Как сделать вычислительную инфраструктуру для большого кластера

Евгений Кирпичёв

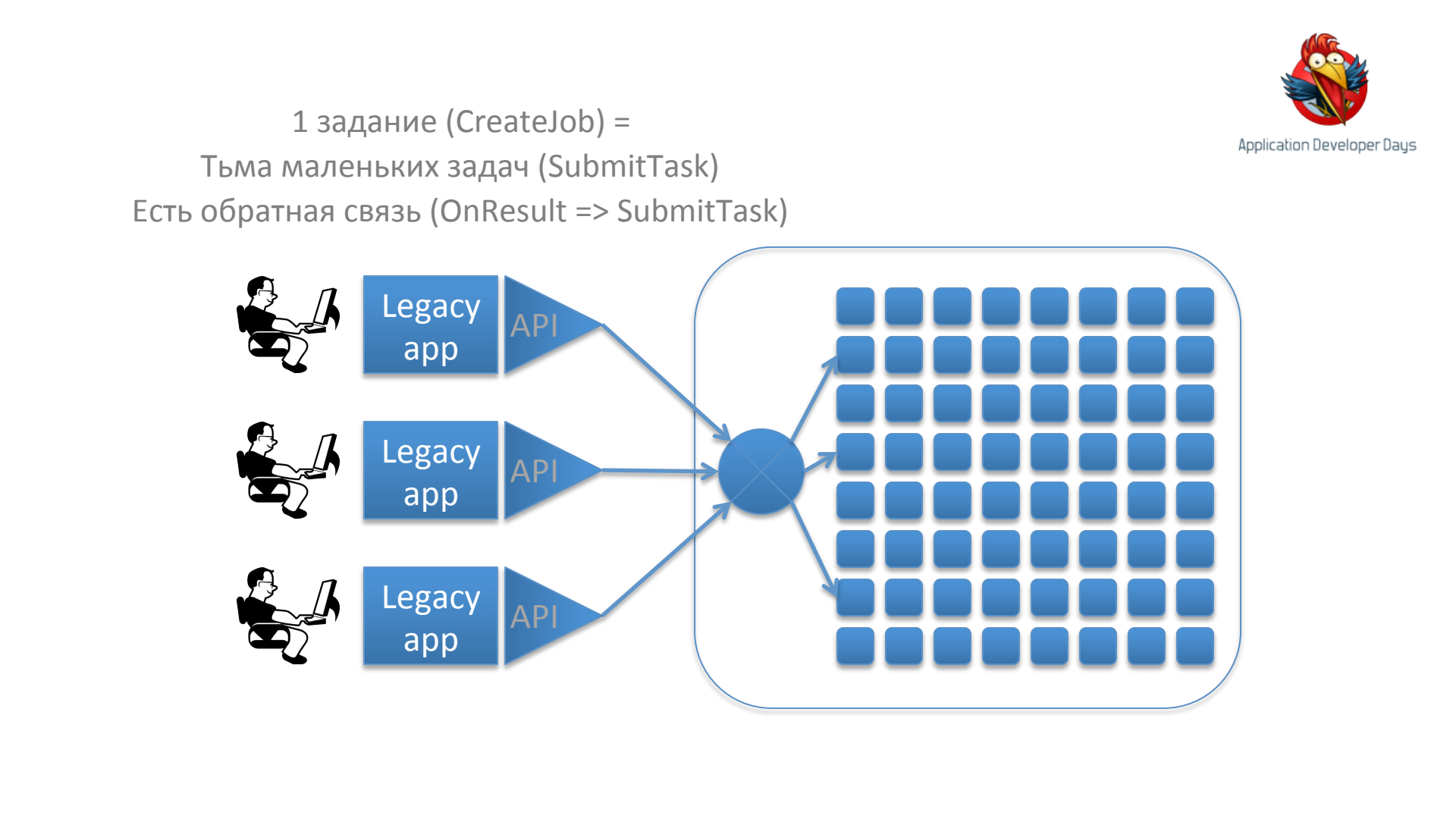
Станислав Лагун

MIRANTIS

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

Что мы строим

- Очень тяжелые вычисления *(но очень параллельные)*
- Много одновременных заданий (Job) и юзеров разной важности
- Простой API – задание = поток задач: *CreateJob, SubmitTask, OnResult*
  - Задание – атомарное и stateless однопоточное вычисление
    (напр. перемножить пару матриц)
- Непредсказуемые вычисления:
  От секунд *(нужна интерактивность)*
  до дней *(но не мешать чужой интерактивности)*
- Задействовать кластер целиком
- Отказоустойчивость *(смерть вычислителей, перезапуск системных сервисов)*


Application Developer Days
1 задание (CreateJob) =
Тьма маленьких задач (SubmitTask)
Есть обратная связь (OnResult => SubmitTask)
Legacy app
API
Legacy app
API
Legacy app
API

Обычно (на суперкомпьютерах) используют:

Platform LSF

MS HPC Server

Oracle Grid Engine

Condor

…

+

MPI

OpenMP

…

Это называется **«batch scheduler»**

Нам не подходит

Они предполагают:

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

Без этих ограничений можно сделать эффективнее.

