www.erlang-solutions.com

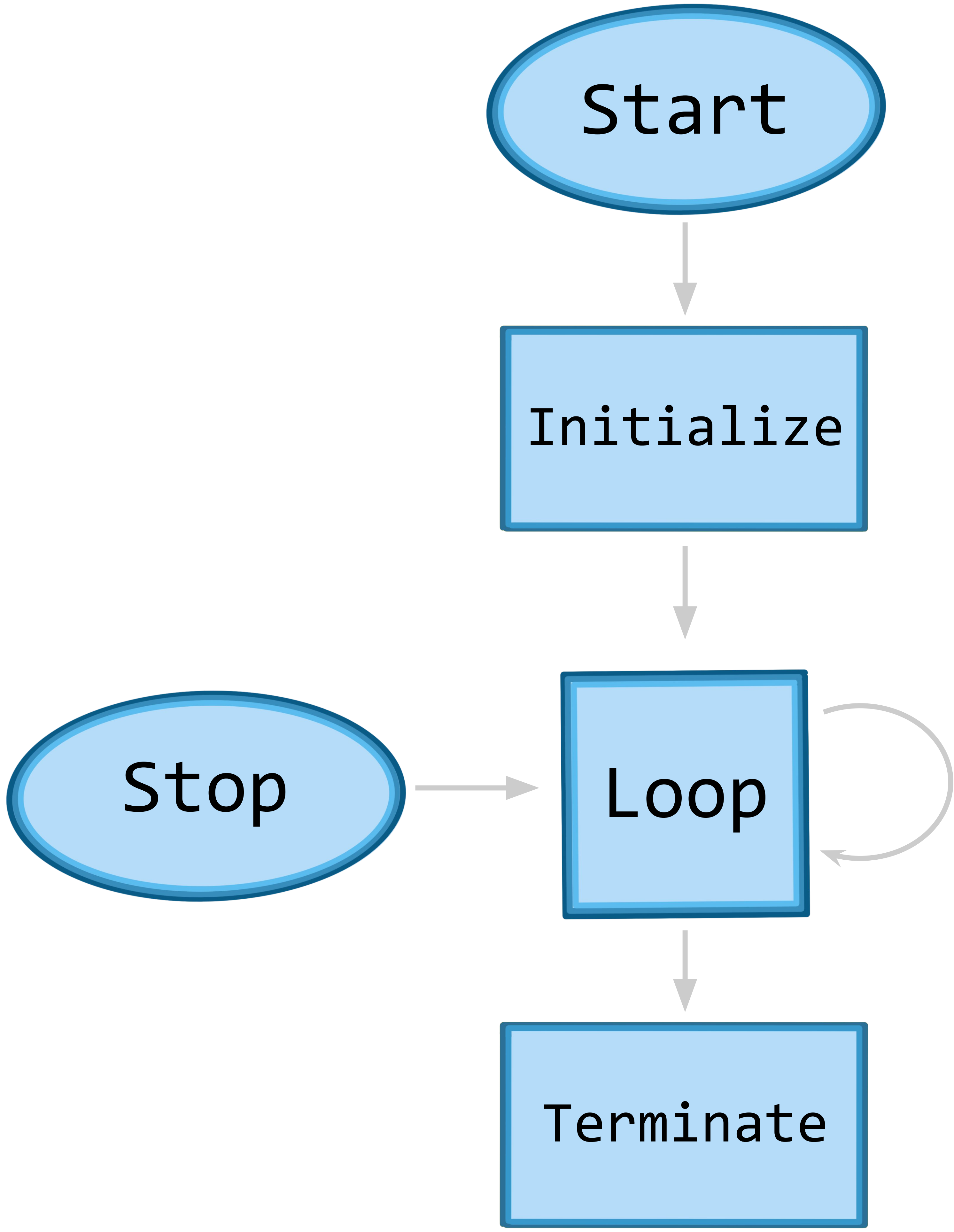
Process Design Patterns

Process Design Patterns

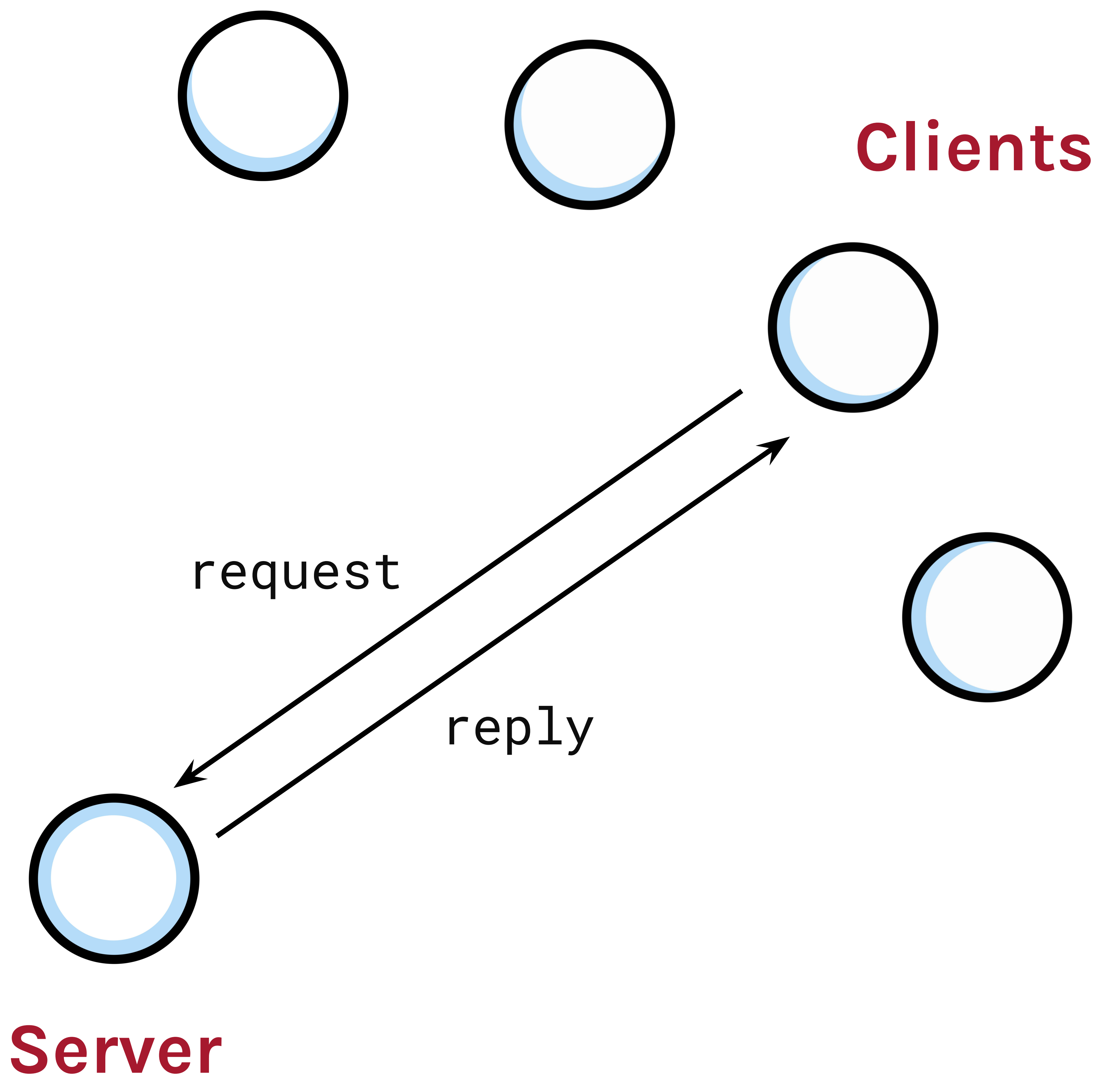
- ▶ Client Server Models
- ▶ A Server Example
- ▶ Finite State Machines
- ▶ Event Managers
- ▶ Supervisors

Client Server Models: process skeleton

```
start(Args) ->
   spawn(server, init, [Args])

init(Args) ->
  State = initialize_state(Args),
  loop(State).

loop(State) ->
   receive
      {handle, Msg} ->
         NewState = handle(Msg, State),
         loop(NewState);
      stop -> terminate(State)
   end.

terminate(State) -> clean_up(State).
```

