
HighLoad++

# Как сделать вычислительную инфраструктуру для большого кластера

Евгений Кирпичёв

Станислав Лагун

Mirantis Inc. www.mirantis.com


HighLoad++

- Введение и Архитектура
- Доставка задач/результатов
- Отладка и анализ
- Обобщение опыта: корректность, надежность, производительность

# О нас

**Mirantis** делает проекты на заказ:

Высокотехнологичные, иногда долгосрочные, для «топовых» заказчиков
(Cisco, Mentor Graphics, Cadence, GE, …)

В основном – масштабируемые системы, клауды, research

# Что мы строим

- Очень тяжелые вычисления *(но очень параллельные)*
- Простой API – задача = поток подзадач: *CreateJob, SubmitTask, OnResult*
- Много одновременных задач разной важности
- Очень разнородные вычисления:
  От секунд *(нужна интерактивность)*
  до дней *(но не мешать чужой интерактивности)*
- Задействовать кластер целиком
- Устойчивость к временным падениям компонент
  (и к перманентным падениям вычислительных узлов)

Обычно (на суперкомпьютерах) используют:

| | | |
|---|---|---|
| **Platform LSF** | | |
| **MS HPC Server** | | **MPI** |
| **Oracle Grid Engine** | **+** | **OpenMP** |
| **Condor** | | **…** |
| **…** | | |

Это называется «**batch scheduler**»

HighLoad++

# Нам не подходит

Они предполагают:

- Планировщик *ничего* не знает про задачу
  - Просто выделяет ядра
- Задачи монолитны
  - «Мне надо 100 ядер»
  - Как появится 100 ядер – запустят
  - На 99 не запустят
- Есть куча сложных правил и квот
- Акцент на фичи, а не на эффективность


Без этих предположений можно сделать эффективнее.

# Терминология

**Задача** = поток **задачек**

Пакетный планировщик:

- Приложение:
  - Дай-ка мне 300 – 400 ядер и запусти на них вот эту команду
- Планировщик:
  - Вот ядра, команду запустил. Разбирайся сам.

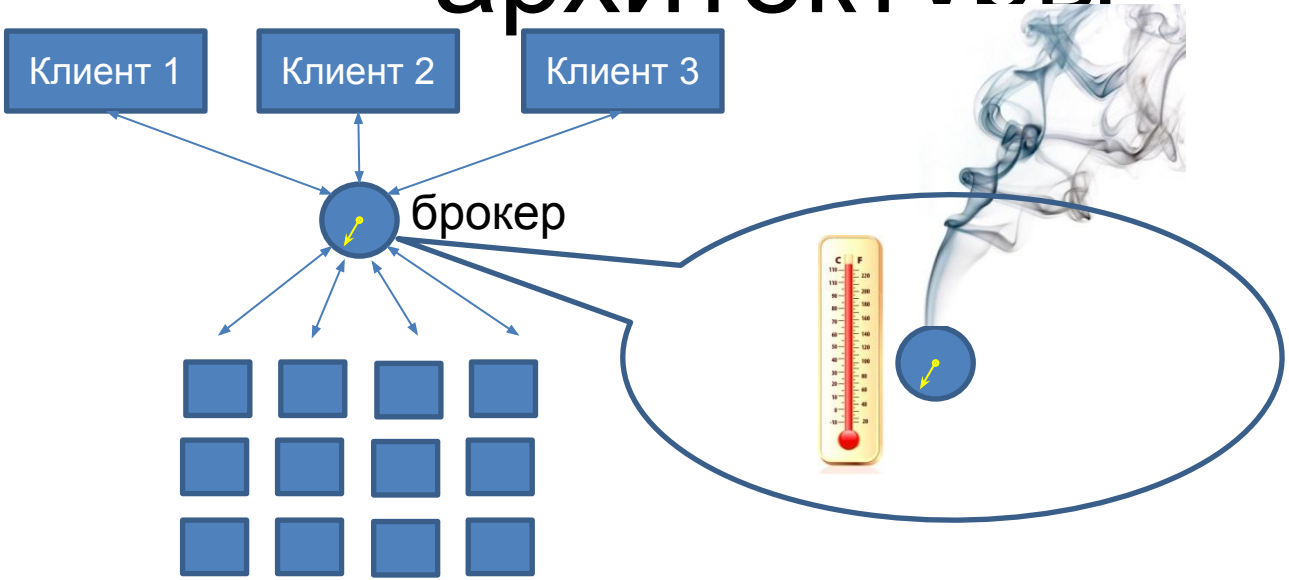
Мы:

- Приложение:
  - Я хочу использовать кластер для вычисления задачек
- Планировщик:
  - Хорошо, кидай задачки сюда, жди результатов отсюда. Я позабочусь об эффективной и справедливой делёжке ресурсов.

Становятся возможными некоторые трюки для повышения утилизации.

But this margin is too small…

# Пример неудачной архитектуры




**Очевидно, single bottleneck – не масштабируется**
**+балансировка нагрузки очень математически нестабильна**

# Более удачная архитектура

- **Планировщик**
  - Слушает команды о запуске-останове задач
  - Приказывает демонам обслуживать задачи
- **Трубы** (как очереди, только шире)
  - Доставляют задачки и ответики
- **Вычислительные демоны** на узлах
  - Слушаются планировщика
  - Тянут из очереди задачки, считают, публикуют ответики

+клиенты

+статистика

+логгирование

+мониторинг

# Пример

- **Клиент – Планировщику:**
  Создай задачу A, важность 30%
- **Планировщик (выбирает несколько демонов) – демонам:**
  Ты, ты и ты – бросайте всё и подключайтесь к трубе A.

