CoreHard C++ Winter 2017

# Шишки, набитые за 15 лет использования акторов в C++

Евгений Охотников

# Откуда ноги растут?

SObjectizer:

- фреймворк для C++11;
- поддерживает Actor Model, Publish-Subscribe, CSP;
- OpenSource, BSD-3-Clause лицензия;
- кросс-платформа (Linux, FreeBSD, Windows, MacOS).

Одна из немногих живых и развивающихся OpenSource-реализаций Модели Акторов для C++.

В работе с 2002-го года.

# Для солидности...

С первого же дня в реальной разработке. В том числе:

- электронная и мобильная коммерция;
- мобильный банкинг;
- агрегация SMS/USSD-трафика;
- имитационное моделирование;
- тестовые стенды для проверки ПО АСУ ж/д транспорта;
- прототипирование распределенной системы сбора измерительной информации.

*Часть систем до сих пор в активной эксплуатации.*

# О чем пойдет речь?

О граблях, подводных камнях и набитых шишках.

Вряд ли что-то новое для Erlang-еров.

Но среди C++ников едва ли много Erlang-еров.

Да и C++ – это не Erlang, есть своя специфика.

# Совсем коротко о Модели Акторов

В основе всего три простых принципа:

- актор – это сущность, обладающая поведением;
- акторы реагируют на входящие сообщения;
- получив сообщение, актор может:
  - отослать некоторое (конечное) количество сообщений другим акторам;
  - создать некоторое (конечное) количество новых акторов;
  - определить для себя новое поведение для обработки последующих сообщений.

Далее "актор" == "агент" и наоборот.

# Акторы в C++ двумя словами: это здорово!

Конкретнее:

- у каждого агента свое изолированное состояние (принцип shared nothing), что упрощает жизнь в многопоточном коде;
- обмен сообщениями – естественный подход к решению некоторых типов задач;
- слабая связность между компонентами;
- очень простая работа с таймерами (отложенные и периодические сообщения);
- низкий порог входа для новичков.

# НО!

Здорово только когда Модель Акторов хорошо ложится на задачу.

Ложится далеко не всегда хорошо.

# Тем не менее...

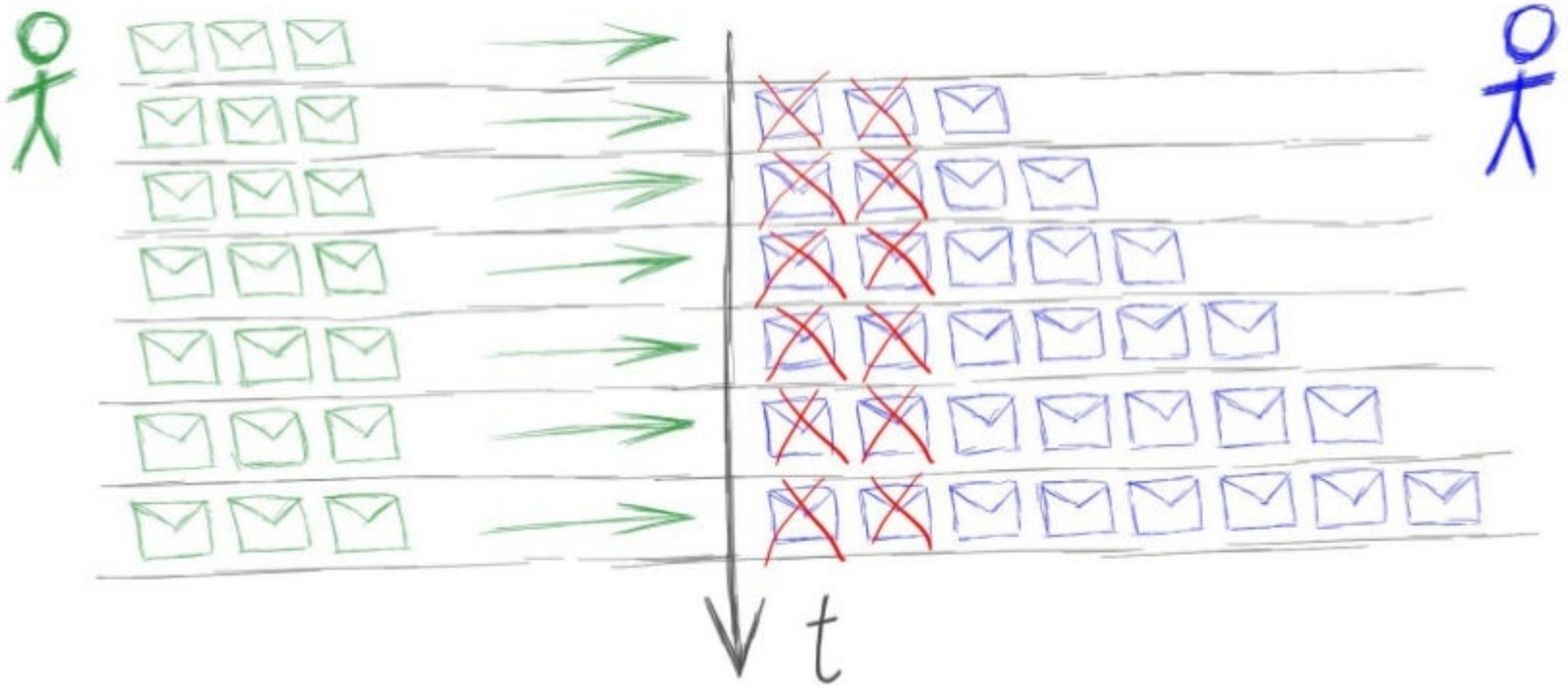
Если Модель Акторов подходит для решения задачи, то правильный фреймворк упростит жизнь еще больше.

