Building a Realtime Game Backend from Scratch

(how we use Erlang and why)

Jeremy Ong © **Quark Games Inc.**

whoami

jeremyong

banachtarski @ freenode

jeremyong.com
quarkgames.com

What are we doing @ qg?

- New real-time, 3D game set to launch in the next couple months
- Active development for only ~5 months (with 3 engineers, although more have been added recently)!

Loose Game Description

- Players can buy and upgrade units
- Players use a subset of their inventory to battle opponents
- Players interact with each other via the game itself, chat, and matchmaking
- Players interact with the server by making purchases, modifying their inventory, and managing their accounts

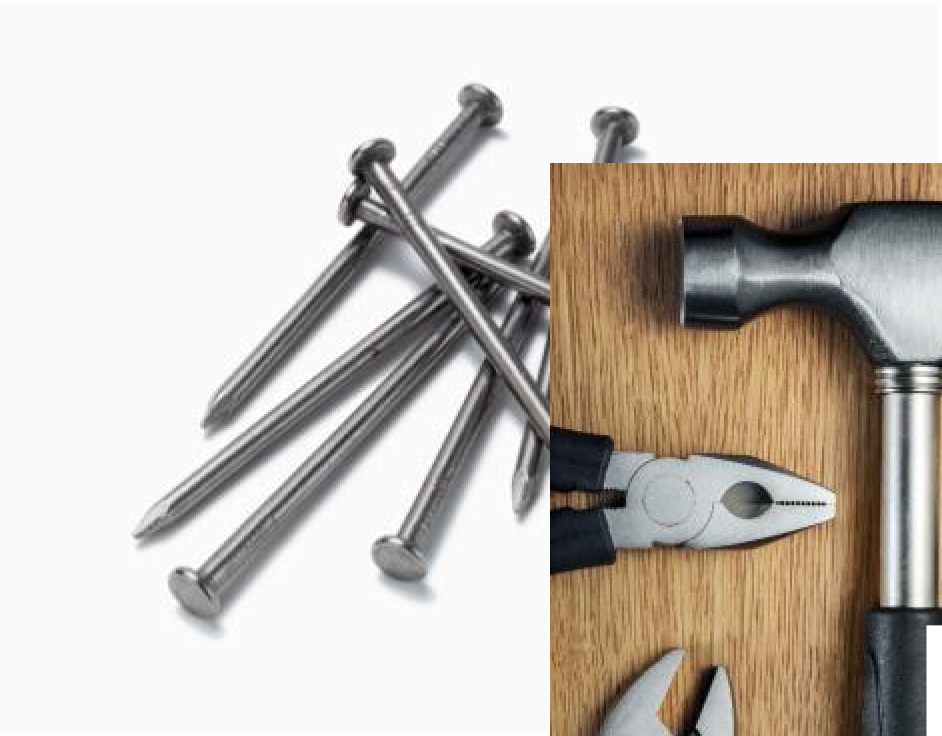
Why this talk?

- Game programmers = OO programmers (traditionally)
- Erlang plays *very* nicely with games for particular roles
- How do I get started?
- What can go wrong?
- I already know Erlang, want to help train up others

The "pitch" - Why Erlang

- Low latency
- Stateful
- Shared nothing
- Good concurrency (a small squadron of beefier machines is more cost-efficient than an army of trash cans)
- Intuitive to use

Real Time Server Applications



Erlang (nailgun)

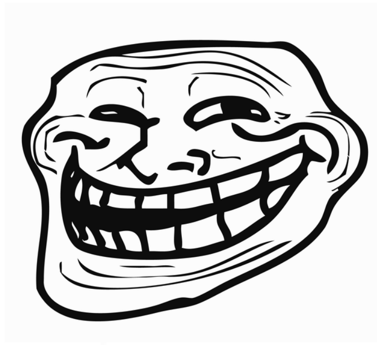
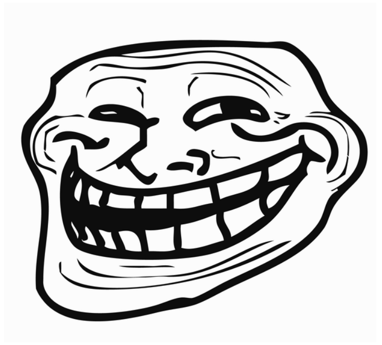
Other various programming languages