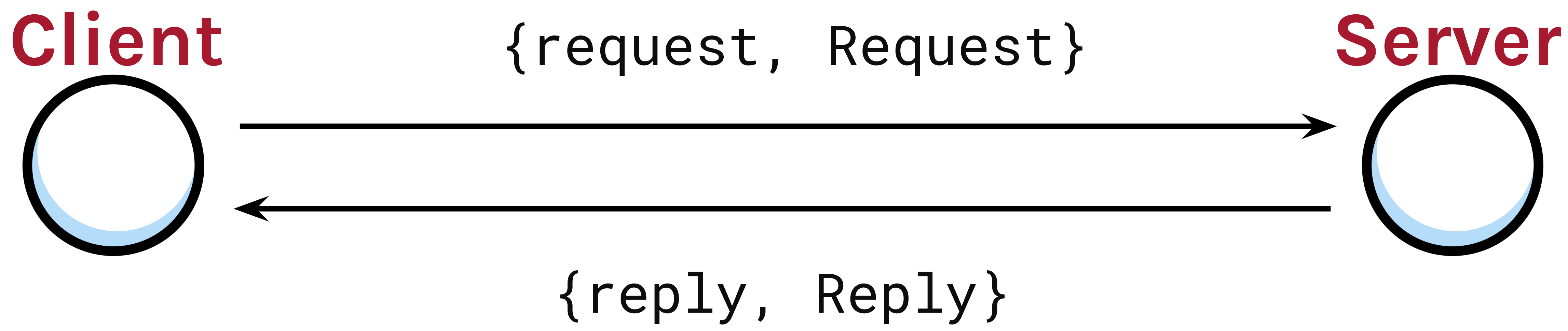


Client Server Models



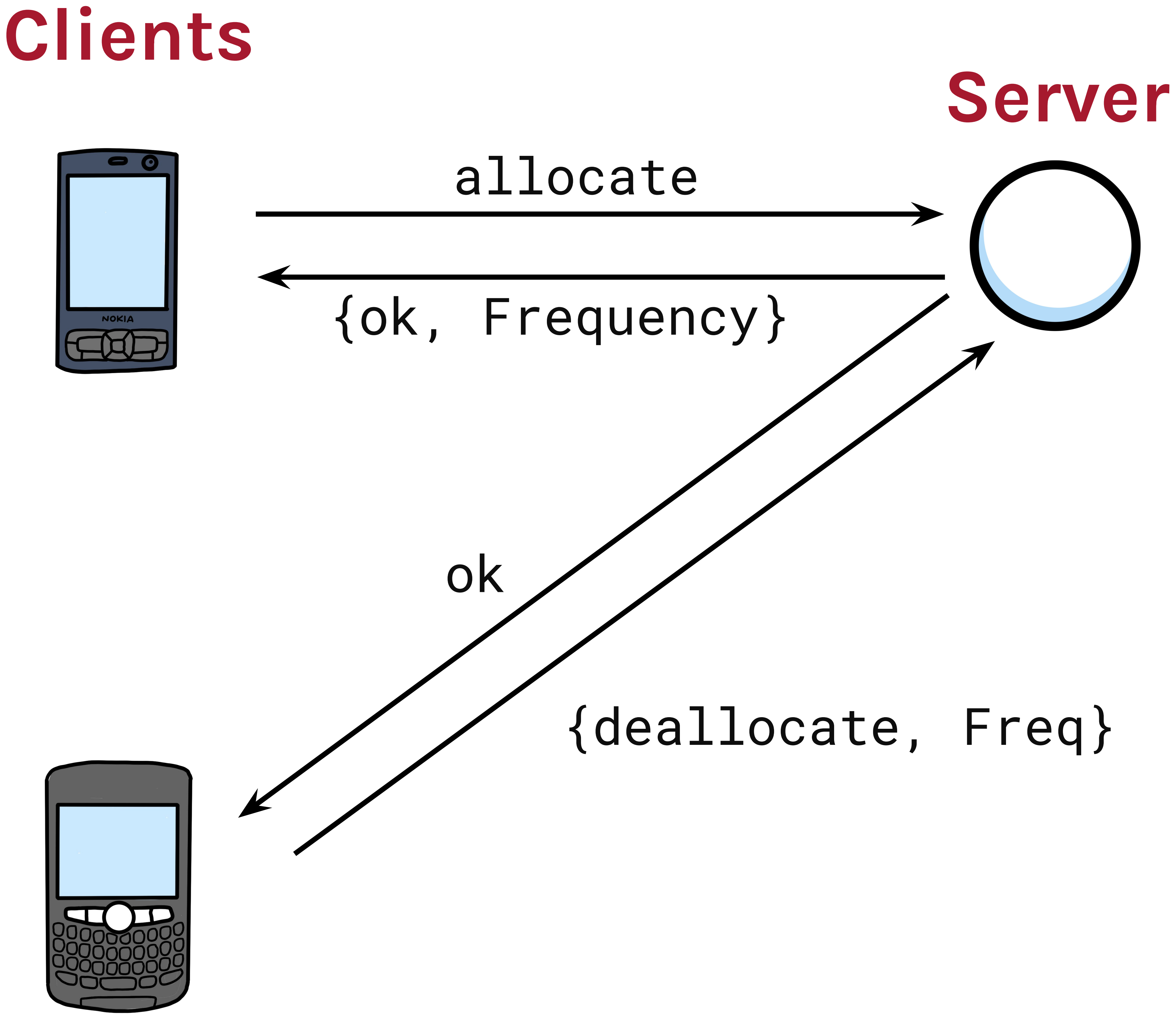
- Processes can be used to implement client server solutions
- A server is usually responsible for providing a service or handling a resource
- Clients are the processes which use these resources