Мы знаем: 1) задачи прерываемы 2) задачи независимы и массивно параллельны

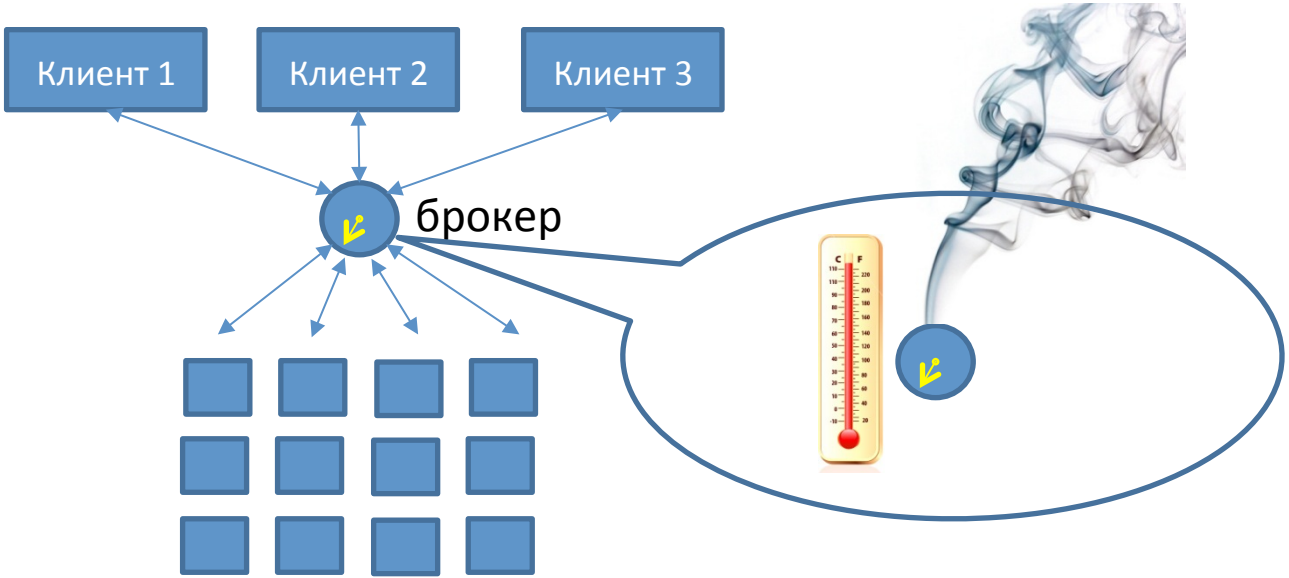
Application Developer Days

Пакетный планировщик	Наш планировщик
Приложение: «Дай-ка мне 300 – 400 ядер и запусти на них вот эту команду»	**Приложение:** Я хочу использовать кластер для вычисления задач
Планировщик: Вот ядра, команду запустил. Разбирайся сам.	**Планировщик:** Хорошо, кидай задачи 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 and NDA too strict…

Пример неудачной архитектуры



Очевидно, single bottleneck – не масштабируется

+балансировка нагрузки очень математически нестабильна

Более удачная архитектура

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

+клиенты

+статистика

+логгирование

+мониторинг

Пример

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

- Клиент шлет задачи в трубу А
- Демоны считают, шлют ответы
- Клиент собирает ответы, шлет новые задачи и т.п.

