Flutter

Полнодуплексное соединение

Матюнин Михаил, 2023 г.

Виды соединений

Симплексное соединение (Simplex): Одно устройство является только отправителем, а другое — только получателем. Как радио.

Полудуплексное соединение (Half-duplex): Устройства могут передавать и принимать данные, но не одновременно. Как рация или HTTP запросы.

Полнодуплексное соединение (Full-duplex): Устройства могут передавать и принимать данные одновременно. Как телефонный разговор или WebSocket.

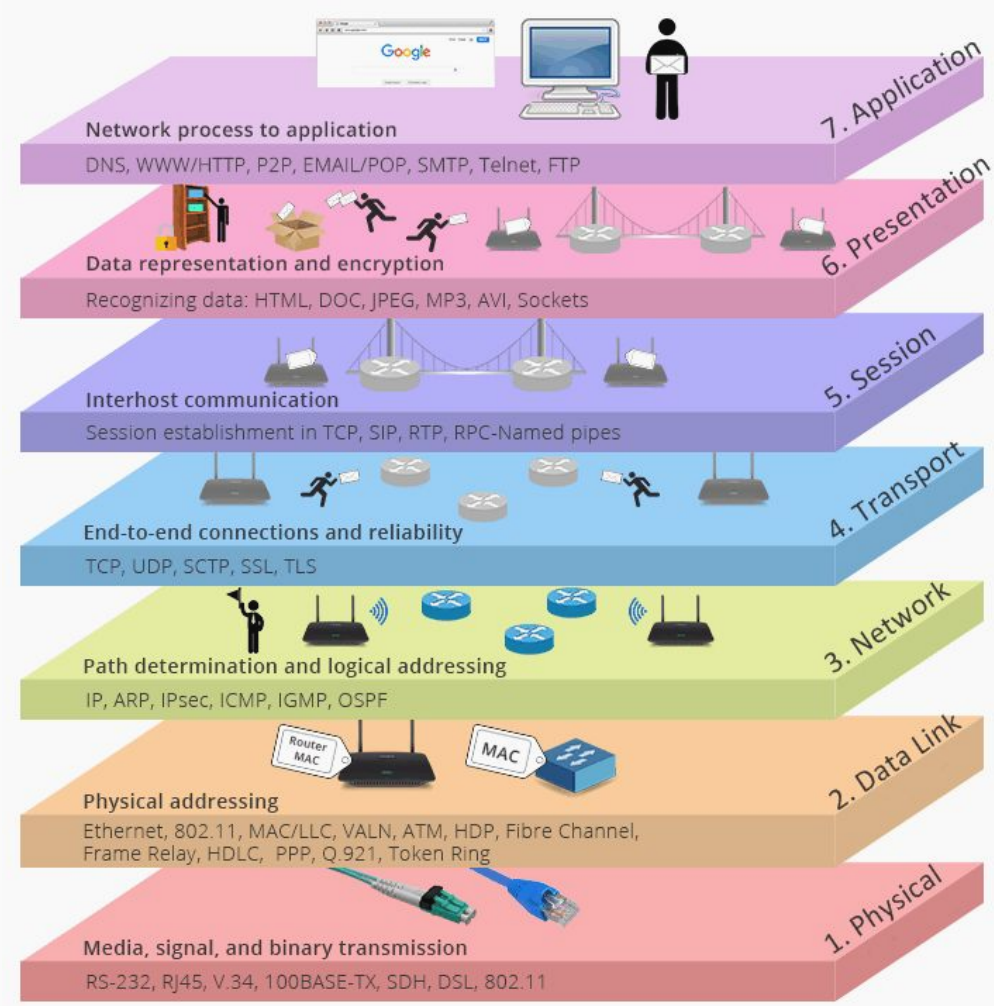
- OSI
- Client & Server
- Push & Pull
- TCP/IP
- DNS
- HTTP 1.1
- Polling
- WebSocket
- Centrifuge
- HTTP/2
- gRPC

OSI

OSI

Модель OSI (Open Systems Interconnection) – это концептуальная модель, которая разделяет функции сетевого протокола на семь уровней или слоев.

Её цель – стандартизировать функции сети в несколько слоев, каждый из которых выполняет конкретные функции.