Client Server Models



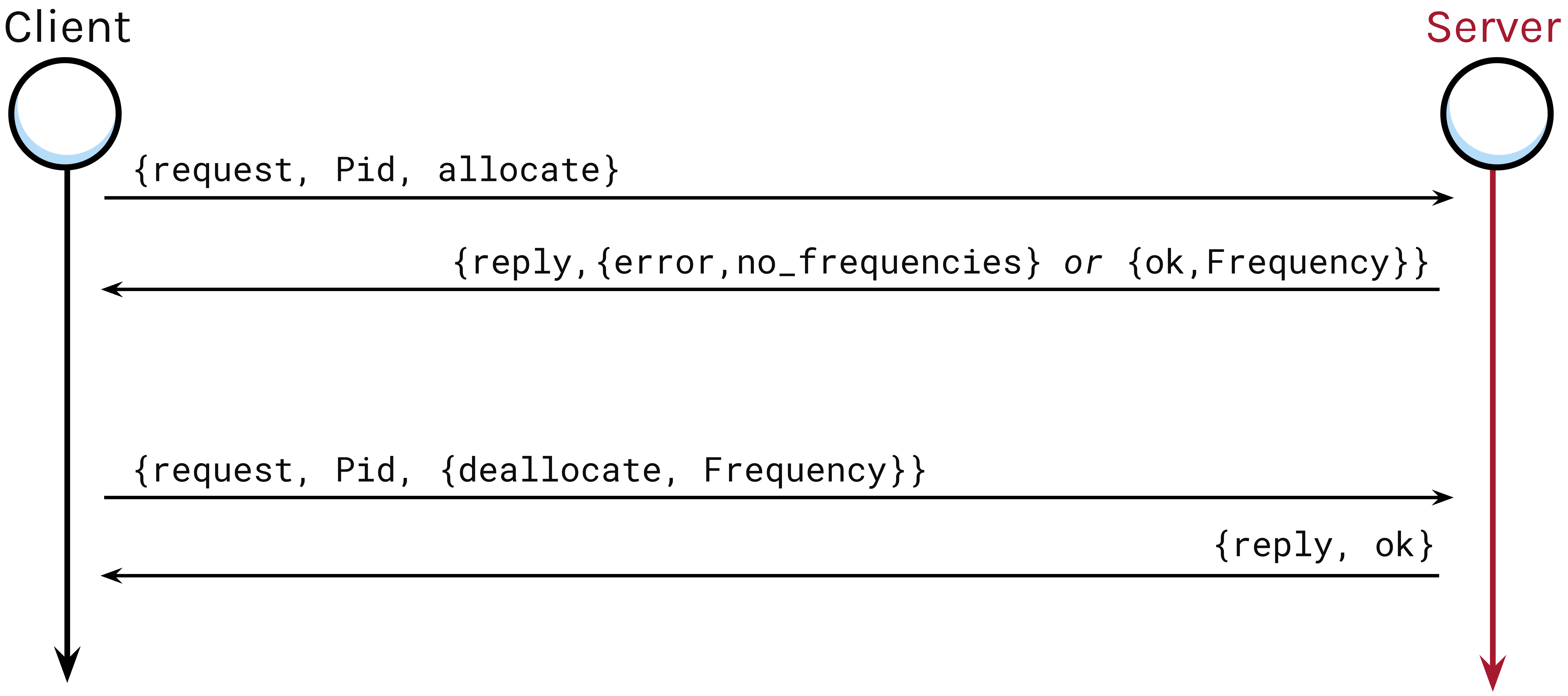
- Clients make requests to the server through message passing
- Message passing is often hidden in functional interfaces
- If the client using the service needs a reply to the request, the call to the server has to be **synchronous**
- If the client does not need a reply, the call to the server can be **asynchronous**

A Server Example



- The following server is responsible for allocating and deallocating frequencies on behalf of mobile phones

A Server Example


Client
Server
{request, Pid, allocate}
{reply,{error,no_frequencies} *or* {ok,Frequency}}
{request, Pid, {deallocate, Frequency}}
{reply, ok}

A Server Example

```
-module(frequency).

-export([start/0, stop/0,  allocate/0, deallocate/1]).
-export([init/0]).

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

get_frequencies() -> [10,11,12,13,14,15].
```

A Server Example

```erlang
stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message protocol in
%% functional interfaces.

call(Message) ->
    frequency ! {request, self(), Message},
    receive
       {reply, Reply} -> Reply
    end.

reply(Pid, Message) ->
    Pid ! {reply, Message}.
```

