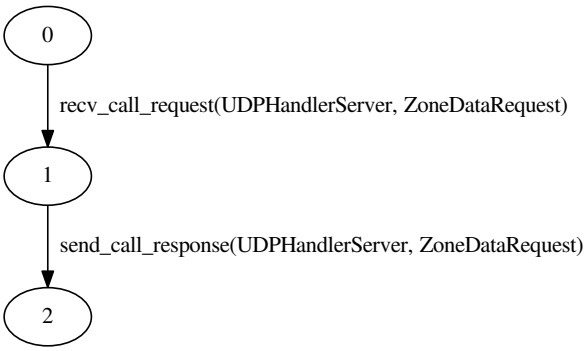
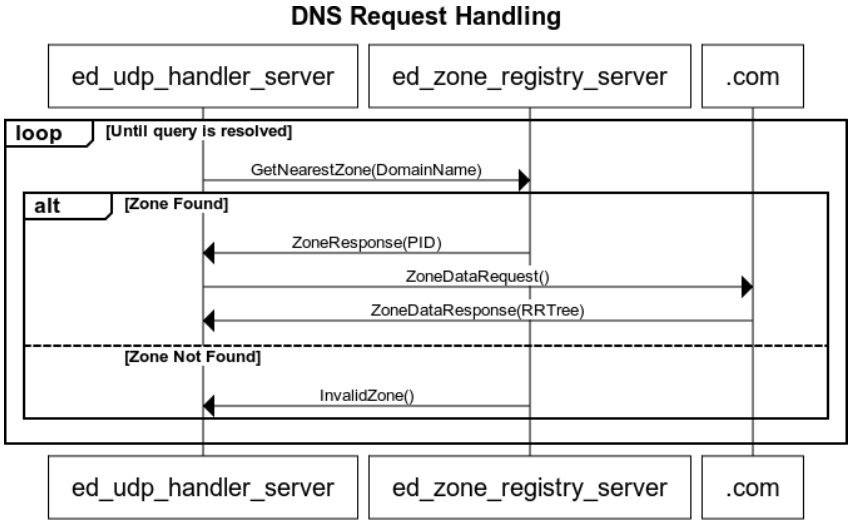


Figure 4: Monitor FSMs for `DNSZoneRegServer` in `GetZoneData`

6 Sequence Diagram



References

[1] Gul A. Agha. Actors: a Model of Concurrent Computation in Distributed Systems. 1985.

[2] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[3] Romain Demangeon and Kohei Honda. Nested Protocols in Session Types. In *CONCUR 2012—Concurrency Theory*, pages 272–286. Springer, 2012.

[4] D. Eastlake. RFC 2535–Domain name system security extensions. 1999.

[5] Simon Fowler. Verified Networking using Dependent Types. BSc dissertation, April 2014.

[6] Simon Fowler. Monitoring Erlang/OTP applications using multiparty session types. Master's thesis, August 2015.

[7] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL `http://portal.acm.org/citation.cfm?id=1624804`.

[8] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. *Scribbling Interactions with a Formal Foundation*, volume 6536 of *Lecture Notes in Computer Science*, chapter 4, pages 55–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-19055-1. doi: 10.1007/978-3-642-19056-8_4. URL `http://dx.doi.org/10.1007/978-3-642-19056-8_4`.

[9] Paul Mockapetris. RFC 1035âĂŤDomain namesâĂŤimplementation and specification. 1987.

[10] Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 131–146. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-662-43376-8_9. URL `http://dx.doi.org/10.1007/978-3-662-43376-8_9`.

[11] The Scribble Team. Scribble Language Reference. `http://www.doc.ic.ac.uk/\~{}rhu/scribble/langref.html`, 2013.