MODERN CONCURRENCY:

ERLANG, SCALA, GO, CLOJURE

BY ALEXEY KACHAYEV, 2013

ABOUT ME

- Position: CTO at Kitapps Inc.
- Production experience: Python, Java, JS, Go, Scala, Clojure, Erlang
- Looked at: Haskell, Lisp, Scheme

WHERE TO FIND?

- Twitter: **@kachayev (http://twitter.com/kachayev)**
- Github: **@kachayev (http://github.com/kachayev)**

WHAT WE ARE GOING TO TALK ABOUT?

- More about concurrency, (much!) less about parallelism
- Tasks: from practice and from theory
- Classical approach and problems: threads/locks/mutex
- One task, different solutions: Actors, Channels, STM
- What to choose and how to live with all this stuff?
- p.s. Why functional programming matters

CONCURRENCY IS NOT PARALLELISM

Rob Pike:

> *“ Concurrency is the composition of independently executing computations. Concurrency is a way to structure software, ... ”*

Video "Concurrency is not parallelism" (http://vimeo.com/49718712)

TASKS

- Theory:
  Sleeping barber problem
  (http://en.wikipedia.org/wiki/Sleeping_barber_problem)
- Practice: Cache manager for image viewer (you want to show several images on a mobile screen downloaded from server, but you have hard limits on free memory and disk space)

- Why should we solve this stupid theoretical task?
- Cause any task from practice is much harder

Need more tasks from practice? Ping me after the talk.

HAH, IT'S EASY. USE LOCKS, LUKE!

"Classical" approach, each thread:

- check resource on disk
- "lock" hash map, check if someone is downloading the same, "unlock"
- check HTTP header for Content-Length
- "lock" free space counter, try to subtract size, "unlock"
- if successful, read from the HTTP body to disk
- if not, call somebody who will remove old file(s)

HAH, IT'S EASY. USE LOCKS, LUKE!

Hmmm, really?

- What to do with crashes while downloading images?
- What to do with "waiters" and how to manage free space on concurrent +/- operations?
- How to manage free space more "intelligently": 10 * 1 mb is better that 1 * 10 mb?
- How to "kill" a task if an image is not needed (before it's downloaded)?
- How will this program work if we run 1000 "downloaders"?
- Are you sure that your locks are good enough???

WHAT IS THE PROBLEM WITH LOCKS?

Jonas Boner,
about fault-tolerant scaling (http://www.slideshare.net/jboner/building-scalable-highly-concurrent-fault-tolerant-systems-lessons-learned)
:

- Locks do not compose
- Locks breaks encapsulation *(you need to know a lot!)*
- Taking too few locks
- Taking too many locks
- Taking the wrong locks
- Taking locks in wrong order
- Error recovery is hard

WHAT IS THE PROBLEM WITH LOCKS?

- It's not the nature of our world!!! (we don't ask the world to stop to do something)
- What to do with crashes in a critical section?
- Impossible to test: you will know about the problem only "in production"

MODERN CONCURRENCY

Think about it! You work in a big free-space office with many programmers (hundreds), you have one board with a piece of paper pinned to it. Each time someone finishes doing a task, he or she should increase the number written on this paper. What options do you have? How will you organize everything? What will you do with paranoid manager who wants to know how many tasks are already done (from time to time, often)?

MODERN CONCURRENCY

- Actors (message passing)
- STM (Software Transactional Memory)
- Dataflow Concurrency (
  your homework
  (http://doc.akka.io/docs/akka/snapshot/scala/dataflow
)

WHAT DOES ACTOR LOOKS LIKE? (THEORY)

- Separated isolated lightweight processes (actor)
- Share nothing, avoid side-effects (FP)
- Pure message passing communication (mailbox)
- Location transparency