*Очевидно, что правильный всего один: название начинается с SObj... ;)*

# Грабли №1

Перегрузка агентов

# Что такое перегрузка агентов?



# Откуда берется перегрузка агентов?

Back-pressure на асинхронных сообщениях – это не просто. Да и не очень надежно.

Агент-отправитель сообщения может не иметь представления о том, насколько загружен получатель.

Еще хуже, когда агентов-оправителей несколько. И они разделяют общую рабочую нить.

Еще хуже, когда агент отправляет сообщение самому себе...

# Очевидно, что агентов нужно защищать, но...

Хороший механизм защиты от перегрузки должен быть "заточен" под конкретную задачу.

# Что хорошо заработало у нас?

Пара агентов: collector + performer.

Обязательно работают на разных нитях.

Агент-collector накапливает сообщения и реализует нужную политику защиты от перегрузки, например:

- выбрасывание самых "свежих";
- выбрасывание самых "старых";
- выбор другого режима обработки и т.д.

Агент-performer забирает сообщения у collector-а. Пачками.

# Хочется чего-то готового "из коробки"

Мы добавили в SObjectizer механизм message limits:

```
class collector : public so_5::agent_t {
public :
  collector(context_t ctx, so_5::mbox_t quick_n_dirty)
    :  so_5::agent_t(ctx
        // Запроса get_status достаточно всего одного.
        + limit_then_drop<get_status>(1)
        // Лишние запросы будут передаваться другому агенту,
        // который работает более грубо, но быстро.
        + limit_then_redirect<request>(50, [quick_n_dirty]{ return quick_n_dirty; } )
        // Если же не успеваем отдавать накопленное, то работать
        // дальше не имеет смысла.
        + limit_then_abort<get_messages>(1))
    ...
};
```

# limit_then_drop

```
class collector : public so_5::agent_t {
public :
   collector(context_t ctx, so_5::mbox_t quick_n_dirty)
      :  so_5::agent_t(ctx
            // Запроса get_status достаточно всего одного.
            + limit_then_drop<get_status>(1)
            // Лишние запросы будут передаваться другому агенту,
            // который работает более грубо, но быстро.
            + limit_then_redirect<request>(50, [quick_n_dirty]{ return quick_n_dirty; } )
            // Если же не успеваем отдавать накопленное, то работать
            // дальше не имеет смысла.
            + limit_then_abort<get_messages>(1))
      ...
};
```

