ДИПЛОМНА РАБОТА


# Проектиране и разработване на библиотека на C++, подпомагаща създаването на уеб приложения

Coroute: Coroutine-Based Multi-Protocol Web Framework


| Изготвил | Факултетен номер | Специалност |
|---|---|---|
| Алекс Цветанов | 121222225 | Компютърно и софтуерно инженерство |

Научен ръководител
доц. д-р Иван Станков
Катедра "Киберсигурност"


Технически университет – София, 8 декември 2025 г.

# Съдържание

## Резюме

Настоящата дипломна работа представя проектирането, реализацията и експерименталната оценка на Coroute — високопроизводителна C++ библиотека за изграждане на уеб приложения, базирана на съвременните възможности на стандарта C++20. Основната цел на изследването е да се демонстрира, че комбинацията от stackless корутини, операционно-специфични асинхронни I/O механизми и алгоритмично оптимизирано маршрутизиране може да постигне производителност, съизмерима с водещите индустриални решения, при значително опростен програмен модел.

Архитектурата на Coroute се основава на няколко ключови иновации. Първо, изпълнителният модел използва типа `Task<T>`, който реализира lazy корутини с поддръжка на detached изпълнение и кооперативно прекратяване. Този модел елиминира традиционните недостатъци на callback-базираното асинхронно програмиране, като запазва ефективността на неблокиращия I/O. Второ, мрежовият слой абстрахира платформено-специфичните механизми — IOCP за Windows и io_uring за Linux — зад унифициран интерфейс, позволявайки zero-copy операции и batching на системни извиквания. Трето, маршрутизирането използва DFA-базиран алгоритъм, публикуван в „Matching Text from Start to Finish Against Multiple Regular Expressions" [20], който постига O(N) времева сложност спрямо дължината на URL-а, независимо от броя на регистрираните маршрути.

Библиотеката поддържа множество протоколи върху един порт чрез ALPN негоциация: HTTP/1.1 с keep-alive, HTTP/2 с HPACK компресия и stream мултиплексиране, и WebSocket за двупосочна комуникация в реално време. Типово-безопасното извличане на параметри използва C++20 concepts за compile-time валидация, елиминирайки цял клас runtime грешки.

Експерименталната оценка, проведена на система с AMD Ryzen 5 3600 (6 ядра, 12 нишки) и до 1024 конкурентни клиента, демонстрира throughput от 1,378,578 заявки в секунда за прости HTTP отговори, с медианна латентност от 212 микросекунди и 0% error rate. Сравнителният анализ с Crow, Boost.Beast и Drogon показва, че Coroute постига сравнима или по-добра производителност при значително по-прост API.

Научният принос на работата включва: интеграция на DFA-базирано маршрутизиране в пълноценен уеб библиотека; корутинен изпълнителен модел, специално проектиран за мрежови приложения; и демонстрация на практическата приложимост на подхода чрез production-scale примерно приложение.



**Ключови думи:** C++20, корутини, уеб сървър, HTTP/2, WebSocket, IOCP, io_uring, DFA маршрутизиране, типова безопасност, асинхронен I/O



## Abstract

This thesis presents the design, implementation, and experimental evaluation of Coroute — a high-performance C++ library for building web applications based on modern C++20 features. The primary objective is to demonstrate that the combination of stackless coroutines, OS-native asynchronous I/O mechanisms, and algorithmically optimized routing can achieve performance comparable to leading industrial solutions while significantly simplifying the programming model.

The architecture of Coroute is founded on several key innovations. The execution model employs the `Task<T>` type, implementing lazy coroutines with support for detached execution and cooperative cancellation. This model eliminates the traditional drawbacks of callback-based asynchronous programming while preserving

the efficiency of non-blocking I/O. The network layer abstracts platform-specific mechanisms — IOCP for Windows and io_uring for Linux — behind a unified interface, enabling zero-copy operations and syscall batching. Routing utilizes a DFA-based algorithm, published in "Matching Text from Start to Finish Against Multiple Regular Expressions" [20], achieving O(N) time complexity relative to URL length, independent of the number of registered routes.

