The Erlang Multi User Dungeon

--

Lars Hesel Christensen
@larshesel

Zurich Erlang User Group
@ZurichErlangUG

This talk

- Me.
- Motivation.
- What's a mud, why a mud?
- Java thoughts/perspective.
- Erlang design and thoughts.
- The thorns using Erlang.
- Future?

self()

Previously on self():
- C, C++ (telecom, embedded, industry.)
- .Net (cat modelling, reinsurance.)

now():
- Work at Trifork GmbH (http://www.trifork.ch.)
- Android, Java (banking, industry.)

Future:
- Erlang?

Motivation

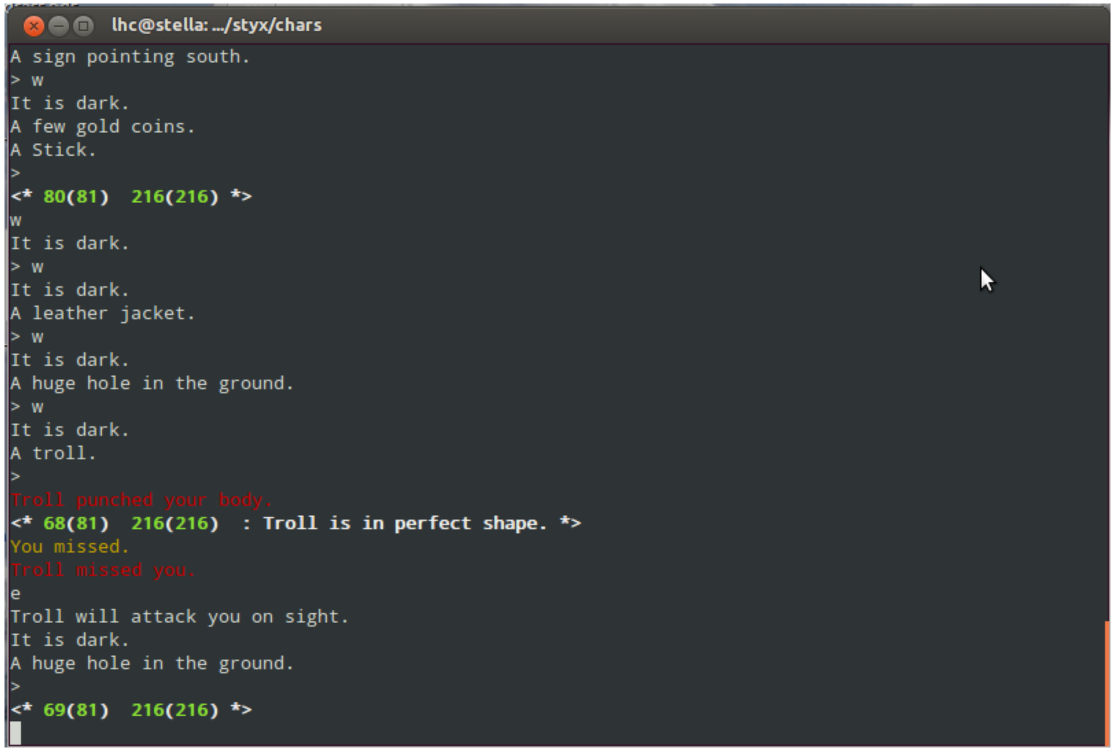
- Build something (larger) in Erlang.
- Something inherently concurrent.
- Something non-trivial (architecture.)
- With behaviors (supervisors and gen_servers.)
- With eunit.
- With specs* (type annotations.)
- Use dialyzer* (static analysis.)

*not really a plan, turned out to be necessary.

What's a MUD?

