

Erlang и Haskell в production: проблемы и решения

Dmitry Groshev (@lambdadmitry),
Fedor Gogolev (@knsd),
@Selectel

FProg 2012-10

04.10.2012

- ▶ Вступление
- ▶ Коротко о пони
- ▶ YAWNDB
- ▶ Selecon-web
- ▶ Коротко об облаках
- ▶ Rainbowdash
- ▶ Twilightsparkle
- ▶ Резюме

Вступление

- ▶ 1.5 года production-experience с Erlang
- ▶ 1 год с Haskell
- ▶ In-memory timeseries database (YAWNDB)
- ▶ Система нотификации (Spike)
- ▶ Веб-консоль (Selecon-web)
- ▶ Облако (Rainbowdash/Twilightsparkle)

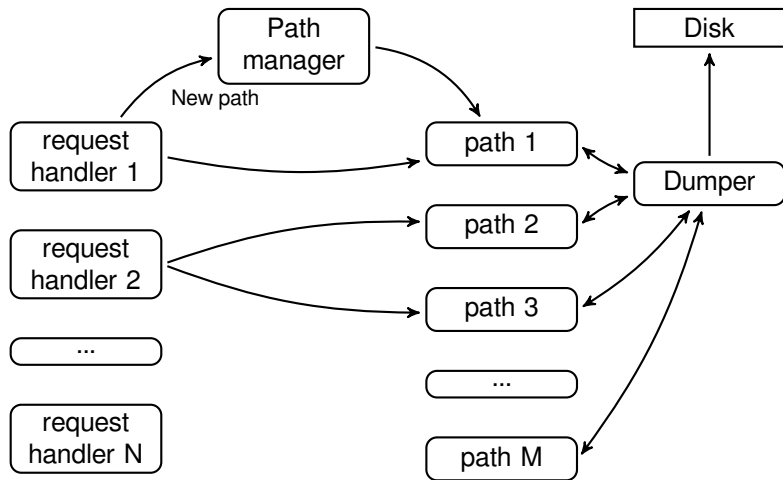
Коротко о пони

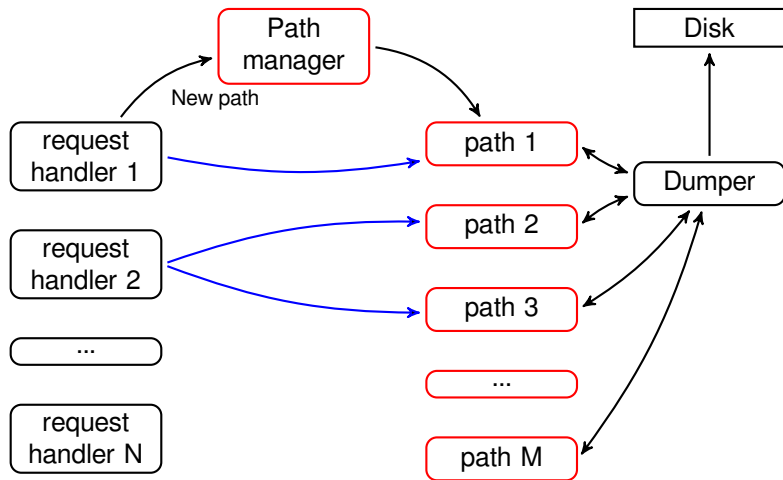


YAWNDB

YAWNDB — Yet Another Weel iNvented

- ▶ Timeseries данные (утилизация CPU per second)
- ▶ Много операций записи (десятки тысяч в секунду — виртуальных машин много)
- ▶ Хочется агрегацию (max/min/avg за период)
- ▶ Graphite медленный, RRD не умеет агрегацию



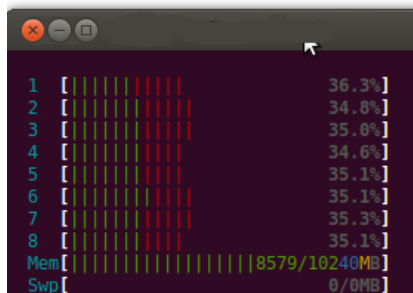


- ▶ NIF (Native Implemented Functions) это круто, но опасно
- ▶ мутабельные NIF binaries предоставляют изолированную мутабельность
- ▶ писать NIF неприятно, документация полна, но не всегда помогает
- ▶ property-based тестирование при использовании NIF необходимо, т.к. ошибки нетривиальны (C же!), а segfault'ы травматичны
- ▶ fuzz-тесты на случайных/бессмысленных данных полезны
- ▶ писать высококонкурентные системы сложно, Erlang не добавляет сложности
- ▶ устойчивость к ошибкам Erlang'a помогает, в продакшне почти полгода был редкий рейс без последствий вообще