WHAT DOES ACTOR LOOKS LIKE? (ERLANG)

barber.erl | Erlang | 2.29 kb

```erlang
-module(barber).
-export([run/2, barber/1, clients/1, simulator/2]).

-define(CUT_DUTAION, 20).

%% XXX: links, monitors etc are not used to focus on messaging!!!
%% XXX: use records instead of long tuples

run(RoomSize, Duration) ->
    % create barber
    BPid = spawn(?MODULE, barber, [?CUT_DUTAION]),
    % run simulartor with barber PID
    SPid = spawn(?MODULE, simulator, [BPid, RoomSize]),
    % spawn clients generator
    CSPid = spawn(?MODULE, clients, [SPid]),
    % create timer to close simulator after duration time
    timer:send_after(Duration, SPid, {close, CSPid}).

barber(Duration) ->
    receive
        {client, Simulator} ->
            timer:sleep(Duration),
            Simulator ! {barber, done},
            barber(Duration)
    end.

clients(Simulator) ->
    Rnd = 7 + random:uniform(28), % XXX: you can play with constants
    receive
        close -> ok
    after Rnd ->
        Simulator ! {sim, client},
        clients(Simulator)
    end.

```

gist (https://gist.github.com/4634226)

WHAT DOES ACTOR LOOKS LIKE? (ERLANG)

```erlang
simulator({BPid, Status}, {RoomSize, Client, Total}) ->
    case serve_simulation(self(), {BPid, Status}, {RoomSize, Client, Total}) of
        {S, C, T} -> simulator({BPid, S}, {RoomSize, C, T});
        {close, ClientsGenerator} ->
            ClientsGenerator ! close,
            io:format("Closed! Served clients (~p)~n", [Total]),
            ok
    end;
simulator(Barber, RoomSize) ->
    simulator({Barber, free}, {RoomSize, 0, 0}).

serve_simulation(Me, {BPid, Status}, {RoomSize, Client, Total}) ->
    receive
        {sim, client} when Status == free ->
            io:format("Client goes to barber~n"),
            BPid ! {client, Me},
            {busy, 0, Total};
        {sim, client} when Status == busy, RoomSize > Client ->
            io:format("Client goes to room, new size (~p)~n", [Client+1]),
            {busy, Client+1, Total};
        {sim, client} ->
            io:format("No free space for client, room size (~p)~n", [Client]),
            {busy, Client, Total};
        {barber, done} when Client > 0 ->
            io:format("Take client from room, new size (~p)~n", [Client-1]),
            BPid ! {client, Me},
            {busy, Client-1, Total+1};
        {barber, done} ->
            io:format("Barber finished, idle~n", []),
            {free, 0, Total+1};
        {close, ClientsGenerator} -> {close, ClientsGenerator}
    end.
```

gist (https://gist.github.com/4634226)

WHAT DOES ACTOR LOOKS LIKE? (SCALA)

- Very close to previous
- Approaches for state handling (rough, in short words): actor is a state (Scala) VS. passing state as an argument (Erlang)

GO CHANNELS

barber.go | Go | 2.24 kb

```go
package main

import (
    "fmt"
    "time"
    "math/rand"
)

const (
    CUTTING_TIME = 20
    BARBERS_AMOUNT   = 1
    HALL_SITS_AMOUNT = 3
)

type Barber struct {
    val int
}

type Client struct {
    val int
}

func main() {
    clients := make(chan *Client)
    go clientProducer(clients)
    go BarberShop(clients)
    time.Sleep(2 * time.Second)
}

func clientProducer(clients chan *Client) {
    for {
        time.Sleep(time.Duration(rand.Intn(28) + 7) * time.Millisecond)
        clients <- &Client{}
    }
}

func cutHear(barber *Barber, client *Client, finished chan *Barber) {
    // Cutting hear
    time.Sleep(CUTTING_TIME * time.Millisecond)
    finished <- barber
}
```

gist (https://gist.github.com/4634932)