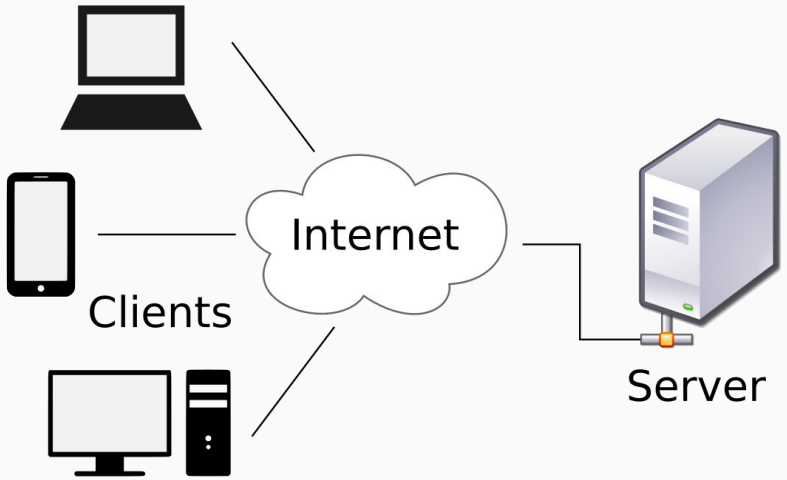


Client & Server

Client & Server

Клиент-сервер – архитектура, в которой один компьютер (клиент) запрашивает данные у другого компьютера (сервера). Клиент обычно представляет собой приложение, в то время как сервер централизованный компьютер.

Примером является посещение веб-сайта: ваш браузер (клиент) делает запрос к веб-серверу, который создает и отправляет запрошенную страницу обратно.

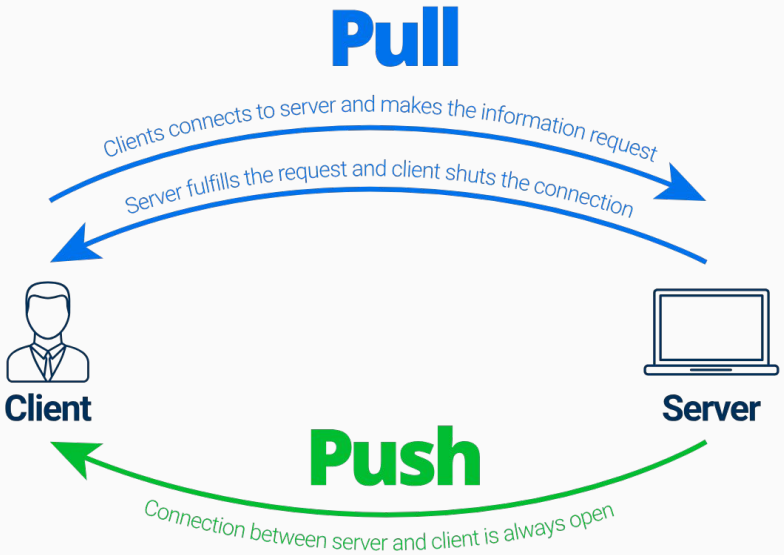


Push & Pull

Push & Pull

Pull: Это когда клиент активно запрашивает или "пуллит" информацию с сервера, когда ему это необходимо. Пример: получение новостной сводки.

Push: Это когда сервер активно отправляет или "пушит" информацию на клиент без предварительного запроса от клиента. Пример: уведомления на вашем смартфоне.



TCP/IP

TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) – это набор сетевых протоколов, который служит основой для передачи данных в современном интернете.

- **TCP** гарантирует, что данные приходят целыми и в правильном порядке. Гарантия точной доставки данных.
- **IP** отвечает за то, чтобы пакеты данных достигали нужного адреса. Это ваш адрес.

TCP handshake – это трехшаговый процесс установления соединения между двумя компьютерами:

1. Клиент отправляет запрос на соединение (SYN).
2. Сервер отвечает подтверждением и своим запросом (SYN, ACK).
3. Клиент отправляет последнее подтверждение (ACK).

Создадим сервер

Код реализует TCP-сервер на Dart:

Кодек: Преобразуют данные между JSON и байтами.

Функция server:
* Запускает сервер.
* При подключении клиента выводит информацию о нем.
* Декодирует входящие JSON-сообщения от клиента.
* Отправляет клиенту ответные JSON-сообщения.
* Возвращает функцию для закрытия сервера.

```dart
import 'dart:async';
import 'dart:convert';
import 'dart:io';

/// Преобразователь JSON-данных в байтовый поток и обратно
typedef JsonMap = Map<String, Object?>;
final codec = (
  encoder: const JsonEncoder().fuse<List<int>>(const Utf8Encoder()),
  decoder: const Utf8Decoder().fuse(const JsonDecoder()).cast<Object?, JsonMap>(),
);

/// Запускает сервер и возвращает функцию, которая закроет сервер
Future<Future<void> Function()> server(InternetAddress address, int port) async {
  // Создаем сервер на указанном адресе и порту
  final server = await ServerSocket.bind(address, port);
  print('Server started on ${server.address.address}:${server.port}');

  // Ожидаем новых соединений
  final sub = server.listen(
    (client) {
      print('Client connected: '
          '${client.remoteAddress.address}:${client.remotePort}');

      // При получении данных от клиента
      client.listen((bytes) {
        // Обрабатываем запрос
        final request = codec.decoder.convert(bytes);
        final <String, Object?>{'id': id, 'message': message} = request;
        print('> $message');

        // Отправляем ответ клиенту
        final responseBytes = codec.encoder.convert(<String, Object?>{
          'id': id,
          'message': 'Hello, Client #$id! I am Server.',
        });
        client.add(responseBytes);
      }, onDone: () {
        print('Client disconnected: '
            '${client.remoteAddress.address}:${client.remotePort}');
        client.close();
      });
    },
    cancelOnError: false,
  );

  /// Возвращаем функцию, которая закроет сервер и все соединения
  return () async {
    await sub.cancel();
    await server.close();
    print('Server closed');
  };
}
```

