IVR using mod_erlang_event

About

An example project scavenged together from various sources.

- **Base logic** section shows the configuration and a `gen_fsm` implementation to handle calls in outbound mode (by Seven Du)
- `freeswitch_msg.erl` by Belaid Areski

Base logic

A mirror of Seven Du's Build a complex hence powerful FreeSWITCH IVR in Erlang.

[freeswitch] [erlang]

FreeSWITCH is powerful, which has cool and awesome applications built in that allows you do almost anything you want. We had built our call center using a combination of dialplan and event_socket. As the logic goes more complex, we decided to re-implement in Erlang.

What is Erlang?

This is Erlang.

Why Erlang?

Erlang is cocurrent, Erlang is functional, and more important, Erlang is designed to program telecom applications.

Erlang has otp, it is very easy to program Finite State Machine with gen_fsm.

How?

_We are running our PBX server in office while it need to fetch data from a very far remote server(could be in a remote datacenter or even in another country) to decide where an incoming call should be distributed to.

FreeSWITCH has mod_http and mod_curl built in, but we think Erlang is more robust when a remote server stops responding which will definitely happen on the world wide Internet. And we can get other advantage from Erlang. We spawn a new process to fetch data from a remote HTTP server immediately when a call comes in without even touching the call flow, and when we need the data for decision, the data already there, therefore no delay for customers.

In addition, we also post call related data to remote servers. We have time conditions stored in local mysql DB, and regular time conditions built in.

We use outbound mode in mod_erlang_event, whenever a call comes in, we send to a Erlang node imediately from dialplan:

```
<extension name="icall_fsm">
  <condition field="destination_number" expression="^fsm$">
    <action application="erlang" data="icall:fsm idp_acd@192.168.1.27"/>
  </condition>
</extension>
```