- Multi User Dungeon
- text based
- go north;
  go east;
  pick up staff;
  kill dragon

- WoW anno
  1992.



```
A sign pointing south.
> w
It is dark.
A few gold coins.
A Stick.
>
<* 80(81)  216(216) *>
w
It is dark.
> w
It is dark.
A leather jacket.
> w
It is dark.
A huge hole in the ground.
> w
It is dark.
A troll.
>
Troll punched your body.
<* 68(81)  216(216)  : Troll is in perfect shape. *>
You missed.
Troll missed you.
e
Troll will attack you on sight.
It is dark.
A huge hole in the ground.
>
<* 69(81)  216(216) *>
```

The styx MUD (styx.dk:3000)

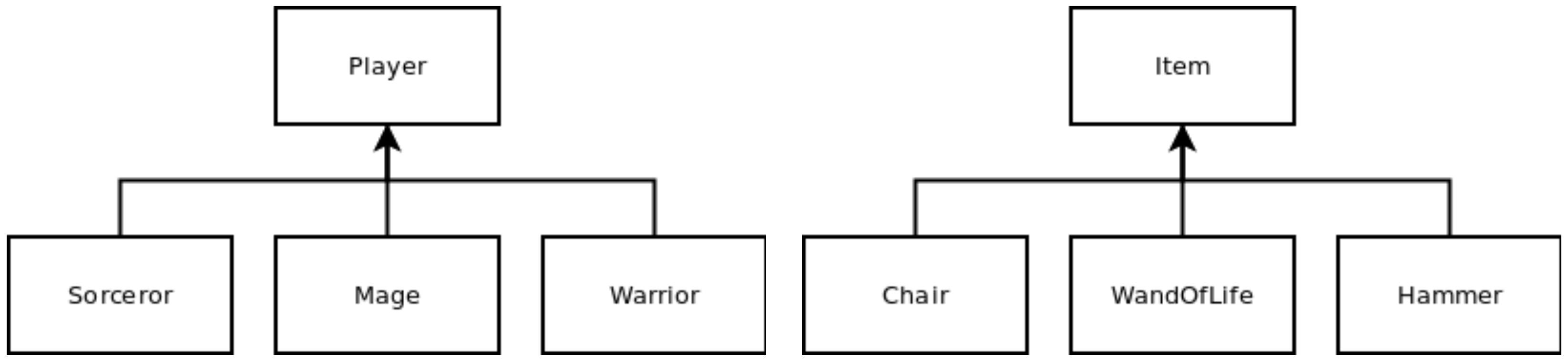
Why a MUD?

- It is inherently concurrent.
- Lots of interaction.
- It has a simple interface (text based.)
- Doesn't need complicated frameworks.
- Tons of possibilities for new stuff (see next episode.)
- It caters to creativity (building worlds/quests.)
- It is fun! Brings out the inner child.

Basic entities

- Players:
  The people who play the game.
- Rooms:
  Form a map/world.
- Items:
  Weapons, armor, beer, ...
- Monsters/AIs:
  Things you can fight/kill.

Java thoughts



- Seems easy to create a basic model.

Java thoughts

- But what about concurrency?
    - A thread pr. player?
    - Locking on mutable state?
        {items, rooms, other players}.
- Consumer/producer queues.

- How to add concurrency seems non-trivial.

