This slide is intentionally left blank

There is nothing to read here

Этюды
о буферизации

Вадим Винник

Мотивация

- Задачи взяты из практики.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.
- Решения пригодны для многократного использования.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.
- Решения пригодны для многократного использования.
- Условия формулируются кратко и выглядят простыми.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.
- Решения пригодны для многократного использования.
- Условия формулируются кратко и выглядят простыми.
- Анализ выявляет неожиданные сложности.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.
- Решения пригодны для многократного использования.
- Условия формулируются кратко и выглядят простыми.
- Анализ выявляет неожиданные сложности.
- Хорошее поле для применения средств C++ 11 / 14 / 17…

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.
- Решения пригодны для многократного использования.
- Условия формулируются кратко и выглядят простыми.
- Анализ выявляет неожиданные сложности.
- Хорошее поле для применения средств C++ 11 / 14 / 17…
- …до сих пор не общеизвестных.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.
- Решения пригодны для многократного использования.
- Условия формулируются кратко и выглядят простыми.
- Анализ выявляет неожиданные сложности.
- Хорошее поле для применения средств C++ 11 / 14 / 17…
- …до сих пор не общеизвестных.
- Хорошо подходит для собеседований.

Мотивация

- Задачи взяты из практики.
- Получившийся код применён в реальных проектах.
- Задачи достаточно общие.
- Решения пригодны для многократного использования.
- Условия формулируются кратко и выглядят простыми.
- Анализ выявляет неожиданные сложности.
- Хорошее поле для применения средств C++ 11 / 14 / 17…
- …до сих пор не общеизвестных.
- Хорошо подходит для собеседований.

Задачи

- Асинхронный оповещатель.
- Репликатор обновлений.
- Объединитель запросов.

Задачи

- Асинхронный оповещатель.
- Репликатор обновлений.
- Объединитель запросов.

Инструменты

- Поток, семафор, переменная условия, атомарная переменная.
- Перемещение, ссылка rvalue, умный указатель.
- Вариадический шаблон, метапрограммирование, класс свойств (traits).

Асинхронный оповещатель

Исходные условия

- Пусть приложение делится на два слоя:
  - верхний: интерфейс пользователя, журнал событий;
  - нижний: вычисления, интерфейс с аппаратурой или ОС.
- Нижний слой посылает оповещения верхнему:
  - количество сделанной работы;
  - вычисленные результаты;
  - диагностические сообщения.
- Обработчик оповещений передаётся с верхнего слоя:
  - указатель на функцию;
  - функциональный объект.