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

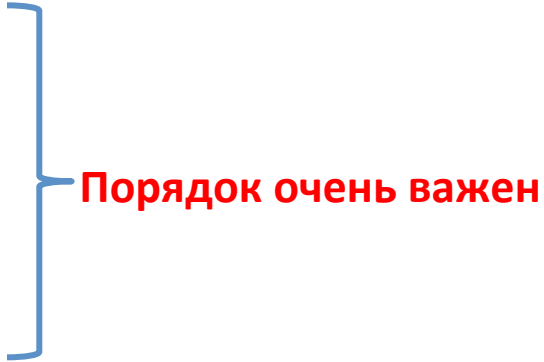
Трубы

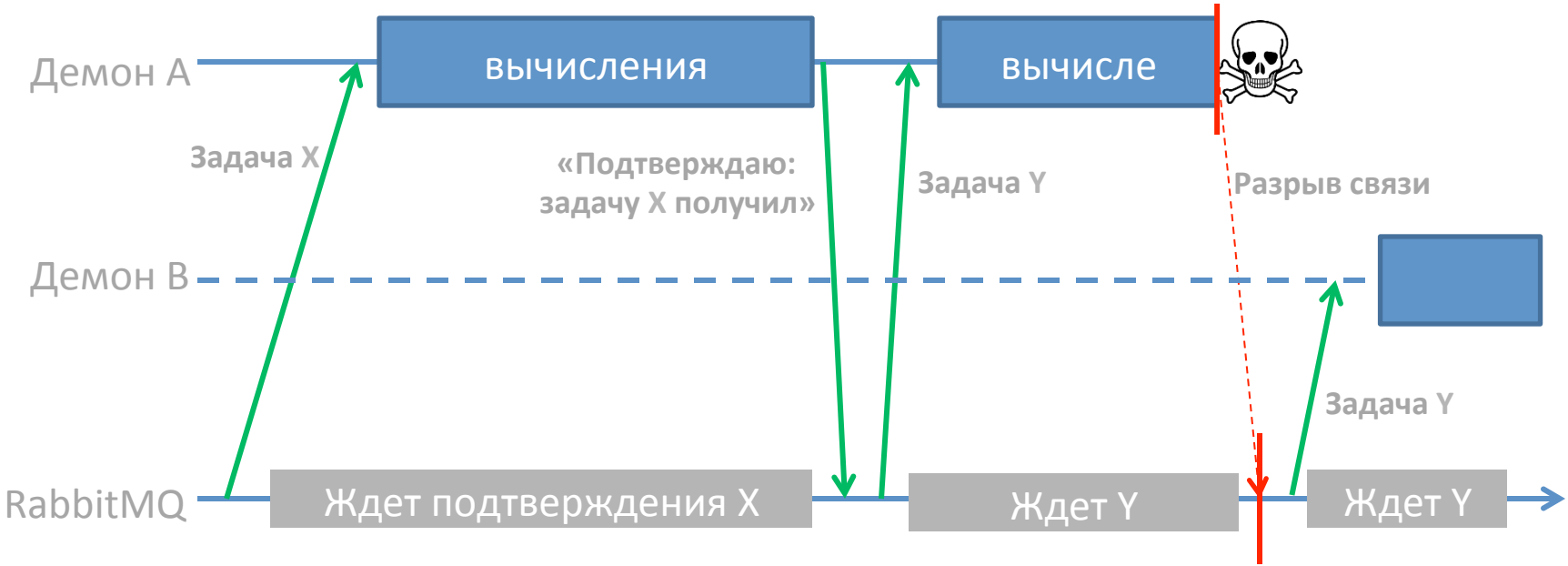
- На основе RabbitMQ
  - Лучший продукт в своем классе (надежная доставка)
  - Но «из коробки» сам по себе не масштабируется

Надежная доставка

Цикл жизни демона:

- Получить задачу
- Посчитать
- Отослать ответ
- Подтвердить получение задачи




Демон A
вычисления
вычисле
Задача X
«Подтверждаю:
задачу X получил»
Задача Y
Разрыв связи
Демон B
Задача Y
RabbitMQ
Ждет подтверждения X
Ждет Y
Ждет Y

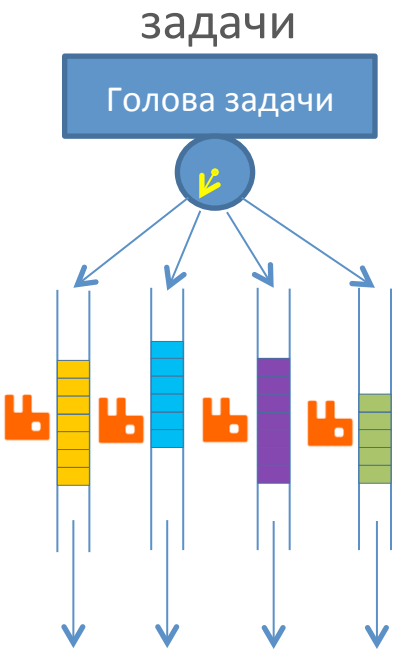
Масштабирование

Из коробки RabbitMQ совсем не подходит

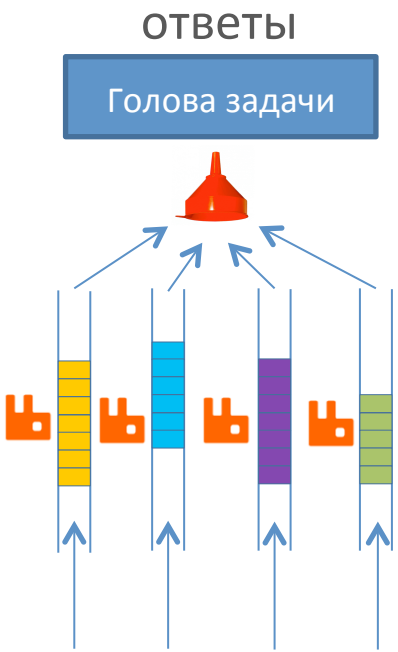
- Одна очередь плохо тянет 10000 клиентов
- Встроенная кластеризация делает не то

  От нее вообще лучше отказаться (одни проблемы)
- Очевидное решение: несколько очередей + load balancing

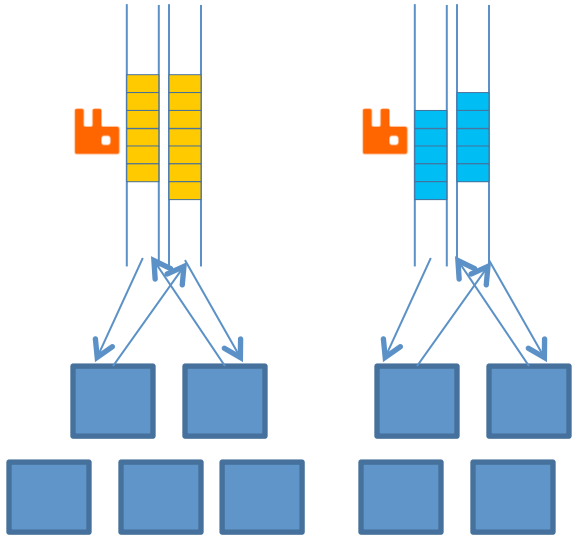
Масштабирование



(подбираются демонами)