YAWNDB: выводы 2

- ▶ код лаконичен (650 строк на C, 1500 строк на Erlang'e)
- ▶ если вы пишете что-то сетевое, нет ни одной причины не использовать Cowboy
- ▶ если ваш проект длиннее 30 строк, нет ни одной причины не использовать gproc
- ▶ поддержка SMP Erlang'ом не миф

```
"write_rps":18633,"processes_now":101044
```



- ▶ Graphite <https://github.com/graphite-project>
- ▶ Ecirca <https://github.com/band115/ecirca>
- ▶ Cowboy <https://github.com/extend/cowboy>
- ▶ Cowboy <https://github.com/uwiger/gproc>
- ▶ McErlang <https://github.com/fredlund/McErlang>

Облака



Xen:

- ▶ Dom0, DomU
- ▶ запуск и остановка доменов
- ▶ простой и неубиваемый

XAPI (XenAPI) от Citrix:

- ▶ Миграция
- ▶ Пулы
- ▶ Динамическое управление памятью
- ▶ ...

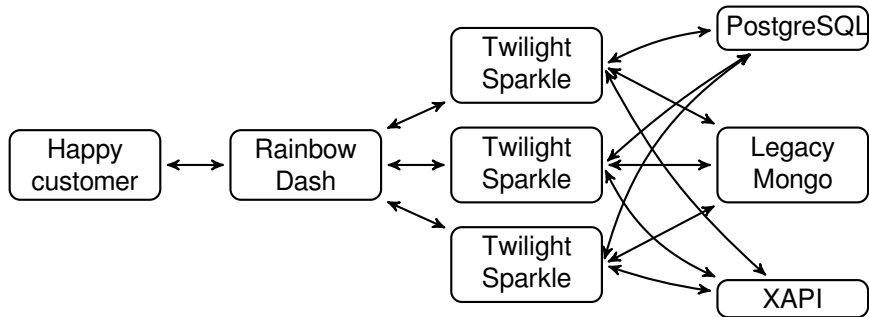
- ▶ учёт используемых ресурсов (биллинг, статистика)
- ▶ управление машинами из биллинга, веб-интерфейса, API и администраторами
- ▶ детекция нештатных ситуаций (падение хранилища либо сети)
- ▶ динамическая балансировка нагрузки
- ▶ предоставление интерфейсов к машине, не предусмотренных XAPI (web-консоль, realtime потребление, MemoryOnDemand)

Много сложной «бизнес»-логики

- ▶ **Make it work**, make it right, make it fast
- ▶ Python+Bash помойка
- ▶ Коммуникация через HTTP, Mongo и redis (aka «как получится»)
- ▶ WTF is summationd? WTF is yawndbtiond-obsolete?
- ▶ боль с Python+Mongo — много отдельного кода, размазывание схемы

- ▶ Make it work, **make it right**, make it fast
- ▶ Построение «от API»
- ▶ VBD/VDI/VIF/BDSM → Disk/Network interface/...
- ▶ Фиксированная схема данных
- ▶ Erlang для сети и рантайма, Haskell для «бизнес»-логики
- ▶ Механизм Erlang ports — stdin/stdout + Erlang External Term Format

Новая архитектура



Rainbowdash



- ▶ Милая, быстрая, немного простоватая, но reliable для друзей
- ▶ REST + RPC: HTTP REST API → RPC API на фронтенде по конфигу с верификацией (type safety!)
- ▶ Асинхронность + синхронность: интерфейс синхронный, rainbowdash асинхронна, twilightsparkle синхронна
- ▶ «Задачи» с уникальным идентификатором для каждого запроса
- ▶ 2-phase commit задачи: проверка корректности и ожидаемой ресурсоёмкости, запуск (возможно, отложенный)
- ▶ балансировка нагрузки на бекендах по ожидаемым ресурсам и внешним характеристикам процессов (ping, mem)
- ▶ отчёты о состоянии системы (ping, mem, cpu, rps, latency)
- ▶ 2-phase commit конфига
- ▶ почти live reloading Haskell-кода с персистентными задачами и HTTP-коннектами

- ▶ переход программистов Python → Erlang занимает неделю-полторы
- ▶ упаковка не-Erlang кода с помощью rebar это боль (Make+bash+cabal+cabal-dev)
- ▶ jobs — прекрасная библиотека, но документации почти нет
- ▶ любить себя полезно, несколько часов на автоматизацию перезагрузки бекенда при изменении бинарника окупились многократно
- ▶ кода до первой работоспособной версии достаточно мало (1k строк)