- Клиент бросает задачки в трубу A
- Демоны считают, бросают ответики
- Клиент собирает ответики, бросает новые задачки и т.п.


HighLoad++

- Введение и Архитектура
- Доставка задач/результатов
- Отладка и анализ
- Обобщение опыта: корректность, надежность, производительность

# Трубы

- Как очереди
  - Только шире, быстрее и неупорядоченные
- На основе RabbitMQ
  - Лучший продукт в своем классе (надежная доставка)
  - Но «из коробки» сам по себе не масштабируется

# Надежная доставка

Цикл жизни демона:

- Получить задачку
- Посчитать
- Отослать ответик
- Подтвердить получение задачки

Если помрет, RabbitMQ заметит и перешлет задачку другому.

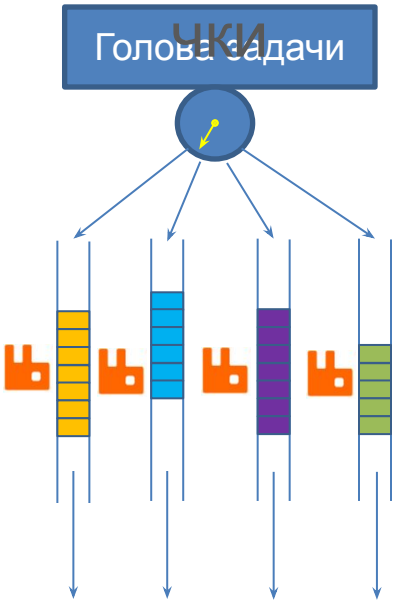
# Масштабирование

Из коробки RabbitMQ совсем не подходит

- Одна очередь плохо тянет 10000 клиентов
- Встроенная кластеризация делает не то
  - От нее вообще лучше отказаться (одни проблемы)
- Очевидное решение: несколько очередей + load balancing