The library supports multiple protocols on a single port via ALPN negotiation: HTTP/1.1 with keep-alive, HTTP/2 with HPACK compression and stream multiplexing, and WebSocket for bidirectional real-time communication. Type-safe parameter extraction leverages C++20 concepts for compile-time validation, eliminating an entire class of runtime errors.

Experimental evaluation on a system with AMD Ryzen 5 3600 (6 cores, 12 threads) and up to 1024 concurrent clients demonstrates throughput of 1,378,578 requests per second for simple HTTP responses, with median latency of 212 microseconds and 0% error rate. Comparative analysis with Crow, Boost.Beast, and Drogon shows that Coroute achieves comparable or superior performance with a significantly simpler API.

The scientific contributions include: integration of DFA-based routing into a full-featured web framework; a coroutine execution model specifically designed for network applications; and demonstration of practical applicability through a production-scale example application.

**Keywords:** C++20, coroutines, web server, HTTP/2, WebSocket, IOCP, io_uring, DFA routing, type safety, asynchronous I/O

# 1 Въведение

## 1.1 Актуалност на проблема

През последните две десетилетия уеб технологиите претърпяха драматична трансформация. От статични HTML страници, обслужвани от прости CGI скриптове, се премина към сложни интерактивни приложения, работещи в реално време. Съвременните уеб услуги трябва да обработват хиляди, а понякога и милиони едновременни връзки, като същевременно поддържат латентност от порядъка на милисекунди.

Традиционните подходи за изграждане на уеб сървъри се базират на модела "една нишка на връзка" (thread-per-connection). При този модел всяка входяща връзка се обслужва от отделна нишка, която блокира докато чака данни от клиента или от базата данни. Макар и концептуално прост, този подход се сблъсква със сериозни ограничения при мащабиране. Създаването на нишка консумира значителни системни ресурси – типично около 1MB памет за стека. При хиляди едновременни връзки консумацията на памет става неприемлива. Освен това, контекстното превключване между нишките въвежда допълнителен overhead, който намалява общата производителност на системата.

Тези ограничения мотивираха разработването на асинхронни модели на изпълнение. При асинхронния подход малък брой нишки (обикновено равен на броя на CPU ядрата) обслужва множество връзки, като използва неблокиращи I/O операции и event loops. Когато една операция не може да завърши веднага, вместо да блокира нишката, тя регистрира callback функция, която ще бъде извикана при завършване на операцията. Междувременно нишката може да обслужва други заявки.

Езикът C++ остава предпочитан избор за системно програмиране и приложения с високи изисквания към производителността. Неговата способност да генерира оптимизиран машинен код, съчетана с богата стандартна библиотека и възможности за нискониво програмиране, го правят идеален за изграждане на високопроизводителни мрежови приложения.

С въвеждането на корутините в стандарта C++20 [11] се открива нова възможност за създаване на асинхронен код, който е едновременно ефективен и четим. Корутините са функции, които могат да спрат изпълнението си в определени точки и да го възобновят по-късно, без да блокират нишката. Това позволява писане на код, който изглежда синхронен и последователен, но всъщност се изпълнява асинхронно. По този начин се елиминира т.нар. "callback hell" – проблемът с дълбоко вложени callback функции, характерен за традиционните асинхронни API-та.

## 1.2 Цели и задачи

Основната цел на настоящата дипломна работа е проектирането и реализацията на модерна C++ библиотека за създаване на уеб приложения, която да отговаря на съвременните изисквания за производителност, надеждност и удобство при разработка.

Библиотеката трябва да **опростява разработката** на уеб приложения чрез интуитивен API, базиран на C++20 корутини. Разработчиците трябва да могат да пишат асинхронен код със синтаксис, подобен на синхронния, без да се налага да управляват ръчно callbacks или promises.

Библиотеката трябва да **осигурява висока производителност** чрез неблокиращ I/O и ефективно управление на ресурсите. Целта е да се постигне производителност, сравнима с водещите C++ уеб библиотеки, като същевременно се запази простотата на API-то.

