C++ CoreHard Autumn 2018

Actors vs CSP vs Tasks...

Евгений Охотников

Коротко о себе

Профессиональный разработчик с 1994-го.

- 1994-2000: АСУ ТП/SCADA и т.п.;
- 2001-2014: телеком, платежные сервисы;
- 2014-...: OpenSource-инструментарий для C++.

Автор блога "Размышлизмы eao197" (http://eao197.blogspot.com/)

Сооснователь stiffstream.com ⇛ (SObjectizer, RESTinio)

Автор ряда докладов про Модель Акторов и C++.

Начнем с банальностей

Многопоточность – это зло

Не грех повторить еще раз:

Многопоточное программирование на C++ посредством голых нитей, mutex-ов и condition variables – это **пот**, **боль** и **кровь**.

Можно не верить мне...

https://habr.com/company/pixonic/blog/426875/

Архитектура мета-сервера мобильного онлайн-шутера Tacticool

ПРЕДЫДУЩАЯ ИТЕРАЦИЯ

- memset(a, 0, sizeof(a))
- Race conditions
- ~10 потоков + блокирующие запросы
- < 200 пассивных CCU
- ~~Горизонтальное масштабирование~~

C/C++

PostgreSQL

Можно не верить мне...

https://habr.

Архитекту
сервера мс
онлайн-и
Tactic

РАЦИЯ

Через пару недель, потраченных на поиск и исправление наиболее критических багов, мы решили, что проще переписать все с нуля, чем пытаться исправить все недостатки текущего решения.

Почему мы ходим по граблям?

Незнание?

Лень?

NIH-синдром?

Есть много чего

Проверено временем и множеством проектов:

- actors
- communicating sequential processes (CSP)
- tasks (async, promises, futures, ...)
- data flows
- reactive programming
- ...

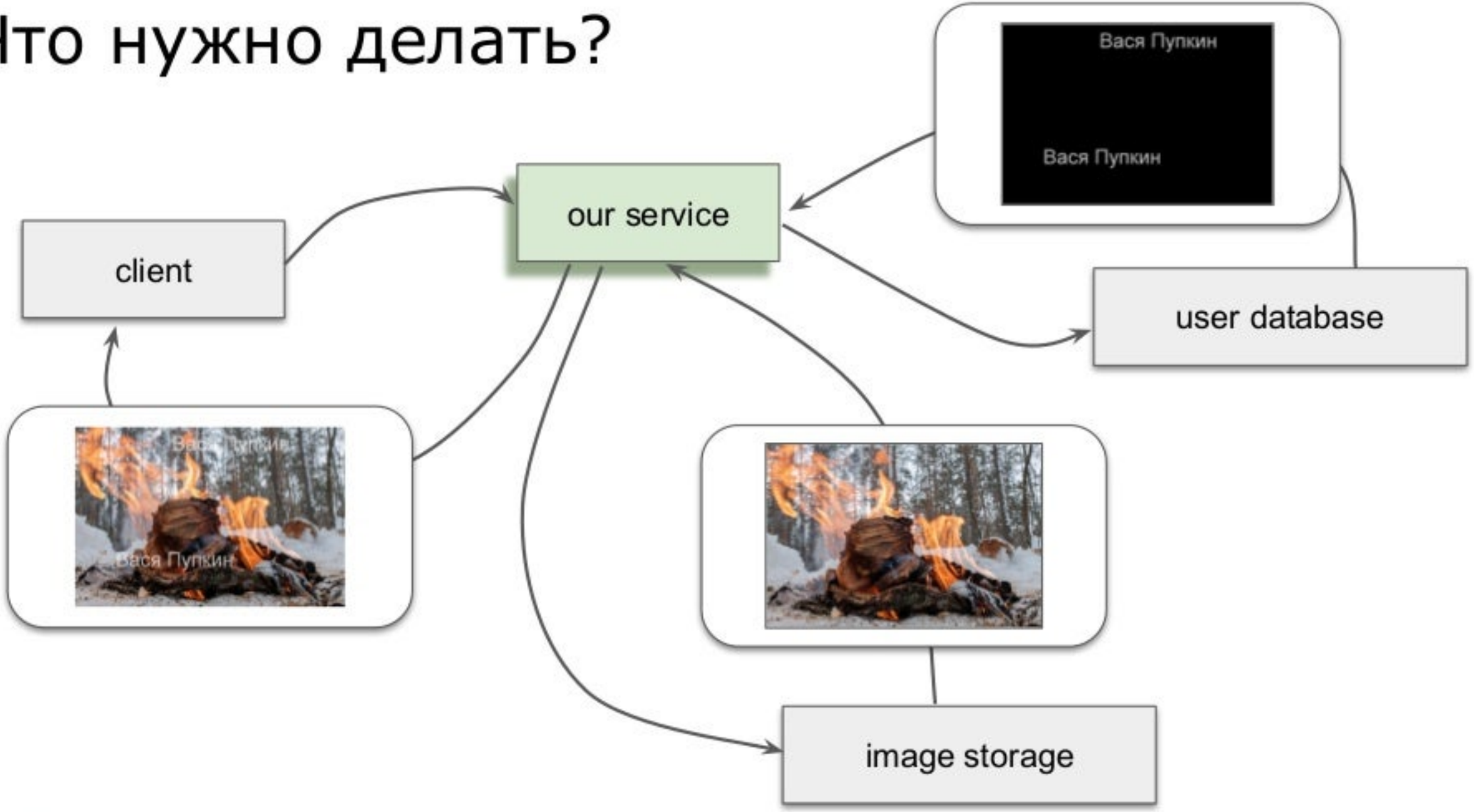
Этому вряд ли учат в ВУЗ-ах.