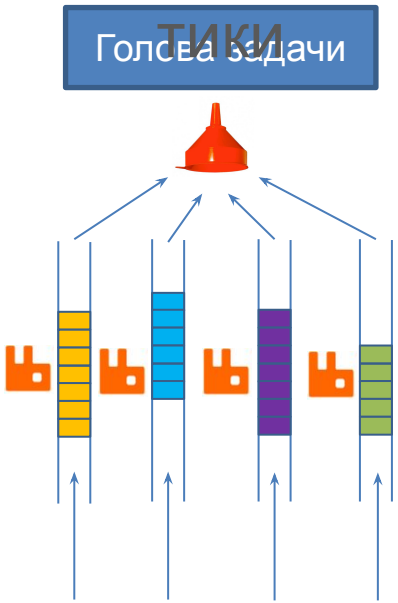
# Масштабирование

зада

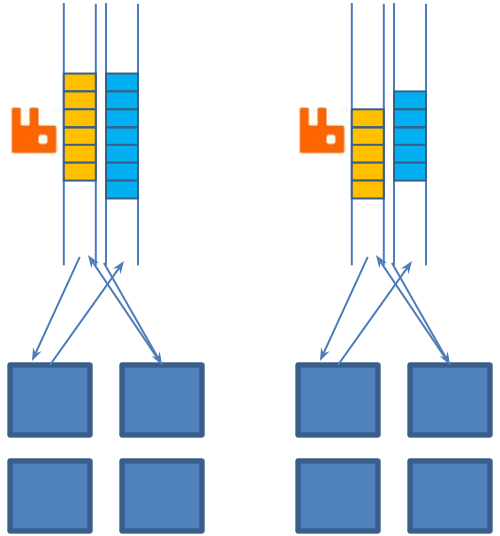



(подбираются

отве




(отсылаются

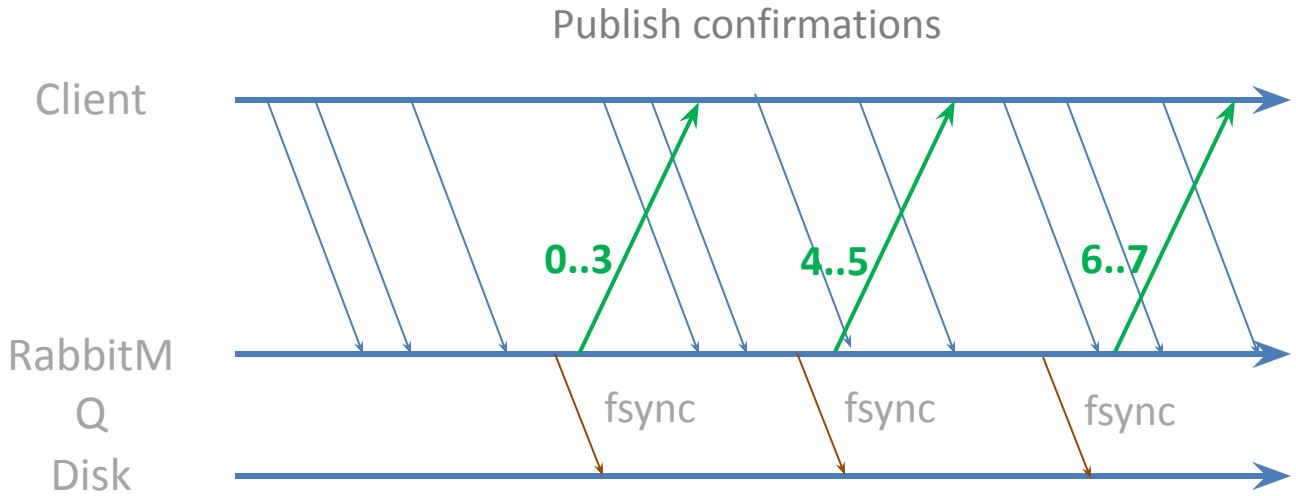


Демон выбирает случайного кролика

# Трудности

- Маленький лимит соединений у RabbitMQ под Windows (не потянет 500-1000 машин)
  - Решено: каждая машина подключается к кому-нибудь одному
- Не терять данные при крахах демонов
  - Решено: подтверждения тасков
- Переустанавливать соединение при случайных разрывах связи
- Не терять данные при крахах RabbitMQ
  - RabbitMQ не гарантирует безопасность данных вне транзакции!
- Не перегружать RabbitMQ
  - Иначе начинаются ужасы (тормоза, разрывы связи, крахи)
- Поддерживать асинхронные прерывания (немедленное переключение) без потери данных
  - Самая сложная часть
- Переключаться на другой RabbitMQ, если в этом кончились задачки (иначе

# Сохранность данных при крахах RabbitMQ

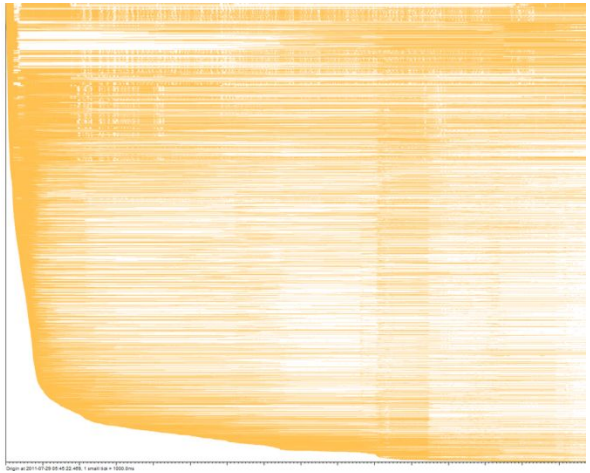



Клиент буферизует сообщения, про которые еще не известно, на диске ли они.

При разрыве связи повторить, возможно утерянные сообщения

# Не перегружать RabbitMQ

- Если слишком яро слать сообщения, RabbitMQ захлебнется (не успевая писать на диск)
- Тормоза, крахи, потеря соединения

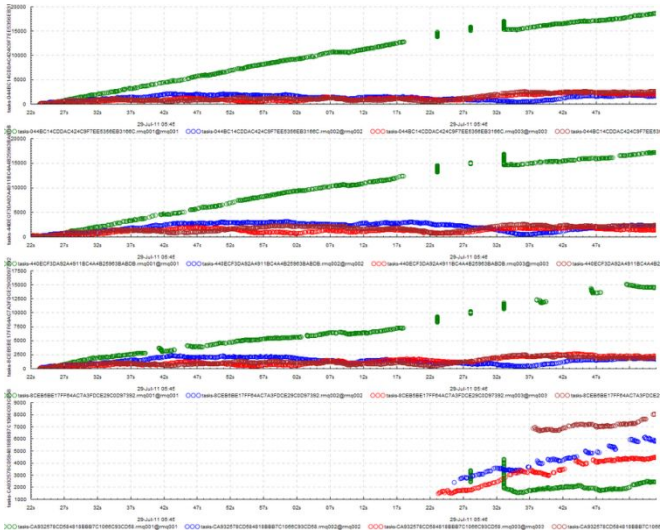


Белое – «ждем задачи»

доставка тормозит, или реконнектимся

# Не перегружать RabbitMQ

- Оказывается, отличная метрика загрузки – число/размер неподтвержденных сообщений