Erlang

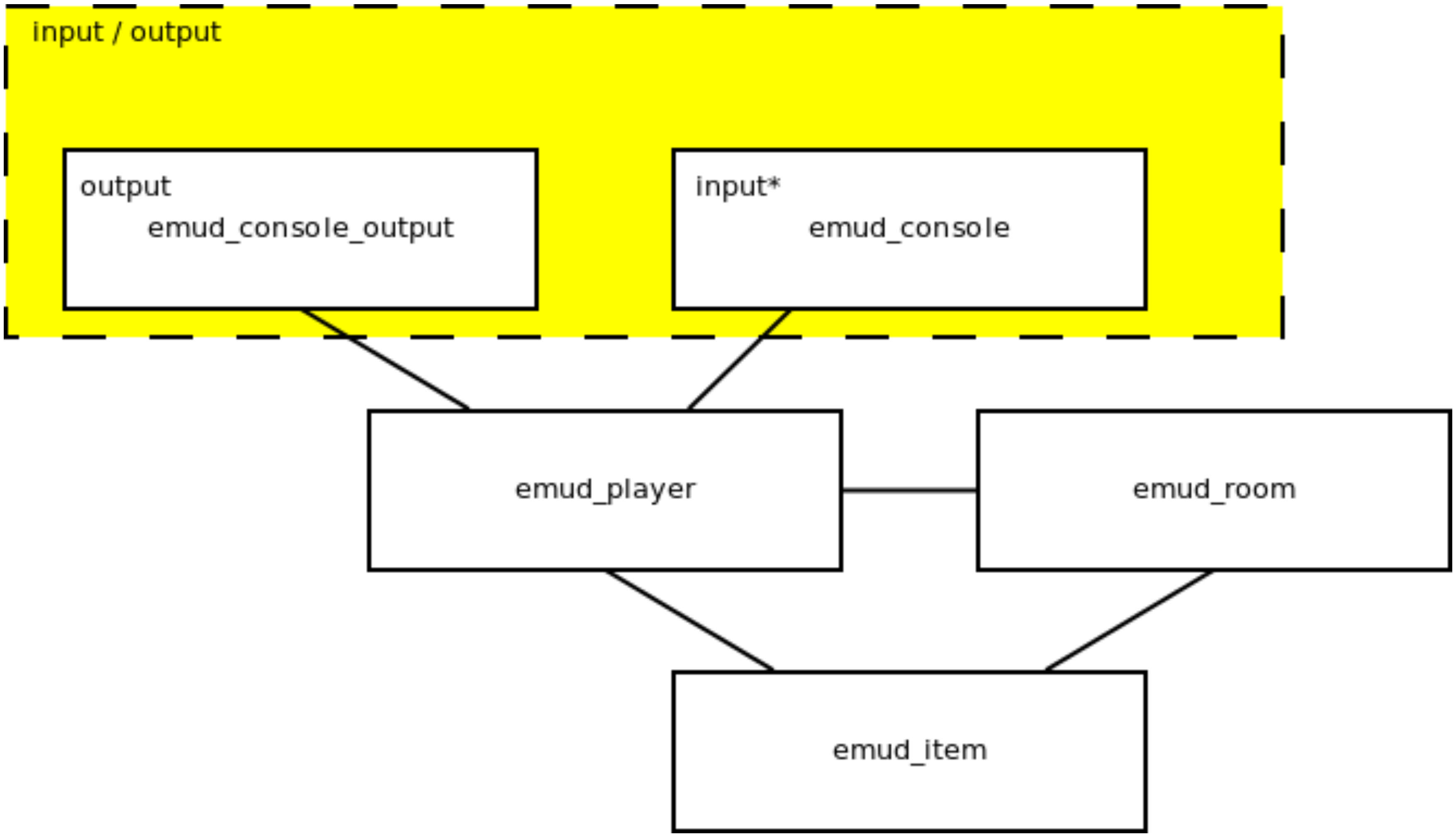
- Initial idea: everything is a gen_server
  players, items, rooms, AIs...

- Processes encapsulate state.