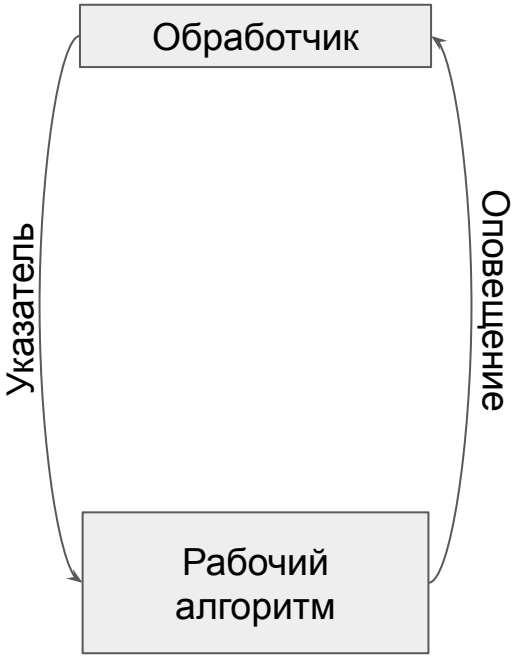
Наивный подход к реализации

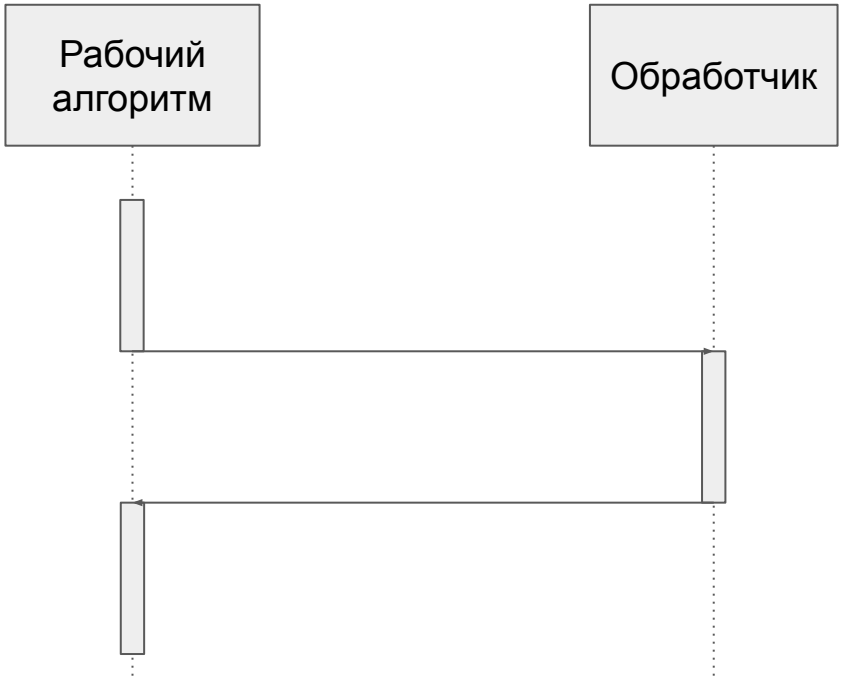
```
void handle_message(std::string const& msg);

int main() {
  run_worker(&handle_message);
  return 0;
}
```

```
void run_worker(void (*notify)(std::string const&)) {
  while (!is_done()) {
    do_some_work();
    notify("one more iteration");
  }
}
```