(отсылаются демонами)



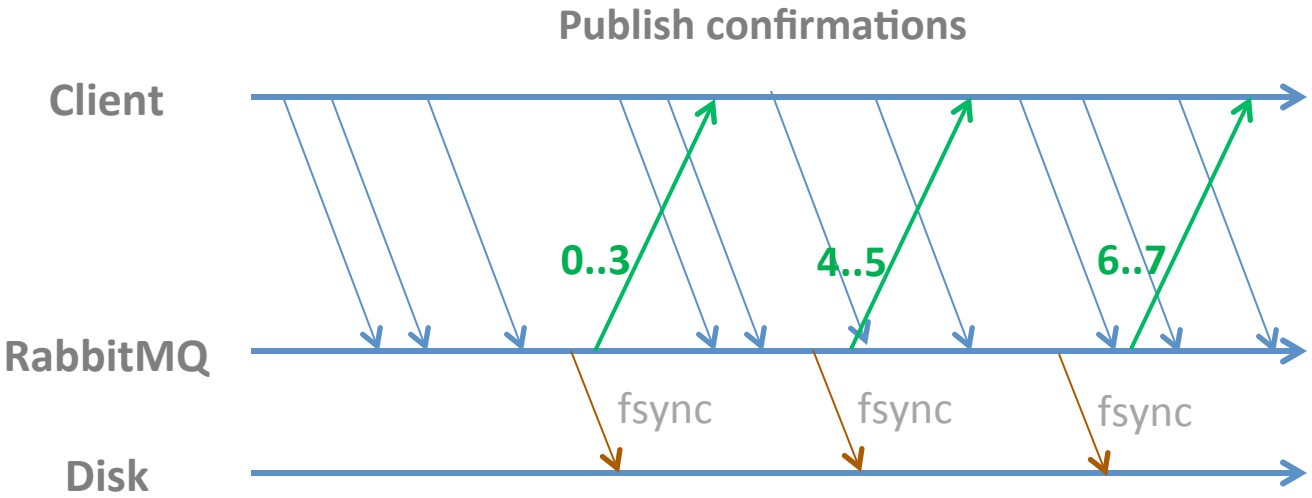
Демон выбирает случайного кролика
и всю жизнь* работает с ним

Трудности работы с очередями

- **RabbitMQ под Windows тянет мало соединений**
  - Решено: каждая машина подключается к кому-нибудь одному
- **При крахах демонов данные теряются**
  - Решено: подтверждения тасков
- **При крахах RabbitMQ данные теряются**
  - RabbitMQ не гарантирует безопасность данных вне транзакции!
- **Соединение может спонтанно рваться**
- **RabbitMQ дохнет под нагрузкой**
  - Надо избегать перегрузки

- **Нужно мгновенное переключение между заданиями (бросай всё и берись за задачу А)**
  - Самая сложная часть
- **В очереди на конкретном RabbitMQ могут кончиться задачи**
  - Надо переключаться, не нарушая остальных свойств

Сохранность данных при крахах RabbitMQ



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

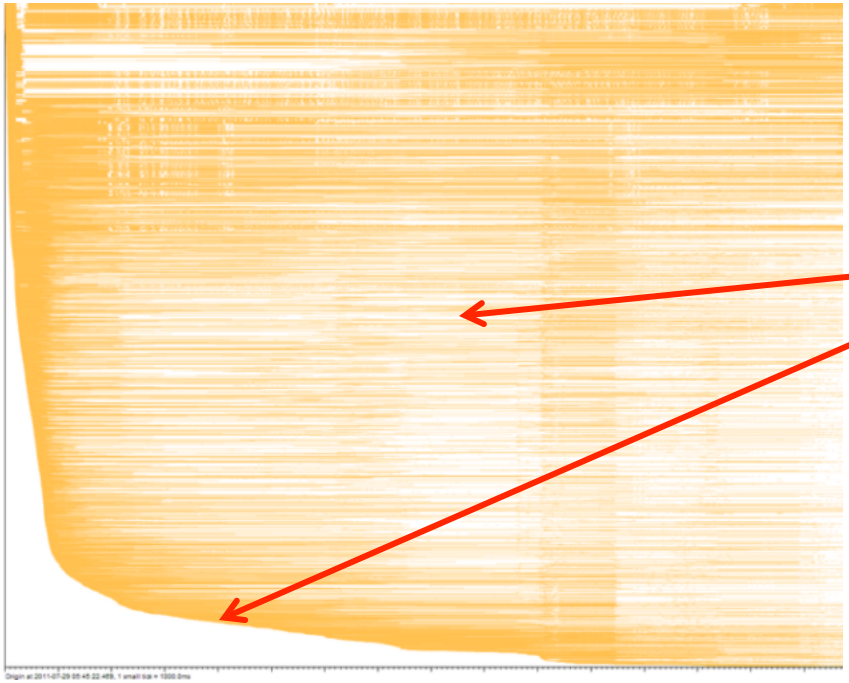
При разрыве связи перепосылает их.

