A generic, lightweight finite state machine implementation in Erlang

Dmitry Kolesnikov (dmkolesnikov@gmail.com)
Mario Cardona (marioxcardona@gmail.com)

April 30, 2013

Chapter 1

XXX

1.1 Rationale

The library proposes simple *behaviour* for finite state machine (FSM) implementation. The major objective is eliminate difference between synchronous, asynchronous and out-of-bound messages processing. The *behaviour* is proposed as an alternative to built gen_fsm.

1.2 Definitions

client process

is an Erlang process that consumes FMS services by sending *synchronous*, *asynchronous* or *out-of-bound* messages.

finite state machine

is an Erlang process that executes Mealy-like FSM, which is defined as the input alphabet Σ (all acceptable messages), finite and non-empty set of all possible states S and state transition function δ (FSM output is performed at state transition).

$$\delta : x, s_i \longrightarrow s_j \quad x \in \Sigma, s \in S$$

message

is an Erlang tuple used for communication between client process and FSM.

1.3 Message passing

The library distinguishes following message passing scenarios used for interprocess communication.

synchronous (aka call)

the execution of message originator is blocked until state machine output successful/unsuccessful result message. The originator process waits for response or *timeout*.

asynchronous (aka cast)

the execution of message originator is not blocked, the state machine output message is delivered to mailbox of originator process. The asynchronous message implies that originator process waits for successful/unsuccessful responses. The scenario allows to execute multiple parallel service calls towards FSM (e.g. asynchronous external I/O, "long-lasting" requests with external worker pool).

out-of-bound (aka send)

the execution of message originator is not blocked, unlike an asynchronous message the originator processes do not have intent to to track results of service execution (fire-and-forget mode).

1.4 Interface specification

1.4.1 Data-types signatures

fsm

unique instance of state machine identity: *pid*|*atom*|*any*.

ref

unique request reference (corresponds to Erlang built-in reference type).

sid

atom name of state transition function

state

any state machine internal data structure

1.4.2 Messaging operation

The chapter below highlights the signatures of message passing interface at the library.

call

the call operation sends *any* message to the state machine and waits for either *any* successful or *any* unsuccessful response

$$call : fsm, any \longrightarrow \{'ok', any\}|\{'error', any\}$$

cast

the cast operation sends *any* message to the state machine and immediately returns unique reference. The reference allows to asynchronously "wait" for either *any* successful or *any* unsuccessful response (e.g. response message signature is $\{'ok'|'error', ref, any\}$). reference generated automatically.

$$cast : fsm, any \longrightarrow ref$$

send

the send operation does not differs from Erlang builtin ! operator or erlang:send function call.

$$send : fsm, any \longrightarrow ok$$

1.4.3 FSM operation

The chapter below highlights the signatures of state machine interface at the library.

init

the function init state machine, builds internal state data *state* and defines initial state (transition function) *sid*. The function is called whenever state machine is started using kfsm:start_link(...)

$$init : [any] \longrightarrow \{ok, sid, state\}|\{'error', reason\}$$

free

the function release state machine, it is called when FSM is about to terminate.

$$free : reason, state \longrightarrow ok$$

state transition

the state transition function receive *any* message, which is sent using message passing interface. The function executes the state transition and generates output. There should be one instance of the state transition function δ for each possible state name. This function performs update of internal state data, continues FSM to next state or terminates execution with a reason. If the input message are sent using *call* or *cast* primitives and state machine transition function do not generate any output then origin of input message is queued in FIFO manner for any further usage.

$$\begin{aligned} \delta : any, state \longrightarrow \quad & \{'next\_state', sid, state, [timeout|hibernate]\} \\ & \{'reply', any, sid, state, [timeout|hibernate]\} \\ & \{'stop', [any], reason, state\} \end{aligned}$$

$\{'next\_state', sid, state, [timeout|hibernate]\}$: continue execute next state *sid* with updated internal state data *state*. No output is generated to external processes by state machine. If an optional integer *timeout* value is provided, a $'timeout'$ event will occur unless any input is received by FSM within given time interval. If hibernate is specified instead of a timeout value, the process will go into hibernation when waiting for the next message to arrive.