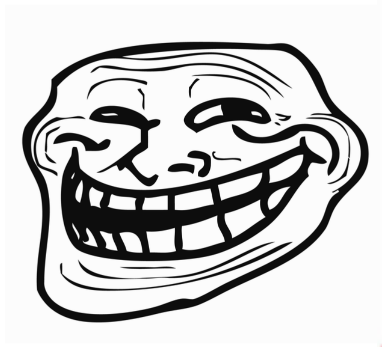
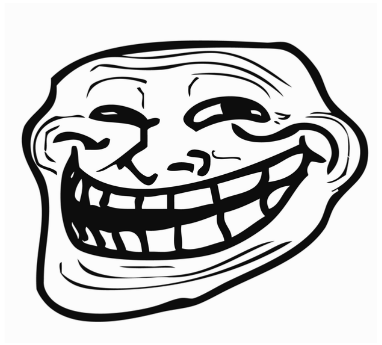
BUT

there are many wrong ways to use a nail gun!

How do we explain how to use and think about Erlang?

cheap 24/7 min-wage workers, factory machines (that we just throw out and replace), build orders, warehouses, repairmen, blueprints that constantly change and can change running machines

Erlang Sweatshop SF 2014??????

Blueprints and Instruction manuals

Code, modules, functions (used to start, create, and run factory machines)

Machines

Processes: the things that do work. Easy to make new ones. We throw them out once they finish

Machines have conveyor belts

Sequential mailbox for both sending and receiving data needed to do something

Example

```
cuz_im_a_machiiiiiiiiine_baby() ->
  receive
    hi ->
      io:format(
         "And I've got the keys, baby"),
      cuz_im_a_machiiiiiiiiine_baby();
    bye ->
      io:format("... I'm not unstoppable")
    end.

spawn(fun cuz_im_a_machiiiiiiiiine_baby/0).
```

Sweatshop Floor Space

Available memory (can only accomodate so many machines)

"Repair"men

Supervisors: they "fix" your broken machines by throwing them out and replacing them with a new one hopped up

Sweatshop Workers

Scheduler: starts, suspends, runs, and terminates the machines (flipping switches)

Power

Your cpu cores. Can only power one machine per core at a time

Workflow

1. Specify system requirements
2. Imagine you had to run the system with manual labor and pencil/paper
3. Describe all the occupations you would need, how many you would need
4. Try to map each occupation with an OTP behaviour, and if this isn't possible, describe the behaviour in full
5. Write it piece by piece until finished