FreeSWITCH will do a rpc call to start a new gen_fsm process to handle the call. What charming is: the FSM can get all related events event you transfer a call out of the erlang FSM.

```
-module(icall).
-behaviour(gen_fsm).

-export([start/0, stop/0, fsm/1, init/1, welcome/2, handle_info/3, handle_event/3, terminate/3]).
-export([welcome_wait_playback/2, main_menu/2, main_menu_wait_dtmf/2, call_hst/2, call_sales/2, call_cc/2,
call_extn/2,
    call_cellphone/2, final_loop/2]).

-import(freeswitch_msg, [get_var/2, get_var/3, sendmsg/3]).
```

```
start() -> gen_fsm:start(icall_fsm, [], []).
stop() ->  gen_fsm:send_all_state_event(self(), stop).
fsm(Ref) ->
    {ok, NewPid} = ?MODULE:start(),
    {Ref, NewPid}.


%% send a stop this will end up in "handle_event"
% stop() -> gen_fsm:send_all_state_event(hello, stopit).
%% -- interface end
% This initialisation routine gets called
init(State) ->
    io:format("icall_fsm init ~p, PID: ~p~n", [State, self()]),
    {ok, welcome, []}.

%% The state machine
welcome([], []) ->
    {next_state, welcome, []};
% welcome(UUID, []) ->
%   {next_state, welcome, [UUID]};
welcome({call, {event, [UUID | Data]}}, []) ->
    CallerID = get_var("Channel-Caller-ID-Number", Data, "00000000"),
    Pid = self(),
    spawn(fun() -> fetch_cc_extn_from_crm(Pid, CallerID) end),
    io:format("New call from [~p]~n", [CallerID]),
    db_pbx:new_call(UUID, Data),
    sendmsg(UUID, playback, "new_sales/1000.wav"),
    {next_state, welcome_wait_playback, UUID}.

welcome_wait_playback({call_event, {event, [UUID | Data]} }, UUID) ->
    Name = get_var("Event-Name", Data),
    App = get_var("Application", Data),

    error_logger:info_msg("welcome_wait_playback: Pid ~p: UUID ~p, Event ~p~n",[self(), UUID, Name]),

    case Name of
        "CHANNEL_EXECUTE_COMPLETE" when App =:= "playback" ->
            NextState = route_time_condition(UUID),
            gen_fsm:send_event(self(), UUID),
            {next_state, NextState, UUID};
        _ ->
            {next_state, welcome_wait_playback, UUID}
    end.

main_menu(UUID, UUID) ->
    sendmsg(UUID, play_and_get_digits, "1 1 3 5000 # new_sales/1001.wav new_sales/9004.wav menu_number [1-
5]"),
    db_pbx:log(UUID, "MainMenu", ""),
    gen_fsm:send_event(self(), {call_event, {event, [UUID]}}),
    {next_state, main_menu_wait_dtmf, UUID}.
main_menu_wait_dtmf({call_event, {event, [UUID | Data]} }, UUID) ->
    Name = get_var("Event-Name", Data),
    App = get_var("Application", Data),

    error_logger:info_msg("Pid ~p: UUID ~p, Event ~p, State: main_menu_wait_dtmf~n",[self(), UUID, Name]),

    case Name of
        "CHANNEL_EXECUTE_COMPLETE" when App =:= "play_and_get_digits" ->
            MenuNumber = get_var("variable_menu_number", Data),
            db_pbx:log(UUID, "MainMenu", MenuNumber),
            case MenuNumber of
                "1" ->
                    gen_fsm:send_event(self(), UUID),
                    {next_state, call_hst, UUID};
                "5" ->
                    %<min> <max> <tries> <timeout> <terminators> <file> <invalid_file> <var_name> <regexp>
                    sendmsg(UUID, play_and_get_digits, "3 3 3 5000 # new_sales/2002.wav new_sales/9004.wav
extn_number [68]\\d\\d"),
                    {next_state, call_extn, no_extn};
                X when X == "2"; X == "3"; X == "4" ->
                    gen_fsm:send_event(self(), UUID),
```

```
                    {next_state, call_sales, UUID};
                _ ->
                    sendmsg(UUID, playback, "new_sales/9003.wav"),
                    timer:sleep(5000),
                    sendmsg(UUID, hangup, ""),
                    {next_state, final_loop, UUID}
            end;
        _ ->
            {next_state, main_menu_wait_dtmf, UUID}
    end.

call_hst(UUID, UUID) ->
    transfer(UUID, "fifo_hst"),
    {next_state, final_loop, UUID}.

call_cellphone(UUID, UUID) ->
    transfer(UUID, "fifo_cellphone"),
    {next_state, final_loop, UUID}.

call_sales(UUID, UUID) ->
    case get(cc_extn) of
        undefined ->
            transfer(UUID, "fifo_sales"),
            db_pbx:log(UUID, "SalesFifo", ""),
            {next_state, final_loop, UUID};
        Extn ->
            play_intransfer(UUID),
            sendmsg(UUID, set, "ringback=${us-ring}"),
            sendmsg(UUID, set, "continue_on_fail=true"),
            sendmsg(UUID, set, "hangup_after_bridge=true"),
            sendmsg(UUID, bridge, "user/" ++ Extn),
            db_pbx:log(UUID, "CallCC", Extn),
            {next_state, call_cc, Extn}
    end.

call_cc({call_event, {event, [UUID | Data]} }, Extn) ->
    Name = get_var("Event-Name", Data),
    App = get_var("Application", Data),

    error_logger:info_msg("Pid ~p: UUID ~p, Event ~p, Extn ~p~n",[self(), UUID, Name, Extn]),

    case Name of
        "CHANNEL_EXECUTE_COMPLETE" when App =:= "bridge" ->
            HangupCause = get_var("variable_originate_disposition", Data),
            DialedUser = get_var("variable_dialed_user", Data),
            sendmsg(UUID, play_and_get_digits, "1 1 2 5000 # new_sales/8" ++ Extn ++
                ".wav new_sales/9004.wav cc_menu_number [12]"),
            db_pbx:log(UUID, "CCFailure", HangupCause),
            {next_state, call_cc, Extn};
        "CHANNEL_EXECUTE_COMPLETE" when App =:= "play_and_get_digits" ->
            CCMenuNumber = get_var("variable_cc_menu_number", Data),
            case CCMenuNumber of
                "1" -> sendmsg(UUID, transfer, "Playcell_" ++ Extn);
                "2" -> sendmsg(UUID, transfer, "VM_" ++ Extn);