# limit_then_redirect

```
class collector : public so_5::agent_t {
public :
   collector(context_t ctx, so_5::mbox_t quick_n_dirty)
      :  so_5::agent_t(ctx
            // Запроса get_status достаточно всего одного.
            + limit_then_drop<get_status>(1)
            // Лишние запросы будут передаваться другому агенту,
            // который работает более грубо, но быстро.
            + limit_then_redirect<request>(50, [quick_n_dirty]{ return quick_n_dirty; } )
            // Если же не успеваем отдавать накопленное, то работать
            // дальше не имеет смысла.
            + limit_then_abort<get_messages>(1))
      ...
};
```

# limit_then_abort

```
class collector : public so_5::agent_t {
public :
  collector(context_t ctx, so_5::mbox_t quick_n_dirty)
    :  so_5::agent_t(ctx
        // Запроса get_status достаточно всего одного.
        + limit_then_drop<get_status>(1)
        // Лишние запросы будут передаваться другому агенту,
        // который работает более грубо, но быстро.
        + limit_then_redirect<request>(50, [quick_n_dirty]{ return quick_n_dirty; } )
        // Если же не успеваем отдавать накопленное, то работать
        // дальше не имеет смысла.
        + limit_then_abort<get_messages>(1))
    ...
};
```

# Грабли №2

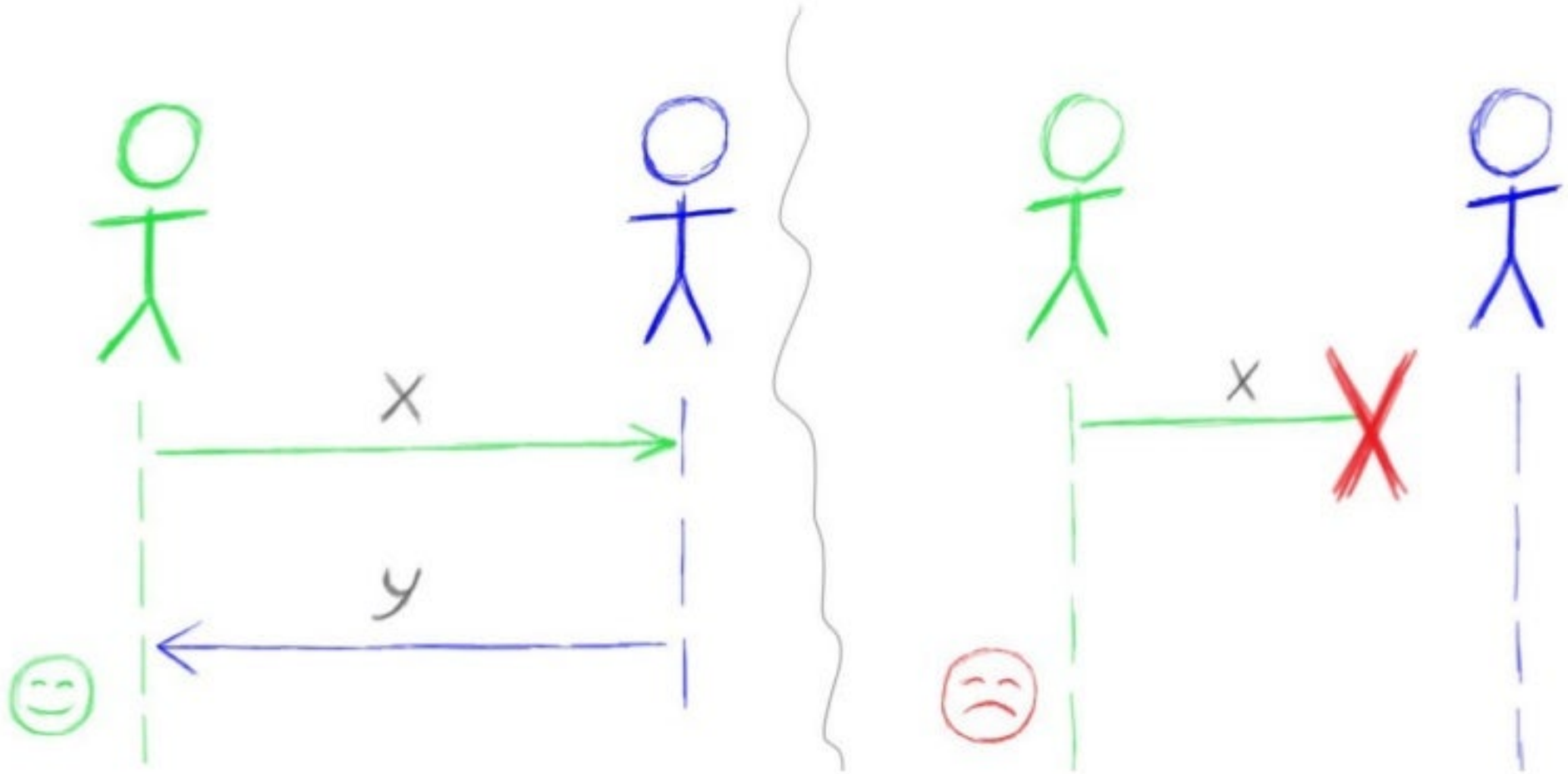
Доставка сообщения не гарантирована

# Сообщение не обязательно будет обработано

Ибо:

- получателя может и не быть. Например, успел исчезнуть из системы;
- получатель есть, но само сообщение ему не интересно;
- получатель есть, сообщение ему интересно, но сработал механизм защиты от перегрузок...

# Чем это может грозить?



# Что делать?

Пара самых простых способов:

1. Перепосылка после тайм-аута.
   Однако, можно спровоцировать перегрузку!
2. Откат операции после тайм-аута. Или выставление статуса "результат неизвестен".

# Забавный парадокс:

Казалось бы, отсутствие гарантий доставки сообщения не способствует написанию надежных приложений...

Однако, на практике оказывается наоборот.