Библиотеката трябва да **поддържа множество протоколи**, включително HTTP/1.1, HTTP/2 и WebSocket. HTTP/2 е особено важен за съвременните уеб приложения, тъй като позволява мултиплексиране на заявки и ефективна компресия на хедъри. WebSocket е необходим за приложения, изискващи двупосочна комуникация в реално време.

Библиотеката трябва да **предоставя типова безопасност** при извличане на параметри от URL ад-

реси. Вместо да връща параметрите като низове, които трябва да се конвертират ръчно, библиотеката трябва автоматично да извлича стойности от правилния тип, като грешките се откриват по време на компилация.

Библиотеката трябва да бъде **платформено независима**, работеща на Windows, Linux и macOS. За всяка платформа трябва да се използват оптимизираните native механизми за асинхронен I/O.

Библиотеката трябва да бъде **разширяема**, позволявайки лесно добавяне на нови протоколи и middleware компоненти. Архитектурата трябва да следва принципите на модулност и разделение на отговорностите.

За постигане на тези цели са поставени следните конкретни задачи:

Първата задача е **анализ на съществуващи C++ уеб библиотеки**. Необходимо е да се проучат популярните решения като Boost.Beast, Drogon, Crow и други, да се идентифицират техните предимства и недостатъци, и да се определи какви функционалности липсват на пазара.

Втората задача е **проектиране на модулна архитектура** с ясно разделение на отговорностите. Архитектурата трябва да позволява независимо развитие на отделните компоненти и лесно добавяне на нови функционалности.

Третата задача е **реализация на корутинна инфраструктура**. Необходимо е да се създаде типът `Task<T>`, който да служи като основа за асинхронните операции в библиотеката.

Четвъртата задача е **имплементация на DFA-базиран маршрутизатор** с O(N) сложност, базиран на алгоритъма, описан в „Matching Text from Start to Finish Against Multiple Regular Expressions" [20]. Този алгоритъм позволява ефективно съпоставяне на URL адреси с множество шаблони.

Петата задача е **разработка на платформено-специфични I/O бекенди** за IOCP (Windows), io_uring (Linux) и kqueue (macOS). Всеки бекенд трябва да използва оптимално native механизмите на съответната операционна система.

Шестата задача е **интеграция на HTTP/2** с ALPN негоциация и HPACK компресия на хедъри. HTTP/2 трябва да работи безпроблемно заедно с HTTP/1.1, като протоколът се избира автоматично при TLS handshake.

Седмата задача е **реализация на WebSocket протокол** за двупосочна комуникация. WebSocket трябва да поддържа както текстови, така и бинарни съобщения.

Осмата задача е **тестване и benchmark анализ** на производителността. Необходимо е да се създадат unit тестове за всички компоненти и да се измери производителността в сравнение с други библиотеки.

## 1.3 Обхват и ограничения

При проектирането на всяка софтуерна система е важно ясно да се дефинира какво влиза в обхвата на разработката и какво остава извън него. Това позволява фокусиране на усилията върху ключовите функционалности и избягване на т.нар. “scope creep” – неконтролираното разширяване на изискванията.

Библиотеката Coroute е проектирана като *library*, а не като самостоятелен сървър или framework. Това е съзнателно архитектурно решение, което дава на разработчиците максимална гъвкавост. Библиотеката предоставя градивни блокове за създаване на уеб приложения, но оставя контрола върху конфигурацията, разгръщането и интеграцията с други системи изцяло на разработчика.

В обхвата на настоящата разработка влизат следните функционалности:

Поддръжката на **HTTP/1.1 и HTTP/2 протоколи** е основна функционалност на библиотеката. HTTP/1.1 е все още широко използван и трябва да се поддържа за съвместимост. HTTP/2 предоставя значителни подобрения в производителността и е необходим за съвременни приложения.

**WebSocket протоколът** позволява двупосочна комуникация между клиент и сървър. Това е необходимо за приложения като чат системи, игри в реално време, колаборативни редактори и dashboard-и с live updates.

**TLS криптирането** е задължително за съвременните уеб приложения. Библиотеката трябва да поддържа TLS 1.2 и TLS 1.3, включително ALPN негоциация за избор на HTTP протокол.