                _ -> sendmsg(UUID, transfer, "Quit")
            end,
            {next_state, final_loop, UUID};
        _ ->
            {next_state, call_cc, Extn}
    end.

call_extn({call_event, {event, [UUID | Data]} }, no_extn) ->
    Name = get_var("Event-Name", Data),
    App = get_var("Application", Data),

    error_logger:info_msg("Pid ~p: UUID ~p, Event ~p, State: call_extn",[self(), UUID, Name]),

    case Name of
        "CHANNEL_EXECUTE_COMPLETE" when App =:= "play_and_get_digits" ->
            Extn = get_var("variable_extn_number", Data),
            db_pbx:log(UUID, "CallExtn", Extn),
```

```
            gen_fsm:send_event(self(), UUID),
            {next_state, call_extn, Extn};
        _ ->
            {next_state, call_extn, no_extn}
    end;

call_extn(UUID, Extn) ->
    io:format("Calling extn: ~p~n", [Extn]),
    % sendmsg(UUID, set, "campon=true"),
    sendmsg(UUID, set, "ringback=${us-ring}"),
    sendmsg(UUID, set, "continue_on_fail=true"),
    sendmsg(UUID, set, "hangup_after_bridge=true"),
    DialString = "user/" ++ Extn,
    sendmsg(UUID, bridge, DialString),
    {next_state, call_cc, Extn}.


final_loop({call_event, {event, [UUID | Data]} }, UUID) ->
    Name = get_var("Event-Name", Data),

    error_logger:info_msg("final_loop Pid ~p: UUID ~p, Event ~p~n",[self(), UUID, Name]),

    {next_state, final_loop, UUID};
final_loop(UUID, UUID) ->
    {next_state, final_loop, UUID}.

handle_info({cc_extn, error}, State, Data) ->
    {next_state, State, Data};
handle_info({cc_extn, Extn}, State, Data) ->
    put(cc_extn, Extn),
    io:format("Found CC Extn: ~p~n", [Extn]),
    {next_state, State, Data};
handle_info(call_hangup, State, Args) ->
    io:format("Hangup ~p ~p ~n", [State, Args]),
    {stop, normal, State};
handle_info({E, {event, [UUID | Data]}} = Event, State, StateData) ->
    Name = get_var("Event-Name", Data),
    App = list_to_atom(get_var("Application", Data, "undefined")),

    error_logger:info_msg("handle_info: ~p ~p ~p ~p~n~p~n",[self(), UUID, Name, State, StateData]),

    case Name of
        %% This may be redundant; `call_hangup` is already handled above
        %% (Is there any other reason to listen to the event as well?)
        "CHANNEL_HANGUP_COMPLETE"->
            db_pbx:hangup(UUID, Data),
            % {next_state, final_loop, UUID};
            {stop, normal, UUID};
        "CUSTOM" ->
            SubClass = get_var("Event-Subclass", Data),
            Action = get_var("FIFO-Action", Data),

            io:format("Fifo: ~p ~p~n", [SubClass, Action]),

            case SubClass of
                "fifo::info" when Action =:= "bridge-caller" ->
                    db_pbx:process(UUID, Data);
                _ -> ok
            end,
            {next_state, State, StateData};
        _ ->
            List = [welcome, welcome_wait_playback, main_menu_wait_dtmf, call_cc, call_extn],
            case lists:any(fun(Elem) -> Elem =:= State end, List) of
                true ->
                    gen_fsm:send_event(self(), Event);
                _ -> ok
            end,
            {next_state, State, StateData}
    end;
handle_info(Info, State, Data) ->
    io:format("Got Other Info: ~p ~p ~p ~n", [Info, State, Data]).
```

```
handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.
terminate(normal, _StateName, _StateData) ->
    io:format("Stop with reason: normal ~p ~p~n", [_StateName, _StateData]),
    ok;
terminate(Reason, _StateName, _StateData) ->
    io:format("Stop with reason: ~p ~p ~p~n", [Reason, _StateName, _StateData]),
    ok.


%% private

route_time_condition(UUID) ->
    case db_pbx:get_time_condition("sales_icall") of
        {Action, Args} ->
            case Action of
                % "idp_acd:" ++ ErlAction ->
                %   Fun = list_to_atom(ErlAction),
                %   db_pbx:log(UUID, "TimeCondition", ErlAction ++ " " ++ Args),
                %   ?MODULE:Fun(UUID, Args);
                Action ->
                    db_pbx:log(UUID, "TimeCondition", Action ++ " " ++ Args),
                    sendmsg(UUID, list_to_atom(Action), Args),
                    final_loop
            end;
        _ ->
            {Date, {Hour, Min, _Sec}} = erlang:localtime(),
            Weekday = calendar:day_of_the_week(Date),
            route_work_time(UUID, Weekday, Hour, Min)
    end.

route_work_time(UUID, Weekday, Hour, Min)
    when Weekday > 5 andalso Hour > 10 andalso (Hour < 20 orelse ( Min < 30 andalso Hour < 21) ) ->
    db_pbx:log(UUID, "Weekend", "10:00 - 20:30"),
    main_menu;
route_work_time(UUID, Weekday, Hour, Min) when Hour > 9 andalso (Hour < 20 orelse (Min < 30 andalso Hour < 21
))->
    db_pbx:log(UUID, "Workday", "Weekend 9:00 - 20:30"),
    main_menu;
route_work_time(UUID, _Weekday, Hour, Min) when (Hour > 21 orelse (Hour > 20 andalso Min > 30)) andalso Hour
< 23 ->
    db_pbx:log(UUID, "Time", "20:30 - 23:00"),
    call_hst;
route_work_time(UUID, _Weekday, _Hour, _Min) ->
    db_pbx:log(UUID, "NonWorktime", "Cellphone"),
    call_cellphone.

transfer(UUID, Dest) ->
    transfer(UUID, Dest, "XML", "new_sales").
transfer(UUID, Dest, Dialplan, Context) ->
    sendmsg(UUID, transfer, Dest ++ " " ++ Dialplan ++ " " ++ Context).

play_intransfer(UUID) ->
    sendmsg(UUID, playback, "new_sales/1002.wav"),
    timer:sleep(3000).

fetch_cc_extn_from_crm(Pid, CallerID) ->
    Extn = case helpers:http_fetch(?CRM_APP, "/voip/cdrs?caller_id=" ++ CallerID) of
        {error, _} -> error;
        Number -> Number
    end,

     Pid ! {cc_extn, Extn}.
```