Создадим клиент

Код реализует TCP-клиент на Dart:

Соединение: Устанавливается соединение с сервером на localhost:8080.
Отправка: Клиент отправляет серверу JSON-сообщение с id и приветственным текстом.
О**жидание ответа:** Клиент ждет ответ от сервера, декодирует его и выводит на экран.
Завершение: Соединение закрывается, ресурсы освобождаются.

```dart
/// Создает клиента, который отправляет сообщение серверу и выводит ответ
Future<void> client(int id) async {
  // Создаем соединение с сервером на адресе 127.0.0.1 (localhost) и порту 8080
  final socket = await Socket.connect('127.0.0.1', 8080);

  // Отправляем сообщение серверу
  final requestBytes = codec.encoder.convert(<String, Object?>{
    'id': id,
    'message': 'Hello, Server! I am Client #$id',
  });
  socket.add(requestBytes);

  // Ожидаем ответ от сервера и выводим его
  final completer = Completer<JsonMap>();
  final sub = socket.map(codec.decoder.convert).listen(completer.complete);
  final response = await completer.future;
  print('< ${response['message']}');

  // Закрыть соединение после завершения обмена данными
  await sub.cancel();
  await socket.close();
  socket.destroy();
}
```

Запустим

Соединение: Устанавливается соединение с сервером на localhost:8080.

Отправка: Клиент отправляет серверу JSON-сообщение с id и приветственным текстом.

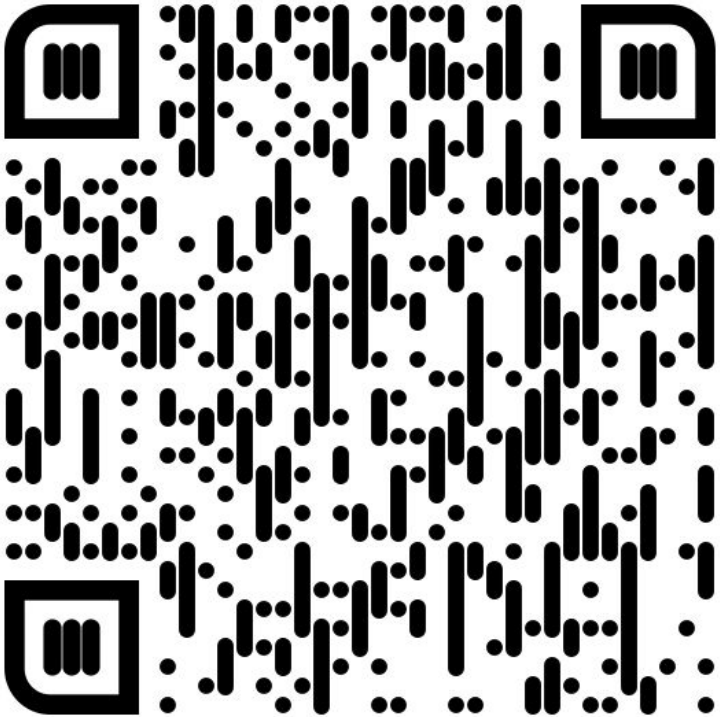
О**жидание ответа:** Клиент ждет ответ от сервера, декодирует его и выводит на экран.

Завершение: Соединение закрывается, ресурсы освобождаются.

```dart
void main([List<String>? arguments]) => Future<void>(() async {
      final delimiter = '\n${'-' * 15}\n';
      Future<void> sleep() => Future<void>.delayed(const Duration(milliseconds: 250));

      // Запускаем сервер на порту 8080
      final close = await server(InternetAddress.loopbackIPv4, 8080);

      print(delimiter);
      await sleep();

      // Запускаем 3 клиента
      for (var i = 1; i <= 3; i++) {
        await client(i);
        await sleep();
        print(delimiter);
      }

      await sleep();

      // Закрываем сервер
      await close();
    });
```

```
Server started on 127.0.0.1:8080

---------------

Client connected: 127.0.0.1:64832
> Hello, Server! I am Client #1
< Hello, Client #1! I am Server.
Client disconnected: 127.0.0.1:64832

---------------

Client connected: 127.0.0.1:64833
> Hello, Server! I am Client #2
< Hello, Client #2! I am Server.
Client disconnected: 127.0.0.1:64833

---------------

Client connected: 127.0.0.1:64834
> Hello, Server! I am Client #3
< Hello, Client #3! I am Server.
Client disconnected: 127.0.0.1:64834

---------------

Server closed
```