GO CHANNELS

```
43  func BarberShop(clients <-chan *Client) {
44      freeBarbers := []*Barber{}
45      waitingClient := []*Client{}
46      syncBarberChan := make(chan *Barber)
47
48      //creating barbers
49      for i := 0; i < BARBERS_AMOUNT; i++ {
50          freeBarbers = append(freeBarbers, &Barber{})
51      }
52
53      for {
54          select {
55          case client := <-clients:
56              if len(freeBarbers) == 0 {
57                  if len(waitingClient) < HALL_SITS_AMOUNT {
58                      // client is waiting in the hall
59                      waitingClient = append(waitingClient, client)
60                      fmt.Printf("Client is waiting in hall (%v)\n", len(waitingClient))
61                  } else {
62                      // hall is full - bye-bye client
63                      fmt.Println("No free space for client")
64                  }
65              } else {
66                  barber := freeBarbers[0]
67                  freeBarbers = freeBarbers[1:]
68                  fmt.Println("Client goes to barber")
69                  go cutHear(barber, client, syncBarberChan)
70              }
71          // barber finish work
72          case barber := <-syncBarberChan:
73              if len(waitingClient) > 0 {
74                  // get client from hall
75                  client := waitingClient[0]
76                  waitingClient = waitingClient[1:]
77                  fmt.Printf("Take client from room (%v)\n", len(waitingClient))
78                  go cutHear(barber, client, syncBarberChan)
79              } else {
80                  // barber is going to sleep
81                  fmt.Println("Barber idle")
82                  freeBarbers = append(freeBarbers, barber)
83              }
84          }
85      }
86
87  }
```

gist (https://gist.github.com/4634932)

GO CHANNELS

"Rough analogy: writing to a file by name (process, Erlang) vs. writing to a file descriptor (channel, Go)." (c) Rob Pike

"Rough analogy: Assume Go channels as portable actor mailboxes" (c) Me

CLOJURE CONCURRENCY PRIMITIVES

- Designed for **concurrency (http://clojure.org/concurrent_programming)**
- Identity and value are different things
- Different approaches for sharing changing state
- Software transactional memory - synchronous and coordinated approach (memory as database, atomicity, consistency, isolation)
- Agent(s) - asynchronous and independent approach
- Atom(s) - synchronous and independent approach

CLOJURE CONCURRENCY PRIMITIVES

agents_VS_atoms.clj | Clojure | 421 bytes

```
$ clj
Clojure 1.4.0
user=> ;; atoms - synchronous sharing state changes
user=> (def w (atom 0))
#'user/w
user=> (swap! w inc)
1
user=> @w
1
user=> (swap! w #(* % 100))
100
user=> @w
100
user=> ;; agents - asynchronous sharing state changes
user=> (def waiters (agent 0))
#'user/waiters
user=> (send waiters inc)
#<Agent@c3fa6cd: 1>
user=> @waiters
1
user=> (send waiters #(+ % 10))
#<Agent@c3fa6cd: 11>
user=> @waiters
11
```

gist (https://gist.github.com/4632568)

CLOJURE CONCURRENCY PRIMITIVES

sleeping-barber-problem.clj | Clojure | 1.09 kb

```clojure
(def open-for-business? (atom true))
(def haircut-count (agent 0))
(def waiting-room (ref []))
(def waiting-room-size 3)

(defn open-shop [duration]
  (do (Thread/sleep duration) (swap! open-for-business? not)))

(defn add-customers []
  (future
    (while @open-for-business?
      (println "Waiting Room:" @waiting-room)
      (dosync
        (if
          (< (count (ensure waiting-room)) waiting-room-size)
          (alter waiting-room conj :customer)))
        (Thread/sleep (+ 10 (rand-int 21))))))

(defn get-next-customer []
  (dosync
    (let [next-customer (first (ensure waiting-room))]
      (when next-customer (alter waiting-room rest) next-customer))))

(defn cut-hair []
  (future
    (while @open-for-business?
      (when-let [next-customer (get-next-customer)]
        (Thread/sleep 20)
        ;; do we want to continue cut hair before (!)
        ;; the actual value is set (possibly) ???
        (send haircut-count inc)))))

(do
  (cut-hair)
  (add-customers)

  (println "Open barber shop for 10 secs")
  (open-shop 10000)

  (println "Number of cuts:" @haircut-count)
  (System/exit 0))
```

gist (https://gist.github.com/3160721)

CLOJURE CONCURRENCY PRIMITIVES

Why functional programming matters in Clojure:

- Side-effect free functions
- Deterministic functions/calculations

HOW TO LIVE WITH THIS

- I can write the same as Scala/Clojure/Go in Java/Python/..!
- hah, really...?