Синхронные оповещения

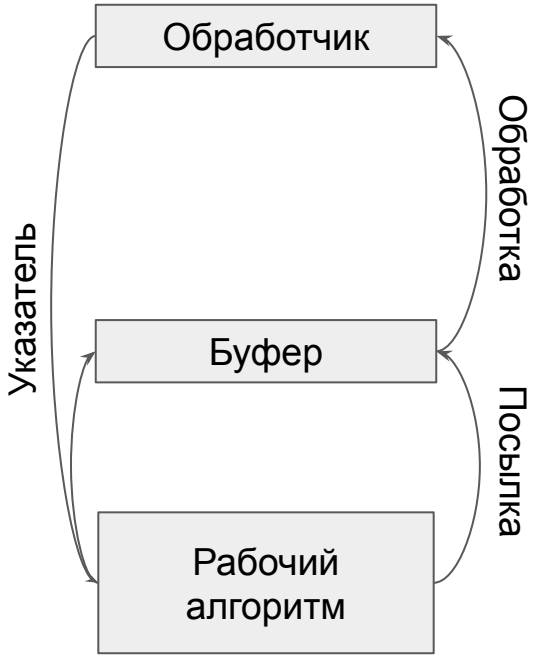


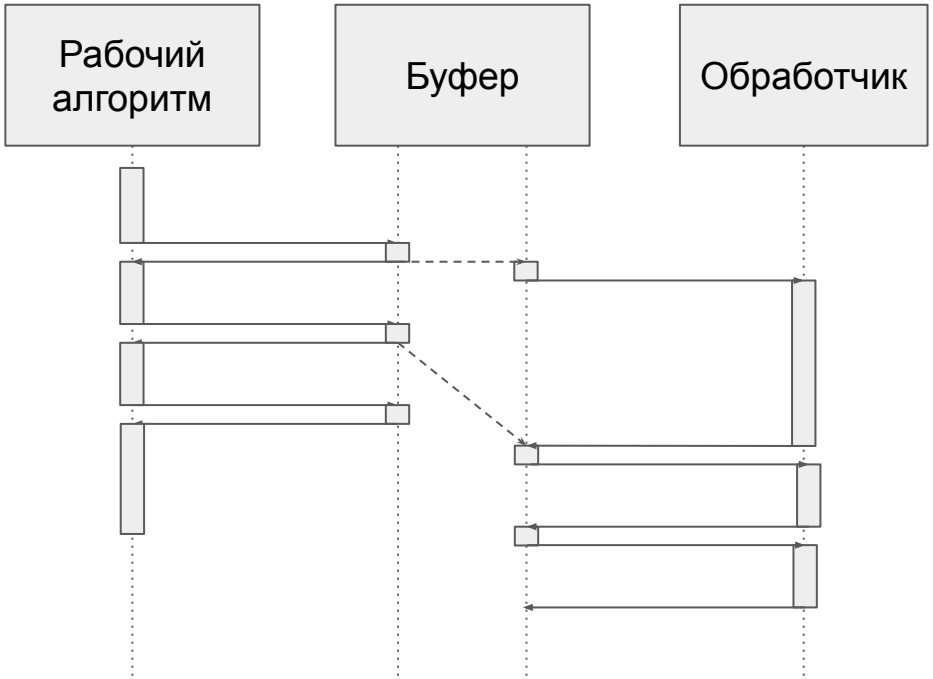


Буфер для развязки уровней

- Нижний уровень не может контролировать работу верхнего:
  - не властен над переданной сверху функцией-обработчиком.
- Производительность нижнего слоя не должна страдать:
  - нижний слой не должен ожидать обработки оповещения верхним;
  - рабочий алгоритм кладёт оповещения в буфер и продолжает работу;
  - оповещения из буфера доставляется наверх отдельным потоком;
  - потенциально медленный обработчик работает во вспомогательном потоке.

Асинхронная доставка оповещений





Дополнительное условие

Два вида оповещений

- Важны все:
  - обрабатываются в порядке очереди;
  - диагностические сообщения; результаты вычислений.
- Важно только самое свежее:
  - верхний уровень не успел обработать сообщение;
  - в это время приходит новое;
  - старое сообщение можно и нужно игнорировать;
  - примеры: количество сделанной работы, текущая фаза работы.

Требования

- Разработать класс буфера оповещений.
  - Служит обёрткой над обработчиком.
- Буфер снимает зависимость скорости рабочего алгоритма от скорости обработчика.
  - Буфер быстро принимает новое сообщение и отпускает рабочий поток.
  - Буфер передаёт сообщения наверх, вызывая обработчик.
  - Передача и приём сообщений друг друга блокируют как можно меньше.
- Должен поддерживать два (по меньшей мере) режима работы:
  - все сообщения в порядке очереди;
  - только самое недавнее сообщение.

Общая идея реализации

- Объект-буфер содержит:
  - поток пересылки сообщений;
  - буфер сообщений;
  - обработчик.
- Поток пересылки запускается конструктором…
- …и останавливается деструктором.
- Поток пересылки работает в цикле:
  - ждёт появления сообщения в буфере;
  - вызывает обработчик.
- Единственный метод: поставить в очередь новое сообщение.

Черновик реализации

```
using lock_t =
  std::unique_lock<std::mutex>;

manager(Handler handler):
  handler_(handler),
  thread_(std::thread(&run, this))
{}

void post(message_t const& m) {
  lock_t l(mx_);
  q_.put(m);
  cv_.notify_one();
}
```

```
void run() {
  while (true) {
    message_t m;
    {
      lock_t l(mx_);
      cv_.wait(
        l,
        is_not_empty);
      m = q_.get_and_remove();
    }
    handler_(m);
  }
}
```

Diabolus in singulis est!

- Как минимизировать блокировки, если в очереди много сообщений?
  - Можно ли избежать блокировки на изъятие из очереди каждого сообщения?
- Как одним решением покрыть режимы очереди и одного последнего?
- Можно ли учесть специфику разных типов данных?
  - “Лёгкие” сообщения (напр., целые числа).
  - Поддерживающие быстрое перемещение (напр., строки).
  - Тяжёлые, с копированием.
- Можно ли заложить резерв гибкости?
  - Очередь с приоритетами.
  - Очередь ограниченной ёмкости.
  - Очередь с ограниченным временем хранения.

Как улучшить производительность

- Под блокировкой быстро забирать всё содержимое буфера
  - и оставить его пустым.
- Вне критической секции вытаскивать элементы
  - и отправлять их обработчику

```
buffer_t buffer {};
{
  lock_t l(mx_);
  cv_.wait(l, is_not_empty);
  std::swap(buffer, buffer_);
}
```

```
// continued

while (!buffer.empty()) {
  handler(buffer.front());
  buffer.pop();
}
```

Да здравствуют абстракции!

- Найти общую абстракцию, охватывающую разные виды буфера:
  - На один элемент.
  - Очередь произвольной ёмкости.
  - С обёртыванием тяжёлого объекта в умный указатель.
- Интерфейс абстрактного буфера:
  - Создать пустой буфер.
  - Проверить на пустоту.
  - Втолкнуть элемент.
  - Ко всем имеющимся элементам применить функцию.
- Параметризовать на этапе выполнения или компиляции?
  - Параметризовать объектом-буфером?
  - …или классом свойств (traits) буфера?