Это универсальный паттерн в распределенных системах (stream replication).

Не перегружать RabbitMQ

- Если слишком интенсивно слать сообщения, RabbitMQ захлебнется (не успевая писать на диск)
- Тормоза, крахи, разрывы соединений

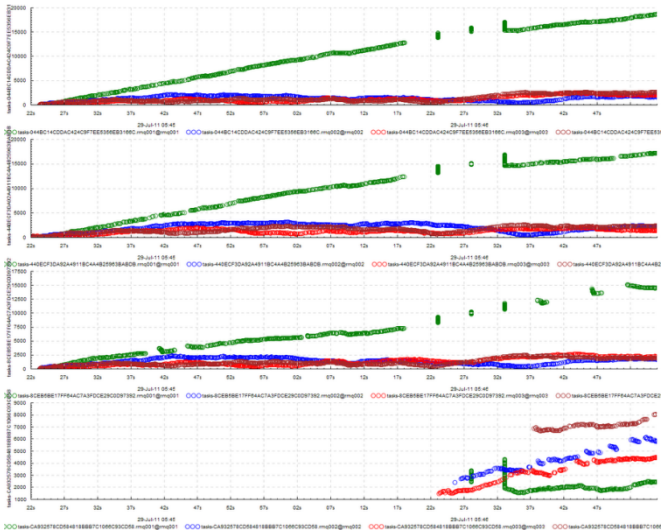
Не перегружать RabbitMQ



Белое – «ждем задачи»
доставка тормозит,
или реконнектимся

Не перегружать RabbitMQ

Оказывается, отличная метрика загрузки –
число неподтвержденных сообщений



4 задания, 4 разноцветных кролика

Один кролик не поспевает.

Не перегружать RabbitMQ

Ограничить число сообщений «в полете»

- Ждать при roundrobin, пока текущему кролику полегчает?
- Нет, тогда один медленный будет всех тормозить.
- А как тогда?
- Давать очередное сообщение случайному неперегруженному.

Поддерживать асинхронные прерывания

Иногда надо всё бросить и заняться другим заданием

- Прервать запущенную задачу и кинуть обратно в трубу
- Или прервать ожидание завершения задачи
- Порвать соединения с трубами предыдущего задания
  - Убедиться, что все ответы *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
- На 5000 ядрах быстро проявляются ВСЕ многопоточные баги

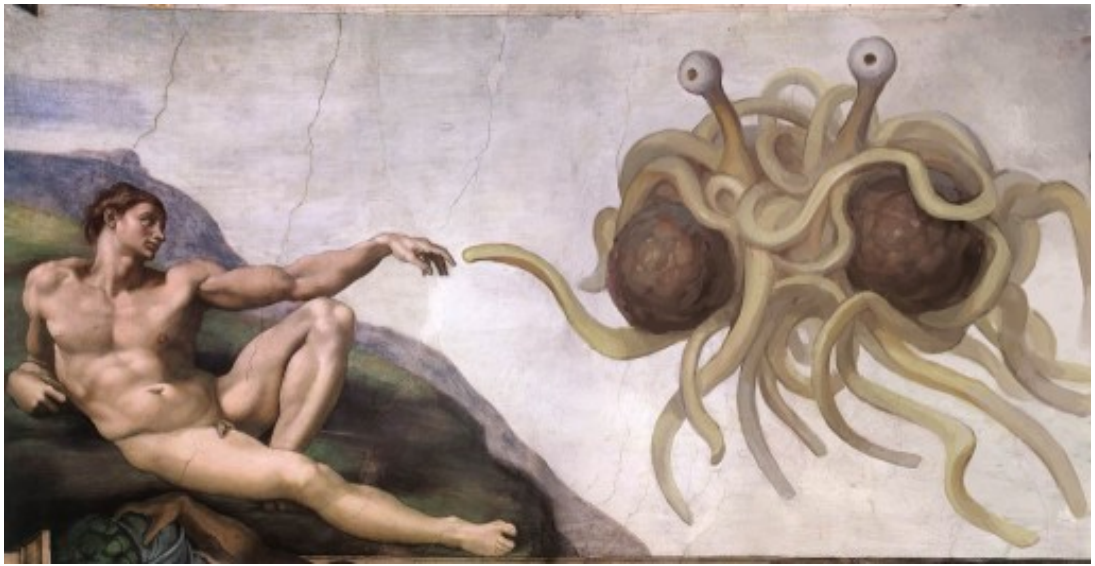
Переключаться между кроликами

- Если кончились задачи в нашем – постучаться в другого
- Если делать наивно, будет куча проблем
  - Бури реконнектов
  - Голодание
  - Дисбаланс
  - Кончатся соединения
  - ...
  - NO PROFIT!!!11

Как это закодировать

Можно сделать лапшу, делающую все сразу

- Реконнекты, подтверждения доставки, переключение, балансировка, асинхронные прерывания…



Как не сойти с ума

Разумеется, слои.