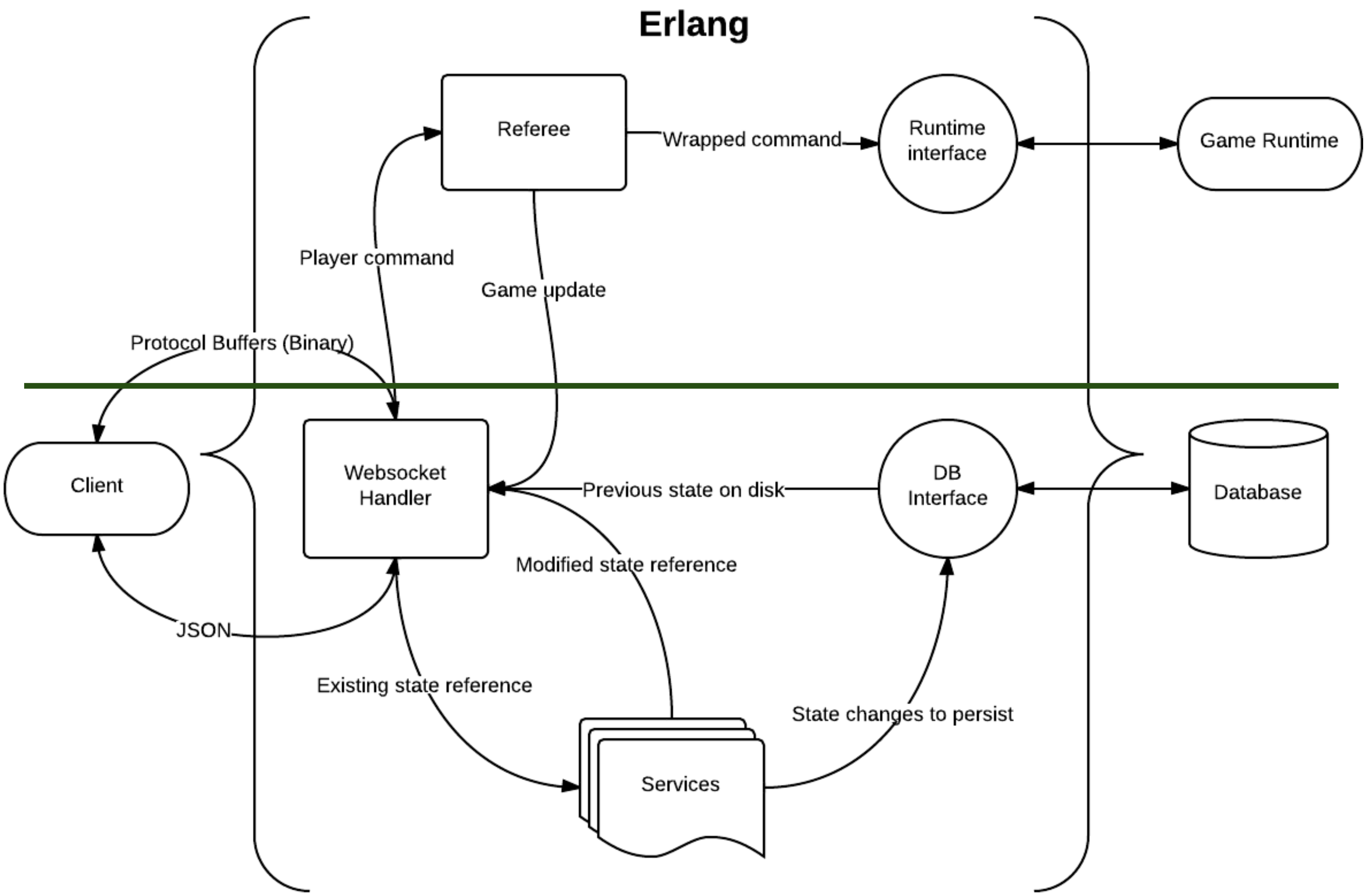
Our Architecture

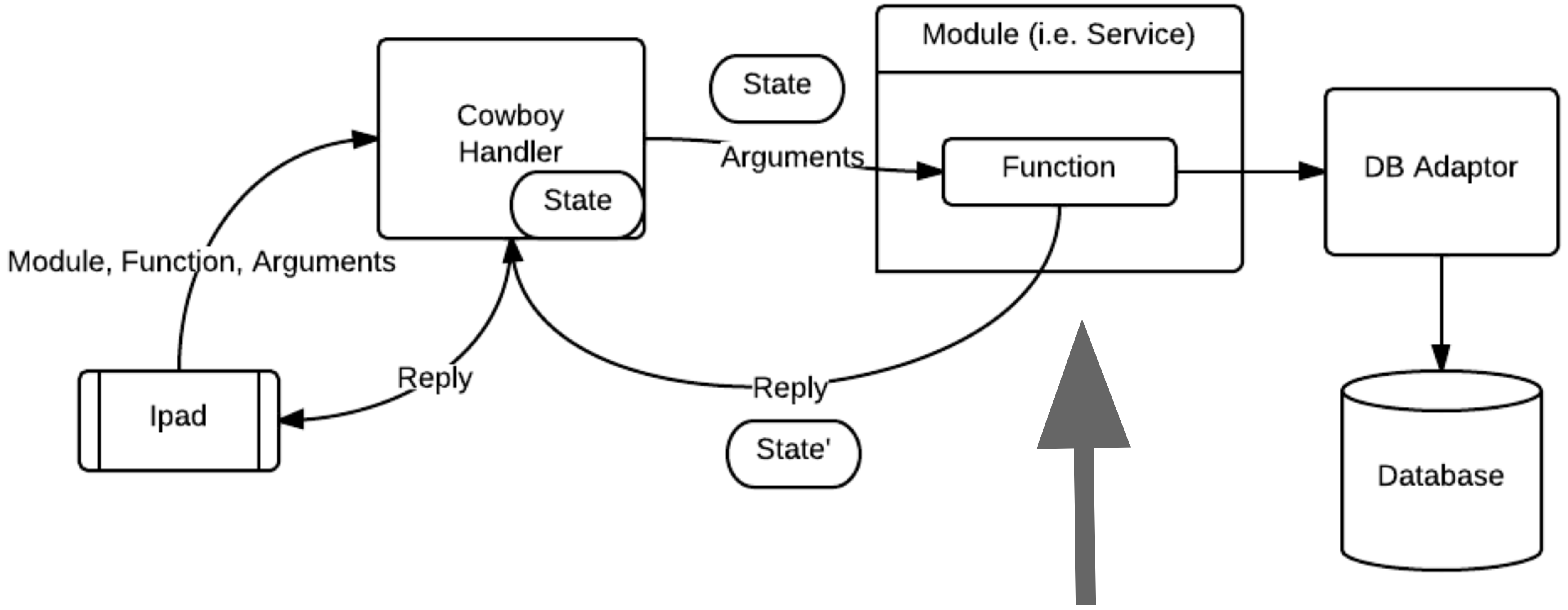
Backend Responsibilities

- Authentication + account creation
- Manage user state (e.g. allow user to purchase units)
- Route data between users and an instance of a C# game runtime
- Monitor activity in real time

Performance Requirements

- Users do many things clumped together
- Fast response times are a must
- Response times should be relatively load insensitive


Erlang
Referee
Wrapped command
Runtime interface
Game Runtime
Player command
Game update
Protocol Buffers (Binary)
Client
Websocket Handler
Previous state on disk
DB Interface
Database
Modified state reference
JSON
Existing state reference
State changes to persist
Services



Exercise: Should this be a separate machine/process?

Not usually, no

Know thy scheduler

Let the user process do (most of) its own work

- Ignore the temptation to handle all requests asynchronously (think first)
- Let the scheduler allocate resources fairly
- Serialized commits = consistent data
- Isolate errors
- Let it die!

Other Architectural Principles

Perform operations in memory

- Users clump their actions, use sessions
- Read once, write multiple times
- Consider how your data is stored (natural transformation between in memory representation, and db representation)

One dirty module ... to find them ... one dirty module to bind them ...

One dirty module per foreign dependency

- Much more easy to test (fewer dependencies to inject)
- Easier (possible) to swap out or modify foreign dependencies
- Mock out your dirty module and provide fixtures (we use meck, websocket_client)

Homogenize your servers

- Don't make any machine "special" if you can help it (but do it if you should)
- Simplify deploys/upgrades/migrations/setup
- Minimize CAP and Murphy's Law exposure
- Coordination is hard!

Erlang supports hot swapping. Use it!

- Annoying to set up, (relatively) painless thereafter
- One of the best features that people seem to avoid