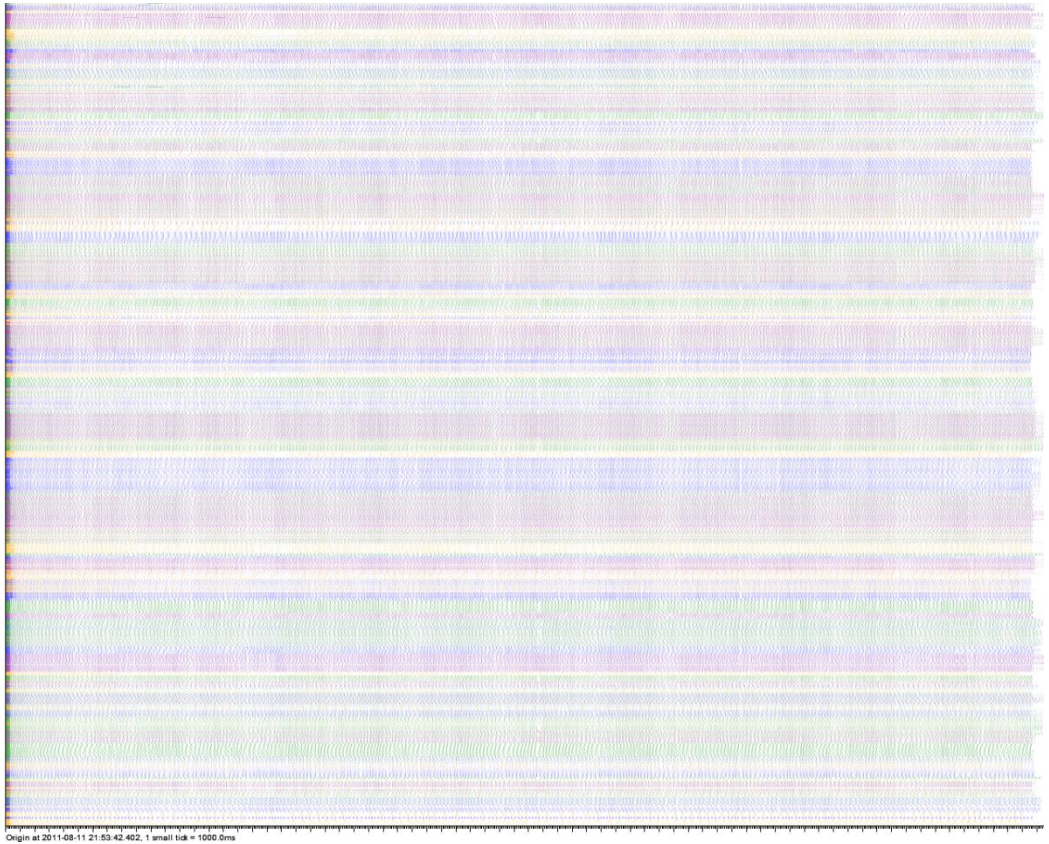
4 задачи, 4 разноцветных кролика

Один кролик не поспевает.

# Не перегружать RabbitMQ

Ограничить число сообщений «в полете»

- На каждого кролика?
- Нет, тогда один медленный будет всех тормозить
- А как тогда?
- Давать очередное сообщение случайному неперегруженному.




Около 5000 ядер, 4 RabbitMQ

Нет перегрузок – нет проблем

# Поддерживать асинхронные прерывания

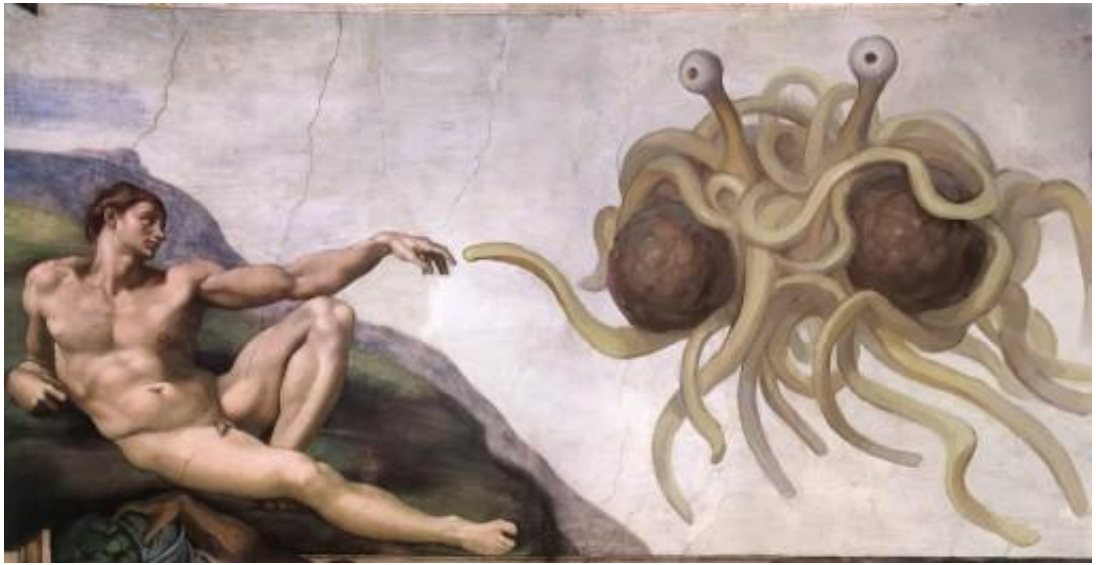
- Иногда надо всё бросить и заняться другой *задачей*
  - Прервать запущенную задачку и кинуть обратно в трубу
  - Или прервать ожидание задачки
  - Порвать соединения с трубами предыдущей задачи
    - Убедиться, что все ответики и отклоненные задачки *точно* сохранены на диск
- Об этом мы узнаем из другого потока
  - Многопоточность – это всегда ад
  - К счастью, это почти единственное использование многопоточности
  - Но все равно ад.

# Переключаться между кроликами

- Задаче досталось несколько демонов. 3 подключены к rmq1, 1 к rmq2
- Дисбаланс
- Голодание
- Нет задачек в нашем – переключимся на другой
  - А если он отвалился?
  - А если в нем тоже нет?
  - А если нигде нет? (избегать бури реконнектов)
  - Нельзя надолго создавать дисбаланс нагрузки на кроликов
  - Нужно найти того, где есть, как можно быстрее
- Решение есть, немножко хитрое, нет времени рассказать ☹

# Как это закодировать

- Можно сделать лапшу, делающую все сразу
  - Реконнекты, подтверждения доставки, переключение, балансировка, асинхронные прерывания…



HighLoad++

# Как не сойти с ума

Разумеется, слои*.