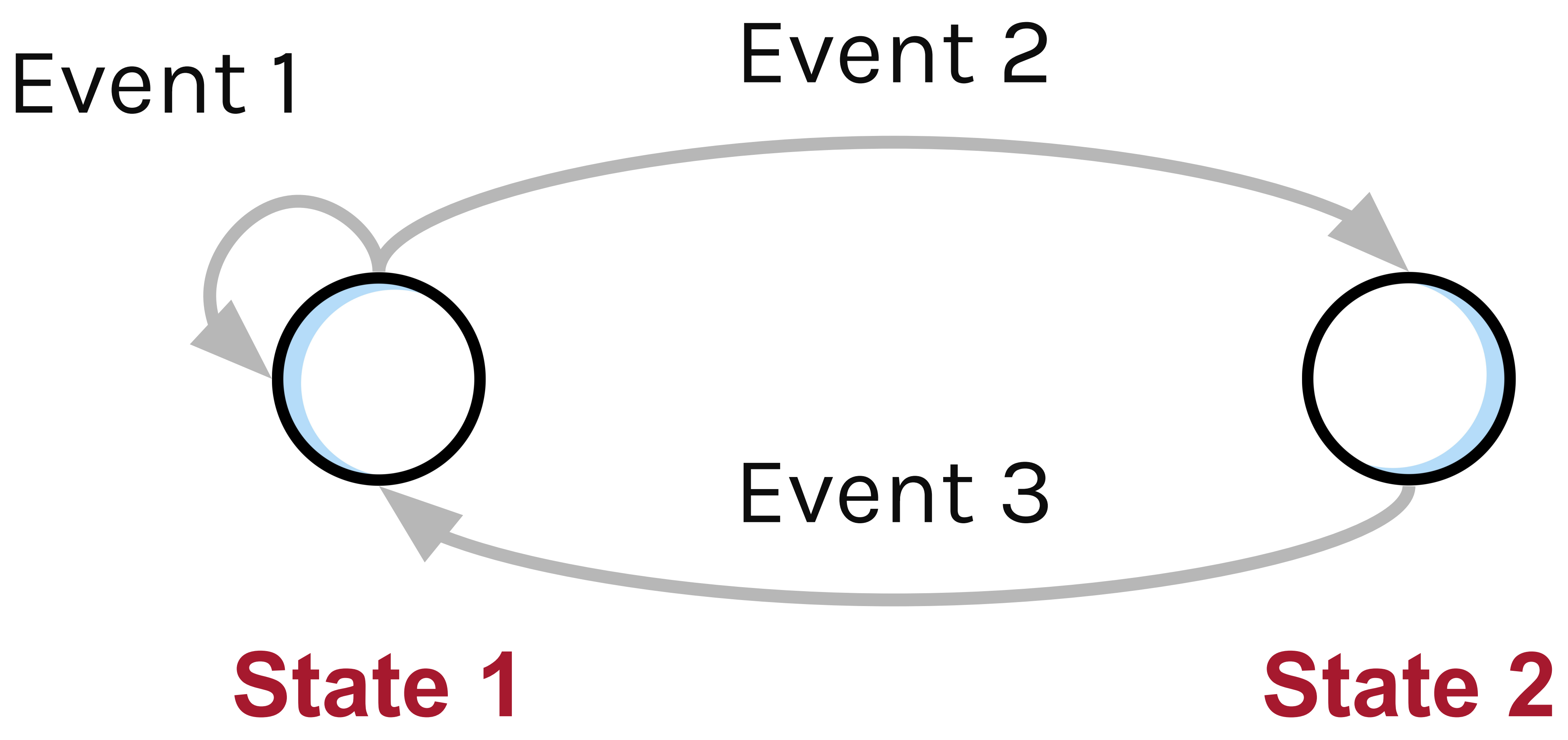
A Server Example

```
%% The main server loop.

loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
       {NewFrequencies,Reply} = allocate(Frequencies,Pid),
       reply(Pid, Reply),
       loop(NewFrequencies);
    {request, Pid , {deallocate, Freq}} ->
       NewFrequencies = deallocate(Frequencies, Freq),
       reply(Pid, ok),
       loop(NewFrequencies);
    {request, Pid, stop} ->
       reply(Pid, ok)
    end.
```