Слои

Отсыльщик:

- Отослать
- Получить/сбросить список неподтвержденных
- Завершиться (возможно, асинхронно)

Слушатель:

- Достать сейчас (blocking + timeout)
- Достать потом (callback)
- Завершиться (возможно, асинхронно)

Слои

«Игнорировать неподтвержденные при закрытии»

«Балансировать отправку между несколькими»

«При ошибке переоткрыться»

«Слушать сразу несколько»

«Преобразовать тип сообщения»

«При ошибке сделать то-то и то-то»

«При ошибке попробовать еще раз»

Например

задачи

Сериализовать
При ошибке повторять до успеха
При ошибке залоггировать
При ошибке переоткрыть (неподтвержденное переслать)
Балансировать
Слать в RabbitMQ

По числу кроликов

ответы

Десериализовать
При ошибке повторять до успеха
При ошибке залоггировать
При ошибке переоткрыть
Слушать сразу несколько
Неограниченная предвыборка
Слушать из RabbitMQ

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

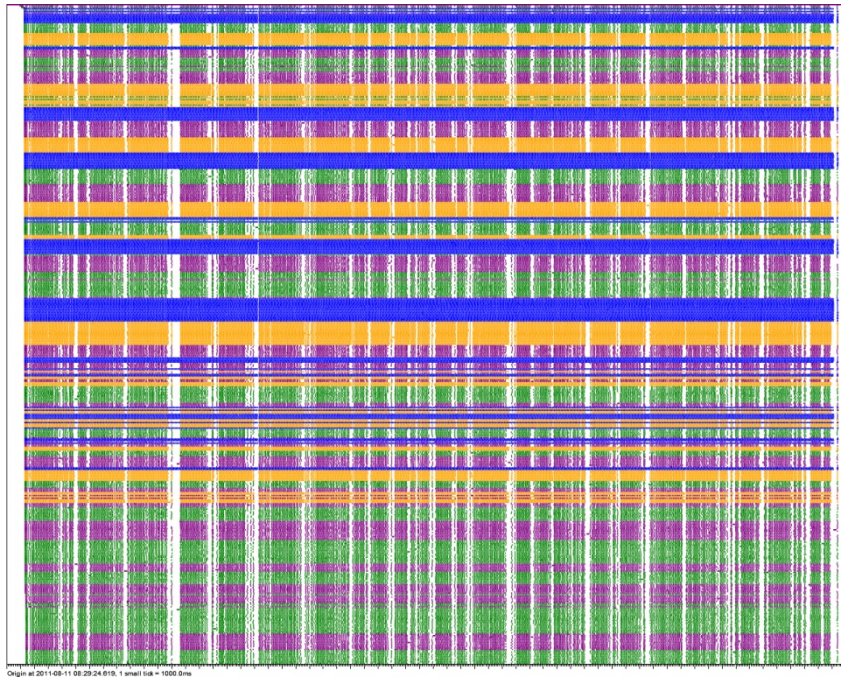
Отладка и анализ

- Дебаггер – не вариант (только для локальных тестов)
- Проблемы тоже распределенные
  - Особенно проблемы производительности
- Post mortem отладка по логам
- Логов, по нашим меркам, ~~дох~~ очень много

  (тысячи важных сообщений в сек., в пике до сотен тысяч)