*Паттерны Adapter, Composite etc, они же Combinator Library

# Слои

API проще некуда:

- Отсыльщик:
  - Отослать
  - Получить/сбросить список неподтвержденных
  - Уничтожиться (возможно, асинхронно)
- Слушатель:
  - Достать сейчас (blocking + timeout)
  - Достать потом (callback)
  - Уничтожиться (возможно, асинхронно)

# Слои

«Игнорировать неподтвержденные при закрытии»

«При ошибке переоткрыться»

«Слушать сразу несколько»

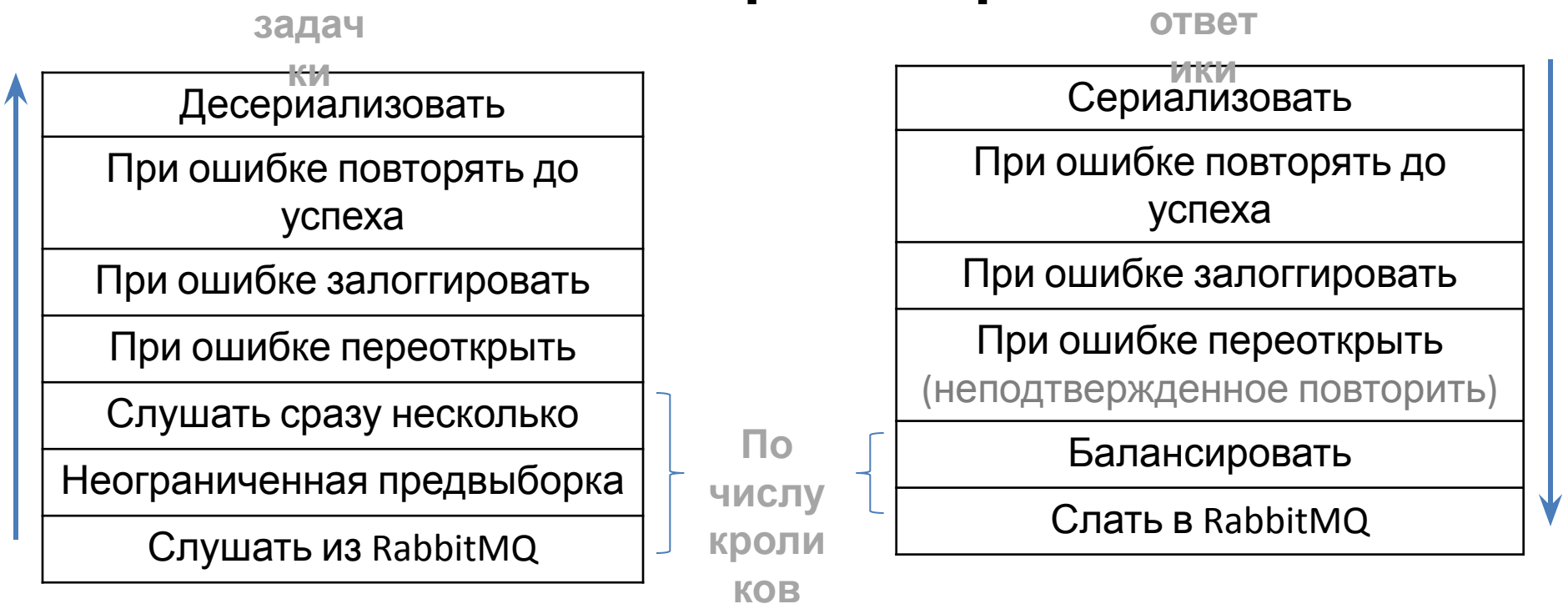
«При ошибке сделать то-то и то-то»


«Балансировать отправку между несколькими»

«Преобразовать тип сообщения»

«При ошибке попробовать еще раз»

# Например


задач
ки
Десериализовать
При ошибке повторять до успеха
При ошибке залоггировать
При ошибке переоткрыть
Слушать сразу несколько
Неограниченная предвыборка
Слушать из RabbitMQ
По числу кроликов
ответ
ики
Сериализовать
При ошибке повторять до успеха
При ошибке залоггировать
При ошибке переоткрыть
(неподтвержденное повторить)
Балансировать
Слать в RabbitMQ


HighLoad++

- Введение и Архитектура
- Доставка задач/результатов
- Отладка и анализ
- Обобщение опыта: корректность, надежность, производительность