Простая задача

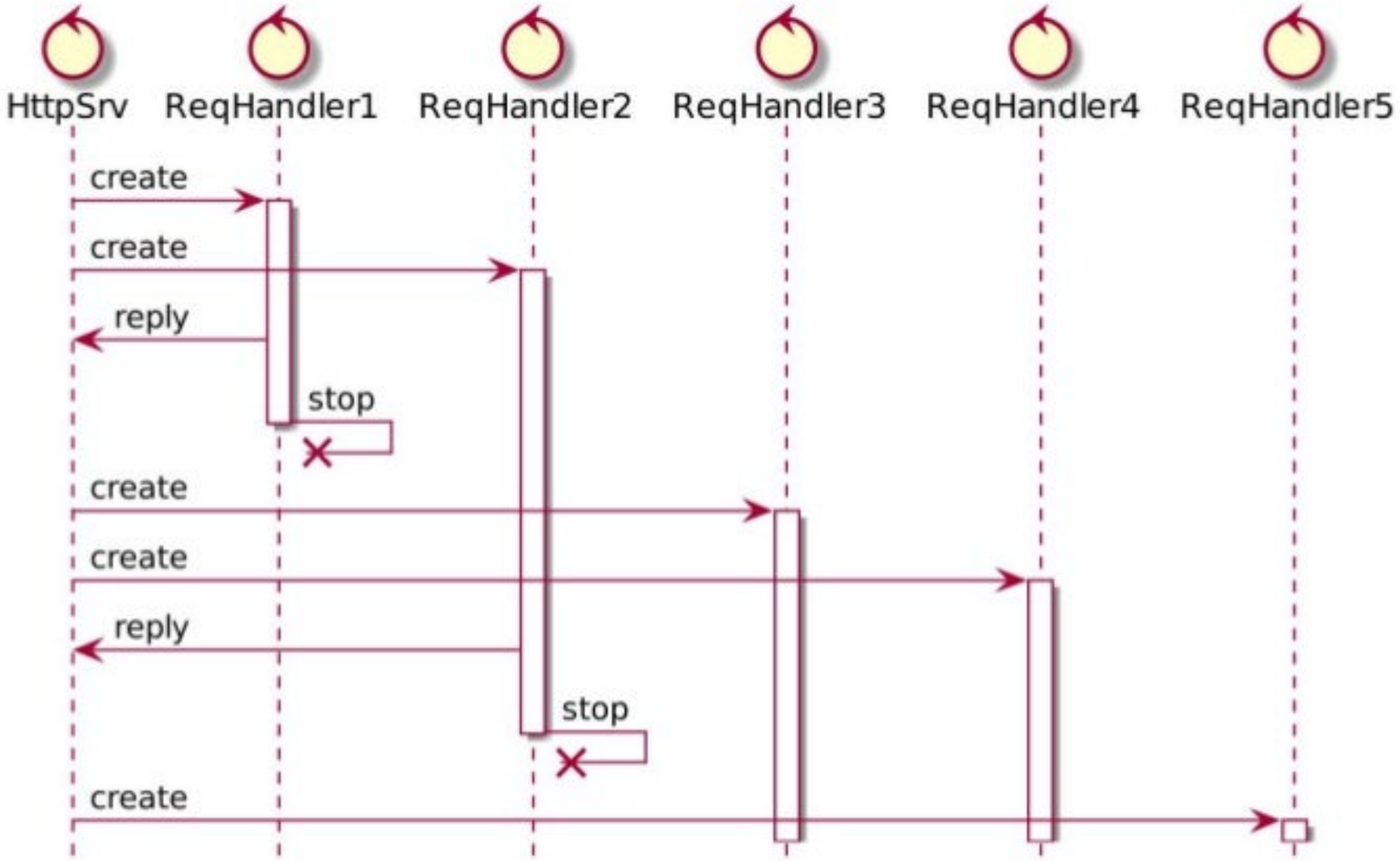
HTTP-сервер, который:

- принимает запрос (ID картинки, ID пользователя);
- отдает картинку с "водяными знаками", уникальными для этого пользователя.

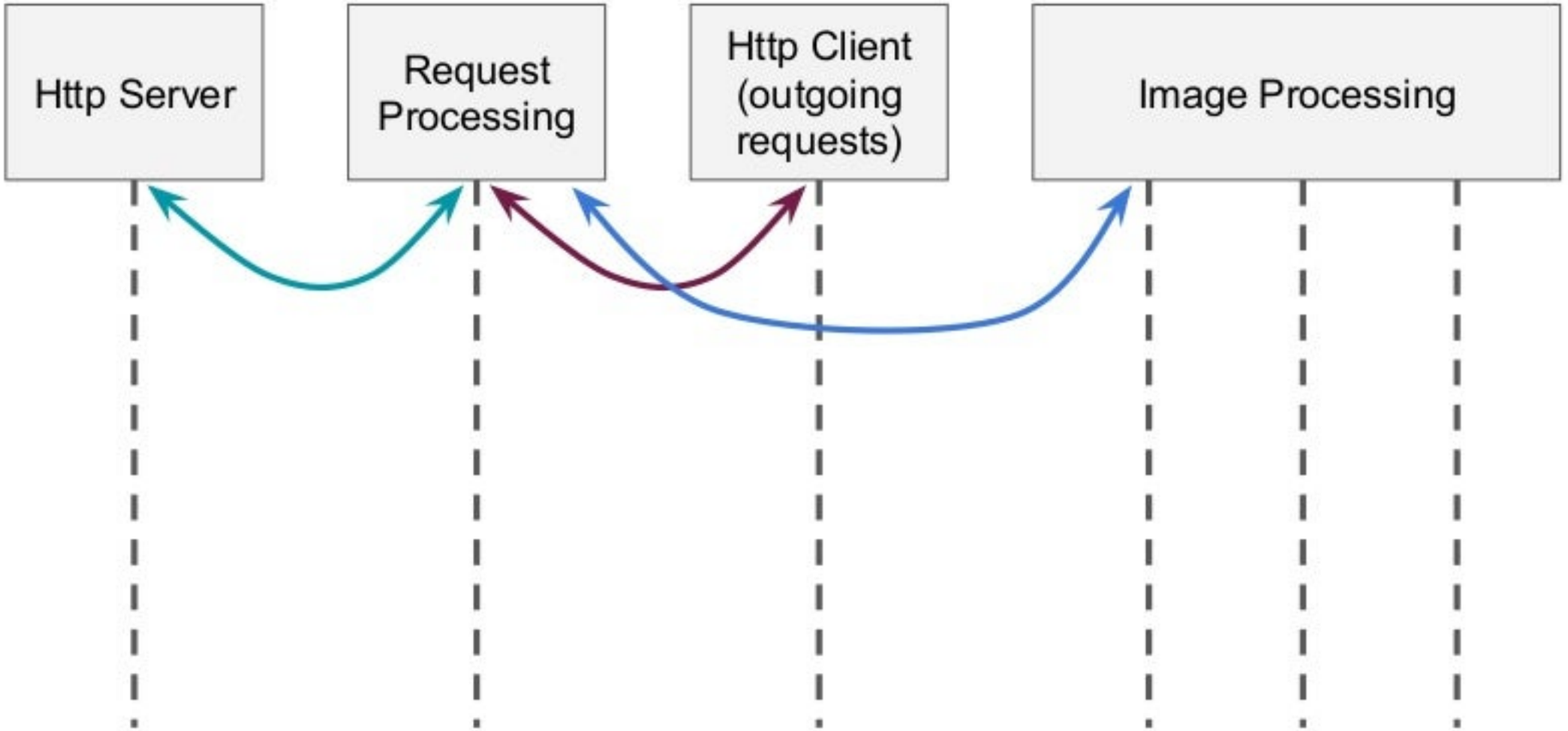
Что нужно делать?