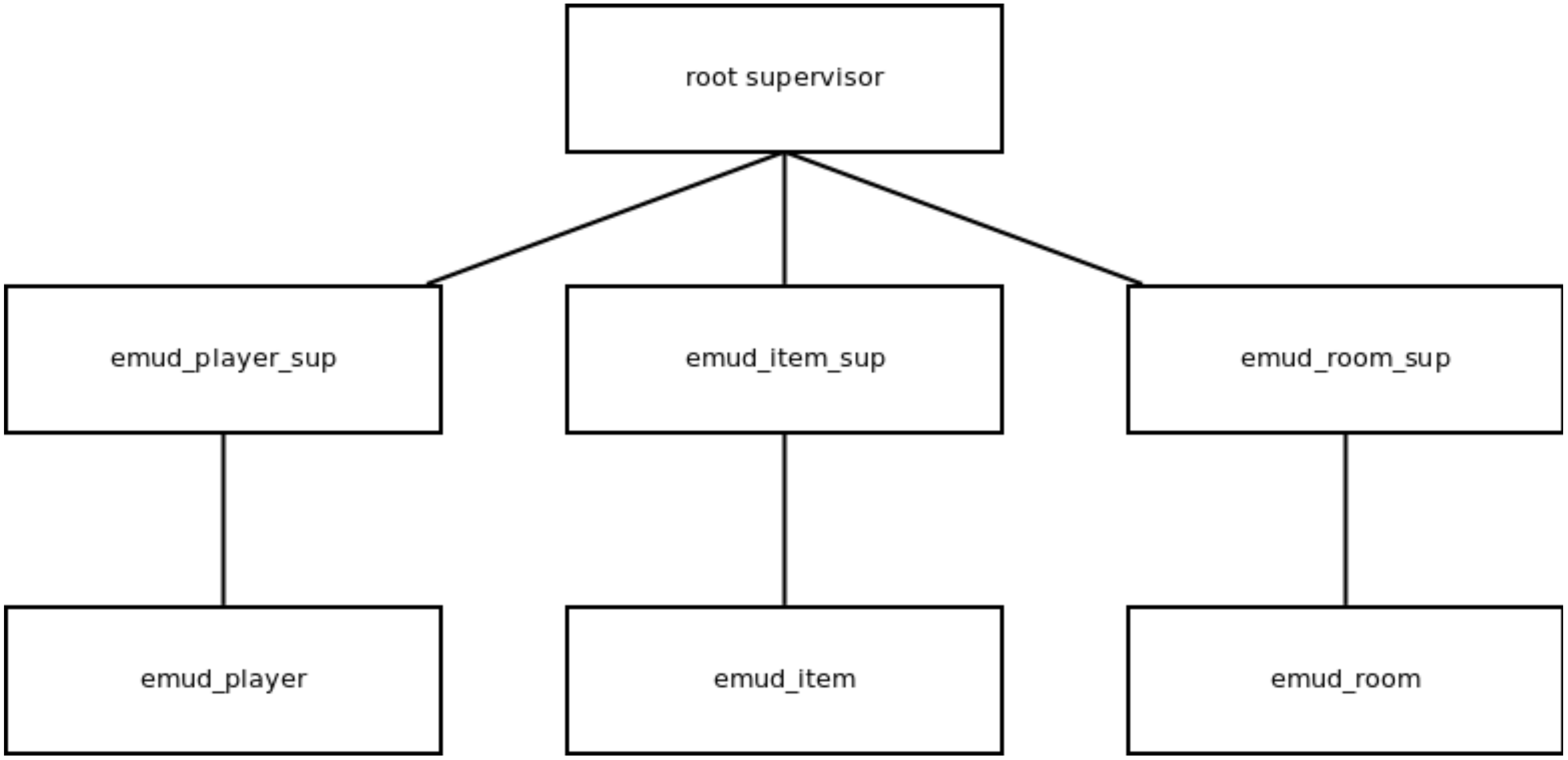
- all interaction through message passing.

- seems easy :)