Буфер для одного легковесного значения

```
template <typename T, T Empty>
struct simple_variable {
  using message_t = T;
  using state_t = T;

  static constexpr state_t
  make_empty_state() noexcept {
    return Empty;
  }

  static constexpr bool
  is_empty_state(state_t const& state)
  noexcept {
    return state == Empty;
  }
```

```
  template <typename M>
  static void
  put(state_t& state, M&& message)
  noexcept {
    state = std::forward<M>(message);
  }

  template <typename F>
  static void
  handle_and_destroy(F f, state_t& state) {
    f(std::move(state));
    state = Empty;
  }
};
```

Буфер-очередь

```
template <typename T>
struct queue {
  using message_t = T;
  using state_t = std::queue<T>;

  static state_t
  make_empty_state() noexcept {
    return {};
  }

  static bool
  is_empty_state(state_t const& state)
  noexcept {
    return state.empty();
  }
```

```
  template <typename M>
  static void
  put(state_t& state, M&& message)
  noexcept {
    state.push(std::forward<M>(message));
  }

  template <typename F>
  static void
  handle_and_destroy(F f, state_t& state) {
    while (!state.empty()) {
      f(std::move(state.front()));
      state.pop();
    }
  }
};
```

Репликатор обновлений

Исходные условия

- Несколько одинаковых потоков работают в цикле:
  - принять данные (например, из сети);
  - обработать данные (долгие вычисления);
  - отправить результат (например, в файл).
- Важна производительность потоков:
  - вычислительная часть алгоритма не должна тормозиться синхронизацией;
  - свести синхронизацию к минимуму: в начале и в конце итерации.
- Рабочие потоки зависят от глобальной конфигурации:
  - используется многократно на всём протяжении вычислетельной части.
- Глобальная конфигурация обновляется редко.
  - Копирование конфигурации может быть медленным.
  - Проверка при отсутствии обновлений должна проходить быстро.

Общая идея

- Рабочий поток содержит локальную копию конфигурации:
  - быстрый доступ;
  - без синхронизации.
- Локальная копия поддерживается в актуальном состоянии:
  - обновляется из глобальной конфигурации;
  - наличие обновлений проверяется в начале каждой итерации;
  - если глобальная конфигурация изменилась, локальная копия обновляется;
  - в худшем случае поток выполнит одну итерацию на старой конфигурации.
- Для проверки актуальности - счётчики обновлений:
  - оригинала;
  - локальной копии.

Требования

- Разработать класс - репликатор объектов данных.
- Канал передачи данных с одним входом и множеством выходов.
- Операции на входном конце канала:
  - проинициализировать начальным значением;
  - установить новое значение;
- Операция на выходном конце:
  - проинициализировать, привязав к источнику;
  - обеспечить актуальность:
    - проверить, изменились ли данные на входе после последнего копирования;
    - если изменились, обновить копию.
  - взять значение-копию.

Пример использования

```
// main thread

source_t<config_t> source{ initial_config };
source_t<config_t>::replica replica { sorce };
start_worker_thread(replica);
source.set(new_config);
```

```
// worker thread

while (!is_finish) {
  replica.ensure_up_to_date();
  do_with_config(replica.const_get());
}
```

Дуальность с асинхронным оповещателем

Оповещатель:

- много издателей;
- один подписчик;
- семантика вталкивания.

Репликатор:

- один издатель;
- много подписчиков;
- семантика втягивания.

Дополнительные требования

- Данные могут не поддерживать присваивания:
  - неизменяемые объекты (immutable);
  - популярны в функциональном и реактивном программирование.
- Репликатор должен поддерживать два механизма обновления:
  - присваивание значения объекту с сохранением идентичности;
  - уничтожение старого объекта и создание нового.
- Репликатор должен быть абстрагирован от этого механизма.

Типы свойств (traits)