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



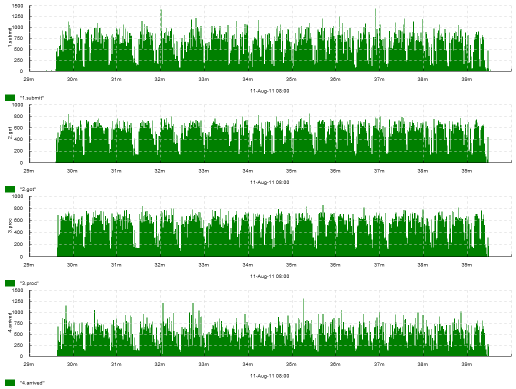
Application Developer Days

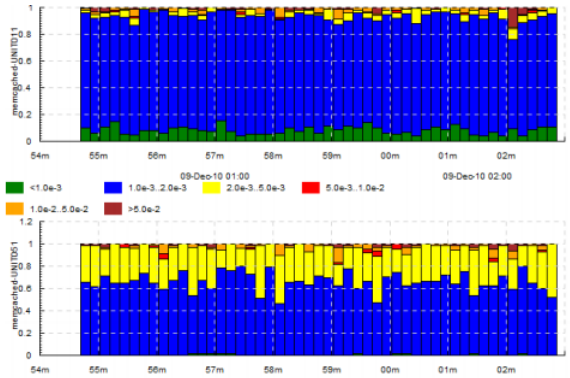
timeplotters

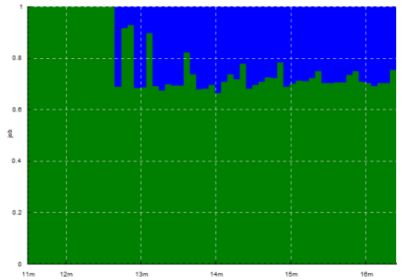
две специальных рисовалки

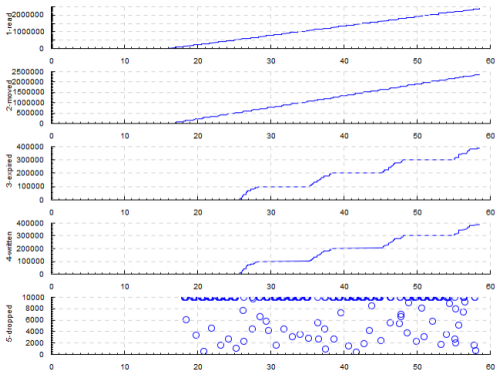
A picture is worth a thousand words

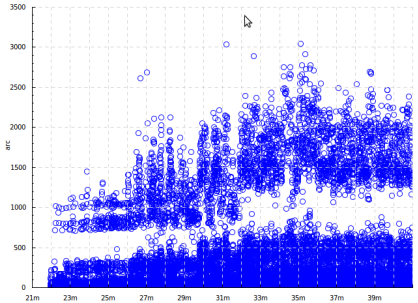
http://jkff.info/software/timeplotters/

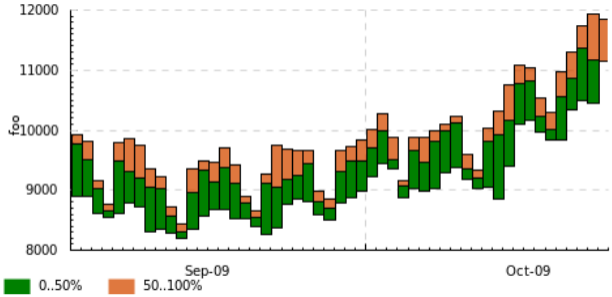












http://jkff.info/software/timeplotters/

Что для этого нужно?

Очень подробные логи.

Еще об этом – позже.

- 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?

- Писать максимально подробные логи
- Писать меньше кода
  (только то, без чего программа не будет работать)
- Минимизировать «ядро корректности»
- Избегать многопоточности и других опасных приемов
  (никакого геройства)
- Использовать eventual consistency
  (не настаивать на strong consistency)
- Использовать «барьеры»

Барьеры

Переводят любое состояние в предсказуемое.

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

- Введение и Архитектура
- Доставка задач/результатов
- Отладка и анализ
- Обобщение опыта:

  корректность, надежность, производительность

Надежность

- Всё перезапускаемо и готово к перезапуску или смерти остальных
  - Инфраструктурные сервисы реплицируют состояние и выбирают «лидера»
- Только асинхронная коммуникация
- Все компоненты готовы к дублям и потерям сообщений
- Явно формулировать переход ответственности за целостность данных
- Eventual consistency

  (система не обязана быть согласованной постоянно)

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, железо, уборщица) – это все равно ваши проблемы.

Application Developer Days

Асинхронные коммуникации

- С синхронными труднее обрабатывать ошибки,
  есть больше классов ошибок.
- Можно использовать софт, *хорошо* разбирающийся
  в асинхронных коммуникациях.

- Лучше использовать connectionless протоколы (UDP):
  исчезает как класс проблема «разрыв связи».
  (если задача позволяет)