Пример обмена данными между клиентом и сервером по протоколу TCP/IP



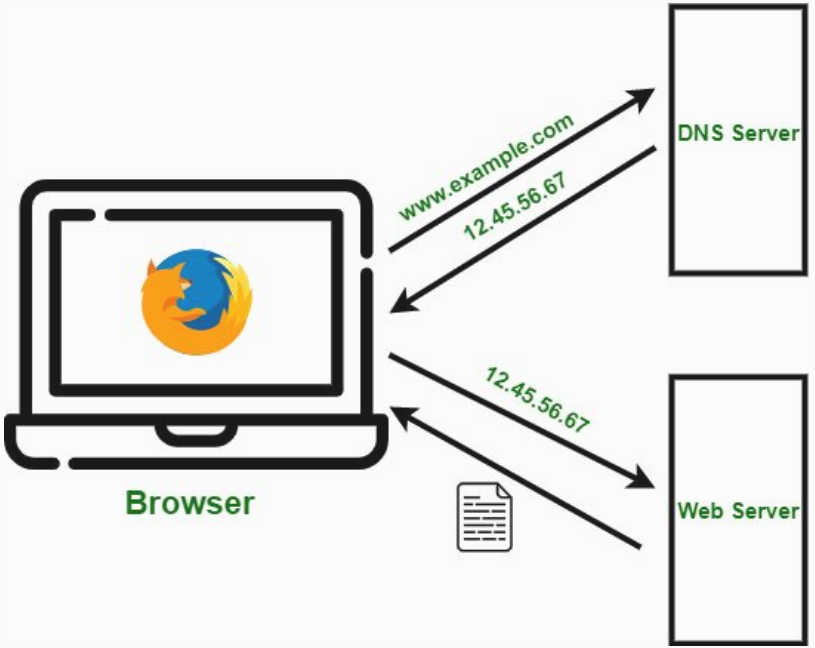
https://gist.github.com/PlugFox/9cc2adb1d8df24fae2164e95dba6a911

DNS

DNS

DNS (Domain Name System) — это система, которая переводит доменные имена в IP-адреса. Она позволяет пользователям и приложениям использовать понятные имена (например, **www.example.com**) вместо сложных IP-адресов (например, **192.0.2.1**).

При обращении к веб-сайту ваш компьютер использует DNS для получения IP-адреса сайта, затем устанавливает соединение и делает HTTP-запрос.

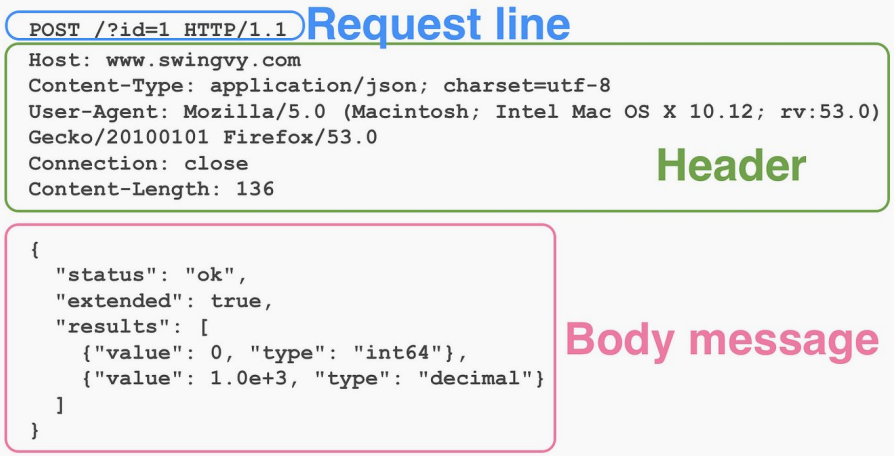


HTTP 1.1

HTTP 1.1

HTTP — это протокол передачи гипертекста.

Соединения: использует постоянные соединения.
Методы: какое действие клиент хочет выполнить с ресурсом.
Статусы: возвращает статус коды.
Заголовки: передача метаданных между клиентом и сервером.
Текстовый: основан на тексте, что делает его удобным для человека.



Пример HTTP сервера

```dart
Future<void> server() async {
  // Создание HTTP-сервера, прослушивающего порт 8080
  final server = await HttpServer.bind(InternetAddress.loopbackIPv4, 8080);
  print('Listening on localhost:${server.port}');

  server.listen((request) async {
    // Отправка ответа клиенту
    request.response.write('Hello from Dart HTTP server!');
    await request.response.close();
  });
}
```

```dart
Future<void> client() async {
  // Создание HTTP-клиента
  final client = HttpClient();

  // Отправка GET-запроса к серверу
  final request = await client.getUrl(Uri.parse('http://localhost:8080'));
  final response = await request.close();

  // Чтение и вывод ответа от сервера
  await response.map(utf8.decode).forEach(print);
}
```

Возможности

HTTP 1.1 принес с собой множество улучшений по сравнению с HTTP 1.0, и некоторые из этих возможностей могут показаться не сразу очевидными.

Постоянные соединения (Keep-Alive): Меньше задержек благодаря пере использованию TCP/IP соединений.