- Абстрагирование от механизма хранения значений:
  - переменная, допускающая присваивание;
  - умный указатель с замещением объекта.
- Тип свойств должен содержать:
  - тип value_t
    - для работы со значениями прикладного уровня;
  - тип wrapped_t
    - механизм хранения, обёртка над значением;
  - static void set(wrapped_t& lhs, value_t const& rhs);
    - обёртка над сохранением нового значения
  - static value_t const& const_get(wrapped_t const& wrapped);
    - обёртка над взятием значения из хранилища.

```
template <typename T>
struct id {
  using value_t = T;
  using wrapped_t = T;

  static wrapped_t construct(
    value_t const& x)
  { return x; }

  static void set(
    wrapped_t& lhs,
    value_t const& rhs)
  { lhs = rhs; }

  static T const& const_get(
    wrapped_t const& wrapped)
  { return wrapped; }
};
```

```
template <typename T>
struct unique_ptr {
  using value_t = T;
  using wrapped_t = std::unique_ptr<T>;

  static wrapped_t construct(
    value_t const& x)
  { return std::make_unique<value_t>(x); }

  static void set(
    wrapped_t& lhs,
    value_t const& rhs)
  { lhs = std::make_unique<value_t>(rhs); }

  static T const& const_get(
    wrapped_t const& wrapped)
  { return *wrapped; }
};
```

Устройство репликатора

- Объект-источник содержит:
  - реплицируемое значение;
  - счётчик обновлений (std::atomic<int>);
  - двоичный семафор (std::mutex).
- Объект-копия содержит:
  - ссылку на объект-источник;
  - значение-копию;
  - номер последнего обновления (достаточно неатомарного типа int).

```
template <typename T, template<typename> typename Traits>
void source_base<T, Traits>::replica::ensure_up_to_date() {
  if (version_ != source_.version_.load(std::memory_order_consume)) {
    std::lock_guard<std::mutex> lock{ source_.mutex_ };
    Traits::set(wrapped_, wrap_traits_t::const_get(source_.wrapped_));
    version_ = source_.version_;
  }
}
```

```
template <typename T, template<typename> typename Traits>
void source_base<T, Traits>::set(value_t const& value) {
  std::lock_guard<std::mutex> lock { mutex_ };
  version_.fetch_add(1, std::memory_order_release);
  Traits::set(wrapped_, value);
}
```

Ещё одна полезная функция

- Если модификация источника не сводится к присваиванию нового значения…
- Втянуть модифицирующую функцию в источник
  - и выполнить в контексте объекта-источника
  - в духе функционального программирования: lifting.

```
template <typename Func>
void source_base<T, Traits>::modify(Func func) {
  std::lock_guard<std::mutex> lock { mutex_ };
  version_.fetch_add(1, std::memory_order_release);
  func(wrap_traits_t::get(wrapped_));
}
```

Объединитель запросов

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (*запросы*) совпадают.

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.
- Умеет распознавать совпадение запросов.

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.
- Умеет распознавать совпадение запросов.
- При первом обращении с новым запросом x вызывает f(x).

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.
- Умеет распознавать совпадение запросов.
- При первом обращении с новым запросом x вызывает f(x).
- При обращении другого потока с тем же запросом x ставит в ожидание.

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.
- Умеет распознавать совпадение запросов.
- При первом обращении с новым запросом x вызывает f(x).
- При обращении другого потока с тем же запросом x ставит в ожидание.
- Результат f(x), в т.ч. исключение, выдаёт всем ожидающим клиентам.

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.
- Умеет распознавать совпадение запросов.
- При первом обращении с новым запросом x вызывает f(x).
- При обращении другого потока с тем же запросом x ставит в ожидание.
- Результат f(x), в т.ч. исключение, выдаёт всем ожидающим клиентам.
- У потока-клиента создаёт иллюзию того, что он вызывает f.