**Статичното обслужване на файлове** е често срещано изискване. Библиотеката трябва да поддържа ефективно обслужване на статични ресурси като CSS, JavaScript и изображения, включително zero-copy трансфер където е възможно.

**Middleware системата** позволява добавяне на cross-cutting concerns като логване, автентикация, компресия и rate limiting. Middleware компонентите трябва да могат да се композират в произволен ред.

**Маршрутизирането с параметри** позволява дефиниране на динамични URL шаблони като `/users/{id}`. Параметрите трябва да се извличат автоматично и да се валидират по тип.

Извън обхвата на настоящата разработка остават следните функционалности:

**HTTP/3 (QUIC)** е най-новата версия на HTTP протокола, базирана на UDP вместо TCP. Въпреки че HTTP/3 предоставя допълнителни подобрения в производителността, неговата имплементация е значително по-сложна и е планирана за бъдещи версии на библиотеката.

**Вградена база данни или ORM** не е част от обхвата. Библиотеката се фокусира върху мрежовия слой и оставя избора на database решение на разработчика. Това позволява интеграция с произволна база данни – SQL, NoSQL или in-memory.

**Шаблонна система за генериране на HTML** също не е включена. Разработчиците могат да използват външни библиотеки като inja, mustache или да генерират HTML програмно.

**Автоматичното мащабиране и load balancing** са функционалности на инфраструктурно ниво, които обикновено се реализират чрез външни инструменти като Kubernetes, nginx или HAProxy.

## 1.4 Структура на дипломната работа

Дипломната работа е организирана в следните глави:

**Глава 2** представя преглед на съществуващите C++ уеб библиотеки, техните характеристики и сравнителен анализ.

**Глава 3** въвежда теоретичната основа – HTTP протоколи, WebSocket, TLS, C++20 корутини, асинхронен I/O и DFA-базирано маршрутизиране.

**Глава 4** описва архитектурата на Coroute – модулна структура, жизнен цикъл на заявките, корутинен модел и middleware верига.

**Глава 5** разглежда реализацията на поддържаните протоколи – HTTP/1.1, HTTP/2 и WebSocket.

**Глава 6** представя системата за типово-безопасно извличане на параметри от URL адреси.

**Глава 7** демонстрира примерно приложение, изградено с библиотеката.

**Глава 8** съдържа резултатите от тестването и анализ на производителността.

**Заключението** обобщава постигнатите резултати и очертава насоки за бъдещо развитие.

# 2 Преглед на съществуващи решения

Анализът на съществуващите решения е критична стъпка в процеса на проектиране на нова софтуерна система. Той позволява да се идентифицират утвърдени архитектурни модели, да се избегнат известни проблеми и да се определят области, в които новото решение може да предостави добавена стойност. В контекста на C++ уеб библиотеки, този анализ е особено важен поради разнообразието от подходи — от нискониво мрежово програмиране до високониво абстракции, подобни на тези в интерпретираните езици.

Настоящата глава представя систематичен преглед на пет водещи C++ библиотеки за уеб разработка: Boost.Beast, Drogon, Crow, Oat++ и Pistache. За всяка библиотека се анализират архитектурните решения, моделът на конкурентност, API дизайнът и документираната производителност. Анализът завършва със сравнителна таблица и обосновка на мотивацията за разработване на Coroute.

## 2.1 Boost.Beast

Boost.Beast [5] е част от утвърдената колекция Boost библиотеки и представлява една от най-зрелите имплементации на HTTP и WebSocket протоколите за C++. Библиотеката е изградена върху Boost.Asio, което й осигурява солидна основа за асинхронни мрежови операции.

Философията на Boost.Beast е да предостави ниско ниво на абстракция, давайки на разработчика пълен контрол върху всеки аспект на HTTP комуникацията. Това означава, че библиотеката не налага конкретна архитектура или модел на използване, а вместо това предоставя градивни блокове, от които разработчикът може да изгради своето решение.

Едно от основните предимства на Boost.Beast е нейната интеграция с Boost екосистемата. Разработчиците, които вече използват други Boost библиотеки, могат лесно да добавят HTTP функционалност към своите проекти. Документацията е изчерпателна и включва множество примери за различни сценарии на използване. Активната общност осигурява бърза помощ при възникнали проблеми.