Process entities

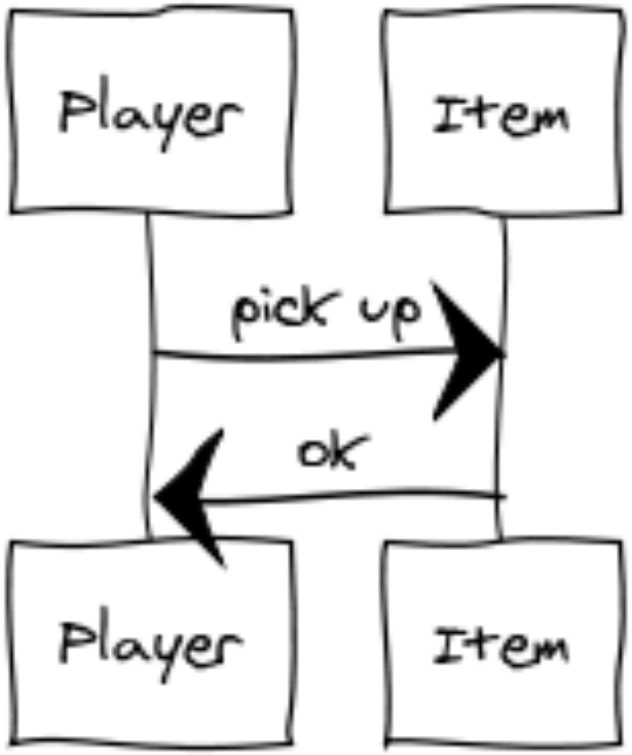


Supervision tree



Concurrency

- Player picks up item.
  Easy! ... Easy?



vebsequencediagrams.com

- Yes easy, but not what
  we want!

+

+

->

THIS is what we want!

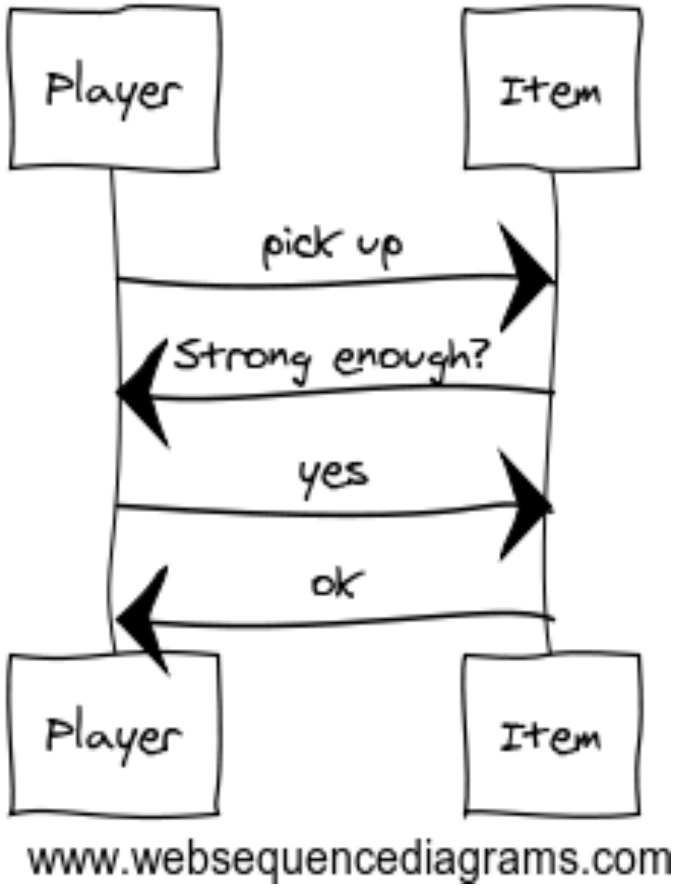
Synchronous pickup, take two

- Player picks up item.

- Outcome depends on player and and item.

- Item might demand anything (strength, class, charisma, props of other objects.)

- not atomic. Race conditions, deadlocks if sync.



Back up a bit, we want:

- Player: request (pickup) synchronous and atomic.

    Observes yes or {no, Reason}.