# Отладка и анализ

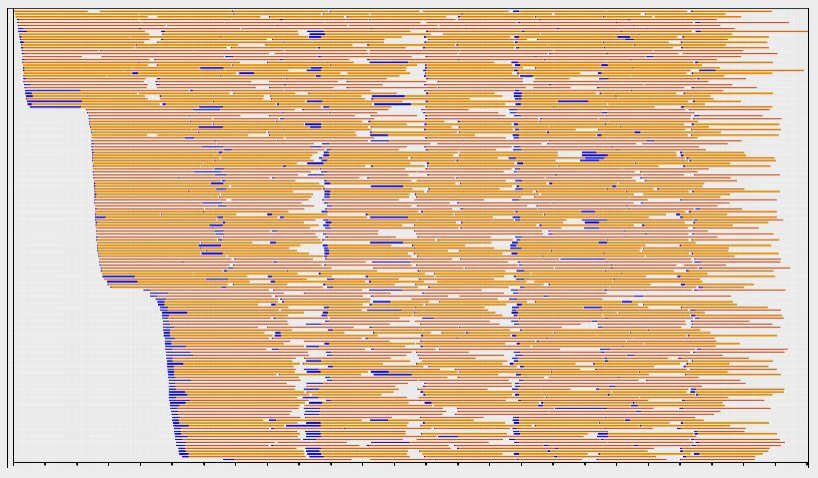
- Дебаггер – не вариант (только для локальных тестов)
  - Где и когда произойдет ошибка – заранее неизвестно
- Post mortem отладка по логам
- And you have to be pretty damn good at it
  - Это не логи вебсервера, где все реквесты независимы
  - Несколько взаимодействующих, иногда многопоточных подсистем
  - Проблемы с корректностью – недетерминированы
  - Проблемы с производительностью – не локальны
  - Логов, по нашим меркам, дофига (тысячи важных сообщений в

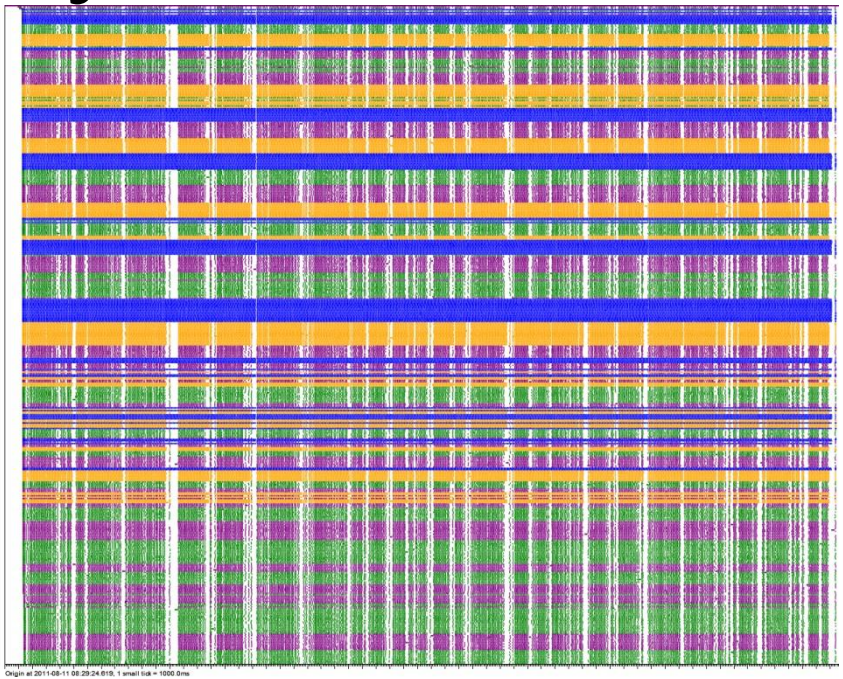
# Пара фокусов в рукаве

- Мощный логгер
  - Глобальная ось времени (точнее, чем NTP)
  - Тянет сотни тысяч сообщений в секунду от тысяч клиентов
  - http://code.google.com/p/greg – опенсорс-версия
  - Ставим 1шт. на кластер, получаем точную глобальную картину
    (без мучений со специальным сбором-слиянием логов)
- GNU textutils + awk (пока хватает, MapReduce не юзаем)
- timeplotters – две специальных рисовалки
  - http://ikiff.info/software/timeplotters/