Конкурентность напрашивается...



Набор рабочих нитей



Disclaimer

Примеры не привязаны к какому-то конкретному инструменту.

Везде есть некий execution_context.

Попытка №1

Actors

Модель Акторов. Основные принципы

- актор – это некая 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.

Модель Акторов. Основные принципы

- актор – эт
- акторы ре
- получив с
  - отосла
    другим
  - создат
  - опред
    после

в;

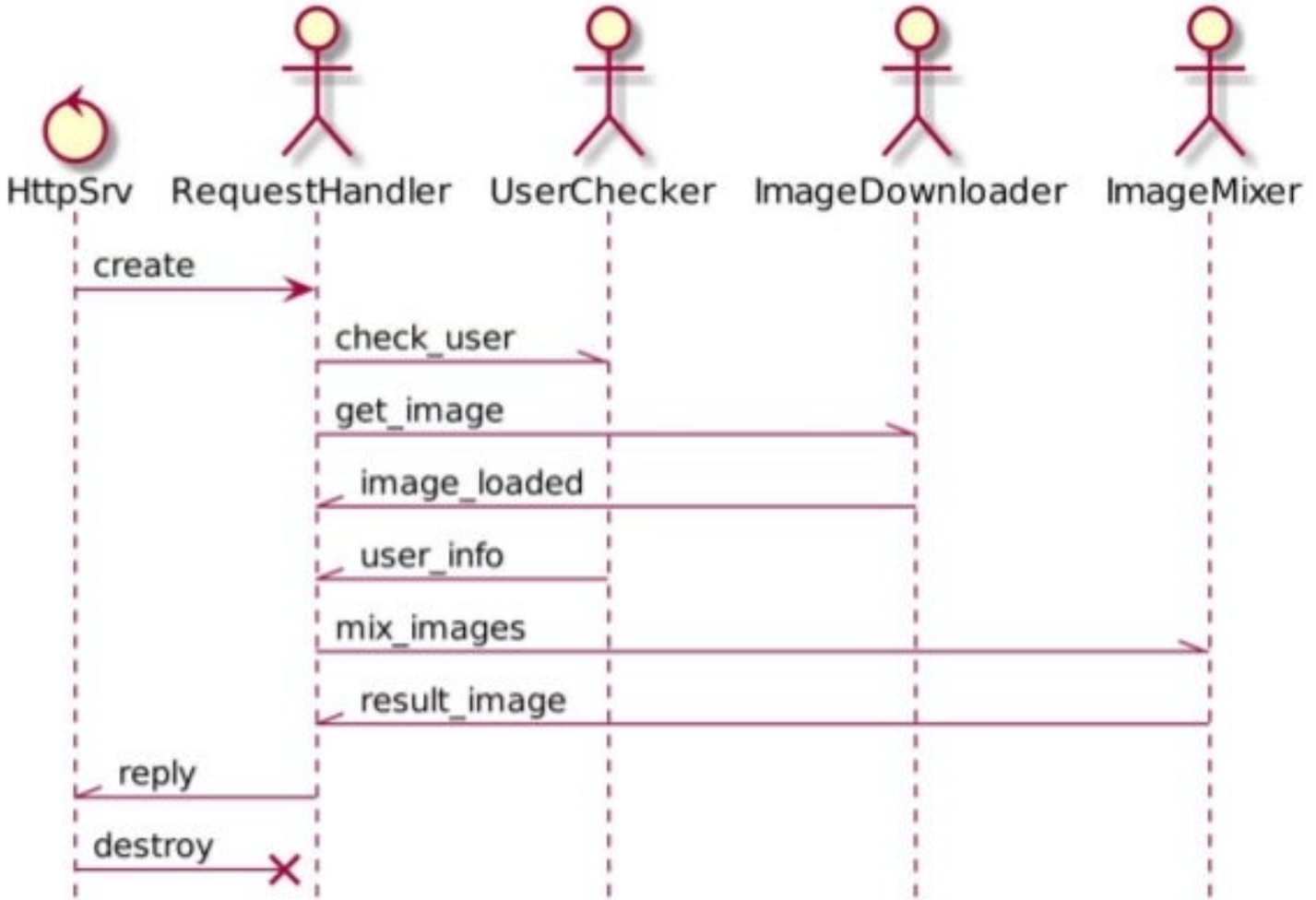
Могут быть реализованы сильно по-разному:

- каждый актор – это отдельный поток ОС;
- каждый актор – это сопрограмма (stackful);
- каждый актор – это объект у которого кто-то вызывает коллбэки.

Сейчас будем говорить про акторов виде объектов с коллбэками.

Сопрограммы оставим для разговора про CSP.

Один из возможных вариантов



Актор request_handler (1)

```
class request_handler final : public some_basic_type {
  const execution_context context_;
  const request request_;

  optional<user_info> user_info_;
  optional<image_loaded> image_;

  void on_start();

  void on_user_info(user_info info);

  void on_image_loaded(image_loaded image);

  void on_mixed_image(mixed_image image);

  void send_mix_images_request();

  ... // вся специфическая для фреймворка обвязка.
};
```

Актор request_handler (2)

```
void request_handler::on_start() {
  send(context_.user_checker(), check_user{request_.user_id(), self()});

  send(context_.image_downloader(), download_image{request_.image_id(), self()});
}
```

Актор request_handler (3)

```
void request_handler::on_user_info(user_info info) {
  user_info_ = std::move(info);

  if(image_)
    send_mix_images_request();
}

void request_handler::on_image_loaded(image_loaded image) {
  image_ = std::move(image);

  if(user_info_)
    send_mix_images_request();
}
```

Актор request_handler (4)

```
void request_handler::send_mix_images_request() {
  send(context_.image_mixer(),
       mix_images{user_info->watermark_image(), *image_, self()});
}

void request_handler::on_mixed_image(mixed_image image) {
  send(context_.http_srv(), reply{..., std::move(image), ...});
}
```

Особенности Модели Акторов (1)

Акторы реактивны.

Работают при наличии входящих сообщений.

Особенности Модели Акторов (2)

Акторы подвержены перегрузкам.

Следствие асинхронной природы взаимодействия.

send не блокирует отправителя.

Особенности Модели Акторов (3)

Много акторов ≠ решение.