Если прикладные агенты умеют переживать потерю сообщений, то они зачастую справляются с запланированными и незапланированными нештатными ситуациями.

# Грабли №3

Коды ошибок или исключения

# Немного истории

SObjectizer-4, 2002-й год, нет исключений, есть коды возврата.

Практика показала: коды ошибок не способствуют написанию надежного кода.

SObjectizer-5, 2010-й год, используются исключения.

Положительное влияние на надежность. По крайней мере мне так кажется.

# Вопрос почти на миллион:

Что делать с исключениями, которые вылетели из агента наружу?

Два важных фактора:

- исключение ловит SObjectizer и он не знает, что означает исключение;
- можно попробовать доставить исключение отправителю сообщения, но:
  - отправителя может уже не быть;
  - отправителю это не нужно;
  - до отправителя эта информация может не дойти (доставка сообщений не гарантируется).

# exception_reaction в SObjectizer

В SObjectizer есть возможность указать, что делать, если агент выпускает наружу исключение:

- abort_on_exception
- shutdown_sobjectizer_on_exception
- deregister_coop_on_exception
- ignore_exception

Вариант deregister_coop_on_exception + механизм dereg_notificator позволяет реализовать механизм супервизоров, если в этом есть необходимость.

# C++ не настолько безопасен, как Erlang

В Erlang есть изоляция процессов. В C++ нет никакой изоляции между агентами.

Если агент выполнит деление на ноль, то в C++ упадет не только этот агент.

Если агент в C++ оставит какой-то мусор в памяти, то это повлияет на все приложение.

Агенты в C++ должны обеспечивать хотя бы базовый уровень гарантии безопасности исключений. Ибо деструкторы для них будут вызываться нормальным образом.

# C++ не прощает стиль "тяп-ляп"

Принцип "let it crash" в C++ выглядит сильно иначе, чем в Erlang. По меньшей мере о базовой гарантии нужно заботиться...

Естественным образом разработчик приходит к стилю, когда:

- обеспечивается nothrow-гарантия;
- любое выскочившее из агента исключение должно убивать все приложение.