Диапазонные запросы (Range Requests): Это позволяет клиентам запрашивать части ресурса, что особенно полезно для больших ресурсов или для возобновления прерванных загрузок.

Пайплайнинг: Отправка нескольких запросов без ожидания ответа.

Кодирование содержимого: Введены способы сжатия, такие как gzip и deflate, позволяя сократить объем передаваемых данных.

Условные запросы: С помощью заголовков, таких как If-Modified-Since и If-None-Match, клиенты могут запрашивать ресурс только в том случае, если он был изменен.

Кэширование: Улучшенные механизмы кэширования с помощью новых заголовков и директив, таких как Cache-Control.

Polling

Polling

Polling — техника, при которой клиент периодически отправляет запросы серверу для получения новых данных или обновлений.

Пример: Проверка новых сообщений
- Клиент посылает запрос каждые N секунд: "Есть ли для меня новые сообщения после сообщения с ID X?".
- Сервер отвечает с информацией о новых сообщениях или говорит клиенту подождать.

Преимущества:

- Простота реализации и понимания.

Недостатки:

- Может создавать лишний трафик.
- Задержка в получении актуальных данных, равная интервалу опроса.

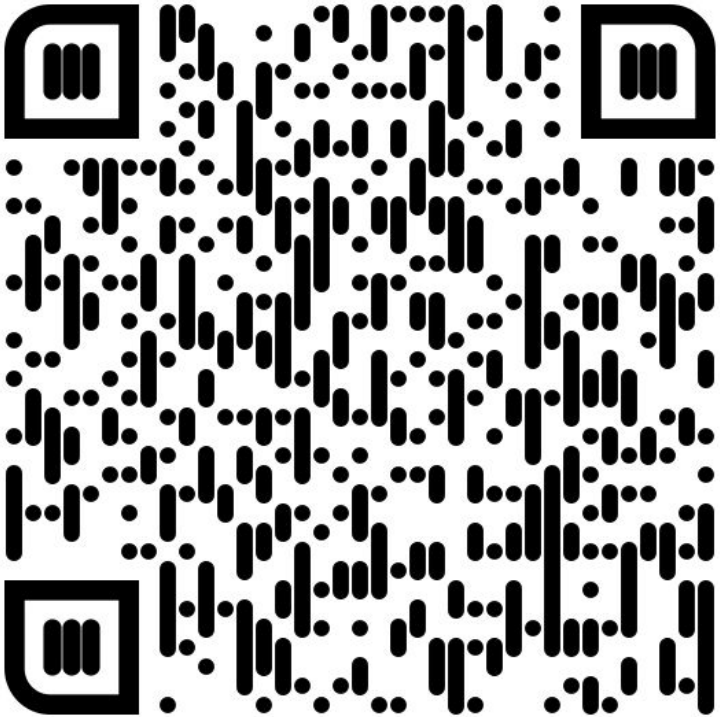
Пример

Код периодически запрашивает сервер на наличие новых сообщений и если таковые есть, выводит их.

- Создаем функцию для поллинга.
- Каждые 12 секунд опрашиваем URL.
- Опционально добавляем идентификатор последнего сообщения.
- Разбираем ответ.
- Возвращаем список сообщений и их идентификаторов.

```dart
import 'dart:async';
import 'dart:convert';
import 'dart:io';
import 'dart:math' as math;

void main([List<String>? arguments]) {
  final fn = polling();
  Timer.periodic(
    const Duration(seconds: 12),
    (timer) async {
      try {
        final messages = await fn();
        messages.forEach(print);
      } on Object {/* ignore */}
    },
  );
}

Future<List<({int id, String text})>> Function() polling() {
  const kEmptyList = <({int id, String text})>[];
  final url = Uri.parse('http://localhost:8080/getUpdates');
  final client = HttpClient();
  int? $lastId;
  Uri createUrl() => switch ($lastId) {
        int id => url.replace(queryParameters: {'id': id.toString()}),
        _ => url,
      };

  return () async {
    final request =
        await client.getUrl(createUrl()).timeout(const Duration(seconds: 6));
    final response = await request.close();
    if (response.statusCode != HttpStatus.ok) throw Exception('bad_status_code');
    final body = await response.transform<String>(utf8.decoder).join();
    if (body.length < 3) return kEmptyList;
    final messages = (jsonDecode(body) as Iterable)
        .whereType<Map<String, Object?>>()
        .expand<({int id, String text})>((e) => switch (e) {
              {'id': int id, 'text': String text} => [(id: id, text: text)],
              _ => kEmptyList,
            })
        .toList(growable: false);
    if (messages.isEmpty) return kEmptyList;
    $lastId = <int>[
      if ($lastId != null) $lastId!,
      ...messages.map<int>((e) => e.id),
    ].reduce(math.max);
    return messages;
  };
}
```

Пример опросов по таймауту



https://gist.github.com/PlugFox/b6e5bbbbfe79157c8229f1ba4808fbe9

Long Polling (Длинные опросы)

Long Polling — это вариация **Polling**, где сервер "удерживает" запрос, пока не появится новая информация или не истечет таймаут.