Зачастую много акторов ⇒ еще одна проблема.

Куда смотреть, что брать?

SObjectizer

https://stiffstream.com/en/products/sobjectizer.html

C++ Actor Framework (CAF)

http://actor-framework.org

QP/C++

https://www.state-machine.com/qpcpp/

https://en.wikipedia.org/wiki/Actor_model#Actor_libraries_and_frameworks

Попытка №2

CSP

(communicating sequential processes)

CSP на пальцах и без матана

Композиция параллельно работающих последовательных процессов.

Взаимодействие посредством асинхронных сообщений.

Сообщения доставляются посредством каналов.

У каналов есть две операции: write (send) и read (receive).

CSP на пальцах и без матана

Композиция п
процессов.

Взаимодейст

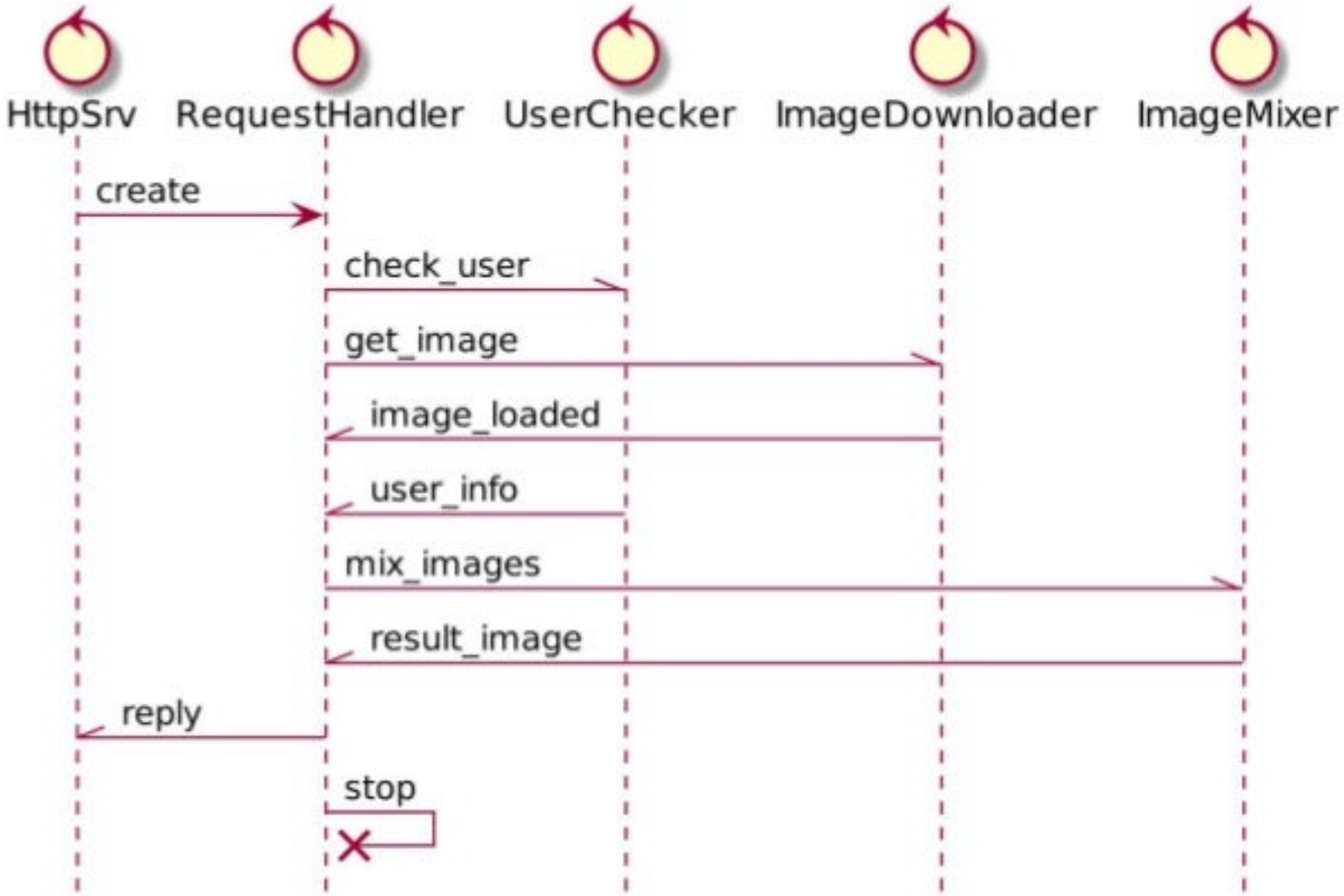
Сообщения д

У каналов ест

"Последовательные процессы" внутри одного процесса ОС могут быть реализованы как:

- отдельные нити ОС. В этом случае имеем вытесняющую многозадачность;
- сопрограммы (stackful coroutines, green threads, fibers, ...). В этом случае имеем кооперативную многозадачность.

Один из возможных вариантов



"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

"Процесс" request_handler

```
void request_handler(const execution_context ctx, const request req)
{
  auto user_info_ch = make_chain<user_info>();
  auto image_loaded_ch = make_chain<image_loaded>();

  ctx.user_checker_ch().write(check_user{req.user_id(), user_info_ch});
  ctx.image_downloader_ch().write(download_image{req.image_id(), image_loaded_ch});

  auto user = user_info_ch.read();
  auto original_image = image_loaded_ch.read();

  auto image_mix_ch = make_chain<mixed_image>();
  ctx.image_mixer_ch().write(
      mix_image{user.watermark_image(), std::move(original_image), image_mix_ch});
  auto result_image = image_mix_ch.read();

  ctx.http_srv_ch().write(reply{..., std::move(result_image), ...});
}
```

Особенности CSP (1)

"Процессы" могут вести себя как захотят: быть реактивными, проактивными или чередовать реактивность с проактивностью.

"Процессы" могут работать даже в отсутствии сообщений.

Особенности CSP (2)

"Родные" механизмы защиты от перегрузки.

Каналы могут быть ограничены по размеру.

Где-то только такими и могут быть.

Отправитель блокируется при записи в полный канал.

Особенности CSP (3)

"Процессы" в виде нитей ОС ⇒ дорого.

Плата за стек. Плата за переключение.

Но вытесняющая многозадачность.

"Процессы" в виде сопрограмм ⇒ дешево.

Но кооперативная многозадачность.

Но сюрпризы с 3rd-party libs (thread-local-storage, например).

Особенности CSP (4)

Много "процессов" ≠ решение.

Зачастую много "процессов" ⇒ еще одна проблема.