# Грабли №4

Народ просит ~~хлеба и зрелищ~~
синхронности

# Из непонятого...

Краеугольный камень Модели Акторов – это асинхронность.

Практически все бонусы проистекают именно из этого.

Бери да пользуйся (используй то, что под рукою и не ищи себе другое)...

Но, нет.

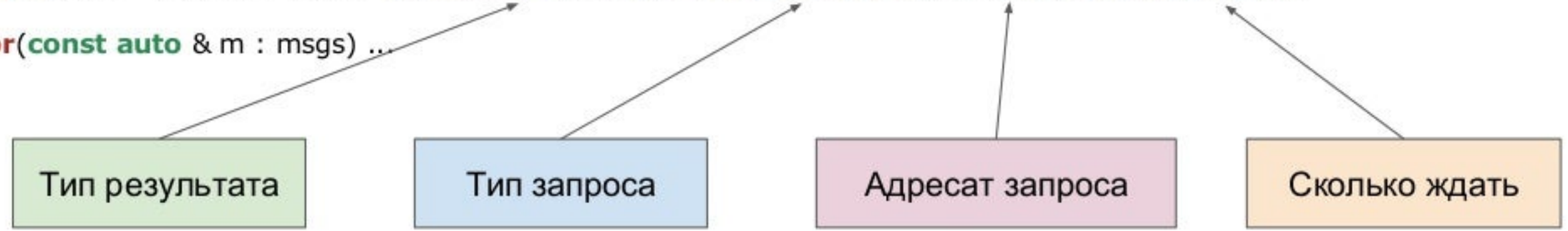
Почему-то народ просит синхронности.

# Ну раз просит :)

Инициация синхронного запроса:

```
struct get_messages : public so_5::signal_t {};
...
auto msgs = request_value<std::vector<message>, get_messages>(mbox, so_5::infinite_wait);

for(const auto & m : msgs) ...
```



# Ну раз просит :)

Обработка синхронного запроса:

```
class collector : public so_5::agent_t {
public :
  ...
  virtual void so_define_agent() override {
    so_subscribe(mbox).event<get_messages>(&collector::on_get_messages);
    ...
  }
private :
  std::vector<messages> collected_messages_;

  std::vector<messages> on_get_messages() {
    std::vector<messages> r;
    std::swap(r, collected_messages_);
    return r;
  }
};
```

# Под капотом

```
struct special_message : public so_5::message_t {
  std::promise<std::vector<messages>> promise_;
  ...
};

collector * collector_agent = ...;
auto actual_message_handler = [collector_agent](special_message & cmd) {
  try {
    cmd.promise_.set_value(collector_agent->on_get_messages());
  }
  catch(...) {
    cmd.promise_.set_exception(std::current_exception());
  }
};

do_special_subscribe<get_messages, special_message>(mbox, actual_message_handler);
```

# request_value: за и против

Использование request_value чревато возникновением deadlock-ов.

request_value следует использовать с большой осторожностью.

request_value оказался очень удобен для реализации общения collector-ов и performer-ов.

# Грабли №5

Распределенность "из коробки":
плюсы и минусы

# В SObjectizer-4 распределенность была

У нас был свой протокол поверх TCP/IP, свой способ сериализации C++ных структур данных.

SObjectizer занимался маршрутизацией, (де)сериализацией, контролем I/O операций, переподключением при разрывах...

Посредством несложных телодвижений получались распределенные приложения.

Все было хорошо, пока нагрузки были небольшими, а трафик – однородным.

# В чем проблема? (1)

Под каждый тип взаимодействия хорошо бы иметь свой протокол.

Обмен большими бинарными блоками – это одно. Например, передача кусков аудио-, видеоданных, архивов и т.д.

Обмен телеметрией – другое. Например, передача мелких сообщений с результатами замеров температуры воздуха.

Еще интереснее, когда в одном канале смешивается трафик с разными приоритетами…

# В чем проблема? (2)

Back pressure для асинхронного взаимодействия – это непросто.
В случае IPC – это еще более непросто.

Задержки в сети или подтормаживание узлов и сразу:

- большие объемы непереданных сообщений;
- перепосылки (нужные и не нужные);
- значительные потери при разрывах каналов.