A Server Example

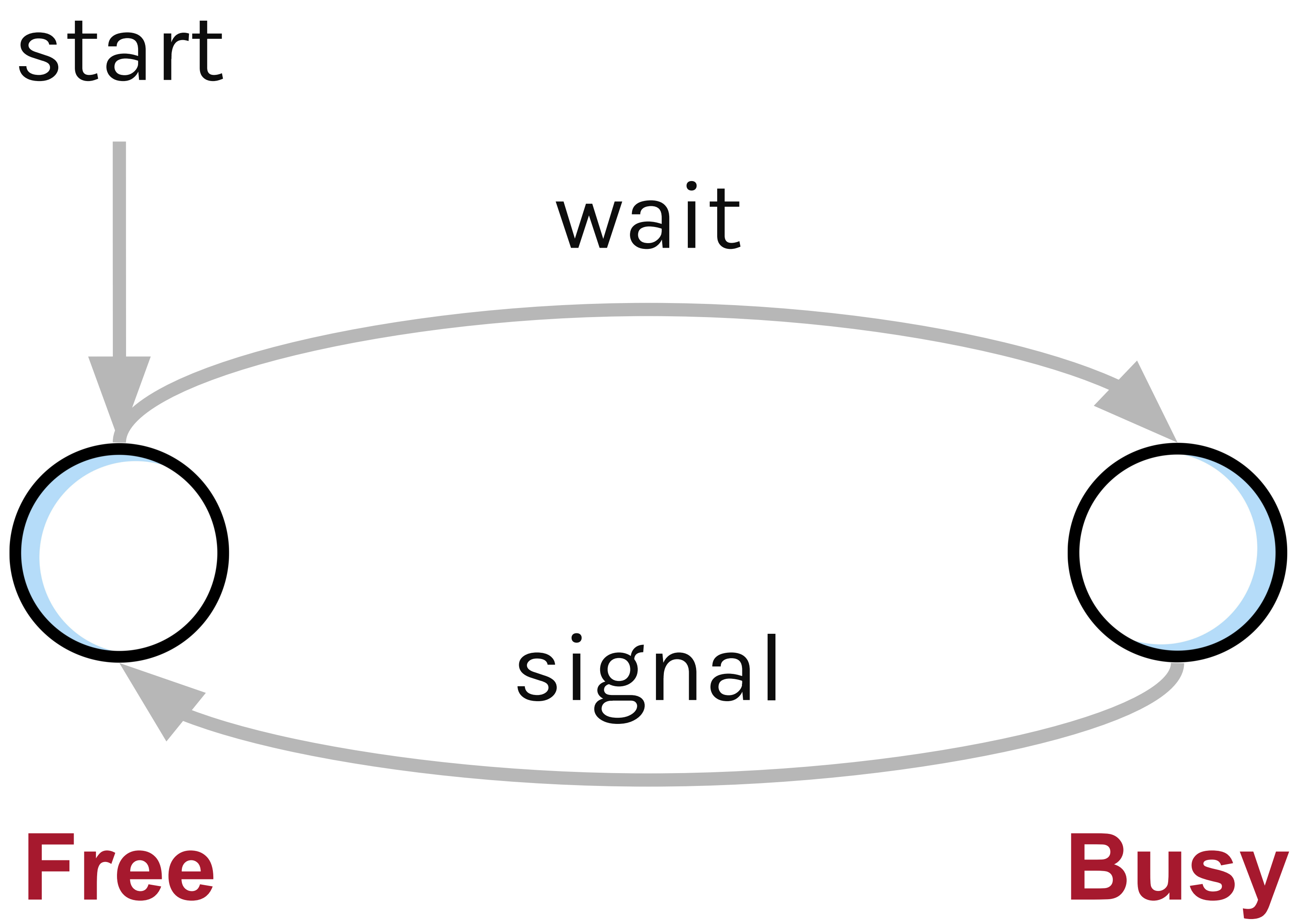
```
%% The Internal Functions
%% Functions used to allocate and deallocate frequencies.

allocate({[], Allocated}, Pid) ->
    {{[], Allocated}, {error, no_frequency}};
allocate({[Freq|Free], Allocated}, Pid) ->
    {{Free, [{Freq, Pid}|Allocated]}, {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    NewAllocated = lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free],  NewAllocated}.
```

Finite State Machines



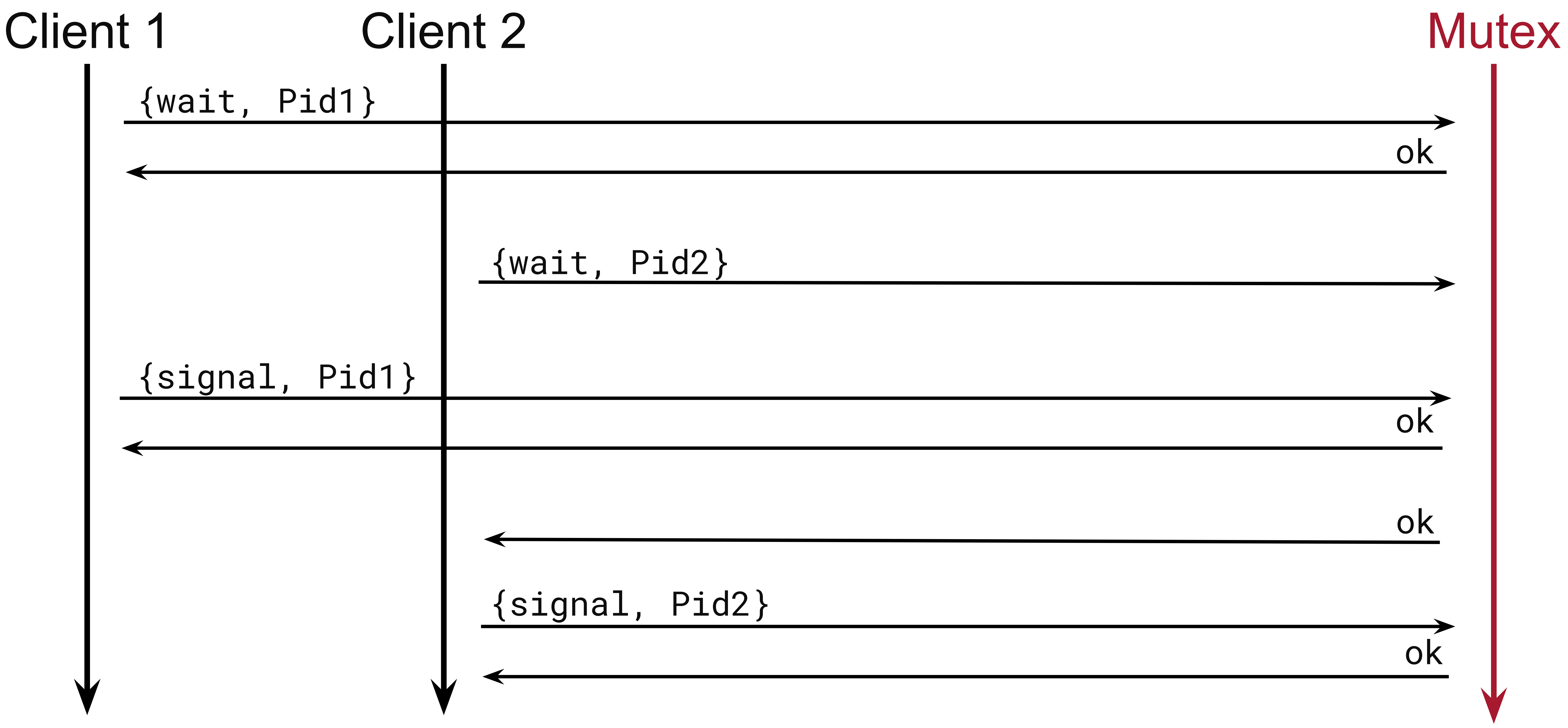
- Processes can be used to implement finite state machines
- Each state is represented as a tail recursive function
- Each event is represented as an incoming message
- Each state transition is achieved by calling the function denoting the new state