Куда смотреть, что брать?

Boost.Fiber

https://www.boost.org

SObjectizer

https://stiffstream.com/en/products/sobjectizer.html

Actors vs CSP

Actors:

один-единственный "канал",
только реактивность,
могут быть сложными конечными автоматами.

CSP-шные процессы:

много каналов,
и реактивность, и проактивность,
не очень хорошо с конечными автоматами,
лучше, когда поддерживаются на уровне языка и stdlib

Попытка №3

Tasks

(async, futures, then, wait_all, ...)

Про Task-based подход в двух словах

Решение строится как набор небольших задач.

Каждая задача выполняет одну операцию.

Задачи запускаются вызовами async. Результатом async является future.

Задачи провязываются в "цепочки" посредством .then(), wait_all(), wait_any().

Реализация обработки запроса (1)

```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data) {
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(...); });
      });
    });
}
```

Реализация обработки запроса (2)

```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data) {
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(...); });
      });
    });
}
```

Реализация обработки запроса (3)

```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data) {
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(...); });
      });
    });
}
```

Реализация обработки запроса (4)

```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data) {
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(...); });
      });
    });
}
```

Реализация обработки запроса (5)

```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data) {
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(...); });
      });
    });
}
```

Реализация обработки запроса (6)

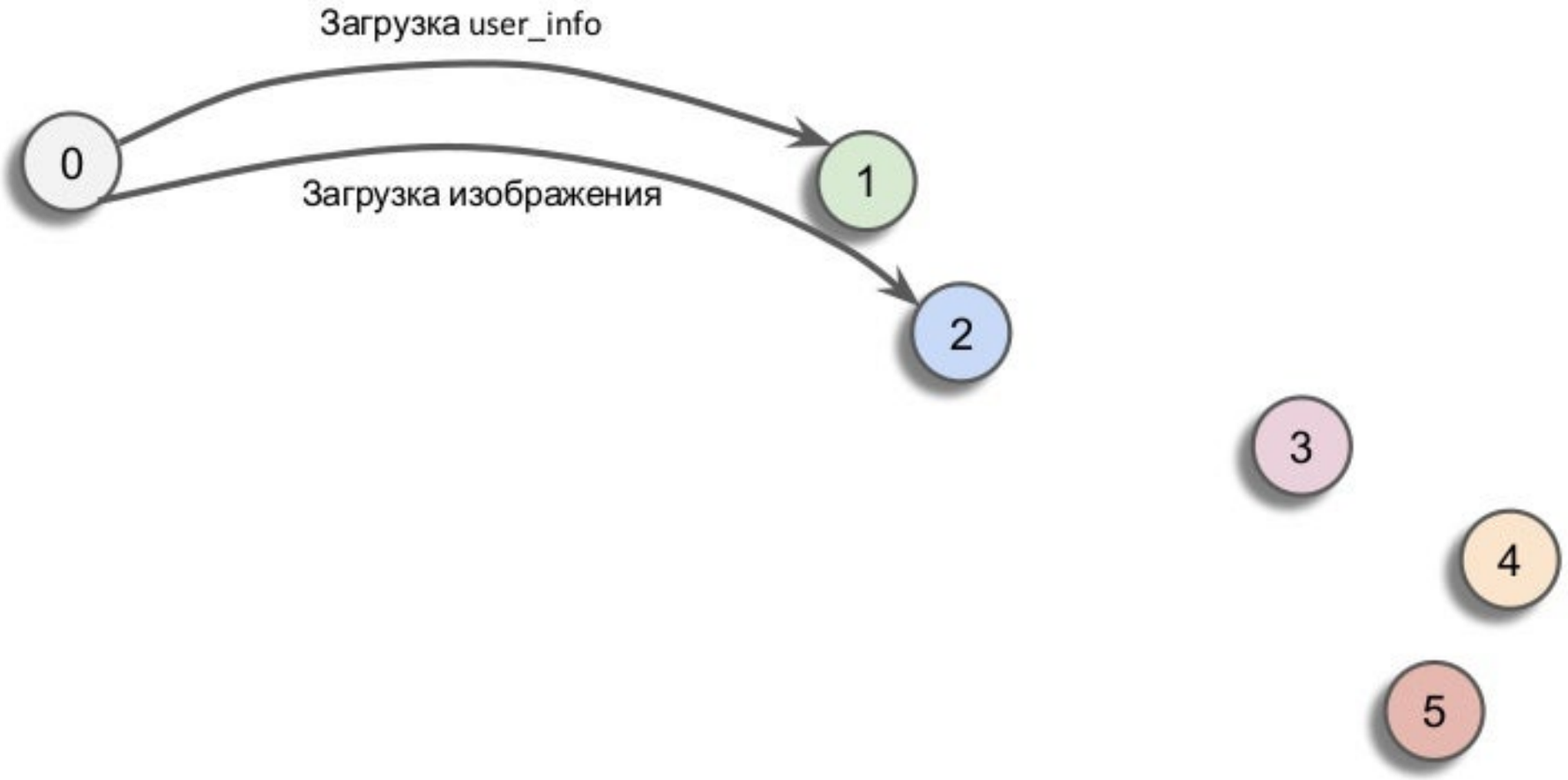
```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data) {
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(...); });
      });
    });
}
```

