

Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 13

Міжсервісна взаємодія: синхронна (REST, gRPC оглядово) та асинхронна (черги повідомлень, RabbitMQ/Redis)

Спеціальність: Комп'ютерна інженерія, 3 курс
Обсяг: 1 год 20 хв (лекція 13 з 18)

План лекції

1. Синхронна та асинхронна взаємодія: коли яка потрібна
2. REST між сервісами на практиці: таймаути й повторні спроби
3. gRPC: огляд альтернативи REST
4. Проблема каскадних збоїв і Circuit Breaker
5. Черги повідомлень: концепція Publish/Subscribe
6. RabbitMQ на практиці
7. Redis: Pub/Sub та черги задач

Мета лекції: опанувати практичні інструменти міжсервісної взаємодії — від прямих REST-викликів з таймаутами до асинхронних черг повідомлень (RabbitMQ, Redis) — і навчитися свідомо обирати між ними залежно від вимог конкретного сценарію.

1. Синхронна та асинхронна взаємодія: коли яка потрібна

У попередній темі ми познайомились із двома підходами до комунікації між сервісами — синхронним (REST) і асинхронним (черга повідомлень). Ця лекція заглиблюється в практичну реалізацію кожного підходу: як саме викликати інший сервіс через REST з урахуванням мережевих збоїв, чим gRPC відрізняється від REST, і як фактично працює черга повідомлень на прикладі RabbitMQ та Redis.

Критерій	Синхронна (REST/gRPC)	Асинхронна (черга повідомлень)
Очікування відповіді	Так, негайно	Ні, обробка з затримкою
Зв'язаність у часі	Обидва сервіси мають бути доступні одночасно	Отримувач може бути тимчасово недоступний
Типовий сценарій	"Отримати дані про товар зараз"	"Повідомити про подію, обробити коли зручно"
Складність	Нижча	Вища (потрібна інфраструктура черги)

2. REST між сервісами на практиці: таймаути й повторні спроби

Коли сервіс звертається до іншого сервісу через мережу (а не до бази даних у межах одного процесу), з'являється принципово новий клас проблем: мережа може бути повільною, тимчасово недоступною чи взагалі "загубити" запит. Код, що звертається до іншого сервісу, повинен це враховувати.

2.1. Таймаути

Без явного обмеження часу очікування запит до недоступного сервісу може "зависнути" на невизначений час, утримуючи ресурси (з'єднання, потоки) сервіса-ініціатора. Таймаут примусово перериває очікування після заданого інтервалу.

```
<?php
$ch = curl_init('http://products-service/api/products/42');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_TIMEOUT, 3);           // максимум 3 секунди на
всю операцію
curl_setopt($ch, CURLOPT_CONNECTTIMEOUT, 1);    // максимум 1 секунда на
встановлення з'єднання

$response = curl_exec($ch);

if (curl_errno($ch) === CURLE_OPERATION_TIMEDOUT) {
    // Обробити випадок недоступності сервіса окремо від інших помилок
    error_log('Products service timeout');
}
```

2.2. Повторні спроби (retry)

Тимчасові мережеві збої (короткочасне перевантаження сервіса, миттєвий розрив з'єднання) часто зникають самі протягом секунд. Замість негайної відмови має сенс повторити запит кілька разів із зростаючою затримкою між спробами (експоненційний backoff).

```
<?php
function callWithRetry(callable $request, int $maxAttempts = 3): mixed {
    $attempt = 0;
    while (true) {
        try {
            return $request();
        } catch (RequestException $e) {
            $attempt++;
            if ($attempt >= $maxAttempts) {
                throw $e;    // вичерпали спроби – передаємо помилку далі
            }
            sleep(2 ** $attempt);    // 2, 4, 8 секунд – експоненційний
        }
    }
}
```

Повторювати варто не всі помилки без розбору: помилку 404 Not Found (тема 10) повторювати безглуздо — товару за цим шляхом просто немає, і повторний запит поверне те саме. Retry має сенс для мережеских таймаутів і помилок 500-го чи 503-го рівня, які можуть бути тимчасовими.

3. gRPC: огляд альтернативи REST

gRPC (Google Remote Procedure Call) — фреймворк для міжсервісної взаємодії, розроблений Google як альтернатива REST для внутрішньої комунікації між мікросервісами. Замість JSON поверх HTTP/1.1, gRPC використовує бінарний формат Protocol Buffers поверх HTTP/2, що дає суттєвий приріст швидкості й пропускну здатності.