HighLoad++

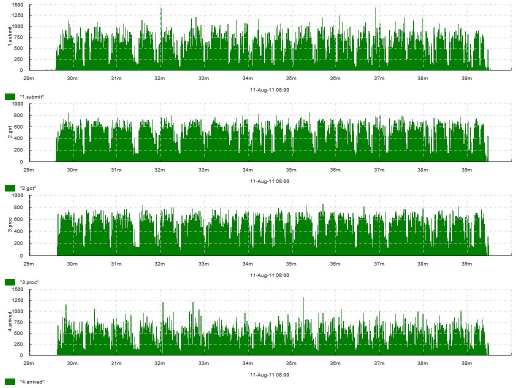
# Мы рисуем

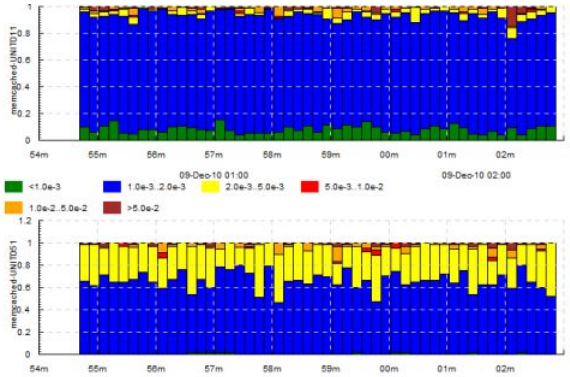


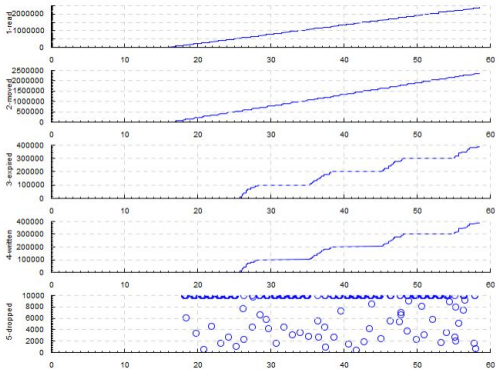


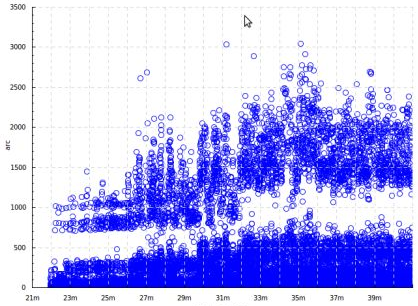
http://jkff.info/software/timeplotters/

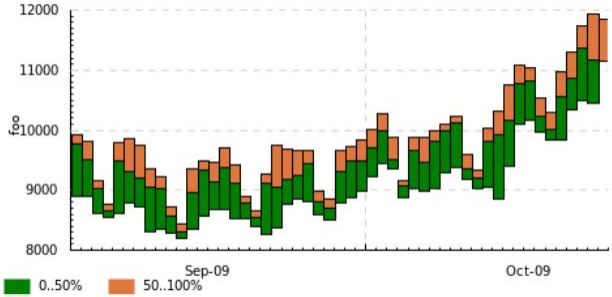
# HighLoad++








http://jkff.info/software/timeplotters/

# Что для этого нужно?

**Очень подробные логи.**

Еще об этом – позже.


HighLoad++

- Введение и Архитектура
- Доставка задач/результатов
- Отладка и анализ
- Обобщение опыта:
  корректность, надежность, производительность

# Корректность: Главный принцип

Как писать правильный код?

# Корректность: Главный принцип

Код *точно* неправильный.

Как быть?

# Как быть?

- Быть скромнее
- Дать себе шанс найти ошибку
- Минимизировать «ядро корректности»
- Минимизировать распространение ошибки
- Избегать опасных приемов

# Быть скромнее

- Не думать «ничего, отладим»
  - Это будет стоить вам увеличения времени разработки в разы
- Не лепить все фичи сразу:
  - Каждый раз приходится разломать и отлаживать по отдельности
- Безжалостно уничтожать некритичные фичи
  - Единственный их эффект – усложнение отладки

# Цитаты в тему

**«Write the simplest thing that could possibly work»**

**Ward Cunningham**

**«Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it»**

**Brian Kernighan**

# Дать себе шанс найти ошибку

- Максимально подробные логи
- Не бывает «слишком много логов»
- Не бывает «от логгирования код некрасивый»

# Минимизировать «ядро корректности»

Часть, от корректности которой зависит работоспособность системы.

## Веб-сервер:

- Неправильное вычисление в процессе обработки запроса  неправильный ответ
- Дедлок в пуле сокетов  виснет весь сервер

# Минимизировать распространение ошибки

Расставлять «барьеры»

- До барьера – будь что будет
- После барьера верны некоторые свойства
- Барьер должен быть очень надежен

# Барьеры

Уничтожение процесса (выполнение действия в отдельном процессе)

- Защищает от утечек ресурсов внутри процесса

Периодический перезапуск системы

- Защищает от неограниченно долгих зависаний

Закрытие соединения с очередью

- В худшем случае (неподтвержденная)задача будет сдублирована

Eventual consistency

- negative feedback, периодическая сверка желаемого и действительного

# Избегать опасных приемов

- Изменяемое состояние
- Многопоточность
- Блокирование, синхронизация
- Обратная связь
- Редко выполняющийся код


HighLoad++

- Введение и Архитектура
- Доставка задач/результатов
- Отладка и анализ
- Обобщение опыта:
  корректность, надежность, производительность

# Надежность

- Всё перезапускаемо и готово к перезапуску остальных
- Asynchronous one-way messaging (противоположность RPC)
- Явно формулировать переход ответственности за целостность данных
- Все компоненты готовы к дублям и потерям данных
- Eventual consistency

# Перезапускаемость

Если она есть:

- Можно перезапустить оборзевший процесс
- Можно навсегда забыть о редких крахах
- Можно перезапускать процесс периодически и забыть навсегда об утечках и зависаниях

Если ее нет:

- Надо вылизывать код, пока не исчезнут самые маловероятные крахи и утечки
- Если крах не по вашей вине (ОС, библиотека...) –
  это все равно ваши проблемы.

# Asynchronous, one-way messaging

Противоположность RPC, прямое следствие из перезапускаемости.