Въпреки тези предимства, Boost.Beast има и съществени недостатъци. Ниското ниво на абстракция означава, че дори прости задачи изискват значително количество код. Липсата на вградено маршрутизиране принуждава разработчиците да имплементират собствена логика за насочване на заявките. Няма middleware система, което затруднява добавянето на cross-cutting concerns като логване, автентикация или компресия. Кривата на обучение е стръмна, особено за разработчици без опит с Boost.Asio.

## 2.2 Drogon

Drogon [1] се е утвърдил като един от най-бързите C++ уеб библиотеки, редовно заемащ водещи позиции в TechEmpower Web Framework Benchmarks [21]. Библиотеката е разработен с фокус върху производителността и предоставя пълен набор от функционалности за изграждане на съвременни уеб приложения.

Архитектурата на Drogon е базирана на неблокиращ I/O модел, използващ event loop за обработка на множество едновременни връзки. Библиотеката поддържа HTTP/1.1, HTTP/2 и WebSocket протоколи, което го прави подходящ за широк спектър от приложения. Вградената ORM система улеснява работата с бази данни, като поддържа PostgreSQL, MySQL и SQLite.

Една от силните страни на Drogon е неговата middleware система, която позволява добавяне на филтри за обработка на заявките. Библиотеката предоставя и инструменти за автоматично генериране на контролери от дефиниции, което ускорява разработката. Документацията е добра, макар и предимно на английски и китайски език.

Въпреки впечатляващата производителност, Drogon има някои недостатъци. Асинхронният модел е базиран на callbacks, което може да доведе до т.нар. “callback hell” при сложна логика. Въпреки че последните версии добавят частична поддръжка на C++20 корутини, тя не е пълноценна и не покрива

всички аспекти на библиотеката. Конфигурацията може да бъде сложна за начинаещи, а размерът на библиотеката е значителен, което увеличава времето за компилация.

## 2.3 Crow

Crow [4] е микробиблиотека, вдъхновен от популярния Python библиотека Flask. Основната цел на Crow е да предостави минималистичен и интуитивен API, който позволява бързо създаване на уеб приложения с минимално количество код.

Една от най-привлекателните характеристики на Crow е неговата простота. Библиотеката е реализиран като header-only библиотека, което означава, че интеграцията в проект се свежда до включване на един header файл. Маршрутизирането се извършва чрез lambda функции, което прави кода четим и лесен за разбиране. За разработчици, идващи от Python или JavaScript екосистемите, синтаксисът на Crow ще изглежда познат и интуитивен.

Crow е особено подходящ за малки проекти, прототипи и вътрешни инструменти, където простотата е по-важна от производителността. Библиотеката поддържа WebSocket комуникация и предоставя базова middleware функционалност.

Въпреки своята елегантност, Crow има съществени ограничения. Липсата на HTTP/2 поддръжка го прави неподходящ за съвременни production приложения, които изискват мултиплексиране и ефективна компресия на хедъри. Синхронният модел по подразбиране означава, че при високо натоварване производителността страда значително. Библиотеката не предоставя вградени инструменти за работа с бази данни, сесии или автентикация, което налага използването на допълнителни библиотеки.

## 2.4 Oat++

Oat++ [13] представлява модерен подход към C++ уеб разработката, поставяйки силен акцент върху типовата безопасност и автоматизацията. Библиотеката е проектиран с идеята, че API документацията трябва да се генерира автоматично от кода, а не да се поддържа ръчно.

Централна концепция в Oat++ е системата от Data Transfer Objects (DTO). Разработчиците дефинират структурите на данните чрез специални макроси, които позволяват автоматична сериализация и десериализация към JSON, както и генериране на OpenAPI/Swagger документация. Това елиминира честите грешки при ръчно писане на документация и гарантира, че тя винаги е синхронизирана с кода.

Библиотеката предоставя както синхронен, така и асинхронен API, давайки на разработчиците гъвкавост при избора на модел на изпълнение. Вградената система за валидация на входни данни допълнително подобрява надеждността на приложенията.