Пример: Проверка новых сообщений
- Клиент посылает запрос: "Есть ли для меня новые сообщения после сообщения с ID X?" и ожидает ответа.
- Сервер задерживает ответ до тех пор, пока не появятся новые сообщения или не истечет время ожидания.

Преимущества:

- Уменьшается общее количество запросов по сравнению с обычным Polling.
- Получение новых данных практически мгновенно после их появления.

Недостатки:

- Ресурсы со стороны сервера на поддержание открытого соединения.

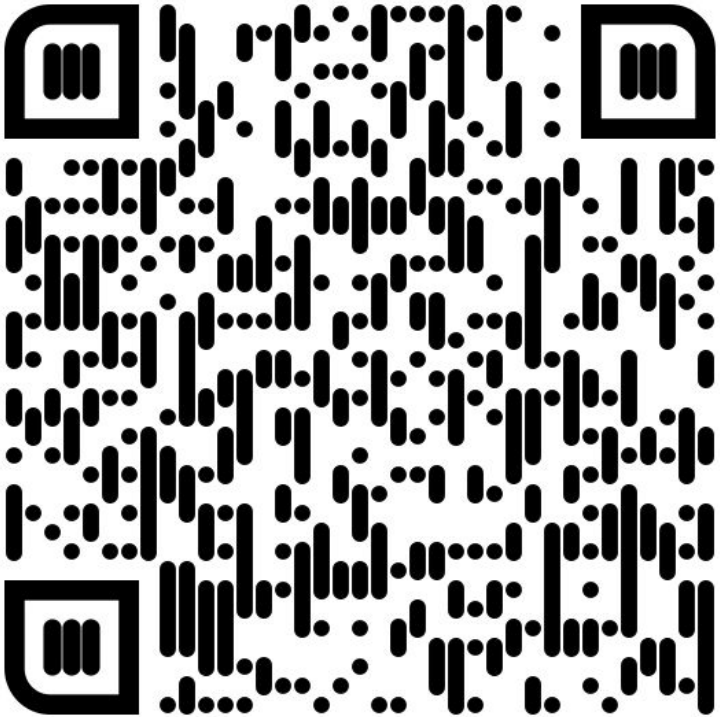
Пример

Код опрашивает сервер на наличие новых сообщений в цикле и если таковые есть, выводит их.

- Создаем функцию для длинных опросов.
- Опрашиваем в цикле.
- Опционально добавляем таймаут ожидания ответа для сервера.
- Опционально добавляем идентификатор последнего сообщения.
- Разбираем ответ.
- Возвращаем список сообщений и их идентификаторов.

```dart
import 'dart:async';
import 'dart:convert';
import 'dart:io';
import 'dart:math' as math;

void main([List<String>? arguments]) {
  final fn = polling();
  Future.doWhile(() async {
    try {
      final messages = await fn();
      messages.forEach(print);
    } on Object {/* ignore */}
    return true;
  });
}

Future<List<({int id, String text})>> Function() polling([
  Duration timeout = const Duration(seconds: 12),
]) {
  const kEmptyList = <({int id, String text})>[];
  final url = Uri.parse('http://localhost:8080/getUpdates');
  final client = HttpClient();
  int? $lastId;
  Uri createUrl() => url.replace(queryParameters: <String, Object?>{
        if ($lastId != null) 'id': $lastId?.toString(),
        'timeout': timeout.inMilliseconds.toString(),
      });

  return () async {
    final request = await client.getUrl(createUrl());
    final response = await request.close();
    if (response.statusCode != HttpStatus.ok) throw Exception('bad_status_code');
    final body = await response.transform<String>(utf8.decoder).join();
    if (body.length < 3) return kEmptyList;
    final messages = (jsonDecode(body) as Iterable)
        .whereType<Map<String, Object?>>()
        .expand<({int id, String text})>((e) => switch (e) {
              {'id': int id, 'text': String text} => [(id: id, text: text)],
              _ => kEmptyList,
            })
        .toList(growable: false);
    if (messages.isEmpty) return kEmptyList;
    $lastId = <int>[
      if ($lastId != null) $lastId!,
      ...messages.map<int>((e) => e.id),
    ].reduce(math.max);
    return messages;
  };
}

```

Пример длинных опросов



https://gist.github.com/PlugFox/51038cb6f587332f6e803790a7d991e5

WebSocket

WebSocket

WebSocket — это протокол, позволяющий двустороннее взаимодействие между клиентом и сервером через постоянное соединение.

Пример: Чат в реальном времени
- Клиент и сервер устанавливают постоянное соединение.
- Любая сторона может отправлять и получать сообщения без необходимости повторного установления соединения.

Преимущества:

- Мгновенная передача данных без задержек.
- Эффективное переиспользование ресурсов.