A State Machine Example



- ▶ A mutex is a program that allows multiple processes to share the same resource
- ▶ It has two states, **Free** and **Busy**
- ▶ It has two events, **wait** and **signal**
- ▶ When started it transitions to state **Free**

A Mutex Example



A State Machine Example

```
-module(mutex).

-export([start/0, stop/0]).
-export([wait/0, signal/0]).
-export([init/0]).

start() ->
    register(mutex, spawn(?MODULE, init, [])).

stop() ->
    mutex ! stop.

init() ->
    free().
```

A State Machine Example

```
wait() ->
    call(wait).

signal() ->
    call(signal).

%% We hide all message passing and the message protocol in
%% functional interfaces.

call(Message) ->
    mutex ! {Message, self()},
    receive
       {reply, Reply} -> Reply
    end.

reply(Pid, Message) ->
    Pid ! {reply, Message}.
```

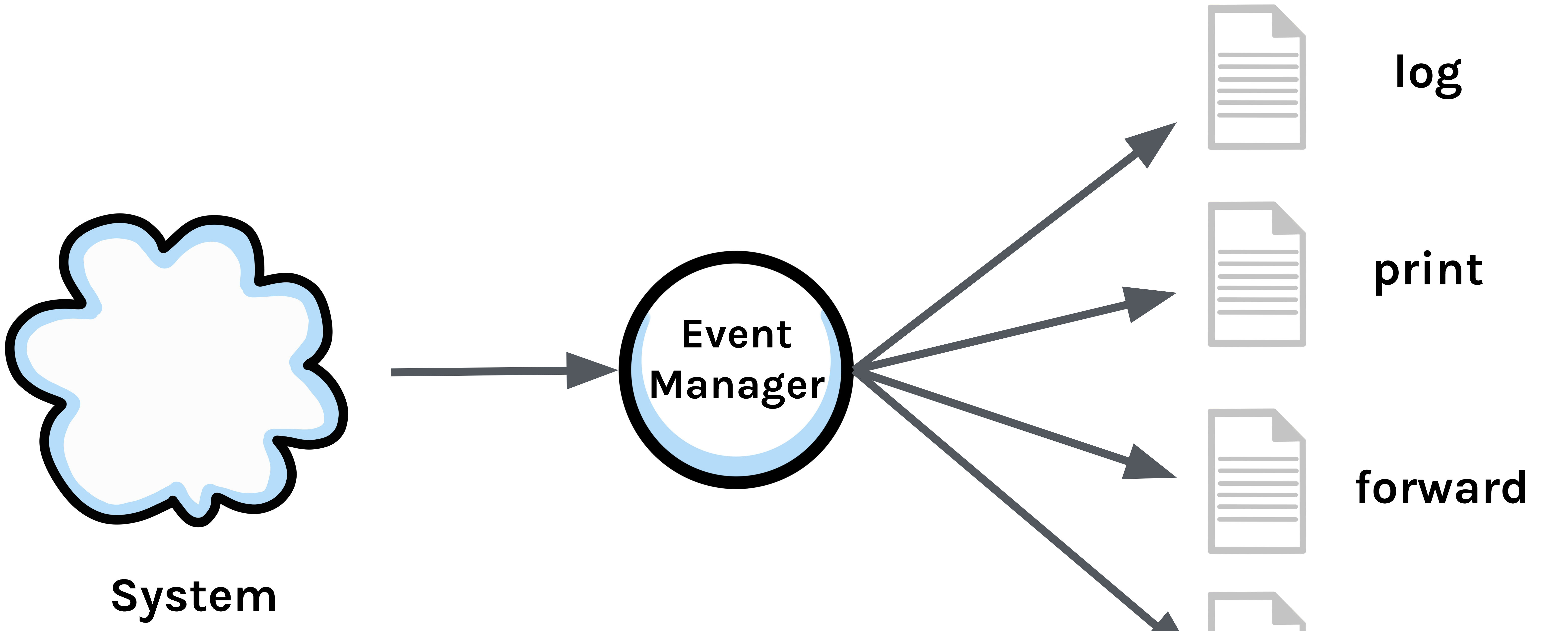
A State Machine Example

```erlang
%% The state functions.

free() ->
    receive
        {wait, Pid} ->
            reply(Pid, ok),
            busy(Pid)
    end.

busy(Pid) ->
    receive
        {signal, Pid} ->
            reply(Pid, ok),
            free()
    end.
```