Критерій	REST	gRPC
Формат даних	JSON (текстовий)	Protocol Buffers (бінарний)
Транспортний протокол	HTTP/1.1 (частіше)	HTTP/2
Швидкість	Нижча (текст, більший розмір)	Вища (бінарний, компактніший)
Читабельність для людини	Висока (звичайний текст)	Низька (потрібні спеціальні інструменти)
Підтримка браузерів	Пряма, нативна	Обмежена (потрібен проксі-шар)
Типова сфера застосування	Публічні API, вебклієнти	Внутрішня комунікація між сервісами

Контракт взаємодії в gRPC описується заздалегідь у файлі `.proto`, з якого генерується код клієнта й сервера для різних мов програмування — це схоже на роль специфікації OpenAPI (тема 11), але з автоматичною генерацією повноцінного клієнтського коду, а не лише документації.

```
// products.proto – контракт сервісу (спрощено)
service ProductService {
    rpc GetProduct (ProductRequest) returns (ProductResponse);
}

message ProductRequest {
    int32 id = 1;
}

message ProductResponse {
    int32 id = 1;
    string name = 2;
    float price = 3;
}
```

У межах цього курсу gRPC розглядається лише оглядово: підтримка gRPC в PHP історично слабша, ніж у Go, Java чи C++, і на практиці для PHP-мікросервісів REST залишається основним вибором. Розуміти принцип gRPC важливо, оскільки в мультимовних командах (де частина сервісів написана на інших мовах) він трапляється дедалі частіше.

4. Проблема каскадних збоїв і Circuit Breaker

Каскадний збій (cascading failure) виникає, коли повільність чи недоступність одного сервіса призводить до вичерпання ресурсів (потоків, з'єднань) у сервісів, що його викликають, які, своєю чергою, стають повільними для тих, хто викликає вже їх — проблема поширюється "хвилею" по всій системі.

Патерн Circuit Breaker (запобіжник) захищає від цього: якщо виклики до певного сервіса стабільно завершуються помилкою, "запобіжник" тимчасово "розмикається" — подальші виклики одразу повертають помилку (чи запасну відповідь) без реальної спроби мережевого запиту, даючи проблемному сервісу час відновитися, замість того щоб продовжувати його "бомбардувати" запитами.

Стан	Поведінка
Closed (замкнений)	Запити проходять як завжди; підраховуються помилки
Open (розімкнений)	Запити одразу відхиляються без звернення до мережі
Half-Open (напіввідкритий)	Пропускається обмежена кількість пробних запитів для перевірки відновлення

Реалізацію Circuit Breaker у production-системах зазвичай не пишуть з нуля — використовують готові бібліотеки (наприклад, ganesha чи Symfony's CircuitBreaker) або виносять цю логіку на рівень інфраструктури (service mesh на кшталт Istio). Для навчальних цілей важливо розуміти сам принцип: система повинна "здаватися швидко" (fail fast) замість нескінченного очікування.

5. Черги повідомлень: концепція Publish/Subscribe

Черга повідомлень (message queue) — проміжний компонент інфраструктури, що приймає повідомлення від сервісів-відправників (producers) і доставляє їх сервісам-отримувачам (consumers), не вимагаючи, щоб обидві сторони були одночасно доступні.

5.1. Основні поняття

Термін	Значення
Producer (видавець)	Сервіс, що надсилає (публікує) повідомлення в чергу
Consumer (споживач)	Сервіс, що отримує й обробляє повідомлення з черги
Queue (черга)	Впорядкований список повідомлень, що очікують обробки
Exchange (обмінник)	Компонент, що маршрутизує повідомлення від видавця до потрібної черги (у RabbitMQ)
Topic (тема)	Іменований канал повідомлень, на який можуть підписатися кілька споживачів

Патерн Publish/Subscribe (видавець/підписник) дозволяє одному повідомленню мати кількох незалежних отримувачів: наприклад, подія "замовлення створено" може одночасно зацікавити сервіс сповіщень (надіслати email), сервіс аналітики (записати статистику) і сервіс складу (зарезервувати товар) — видавцю не потрібно знати про жодного з них.

Ключова перевага такої розв'язки (decoupling): сервіс Orders публікує подію "order.created" один раз, а завтра можна додати новий сервіс-підписник (наприклад, сервіс лояльності для нарахування балів), не змінюючи жодного рядка коду в Orders.

6. RabbitMQ на практиці

RabbitMQ — один із найпопулярніших брокерів повідомлень, що реалізує протокол AMQP (Advanced Message Queuing Protocol). У PHP для роботи з RabbitMQ найчастіше використовують бібліотеку `php-amqplib`.

6.1. Публікація повідомлення (Producer)

```
<?php
use PhpAmqpLib\Connection\AMQPStreamConnection;
use PhpAmqpLib\Message\AMQPMessage;

$connection = new AMQPStreamConnection('localhost', 5672, 'guest',
    'guest');
$channel = $connection->channel();

$channel->queue_declare('order_created', false, true, false, false);

$data = json_encode(['orderId' => 501, 'userId' => 42]);
$message = new AMQPMessage($data, ['delivery_mode' => 2]); // 2 =
зберегти на диску

$channel->basic_publish($message, '', 'order_created');

$channel->close();
$connection->close();
```

6.2. Отримання повідомлення (Consumer)

```
<?php
$channel->queue_declare('order_created', false, true, false, false);

$callback = function (AMQPMessage $msg) {
    $data = json_decode($msg->getBody(), true);
    echo "Обробка замовлення #" . $data['orderId'] . "\n";

    // Надіслати email, оновити статистику тощо

    $msg->ack(); // підтвердити успішну обробку – повідомлення
видаляється з черги
};

$channel->basic_consume('order_created', '', false, false, false, false,
    $callback);

while ($channel->is_consuming()) {
    $channel->wait(); // очікувати на нові повідомлення нескінченно
}
```

Виклик `$msg->ack()` критично важливий: без нього RabbitMQ вважатиме, що повідомлення не було успішно оброблено (наприклад, якщо `consumer` впав під час обробки), і поверне його в чергу для повторної доставки — це забезпечує гарантію "хоча б один раз доставлено" (*at-least-once delivery*), навіть якщо обробник тимчасово недоступний.