Eventual consistency

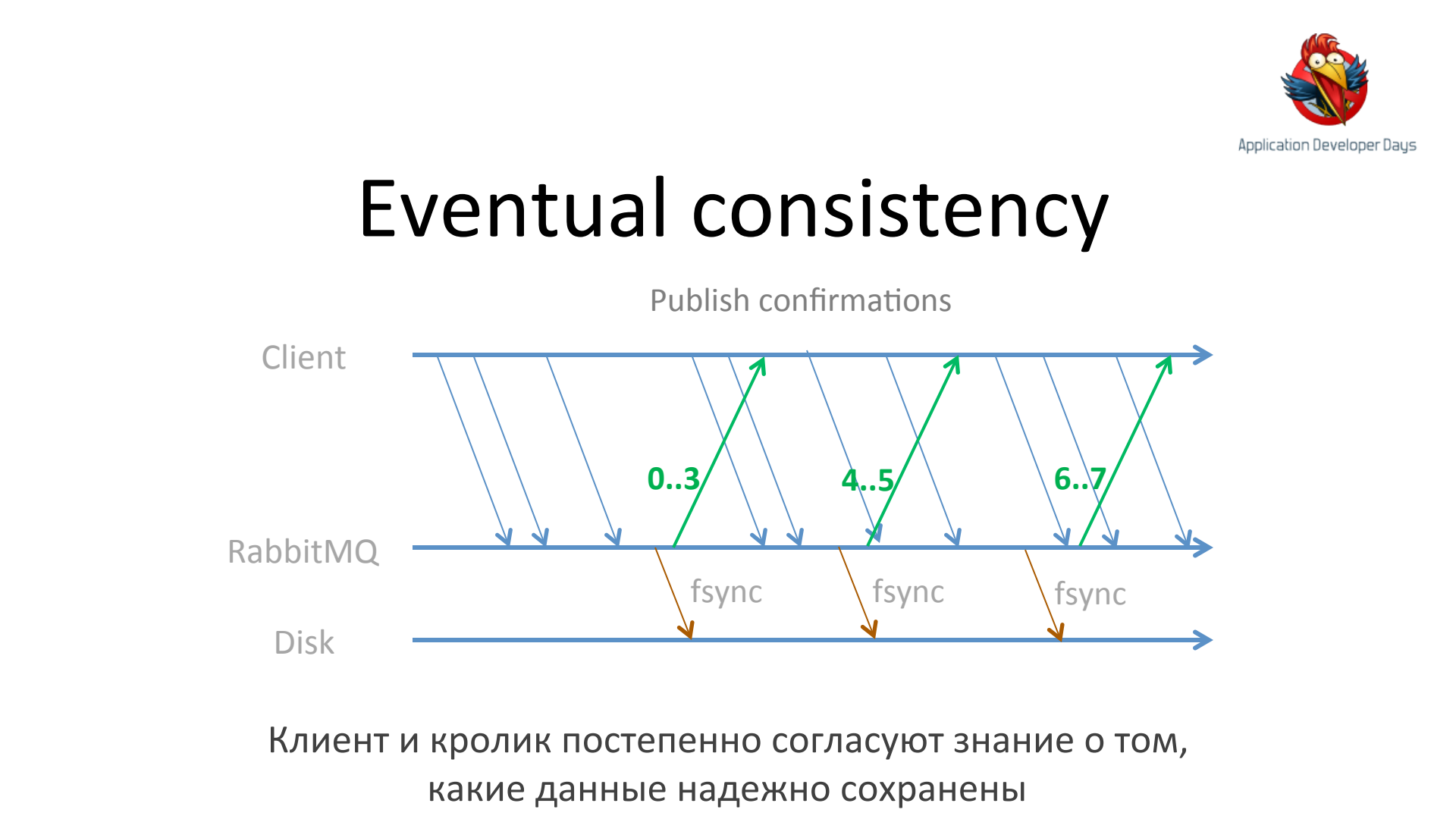
Стремление к согласованности.

Строгая глобальная согласованность

трудна

далеко не всегда нужна

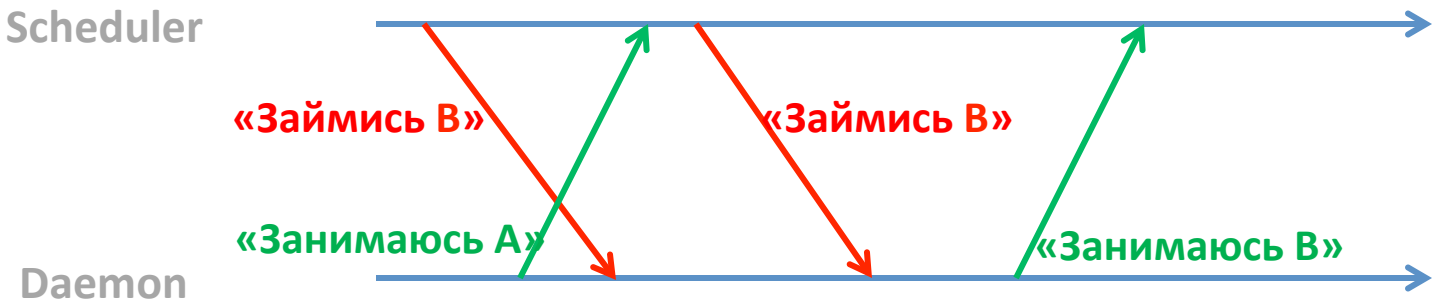
и далеко не всегда возможна



Eventual consistency

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

Eventual consistency



Планировщик и тысячи демонов постепенно согласуют представление о том, чем демонам надо заниматься

- 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

Что кончалось у нас

Соединения с RabbitMQ

Erlang-процессы в RabbitMQ

Синхронные AMQP-операции / сек. (e.g. queue.declare)

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 машины, куда погрузили два сервиса сразу

Успешные UDP-пакеты по нагруженному каналу

Транзакции RabbitMQ в секунду / 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 (в т.ч. сеть)
- Учитывайте характер нагрузки!
  - Его бывает трудно предсказать
  - Наивные бенчмарки нерепрезентативны

Пропускная способность

Избегайте зависимости от обратной связи

Иначе задержка начинает уменьшать пропускную способность
(печальный пример – TCP)

Задержку оптимизировать гораздо труднее

Задержка

- Измеряйте
- Избегайте централизованных компонентов на пути запроса
- Не делите ресурсы между batch и real-time компонентами
- Рано или поздно придется управлять приоритетами запросов/ действий вручную
  - Понадобятся не просто очереди, а приоритетные очереди

That’s all folks.
Поставьте мне оценку :)

*

* Это не наше, но похоже.