# В чем проблема? (3)

В современных больших распределенных приложениях используется далеко не только C++.

Интероперабельность с другими языками must have.

Кастомные протоколы, заточенные исключительно под C++ усложняют жизнь, а не упрощают ее.

# В итоге в SObjectizer-5 распределенности нет

От поддержки распределенности "из коробки" в SObjectizer-5 мы отказались.

Наш интерес в том, чтобы упростить агентам общение с внешним миром посредством де-факто стандартных средств:

- AMQP, HTTP, MQTT,...
- JSON, Protobuf, Thrift,...

# Грабли №6

Много агентов – это проблема,
а не решение.
SEDA-вэй форева!

# Акторы легковесны...

...поэтому их можно создавать тысячами, десятками тысяч, миллионами, десятками миллионов.

Такие слова можно отыскать в маркетинговых материалах практически всех реализаций Модели Акторов (а мы что, лысые что ли?)

Действительно, создать миллион акторов не сложно.

От этой возможности поначалу съезжает крыша.

# Эффект птичьей стаи

Каждый агент ведет себя по простым и понятным правилам.

Понять, что происходит в программе с 10K агентами сложно.

Предсказать, как поведет себя программа с 10K агентами – еще сложнее.

# Интроспекция важна

Иногда бывает важно понять, что происходит с конкретным агентом. Насколько он загружен? Сколько времени тратит?

В Erlang-е есть готовые инструменты для интроспекции.

В мире C++ все не так радужно. В том числе потому, что зачастую C++ инструментарий разрабатывается на чистом энтузиазме.

# Внезапные всплески активности

В какой-то момент может возникнуть столько сообщений, сколько прикладные агенты не смогут "переварить".

Допустим:

- 100K агентов, каждый инициирует отложенное сообщение;
- за короткое время срабатывает 10K таймеров;
- на обработку каждого такого сообщения требуется 10ms;
- 100s при обработке в один поток, 25s – в четыре потока, 10s – в десять параллельных потоков.

# Одно другому не мешает

Внезапные всплески активности очень хорошо сочетаются с эффектом птичьей стаи.

Добавляем сюда отсутствие интроспекции.

Желание создавать 10K агентов резко сменяется желанием обойтись всего сотней агентов. Еще лучше – десятком.

# SEDA-way

SEDA: Staged Event-Driven Architecture

Начало 2000-х. Исследовательская работа маленького коллектива.

Развитие давным-давно прекратилось.