Event Managers and Handlers

- Processes can be used to implement event managers
- A manager will receive a specific type of event, e.g.
  - Alarms
  - State Changes
  - Commands
  - Errors
- When an event is received, one or more operations are applied on the event
- Some or all of the operations can be enabled and disabled during run time

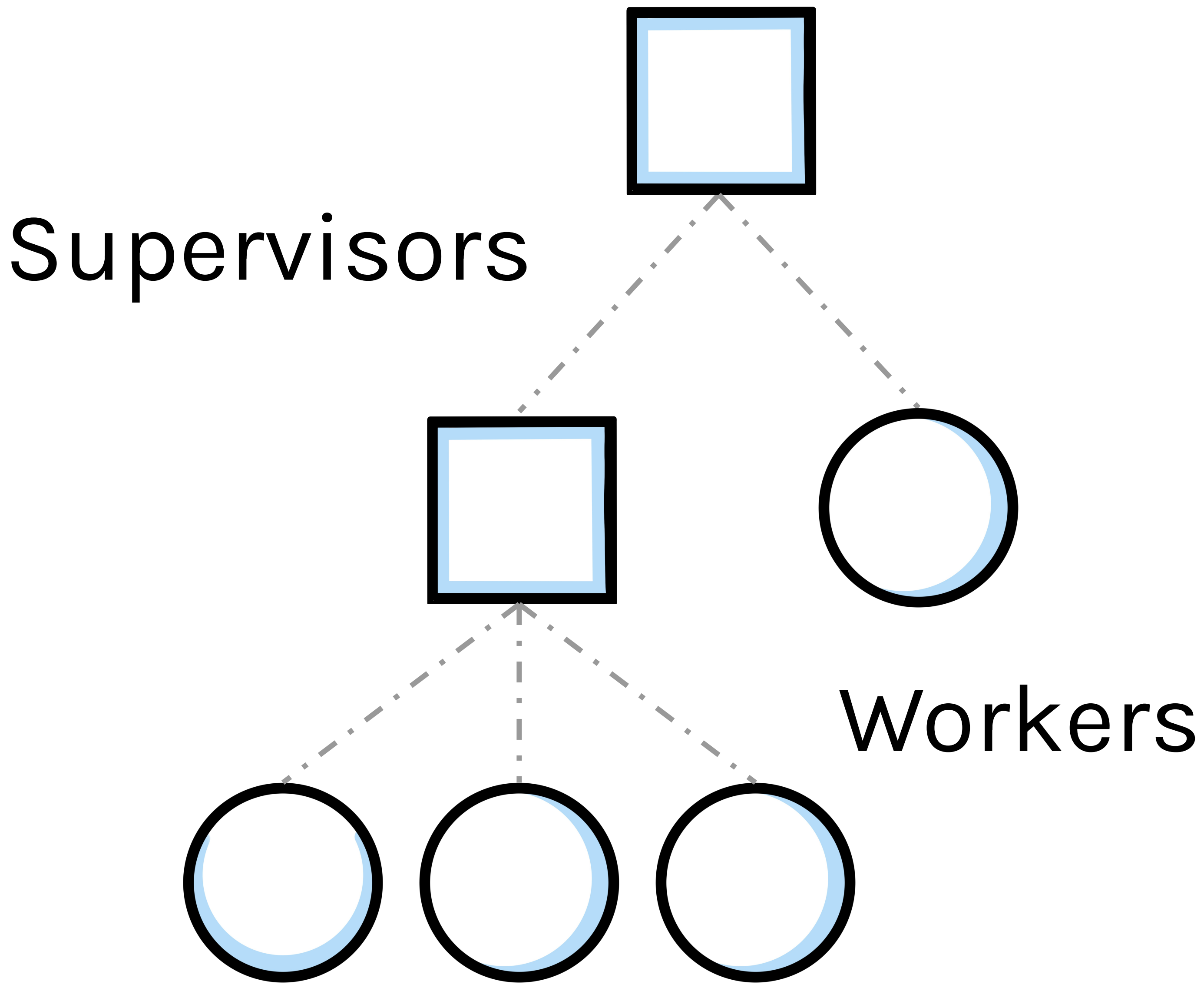
Event Managers: example



- ➢ Alarm managers are implemented as event managers with handlers

Supervisors

- Supervisors are processes whose only task is to start, monitor, and manage children.
- Child processes are either
  - Workers
  - Supervisors
- Supervisors will monitor their children
- Supervisors can restart the children when they terminate



Process Design Patterns

- ▶ Client Server Models
- ▶ Finite State Machines
- ▶ Event Managers
- ▶ Supervisors