Последовательность операций

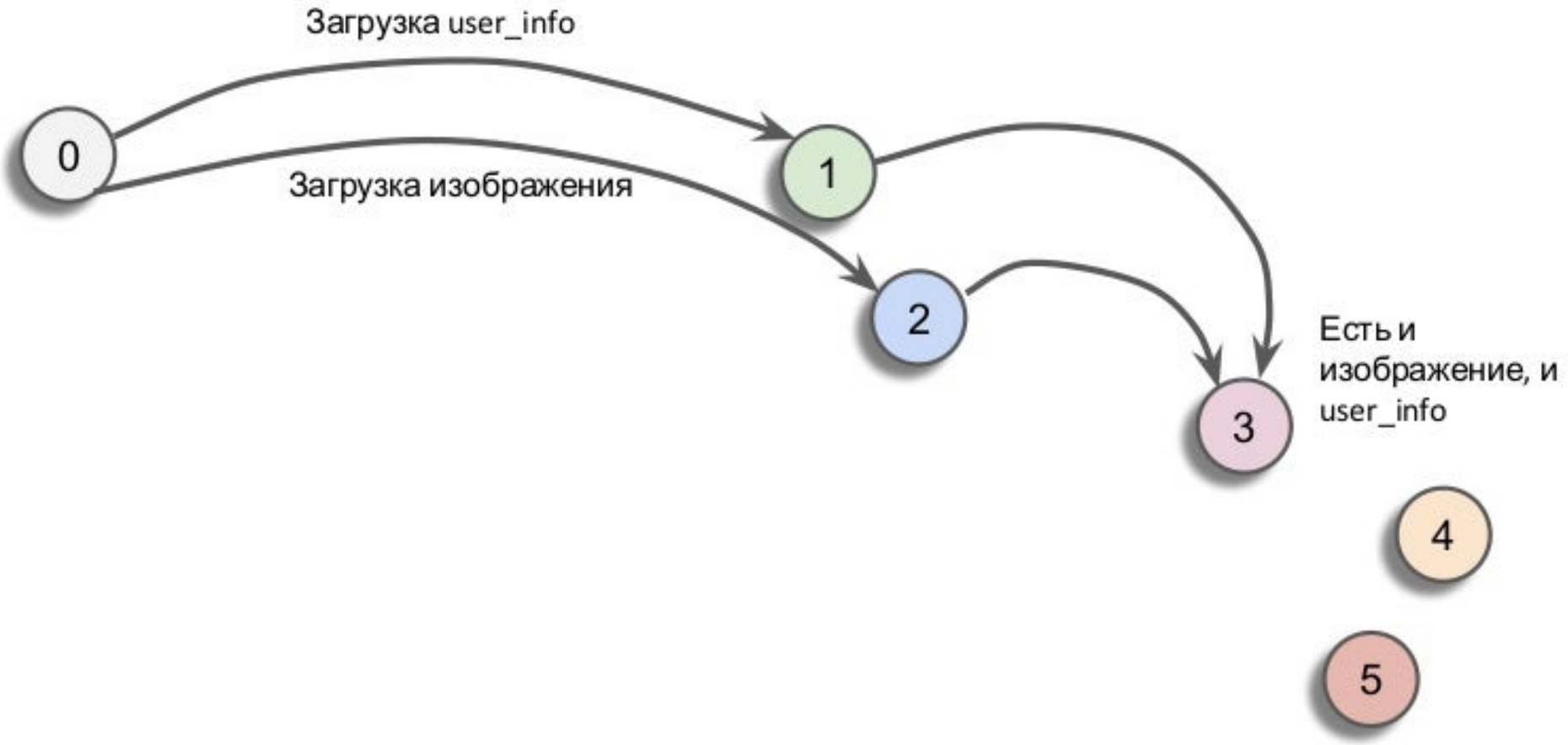
```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data)
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(req, im.get()); });
      });
    });
}
```