Недостатки:

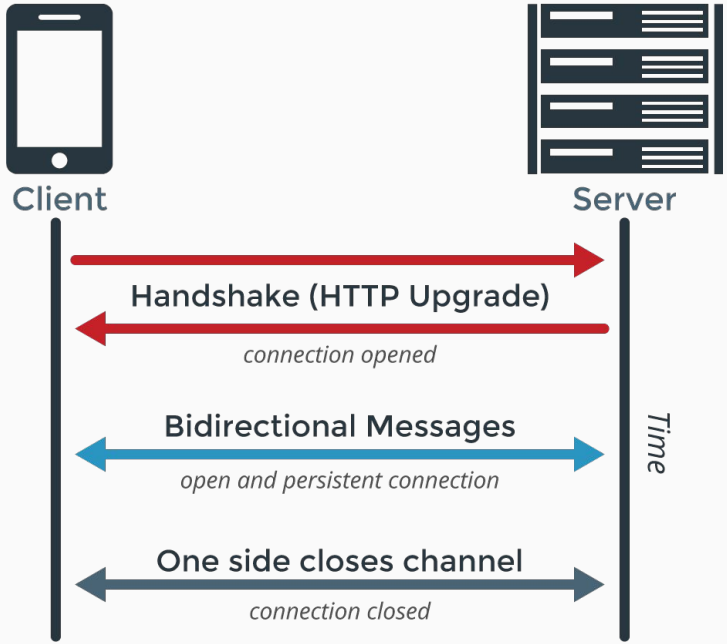
- Сложнее в реализации по сравнению с опросами.

Как происходит WebSocket подключение

Запрос на установку: Клиент отправляет специальный HTTP-запрос, называемый "WebSocket handshake", на сервер. Этот запрос содержит заголовок **Upgrade: websocket**.

Подтверждение сервера: Если сервер поддерживает WebSocket и готов установить соединение, он отправляет ответ со статусом 101 и заголовком **Connection: Upgrade**.

Соединение установлено: После того как рукопожатие завершено, соединение превращается из HTTP в WebSocket, и данные могут передаваться в обоих направлениях без необходимости повторного установления соединения.



Нюансы при использовании WebSocket

Заголовки подключения: В браузерной среде Dart (**dart:html**) нельзя напрямую задать заголовки при установке WebSocket-соединения.

Статус соединения: Спецификация предполагает пинг/понг фреймы, но хорошей идеей будет добавить свою логику.

Гарантии доставки: WebSocket не предоставляет встроенных гарантий доставки сообщений.

Вызов удаленных процедур (RPC): WebSocket не имеет встроенного механизма RPC.

Переподключение: WebSocket по умолчанию не предоставляет механизма автоматического переподключения.

Различия в API: Dart имеет разные API для WebSocket в веб-среде (**dart:html**) и на мобильных/десктопных платформах (**dart:io**).

Пакет “ws”

- Кроссплатформенность
- Указание протоколов
- Состояния подключения
- Очередность
- Переподключение
- “Jitter” стратегия
- Шорткаты у стримов
- Метрики и счетчики
- Платформенные опции
- Расширяемость
- Покрытие > 90%

```dart
void main([List<String>? args]) {
  // The Web Socket server URL.
  // Pass it as `--define=URL=...` for `String.fromEnvironment('URL')`
  // or extract from env by `Platform.environment['URL']`
  // or parse it using `dart:args`
  const url = String.fromEnvironment('URL',
      defaultValue: 'wss://echo.plugfox.dev:443/connect');

  // Setup a WebSocket client with auto reconnecting
  // Also, we can enqueue sending before the connection is established.
  final client = WebSocketClient(
    // Common options for all platforms
    // Or use `.js(..)`, `.vm(..)`, `.selector(..)` instead of `.common(..)`
    WebSocketOptions.common(
      // The delay between reconnection attempts will be between 500ms and 15s
      connectionRetryInterval: (
        min: const Duration(milliseconds: 500),
        max: const Duration(seconds: 15),
      ),
    ),
  )
    // Observing the incoming messages from the server
    ..stream.listen((message) => print('< $message'))
    // Observing the state changes (connecting, open, disconnecting, closed)
    ..stateChanges.listen((state) => print('* $state'))
    // Connect to the server url
    ..connect(url)
    // Send a message
    ..add('Hello, ') // > Hello,
    // One more message after first is sent
    ..add('world!'); // > world!

  // Close the connection after 1 second
  Timer(const Duration(seconds: 1), () async {
    await client.close(); // Close the connection
    print('Metrics:\n${client.metrics}'); // Print the metrics
    io.exit(0); // Exit the process
  });
}
```

https://pub.dev/packages/ws

Структура запроса и ответа

По WebSocket вы можете передавать данные в абсолютно любом виде, от строк до двоичных данных.