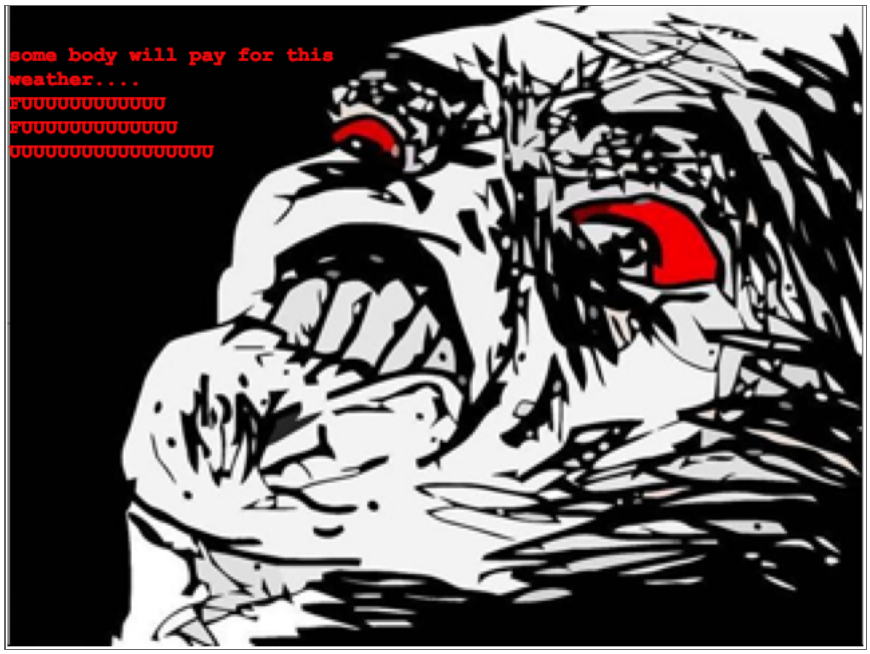
Everything that can go wrong will go wrong (c) Murphy

HOW TO LIVE WITH THIS

Question (http://stackoverflow.com/questions/14298523/why-does-adding-concurrency-slow-down-this-golang-code)

: Why does adding concurrency slow down this golang code?

Answer: The issue seems to come from your use of `rand.Float64()`, which uses a shared global object with a Mutex lock on it.


some body will pay for this
weather....
FUUUUUUUUUUU
FUUUUUUUUUUUU
UUUUUUUUUUUUUUUU

WHAT TO CHOOSE

- There is no silver bullet
- Use most natural approach for your technology / OP (there are many STM implementations, but in Clojure it's idiomatic and part of the core)
- Look deeply into your domain (presenters usually use most convenient case(s) to tell about functionality)
- Use "unusual" approach but remember that on this way only conventions rule
- If you need distributed concurrency - look at actors

WHAT TO CHOOSE

For Scala developers from
Jonas Boner
(http://www.slideshare.net/jboner/building-scalable-highly-concurrent-fault-tolerant-systems-lessons-learned)
:

- Start with deterministic, declarative, immutable core
- Add indeterminism selectively (actors)
- Add mutability selectively (STM)
- Finally: add locks and explicit threads

ANY QUESTIONS?

- **These slides (https://github.com/kachayev/talks/tree/master/kharki** (PDF is coming!)
- **Other presentations (https://github.com/kachayev/talks)**
- Me on **Github (http://github.com/kachayev)**
- Me on **Twitter (http://twitter.com/kachayev)**

THANKS FOR YOUR ATTENTION