- ▶ jobs <https://github.com/uwiger/jobs/>

Twilightsparkle

- ▶ Общая архитектура
- ▶ Template Haskell и генерация сервера
- ▶ Контроль ошибок на уровне типов
- ▶ Барьеры — откат изменений
- ▶ Persistent ORM
- ▶ Проблемы при разработке

TwilighSparkle: общая архитектура

- ▶ Сервер, занимающийся чтением запросов из stdin и пишущий ответы в stdout
- ▶ Используется стандартный для Эрланга способ коммуникации — порты
 - ▶ Был написан модуль реализующий ETF (External Term Format)
- ▶ Воркеры, выполняющиеся в отдельных процессах
- ▶ Сложное ядро, максимально простой API для написания непосредственно обработчиков запросов
- ▶ Каждый запрос определяется тремя параметрами: source, input и result
 - ▶ source — Источник задачи, в нашем случае это пользователь API, администратор или внутренний сервис
 - ▶ input — Входные данные запроса
 - ▶ result — Результат на выполнение запроса

```
class TaskSource source => Task source input result
```

TwilighSparkle: Template Haskell и генерация сервера

Template Haskell используется для генерации функций разбора запросов от сервера. Например из кода:

```
[erlServer|  
    vm_start :: TS User VMStartTask ()  
|]
```

генерируется код:

```
dispatch ref "vm_start"  
    (ErlTuple [ErlBinary "user", ErlInt userId])  
    inputV = case fromErl inputV of  
        Nothing -> invalidTask  
        Just (i :: VMStartTask) -> do  
            ... -- Process request  
dispatch ref "vm_start" _user _input = invalidTask
```

- ▶ Прерывание выполнения подобно ErrorT трансформеру
- ▶ Отдельные типы для каждого вида ошибок
- ▶ Требование явного декларирования списка возможных ошибок

```
instance AllowError source VMStartTask () VMNotFound
instance AllowError source VMStartTask () VMIsAlreadyRunning
```

- ▶ Пока нет, но хочется контроль декларированных и не вызываемых ошибок

- ▶ Существует необходимость отката изменений в случае ошибок
- ▶ Барьеры устанавливаются после изменения, для которого требуется откат и выполняются в случае возникновения ошибки

```
vm <- createVm  
barrier $ destroyVm vm  
disk <- createDisk  
barrier $ destroyDisk disk  
error "Any error"
```

- ▶ Технически реализовано как [TS source input result ()] внутри TVar контекста ReaderT

TwilightSparkle: Persistent ORM

- ▶ Первая версия не использовала ORM

```
vm@(VM { vmUuid, vmTemplate }) <-  
  fetch VMCollection [ "id" == iVmId ]
```

- ▶ Регулярно возникали опечатки в названиях полей, в передаваемых данных
- ▶ Для Хаскеля нет рабочих альтернативных ORM кроме persistent

```
vm@(VM { vmUuid, vmTemplate }) <- fetch [ VmId ==. iVmId ]
```

- ▶ Пришлось использовать патченную версию persistent-mongodb, так как модель хранения отличается от подразумеваемой разработчиками

- ▶ Space leaks
 - ▶ Очень трудно найти источник проблемы при большом объёме кода
- ▶ Не хватает некоторых пакетов на haskage, либо не устраивает их состояние
 - ▶ Написали библиотеку для работы с ETF
 - ▶ Стали поддерживать библиотеки bson и mongoDB
- ▶ Вероятно, более высокий порог вхождения
 - ▶ Тем не менее в проект, кроме ядра, успешно пишут программисты без какого-либо функционального бэкграунда

- ▶ Заметно упало количество не логических ошибок, например:
 - ▶ Параметризованные идентификаторы, например (UUID VM), не допускают их использование для не VM объектов
 - ▶ Отдельные типы для различных семантически данных, например DiskSize
- ▶ Template Haskell — позволяет удобно решать проблемы, но катастрофически плохо читается
- ▶ Использовать cabal-dev — очень хорошая идея

Вопросы?

Copyrighted stuff:

https://en.wikipedia.org/wiki/File:Cirrus_sky_panorama.jpg

http://www.wallpapervortex.com/wallpaper-15684_1_other_wallpapers_my_little_pony.html

<http://www.tikihumor.com/3287/rainbow-dash-makes-it-rain/rainbow-dash-makes-it-rain-2/>

<http://qeinone.deviantart.com/art/Twilight-Sparkle-205789859>