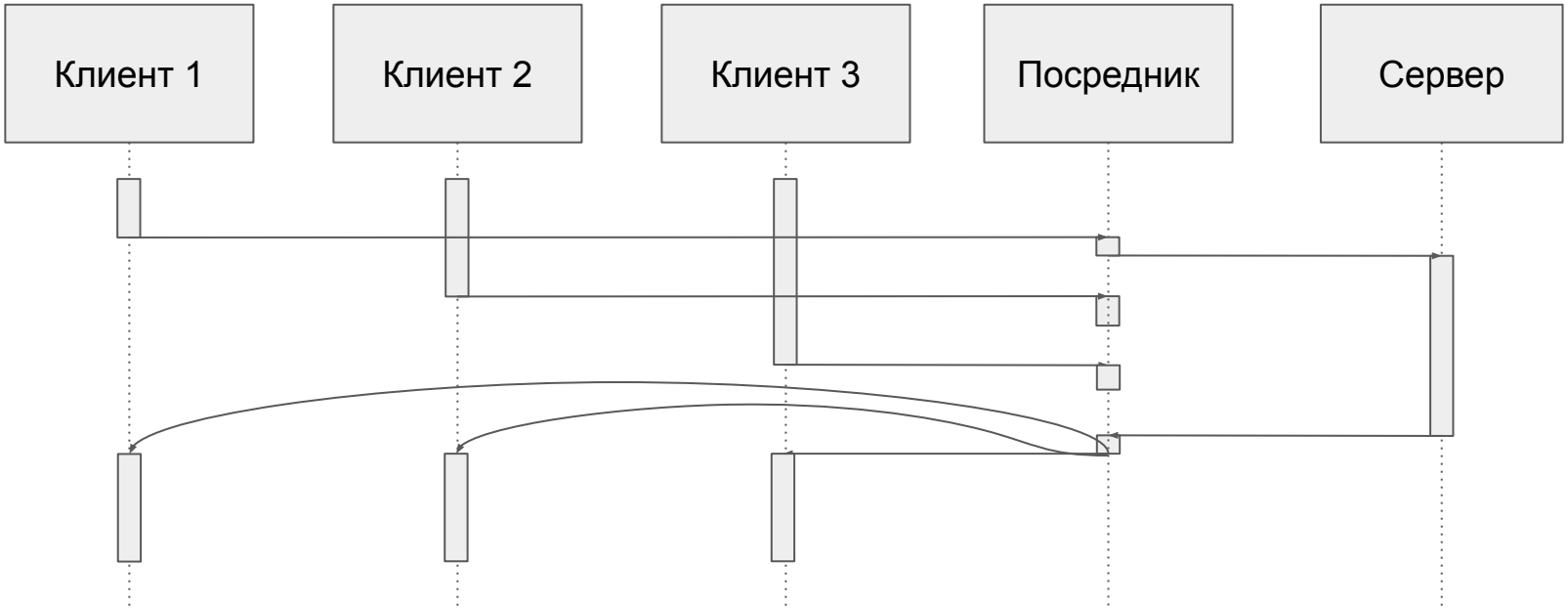
Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.
- Умеет распознавать совпадение запросов.
- При первом обращении с новым запросом x вызывает f(x).
- При обращении другого потока с тем же запросом x ставит в ожидание.
- Результат f(x), в т.ч. исключение, выдаёт всем ожидающим клиентам.
- У потока-клиента создаёт иллюзию того, что он вызывает f.
- При уходе последнего клиента, запросившего f(x), результат забывает.

Требования

- Пусть есть функция f, которая выполняется долго (*сервер*).
- Приложение из разных потоков (*клиентов*) вызывает f часто.
- С большой вероятностью аргументы (запросы) совпадают.
- Разработать класс-обёртку, который объединяет одинаковые запросы.
- Умеет распознавать совпадение запросов.
- При первом обращении с новым запросом x вызывает f(x).
- При обращении другого потока с тем же запросом x ставит в ожидание.
- Результат f(x), в т.ч. исключение, выдаёт всем ожидающим клиентам.
- У потока-клиента создаёт иллюзию того, что он вызывает f.
- При уходе последнего клиента, запросившего f(x), результат забывает.

Асинхронная доставка оповещений



Идея решения

- В буфере хранить дескрипторы выполняющихся вызовов.
- Дескрипторы хранить в ассоциативном массиве:
  - ключ вычисляется на основе запроса;
  - запросы с совпадающими ключами отождествляются;
  - в пределе: ключ - весь запрос.
- Дескриптор содержит:
  - флаг “вызов сервера сделан”;
  - счётчик ожидающих клиентов (с этим ключом);
  - результат вызова (в т.ч. исключение) или признак его отсутствия;
  - двоичный семафор;
  - условная переменная: “результат получен”.

Алгоритм постановки запроса на выполнение

- В ассоциативном массиве найти дескриптор или создать пустой
  - нарастив в дескрипторе счётчик ожидающих клиентов;
  - под блокировкой всего ассоциативного массива дескрипторов.
- Взять блокировку дескриптора.
- Дождаться, когда истинным станет условие:
  - выполнение этого запроса ещё не начато или
  - выполнение запроса закончилось (возвратом значения или выбросом исключения).
- Если выполнение запроса не начато:
  - синхронным образом вызвать функцию-сервер с заданным запросом;
  - запомнить в дескрипторе её результат (возможно, исключение).
- Вернуть (пробросить) клиенту результат, сохранённый в дескрипторе.
- Уменьшить счётчик и, если он достиг 0, удалить дескриптор.

https://github.com/vadimvinnik/

- replicable
- async_notification
- execution_collator

```
} // the talk
```