Other discussion

1) Erlang is completely async, so you have to wait for the channel_execute_complete event to decide if an application(for example playback and bridge) is done before you send other messages which is not as convenient as in dialplan or Lua and other languages. Or you could do a timer:sleep() treak on playback if you know the length of the recording.

2) We transfer to XML dialplan as soon as we don't need erlang features, edit XML is easier in some sense.

3) mod_fifo doesn't work well in Erlang unless you transfer to orbit extensions since the channel is parked(controlled in erlang) and when fifo bridge to an extension their are no sound, I didn't find a way to unpark a channel, let me know if I'm wrong.

4) Code is kind of clear, however, it would be nicer to make a gen_fs_fsm behavior to wrap all the FreeSWITCH message processes :).

`freeswitch_msg.erl`

Seven Du 's code above refers to modules `db_pbx` and `freeswitch_msg.erl`; the former is not available anywhere, and the latter has been (I assumed) reverse engineered by Belaid Areski (many thanks!). From his Github repo:

freeswitch_msg.erl

```
-module(freeswitch_msg).
-compile([export_all]).

%[get_var/2, get_var/3, sendmsg/3]

-define(FS_NODE, 'freeswitch@newbalance').

send_msg(UUID, App, Args) ->
    Headers = [{"call-command", "execute"},
        {"execute-app-name", atom_to_list(App)}, {"execute-app-arg", Args}],
        send_msg(UUID, Headers).

send_lock_msg(UUID, App, Args) ->
    Headers = [{"call-command", "execute"}, {"event-lock", "true"},
        {"execute-app-name", atom_to_list(App)}, {"execute-app-arg", Args}],
        send_msg(UUID, Headers).

send_msg(UUID, Headers) -> {sendmsg, ?FS_NODE} ! {sendmsg, UUID, Headers}.

%sendmsg(UUID, playback, "new_sales/1000.wav"),

sendmsg(UUID, App, Args) ->
    send_msg(UUID, App, Args).


%get_var("Application", Data)
%CallerID = get_var("Channel-Caller-ID-Number", Data, "00000000"),
get_var(Header, Event) ->
    %proplists:get_value(<<Header>>, Event).
    1.

get_var(Header, Event, Default) ->
    proplists:get_value(<<Header>>, Event, Default).
```