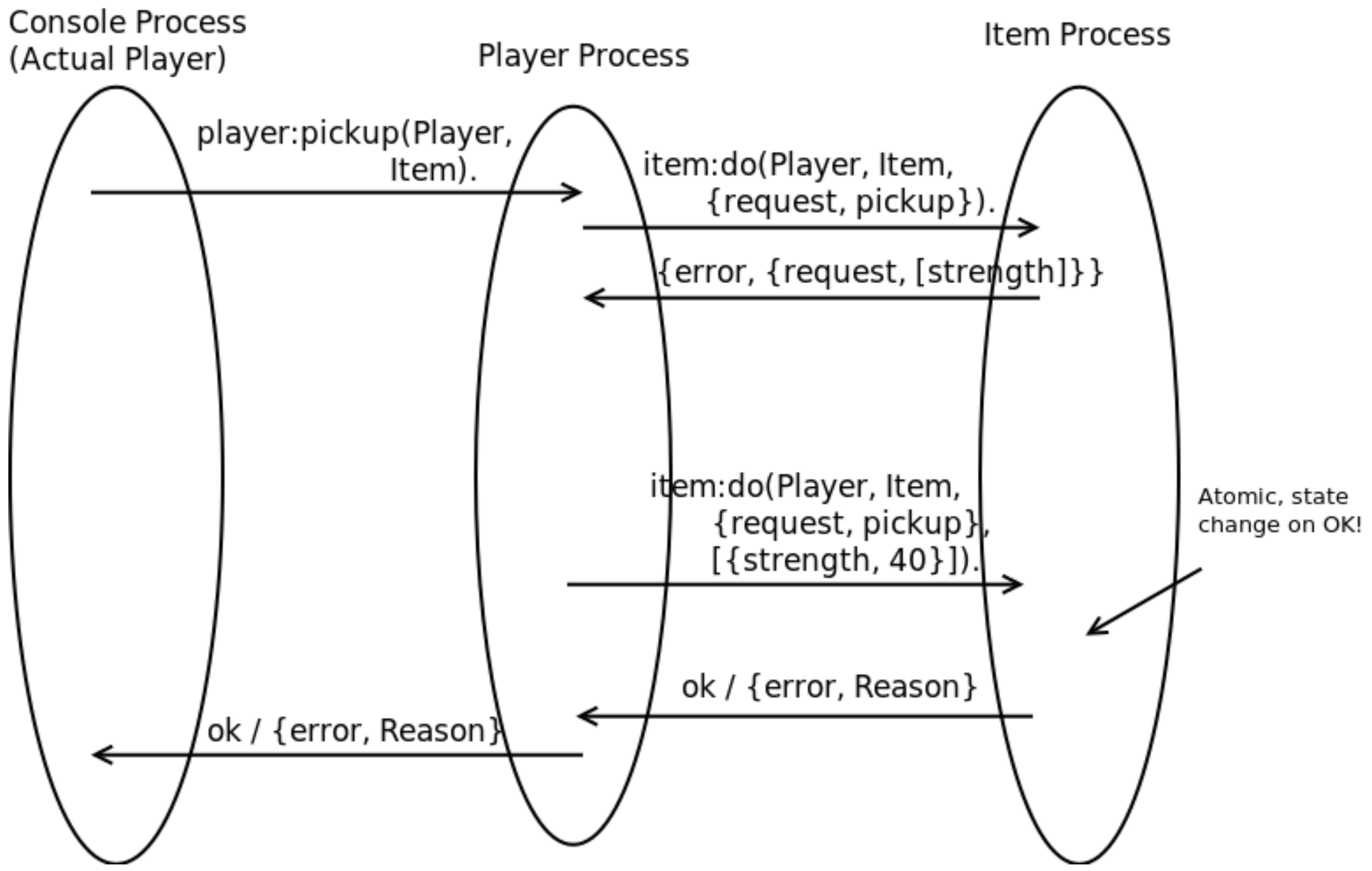
- If player qualified:
    - fails (State says: not pick-upable)
    - request succeeds + state change in object.

The Item implementation

Must obey:

- if it acknowledges a request it must change state.
- new state prevents the event from happening again.
- and do so atomically.