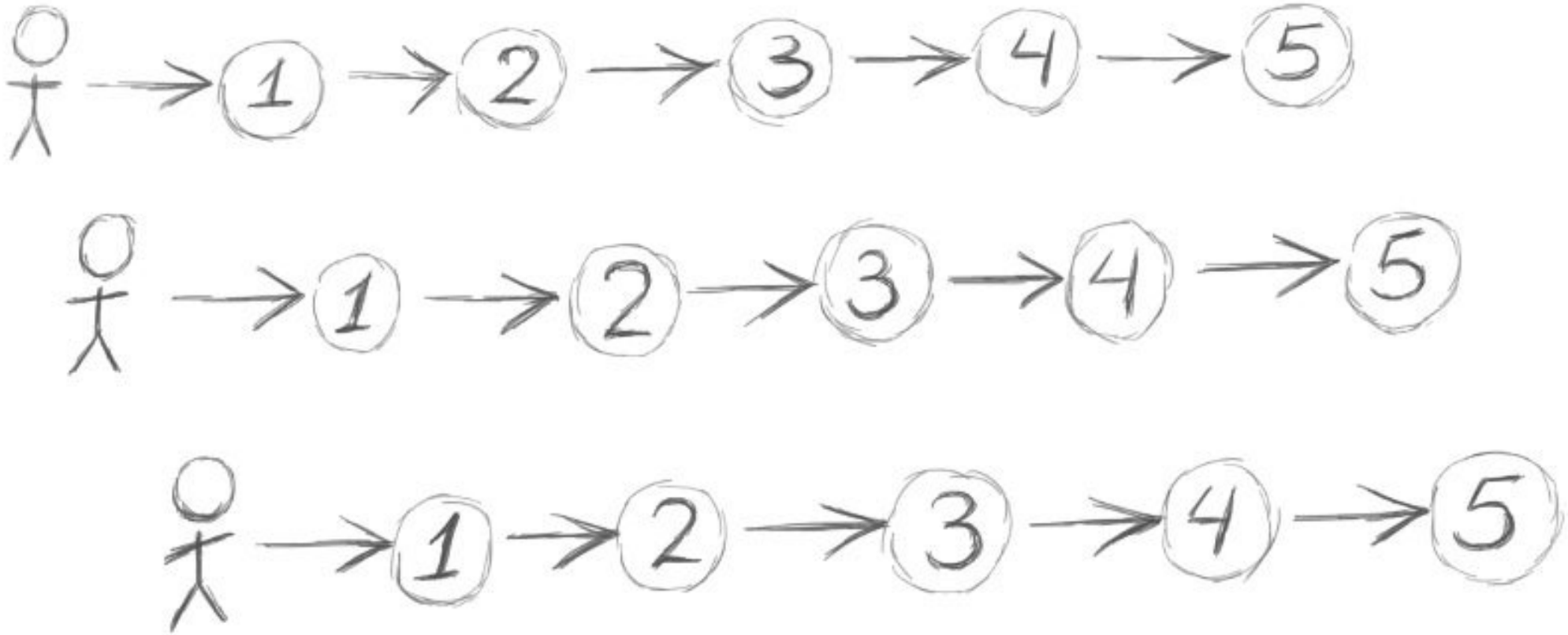
Но идеи более чем здравые.

# Пример с потолка: обслуживание платежа

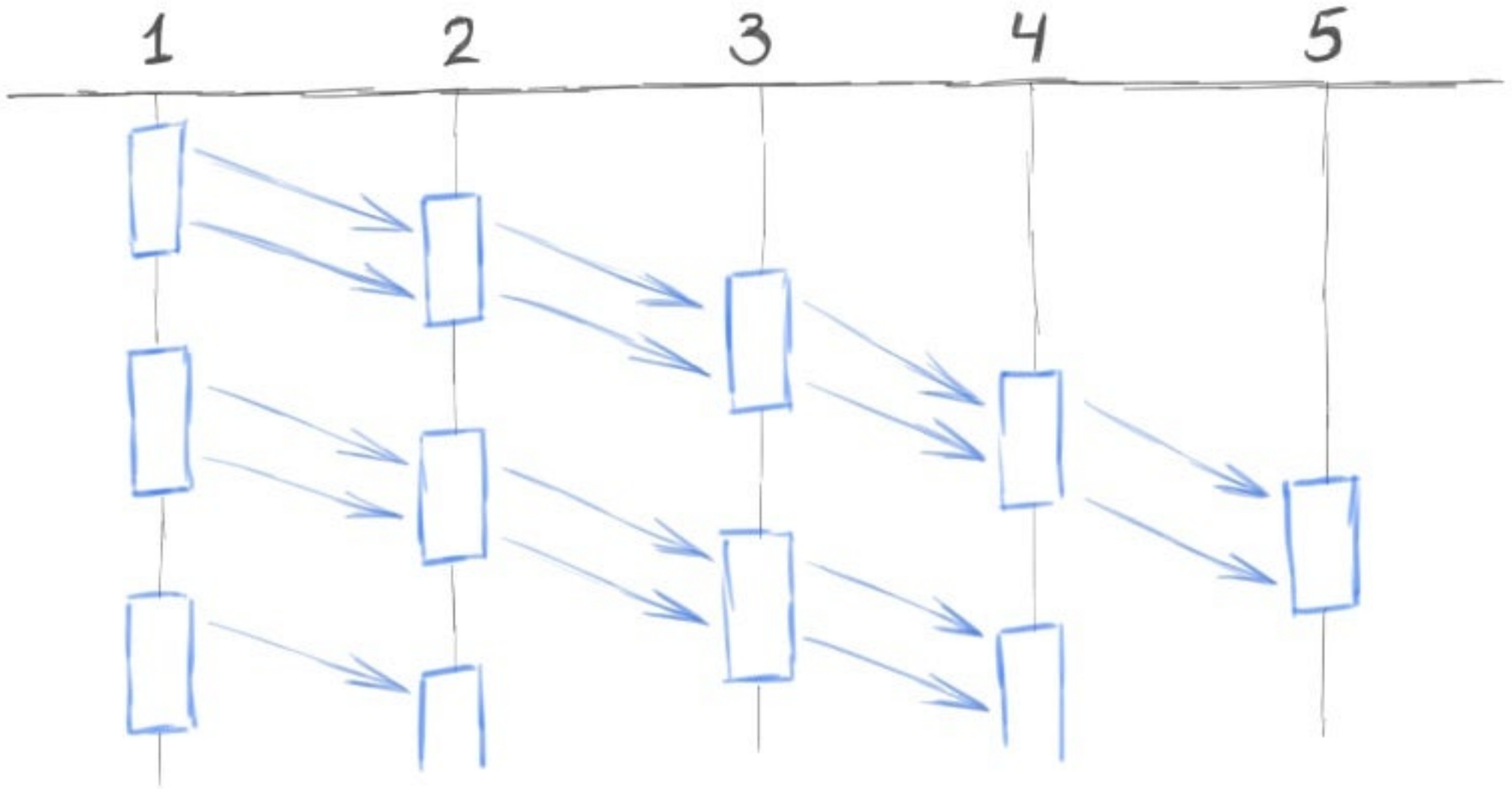
Совсем грубо и не точно, но наглядно:

1. Получить запрос.
2. Проверить корректность параметров платежа.
3. Проверить возможность платежа для клиента (например, превышение суточных лимитов).
4. Проверить рисковость платежа (анти-фрод).
5. Провести списание средств.

# Обслуживание платежа: решение "в лоб"



# Обслуживание платежа: в стиле SEDA



# Обслуживание платежа: бонусы от SEDA-стиля

Упрощается контроль и мониторинг.

Упрощается защита от перегрузок.

При работе СУБД появляется возможность использовать bulk-операции.

Появляется возможность дозирования активности агентов.

# Уже почти все...

# Модель Акторов отнюдь не серебряная пуля

Для успешного применения нужны:

1. Голова на плечах прикладного разработчика. Это редкость, но все же...
2. Набор батареек в акторном фреймворке:
   - для защиты акторов от перегрузок;
   - для обнаружения и преодоления ошибок;
   - для интроспекции (а также сбора статистики и мониторинга);
   - для упрощения отладки и пр.

Мы постепенно оснащаем SObjectizer такими батарейками, но это небыстрый и непростой процесс.

# Маленький совет

При выборе акторного фреймворка обратите внимание на то, что идет в комплекте.

Интересные идеи и красивые примеры – это хорошо. Но стоит недорого.

Дополнительные инструменты, вроде средств для интроспекции и мониторинга – вот это показатель качества и зрелости.

Ну и делать свой акторный фреймворк для C++ сейчас не самая хорошая идея.

# Спасибо за терпение!

Вопросы?

Документация по SObjectizer: https://sourceforge.net/p/sobjectizer/wiki/Home/

Серия статей о SObjectizer на русском:

SObjectizer: что это, для чего это и почему это выглядит именно так? От простого к сложному: Часть I, Часть II, Часть III. Акторы в виде конечных автоматов – это плохо или хорошо? Проблема перегрузки агентов и средства борьбы с ней. Нежная дружба агентов и исключений.

Серия презентаций о SObjectizer на английском "Dive into SObjectizer-5.5":

Intro, Agent's States, More About Coops, Exceptions, Timers, Synchonous Interaction, Message Limits, Dispatchers, Message Chains.

Блог автора доклада: eao197.blogspot.com

О SObjectizer в блоге: eao197.blogspot.com/search/label/SObjectizer

Почта автора доклада: eao197 на gmail тчк com