**Лучше возложить ответственность за доставку сообщений на софт, который хорошо умеет это делать.**

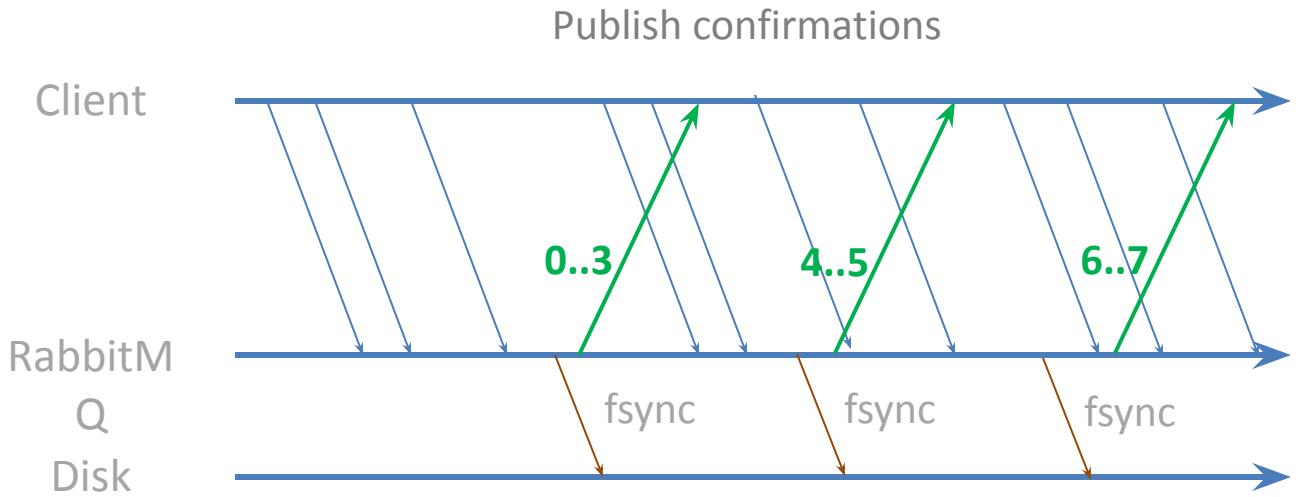
**Или использовать ненадежный транспорт (UDP).**


HighLoad++

# Eventual consistency

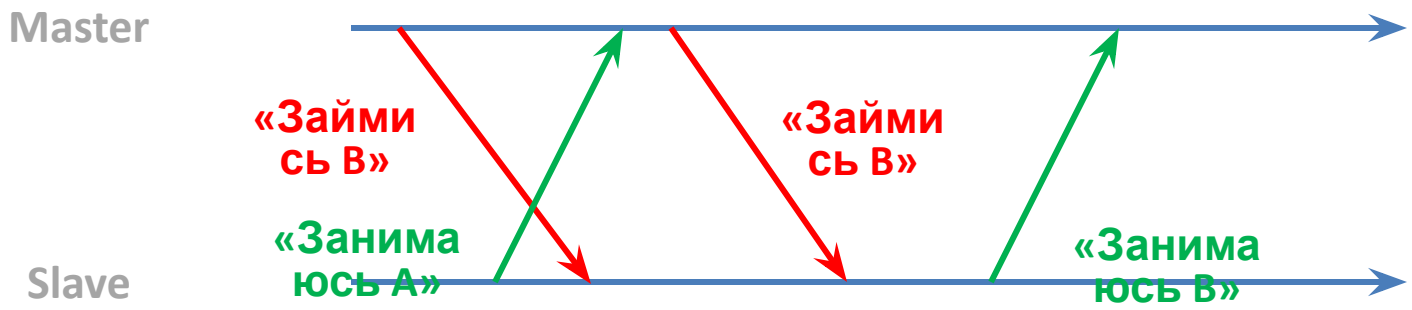
Стремление к согласованности

# Eventual consistency




Клиент и кролик постепенно согласуют знание о том, какие данные надежно сохранены

# Eventual consistency




Хозяин и раб постепенно согласуют представление о том, чем рабу надо заниматься


HighLoad++

- Введение и Архитектура
- Доставка задач/результатов
- Отладка и анализ
- Обобщение опыта:
  корректность, надежность, производительность

# Производительность

Несколько аспектов:

- Стабильность под нагрузкой
- Пропускная способность
- Задержка

# Главное

Ресурсы конечны

# Какие ресурсы конечны

Вот что кончалось у нас:

- Соединения с RabbitMQ
- Erlang-процессы в RabbitMQ
- Синхронные AMQP-операции / сек. (e.g. queue.declare) с RabbitMQ
- Установленные соединения / сек. с RabbitMQ
- Установленные соединения / сек. с логгером
- Внутренние буферы сообщений в логгере
- Место в пуле потоков (медленно разгребался)
- Одновременные RPC-вызовы
- Место на диске
- CPU и диск одной машины, куда погрузили два сервиса сразу
- Успешно проходящие UDP-пакеты по нагруженному каналу
- Транзакции RabbitMQ в секунду (чего уж там – в минуту)
- Терпение при анализе больших логов
- Память у инструментов рисования логов
- …

# Мораль

- Планируйте потребление ресурсов
- Особенно таких, потребление которых растет с масштабом
- Особенно централизованных
  - Центральные одноэкземплярные сервисы
  - Сеть
- Учитывайте паттерн загрузки!
  - Его бывает трудно предсказать
  - Наивные бенчмарки нерепрезентативны

# Пропускная способность

- Избегайте обратной связи
  - Из-за нее задержка начинает уменьшать пропускную способность
  - Задержку оптимизировать гораздо труднее

# Задержка

- Прогнозируйте и измеряйте
- Уменьшайте длину цепи задержки
- Избегайте компонентов с непредсказуемой задержкой
- Избегайте централизованных компонентов на пути запроса
- Не делите ресурсы между throughput-sensitive и latency-sensitive компонентами
  - Плохая идея использовать один и тот же RabbitMQ и для команд, и для задач
- Рано или поздно придется управлять приоритетами запросов/действий вручную
  - Понадобятся не просто очереди, а приоритетные очереди