$\{'reply', any, sid, state, [timeout|hibernate]\}$: continue execute next state *sid* with updated internal state data *state*. The message *any* is sent out to origin process of input message. If the origin is not know (e.g. pure sent or Erlang built-in functions where used) then previously queued request origins are used. If the queue is empty then the message *any* is silently dropped. If an optional integer *timeout* value is provided, a *timeout* event will occur unless any input is received by FSM within given time interval. If hibernate is specified instead of a timeout value, the process will go into hibernation when waiting for the next message to arrive.

TODO: do we need a concept of subscribe here so that if origin to reply is not known then subscribed process got messages (type-of-fallback scenario).

$\{'stop', [any], reason, state\}$: terminate state machine execution with *reason*. Note that for any other reason than $'normal'|'shutdown'$ FSM is assumed to terminate due to an error and an error report is issued. If state machine outputs optional *any* messages the it is replied to client process using same logic as previous reply scenario. Any outstanding synchronous calls are terminated with no_proc exception.

1.4.4 OTP compatibility

gen_server

gen_server:call is translated to synchronous message passing, gen_server:cast is translated to asynchronous send.

gen_fsm

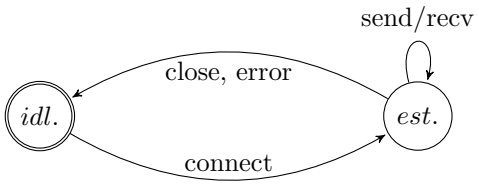
no support.

Chapter 2

Appendix A: Example

2.1 tcp/ip client

the state machine on the diagram below show a potential state machine for tcp/ip communication using socket api.



client interface

```
-module(tcp).

%%
%% start new tcp/ip state machine
-spec(start_link/0 :: () -> {ok, pid()} | {error, any()}).

start_link() ->
   kfsm:start_link(?MODULE, []).

%%
%% synchronous / asynchronous connect
-spec(connect/2  :: (pid(), peer()) -> {ok, peer()} | {error, any()}).
-spec(connect_/2 :: (pid(), peer()) -> reference()).

connect(Pid, Peer) ->
```

```
   % call is blocked until socket is connected
   kfsm:call(Pid, {connect, Peer}).

connect_(Pid, Peer) ->
   % call is released immediately the message
   % {xxx, xxx, {ok, Peer}} is delivered to mailbox
   % when socket is connected
   kfsm:cast(Pid, {connect, Peer}).

%%
%% synchronous / asynchronous message send
-spec(send/2  :: (pid(), binary()) -> ok | {error, any()}).
-spec(send_/2 :: (pid(), binary()) -> reference()).

send(Pid, Msg) ->
   kfsm:call(Pid, {send, Msg}).

send_(Pid, Msg) ->
   % call is released immediately the message
   % {xxx, xxx, ok} is delivered to mailbox
   % when socket is connected
   kfsm:cast(Pid, {send, Msg}).

%%
%% synchronous / asynchronous message recv
-spec(recv/1  :: (pid()) -> {ok, binary()} | {error, any()}).
-spec(recv_/1 :: (pid()) -> reference()).

send(Pid, Msg) ->
   kfsm:call(Pid, recv).

send_(Pid, Msg) ->
   % call is released immediately the message
   % {xxx, xxx, {ok, binary()}} is delivered to mailbox
   % when socket is connected
   kfsm:cast(Pid, recv).
```

tcp fsm

```
-module(tcp).

...

'IDLE'({connect, Peer}, S) ->
   {ok, Sock} = gen_tcp:connect(...),
   {reply, {ok, Peer}, 'ESTABLISHED', S#tcp{sock=Sock}};
```

```
...

'ESTABLISHED'({send, Msg}, S) ->
   Result = gen_tcp:send(S#tcp.sock, Msg),
   {reply, Result, 'ESTABLISHED'};

...

'ESTABLISHED'({recv, Msg}, S) ->
   Result = gen_tcp:recv(S#tcp.sock, 0),
   {reply, Result, 'ESTABLISHED'};
```

2.2 basic client-server

```
-module(server).

-export([
   start_link/0, ping/1,
   init/1, free/2, handle/2,
]).

start_link() ->
   kfsm:start_link({local, ?MODULE}, ?MODULE, []).

init(_) ->
   {ok, handle, undefined}.

free(_, _) ->
   ok.

ping(Pid) ->
   kfsm:call(Pid, ping).

handle(ping, S) ->
   {reply, pong, handle, S}.
```