Let's draw it


Console Process
(Actual Player)
Player Process
Item Process
player:pickup(Player, Item).
item:do(Player, Item, {request, pickup}).
{error, {request, [strength]}}
item:do(Player, Item, {request, pickup}, [{strength, 40}]).
Atomic, state change on OK!
ok / {error, Reason}
ok / {error, Reason}

So, why is this easier than in Java?

Well..

- All state is encapsulated - no shared state.
- Also means all state is synchronized.
- No need to argue about locking.
- Concurrency is and feels natural.
- Only think about concurrency, not worry about state!

Bootstrapping/building world

Happens in mymap.erl:

- Supervisors start room, player, item children.

```
{ok, Chair} = supervisor:start_child(emud_item_sup,
                            emud_specs:childspec_item(chair)),
```

- Everything from child specs, except:

```
ok = emud_room:add_item(RestRoom, Chair),
emud_room:link_rooms(StartRoom, RestRoom, n),
```

Child specs / state

```
{ok, Chair} = supervisor:start_child(emud_item_sup,
                                     emud_specs:childspec_item(chair)),

childspec_item(Name) ->
    {Name, {emud_item, start_link,
               [create_item_state(Name)]},permanent, 2000, worker,
               [emud_item]}.

create_item_state(chair) ->
    S1 = emud_create_item:create_state(),
    S2 = emud_create_item:set_short_description(S1, "A chair.\n"),
    S3 = emud_create_item:set_description(S2, "A dingy looking
chair, made of driftwood.\n"),
    S4 = emud_create_item:set_interaction_names(S3, ["chair"]),
    S4;
```

Problems/challenges

- save and load specs to persistent store.
- linking between dynamic entities.
- ref entities when persisted or lazy.

Demotime

The thorns

- Refactoring is hard.
- Too easy to break stuff.
- Feels silly to not have many things caught by type checker.

- Dialyzer + specs helps.
- Lots of unit and integration tests helps.

- Maybe this is a good thing!? Forces testing!

The thorns

- Hard to understand errors and stack traces.

- This will get better over time - but right now I'm not there - takes too long.

```
{"init terminating in do_boot",{{badarith,[{emud_player,handle_call,3,[{file,"
src/emud_player.erl"},{line,130}]},{gen_server,handle_msg,5,[{file,"gen_server.
erl"},{line,588}]},{proc_lib,init_p_do_apply,3,[{file,"proc_lib.erl"},{line,227}]}]},
{gen_server,call,[<0.78.0>,{crash}]}}}
/home/lhc/dev/install/erlang/otp-r15b01/lib/os_mon-2.2.9/priv/bin/memsup:
Erlang has closed.
```

(they don't have to look this bad...)

The thorns

- Miss enum types, atoms without types feel insufficient.

- not sure how to manage large erlang modules. How to split them? When to split them?

- Experience... experience!

Next episode

- (Major) rewrite:
  - Move everything into specs/state.
  - Generalize property code. Dynamically add properties.
- Persisting state:
  - Save and load state (to dets, amnesia??)
  - Spawn sup children from persisted states.
- Creating ais that do stuff.
- Create interesting content!

Next episode

- Use bitstrings internally.
- Players classes and characteristics.
- Fighting.
- Create login.
- Player creation.
- Restart a console (not a gen_server yet.)
- Create a better console (really, it sucks.)
- Package and release.
- Make a web client.

Conclusion

- Erlang and concurrency is awesome!
- Feels a bit old and primitive. Learn to love ...?
- Would love to see the same principles, but wrapped in something modern.

- But, it feels good, and I'm going to continue to dive deep into the language and OTP.

Resources

- https://github.com/larshesel/emud
- http://learnyousomeerlang.com/ (awesome)
- http://en.wikipedia.org/wiki/MUD
- http://daimi.au.dk/~clemen/styx/

Illustrations:
http://openclipart.org

Thanks!

twitter: @larshesel
private: larshesel at gmail.com
work: lhc at trifork.com