Основният недостатък на Oat++ е неговата собствена система от типове. Вместо да използва стандартни C++ типове, библиотеката изисква използването на специални wrapper класове, което може да бъде объркващо за начинаещи. Кривата на обучение е по-стръмна в сравнение с по-простите библиотеки. Липсата на HTTP/2 поддръжка е съществен пропуск за приложения, изискващи висока производителност при множество едновременни заявки.

## 2.5 Pistache

Pistache [16] е библиотека, специализиран в създаването на RESTful API услуги. Проектиран е с фокус върху елегантността на кода и лесната употреба, следвайки принципите на модерния C++ дизайн.

API-то на Pistache е чисто и изразително, позволявайки дефиниране на endpoints с минимално количество boilerplate код. Библиотеката използва асинхронен модел на изпълнение, базиран на Linux epoll, което осигурява добра производителност при обработка на множество едновременни връзки. Вградената поддръжка за HTTP методи, статус кодове и content negotiation улеснява изграждането на стандартни REST API-та.

Pistache предоставя и допълнителни функционалности като rate limiting, timeout management и graceful shutdown, които са важни за production deployments. Документацията е ясна и включва примери за често срещани сценарии.

Най-съществените ограничения на Pistache са свързани с платформената поддръжка и функционалността. Библиотеката работи само на Linux, което го прави неподходящ за проекти, изискващи cross-platform съвместимост. Липсата на HTTP/2 и WebSocket поддръжка ограничава приложимостта му в съвременни уеб приложения. Активността на проекта е намаляла през последните години, което поражда въпроси относно дългосрочната поддръжка и развитие.

## 2.6 Сравнителен анализ

След детайлния преглед на всеки библиотека, е полезно да направим систематично сравнение по ключови характеристики. Таблица 1 обобщава поддръжката на различни функционалности от всеки от разгледаните библиотеки, включително Coroute.

Таблица 1: Сравнение на C++ уеб библиотеки по ключови характеристики

| **Характеристика** | **Beast** | **Drogon** | **Crow** | **Oat++** | **Pistache** | **Coroute** |
|---|---|---|---|---|---|---|
| HTTP/1.1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HTTP/2 | – | ✓ | – | – | – | ✓ |
| WebSocket | ✓ | ✓ | ✓ | ✓ | – | ✓ |
| C++20 корутини | – | частично | – | – | – | ✓ |
| Middleware | – | ✓ | ✓ | ✓ | ✓ | ✓ |
| Типово маршрут. | – | – | – | ✓ | – | ✓ |
| Cross-platform | ✓ | ✓ | ✓ | ✓ | – | ✓ |

От таблицата се вижда, че всеки библиотека има своите силни страни, но нито един не предоставя пълния набор от желани функционалности. Boost.Beast е мощен, но ниско ниво. Drogon е бърз, но callback-базиран. Crow е прост, но ограничен. Oat++ е типово-безопасен, но без HTTP/2. Pistache е елегантен, но само за Linux.

## 2.7 Мотивация за Coroute

Анализът на съществуващите решения разкрива съществена празнина на пазара. Няма C++ библиотека, който да комбинира всички следни характеристики в едно цялостно решение:

Първо, **пълноценна поддръжка на C++20 корутини**. Корутините са революционна функционалност, която позволява писане на асинхронен код със синтаксис, подобен на синхронния. Въпреки че C++20 е наличен от 2020 година, повечето библиотеки все още използват callback-базирани модели или предоставят само частична поддръжка на корутини.

Второ, **HTTP/2 с ALPN негоциация**. HTTP/2 е стандарт от 2015 година и предоставя значителни подобрения в производителността чрез мултиплексиране на потоци и компресия на хедъри. Въпреки това, много C++ библиотеки все още не го поддържат или изискват сложна конфигурация.

Трето, **типово-безопасно маршрутизиране с compile-time проверки**. Традиционните библиотеки извличат параметри от URL като низове, оставяйки валидацията на runtime. Използването на C++ шаблони позволява проверка на типовете по време на компилация, елиминирайки цял клас грешки.