Вот несколько идей для стандартизации:

- JSON Object
- Первые N байт указывают на структуру
- Protobuf
- Multipart/Form-Data

```
{
    "status": "ok" || "error",
    "data": Object,
    "errors": [
        {
            "code": String,
            "message": String
        }
    ]
}
```

Centrifuge

Centrifuge

Centrifuge – это высокопроизводительный сервер на Go от Александра Емелина для работы с веб-сокетами и другими транспортами в режиме реального времени. Он является частью проекта **Centrifugo**, который также предоставляет клиентские библиотеки для различных платформ. Можно использовать как библиотеку, так и самостоятельный софт.

https://centrifugal.dev

https://github.com/centrifugal/centrifugo

Транспорт: Centrifuge поддерживает несколько транспортных протоколов, включая WebSocket, SockJS и gRPC.
Публикация-Подписка (Pub/Sub): Клиенты могут подписываться на каналы и получать обновления.
Присутствие: Сервер может отслеживать, какие клиенты подписаны на определенный канал.
История: Centrifuge может хранить историю сообщений в каналах.
Аутентификация: Безопасность поддерживается с помощью JWT (JSON Web Tokens).
Горизонтальное масштабирование: С помощью Redis можно организовать горизонтальное масштабирование.
Клиентские библиотеки: Существуют клиентские библиотеки для множества языков и платформ.
Переподключение: Centrifuge имеет встроенный механизм автоматического переподключения.
Пинг: Есть механизм обнаружения проблем с соединением.
RPC вызовы: Centrifuge предлагает механизм удаленных вызовов процедур (RPC — Remote Procedure Call).

HTTP/2

HTTP/2

HTTP/2 — это вторая версия протокола передачи гипертекста (HTTP), которая призвана улучшить производительность веб-приложений, уменьшить задержки и усилить безопасность.

https://pub.dev/packages/http2

Полнодуплексное соединение: Возможность одновременной передачи данных в обе стороны без задержек.
Многопоточность: Несколько параллельных потоков данных в одном соединении, что ускоряет обмен информацией.
Стриминг: Разбивка данных на фреймы для эффективного и непрерывного потокового ввода-вывода.
Сжатие заголовков: HPACK сжатие сокращает объем передаваемых заголовков, делая передачу данных быстрее.
Приоритизация: Клиент может указывать, какие ресурсы ему нужны быстрее всего, оптимизируя загрузку.
Устойчивость к задержке: Благодаря многопоточности, задержка одного потока не замедляет остальные.
Бинарный протокол: Позволяет эффективнее передавать и интерпретировать данные, упрощая обработку на стороне сервера.
Серверный push: Сервер может отправлять клиенту данные, которые скоро понадобятся, без явного запроса.

gRPC

gRPC

gRPC — это открытый фреймворк для удаленного вызова процедур (RPC), который разработан Google.

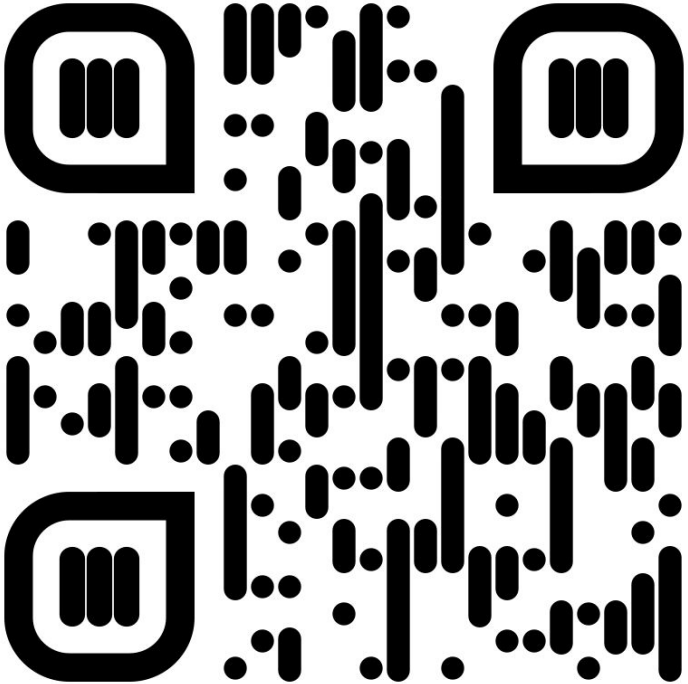
Основные детали gRPC:

1. Protobuf (Protocol Buffers)
2. HTTP/2
3. RPC

https://pub.dev/packages/grpc

```proto
syntax = "proto3";

package echo;

message EchoRequest {
    string message = 1; // The message to be echoed.
}

message EchoResponse {
    string message = 1; // The echoed message.
}

message ServerStreamingEchoRequest {
    string message = 1; // The message to be echoed.
    int32 message_count = 2; // The number of times to echo the message.
    int32 message_interval = 3; // The interval between each echoed message in ms.
}

message ServerStreamingEchoResponse {
    string message = 1; // The echoed message.
}

service EchoService {
    // Echo is a simple RPC that echoes the message in the request.
    rpc Echo(EchoRequest) returns (EchoResponse);

    // A server streaming RPC that echoes the message in the request multiple times.
    rpc ServerStreamingEcho(ServerStreamingEchoRequest)
        returns (stream ServerStreamingEchoResponse);
}
```

Сайт



https://plugfox.dev