- Test, test, test.
- Automate, automate, automate.

Thread the entire state

- Old approach: provide each user accessible function with exactly the information needed
- Just pass the state variable
- Leverage Erlang's immutability
- The calling module should not know how to decompose and recompose the data

Sample code:

```
handle_response(State, Text) ->
  {Module, Function, Args} = extract_mfa(Text),
  Module:Function(State, Args).
```

Make this secure obviously!!

Example: adding a unit

```
add_unit(State, [Unit]) ->
    Currency = lookup_currency(State),
    Cost = cost_of(Unit)
    case deduct_currency(State, Cost) of
      {ok, State1} ->
        State2 = append_unit(State, Unit),
        db_lib:persist(State, manager),
        {reply, success, State2};
      error ->
        {reply, cannot_afford, State}
    end.
```

Standardize your protocols

- MFA style API (natural mapping to erlang MFA)
- Data that can't or shouldn't be human readable is serialized and deserialized using protocol buffers
- Protocol buffers + Interoperability = <3

Tips for building an Erlang system

The tips (do the right things (and actually do them))

- Benchmark
- Log
- Test (unit, integration, system)
- Actually use your tests
- Document
- Typespec (and actually use dialyzer)

Libraries We Use at a Glance

- Application
  - cowboy* (extend)
  - libprotobuf* (TensorWrench)
  - jsx (talentdeficit)
  - lager (basho)
  - bcrypt* (smarkets)
- Testing/Benchmarking
  - common_test
  - meck (eproxus)
  - websocket_client* (jeremyong)
  - basho_bench (self-evident)

* indicates pull request pending or accepted

Conclusion

- Think like a ruthless sweatshop owner
- Teach others around you to think like ruthless sweatshop owners
- Don't do the above two actions literally

We're Hiring!

- If you like games and solving tough programming problems, get in touch!
- jeremy@quarkgames.com

Private Beta

https://www.surveymonkey.
com/s/qgprivatebeta

Need iPad 2 or newer