1 2 3 4 5

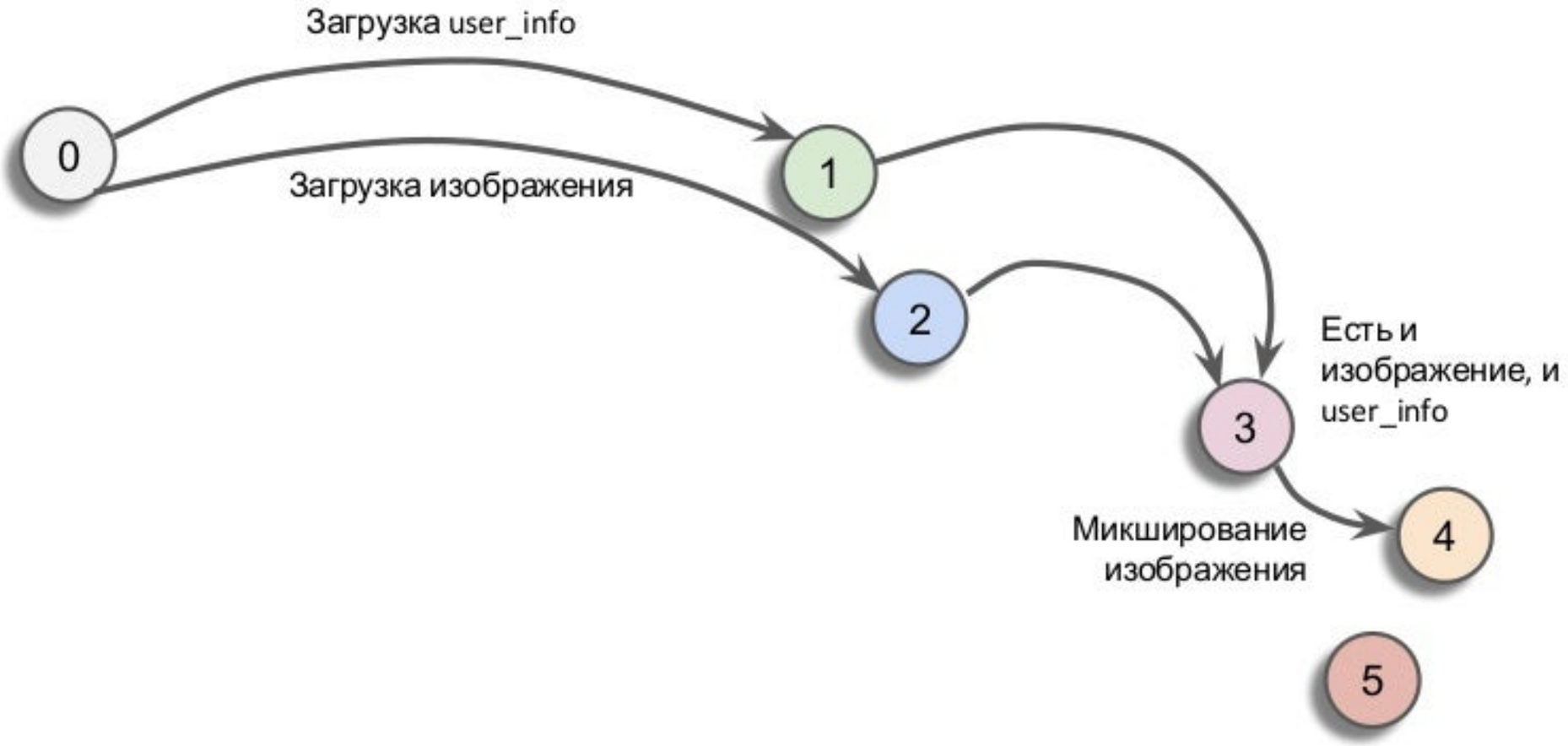
Последовательность операций



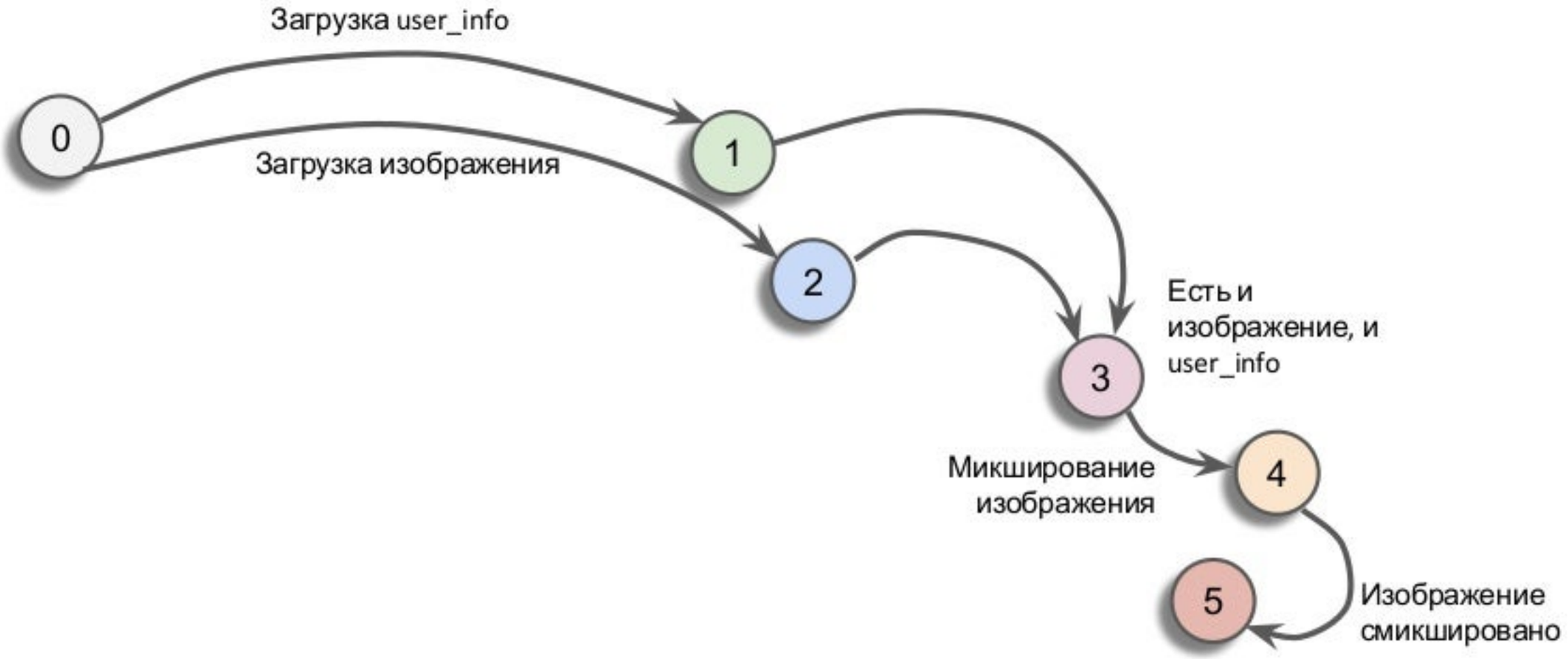
Последовательность операций



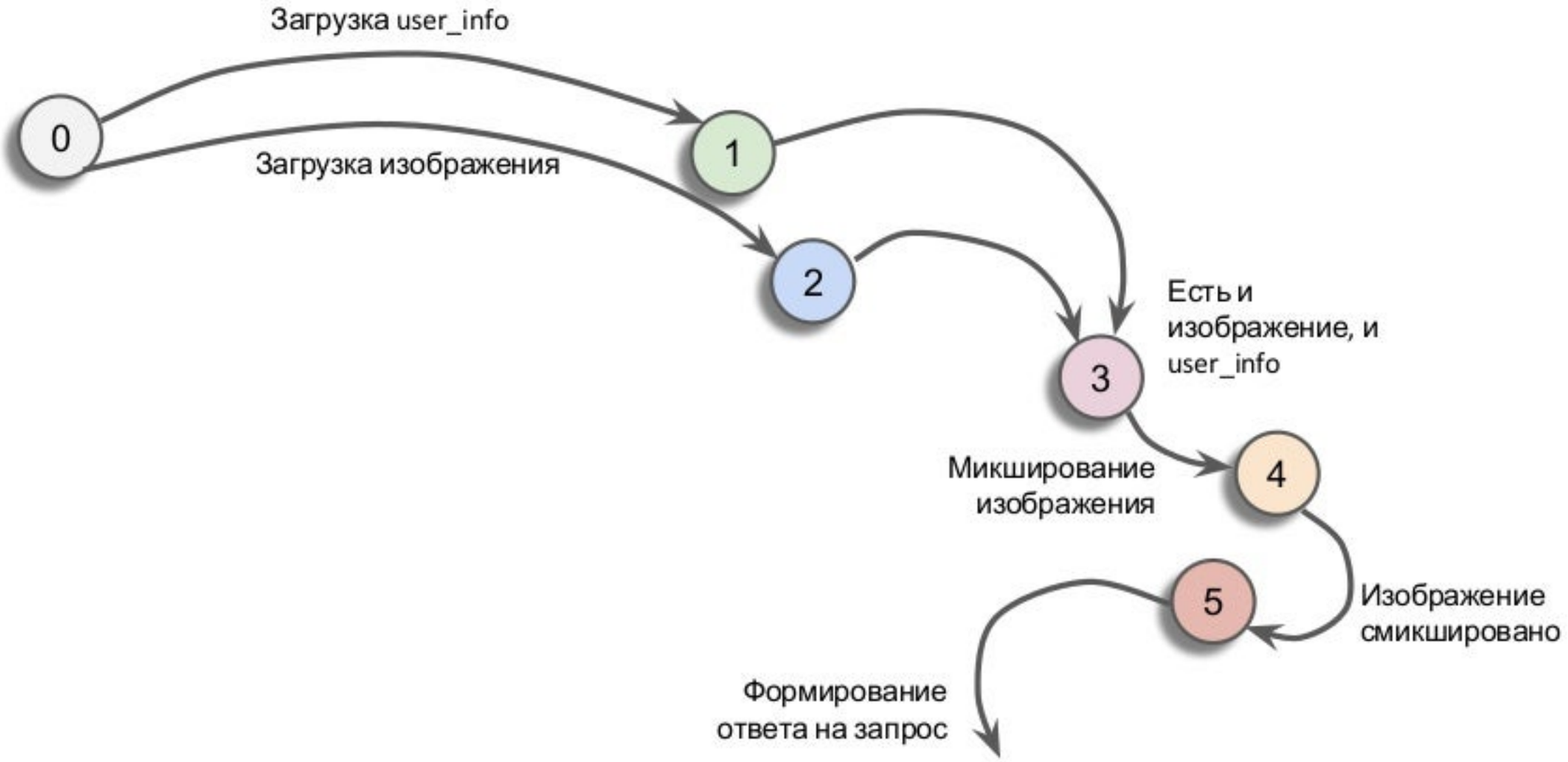
Последовательность операций



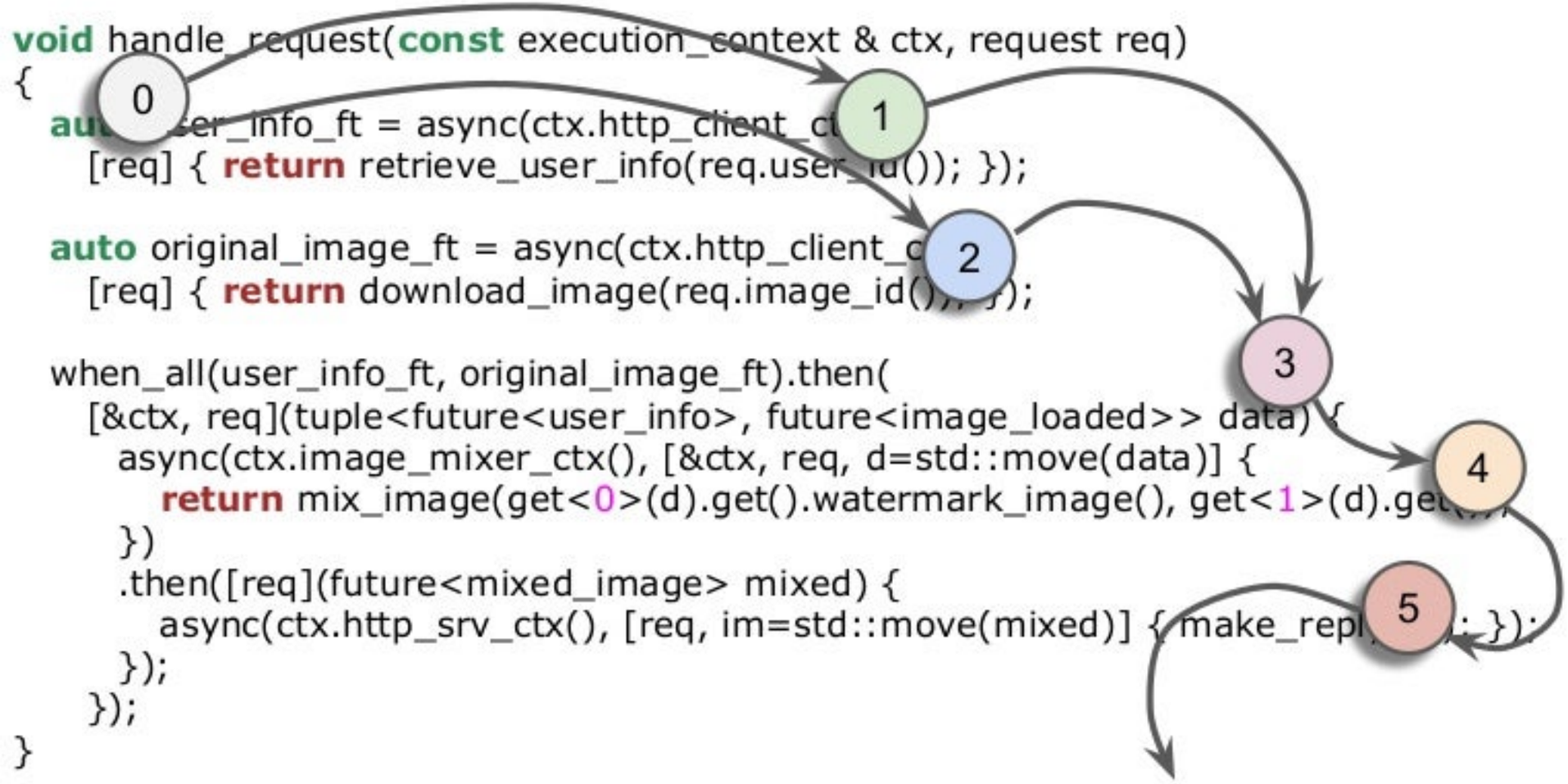
Последовательность операций



Последовательность операций



Последовательность операций



Особенности Task-ов (1)

Обозримость. С ней не очень хорошо.

Callback hell. И вот это вот все.

Особенности Task-ов (2)

Обработка ошибок.

Особенности Task-ов (3)

Отмена task-ов.

Таймеры, таймауты?

Читинг

```
void handle_request(const execution_context & ctx, request req)
{
  auto user_info_ft = async(ctx.http_client_ctx(),
    [req] { return retrieve_user_info(req.user_id()); });

  auto original_image_ft = async(ctx.http_client_ctx(),
    [req] { return download_image(req.image_id()); });

  when_all(user_info_ft, original_image_ft).then(
    [&ctx, req](tuple<future<user_info>, future<image_loaded>> data) {
      async(ctx.image_mixer_ctx(), [&ctx, req, d=std::move(data)] {
        return mix_image(get<0>(d).get().watermark_image(), get<1>(d).get());
      })
      .then([req](future<mixed_image> mixed) {
        async(ctx.http_srv_ctx(), [req, im=std::move(mixed)] { make_reply(...); });
      });
    });
}
```

Actors/CSP vs Tasks

Actors/CSP: акцент на обмене информацией.

Tasks: акцент на выполняемых действиях.

Куда смотреть, что брать?

C++20 stdlib

https://en.cppreference.com/w/cpp/experimental/concurrency

async++

https://github.com/Amanieu/asyncplusplus

Microsoft's PPL

https://msdn.microsoft.com/en-us/library/dd492427.aspx

Еще интересное

Антон Полухин

"Готовимся к C++20. Coroutines TS на реальном примере"
https://habr.com/company/yandex/blog/420861/

And now, the end is near...

Пора переходить к итогам

Забудьте про голую многопоточность

Голой многопоточности есть место только внутри фреймворков и библиотек.

Есть из чего выбирать

Проверено временем и множеством проектов:

- actors
- communicating sequential processes (CSP)
- tasks (async, promises, futures, ...)
- data flows
- reactive programming
- ...

Для всего этого в C++ есть готовые инструменты (в том числе и хорошие)

The (Happy?) End

Спасибо за внимание!

stiffstream.com