7. Redis: Pub/Sub та черги задач

Redis — сховище даних у пам'яті (in-memory), яке часто використовують для кешування, але воно також має вбудовані механізми Pub/Sub і може слугувати простою чергою повідомлень для менш критичних сценаріїв, де RabbitMQ був би надмірно складним рішенням.

7.1. Redis Pub/Sub

Redis Pub/Sub реалізує класичний патерн видавець/підписник: видавець публікує повідомлення в канал (channel), а всі підписники, що слухають цей канал у момент публікації, миттєво отримують його.

```
<?php
// Видавець
$redis = new Redis();
$redis->connect('localhost', 6379);
$redis->publish('order_created', json_encode(['orderId' => 501]));

// Підписник (в окремому процесі)
$redis = new Redis();
$redis->connect('localhost', 6379);
$redis->subscribe(['order_created'], function ($redis, $channel, $message)
{
    $data = json_decode($message, true);
    echo "Отримано замовлення #" . $data['orderId'];
});
```

Важлива відмінність від RabbitMQ: Redis Pub/Sub не зберігає повідомлення, якщо жоден підписник не був підключений у момент публікації — повідомлення просто втрачається (fire-and-forget). RabbitMQ, навпаки, зберігає повідомлення в черзі, доки якийсь споживач не забере й не підтвердить його. Redis Pub/Sub підходить для сценаріїв, де втрата окремого повідомлення не критична (напр. оновлення "хто зараз онлайн"), а RabbitMQ — коли кожне повідомлення важливе (напр. підтвердження оплати).

7.2. Redis як проста черга задач

Крім Pub/Sub, структуру даних "список" (list) у Redis можна використати як просту чергу FIFO: видавець додає елементи в кінець списку (LPUSH), а споживач забирає їх з іншого кінця (BRPOP), блокуючись в очікуванні, якщо черга порожня — на відміну від Pub/Sub, тут повідомлення зберігається, доки хтось його не забере.

```

<?php
// Видавець додає задачу в чергу
$redis->lPush('email_queue', json_encode(['to' => 'user@example.com']));

// Споживач забирає задачі одну за одною (блокується, якщо черга порожня)
while (true) {
    $job = $redis->brPop('email_queue', 0);
    $data = json_decode($job[1], true);
    // Надіслати email...
}

```

Критерій	RabbitMQ	Redis (Pub/Sub або списки)
Гарантія доставки	Висока (підтвердження, збереження на диску)	Нижча (Pub/Sub) або базова (списки)
Складність налаштування	Вища	Нижча — часто вже є в проєкті для кешування
Типовий сценарій	Критичні бізнес-події (оплати, замовлення)	Легкі, некритичні задачі (сповіщення, кеш-інвалідація)

Питання для самоперевірки

- Чому запит до іншого сервісу обов'язково повинен мати таймаут, навіть якщо цей сервіс "завжди працює стабільно"?
- Яку помилку REST-запиту доцільно повторити (retry), а яку — ні? Наведіть по одному прикладу.
- Порівняйте REST і gRPC за форматом даних і типовою сферою застосування.
- Що таке каскадний збій і як Circuit Breaker йому запобігає?
- Опишіть три стани Circuit Breaker і поведінку системи в кожному з них.
- Чим Publish/Subscribe принципово відрізняється від прямого виклику REST API?
- Для чого потрібен виклик `msg->ack()` при обробці повідомлення з RabbitMQ?
- Чим Redis Pub/Sub відрізняється від черги на основі списків Redis (LPUSH/BRPOP) з погляду гарантії доставки?

Короткий глосарій

Термін	Означення
Таймаут	Максимальний час очікування відповіді перед примусовим перериванням запиту
Retry / backoff	Повторна спроба запиту зі зростаючою затримкою між спробами
gRPC	Фреймворк RPC-взаємодії на основі Protocol Buffers та HTTP/2
Каскадний збій	Поширення проблеми одного сервіса на всю систему через ланцюг залежностей
Circuit Breaker	Патерн, що тимчасово блокує виклики до сервіса, який стабільно повертає помилки
Producer / Consumer	Сервіс-відправник і сервіс-отримувач повідомлень у черзі
Publish/Subscribe	Патерн, у якому одне повідомлення отримує кілька незалежних підписників
At-least-once delivery	Гарантія, за якою повідомлення доставляється хоча б один раз (можливі повтори)