Четвърто, **DFA-базирано маршрутизиране с O(N) сложност**. Повечето библиотеки използват последователно съпоставяне на маршрути, което води до O(N × M) сложност. Алгоритъмът, описан в „Matching Text from Start to Finish Against Multiple Regular Expressions" [20], позволява съпоставяне с линейна сложност спрямо дължината на URL-а, независимо от броя на маршрутите.

Пето, **платформена независимост с оптимизирани I/O бекенди**. Различните операционни системи предоставят различни механизми за асинхронен I/O – IOCP на Windows, io_uring на Linux, kqueue на macOS. Оптималната производителност изисква използване на native механизми за всяка платформа.

Coroute е създаден с амбицията да запълни тази ниша. Библиотеката комбинира всички изброени характеристики в едно цялостно, модерно и ефективно решение за C++ уеб разработка. В следващите глави ще разгледаме детайлно архитектурата и имплементацията на всяка от тези функционалности.

# 3 Теоретична основа

Преди да пристъпим към описание на архитектурата и имплементацията на Coroute, е необходимо да представим теоретичните концепции, върху които е изградена библиотеката. Тази глава обхваща HTTP протоколите, WebSocket, TLS криптирането, C++20 корутините, асинхронния I/O и DFA-базираното маршрутизиране.

Разбирането на тези концепции е от съществено значение за правилното използване на библиотеката и за вземане на информирани архитектурни решения при изграждане на уеб приложения.

## 3.1 HTTP протокол

Hypertext Transfer Protocol (HTTP) е основният протокол за комуникация в World Wide Web. От създаването си през 1991 година, HTTP е претърпял значителна еволюция, като всяка нова версия адресира ограниченията на предходната.

### 3.1.1 HTTP/1.1

HTTP/1.1, дефиниран в RFC 2616 [7], е текстово-базиран протокол, използващ модела заявка-отговор (request-response). При този модел клиентът изпраща заявка към сървъра, а сървърът отговаря с резултата. Всяка заявка е независима и не съхранява състояние между отделните обмени (stateless протокол).

HTTP/1.1 въведе редица подобрения спрямо HTTP/1.0, включително persistent connections (keep-alive), chunked transfer encoding и подобрена кеширане. Въпреки тези подобрения, протоколът има фундаментални ограничения, които стават все по-проблематични при съвременните уеб приложения.

Структура на HTTP заявка:

```
GET /path/to/resource HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

Структура на HTTP отговор:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234

<html>...</html>
```

Въпреки широкото си разпространение, HTTP/1.1 има няколко съществени ограничения, които влияят негативно на производителността на съвременните уеб приложения.

Първото и най-значимо ограничение е **head-of-line blocking**. При HTTP/1.1 заявките по една връзка се обработват последователно – сървърът трябва да изпрати пълния отговор на първата заявка, преди да започне обработката на втората. Ако първата заявка е за голям ресурс, всички следващи заявки трябва да чакат, дори ако техните отговори са готови.

Второто ограничение е свързано с **текстовите хедъри**. HTTP/1.1 хедърите са текстови и се повтарят при всяка заявка. При съвременните уеб приложения, които изпращат десетки заявки за една страница, това води до значителен overhead. Cookies, User-Agent и други хедъри могат да заемат стотици байтове, които се предават отново и отново.

Третото ограничение е необходимостта от **множество TCP връзки** за постигане на паралелизъм. Браузърите обикновено отварят 6-8 паралелни връзки към един домейн, за да заобиколят head-of-line

blocking. Това обаче води до допълнителен overhead от TCP handshakes и увеличена консумация на ресурси.

### 3.1.2 HTTP/2

HTTP/2, дефиниран в RFC 7540 [3] и актуализиран в RFC 9113 [23], е бинарен протокол, проектиран да адресира ограниченията на HTTP/1.1. Разработен първоначално от Google под името SPDY, HTTP/2 беше стандартизиран през 2015 година и днес се поддържа от всички съвременни браузъри и сървъри.

Ключовата иновация в HTTP/2 е **мултиплексирането**. Вместо да обработва заявките последователно, HTTP/2 позволява множество потоци (streams) да споделят една TCP връзка. Всеки поток има уникален идентификатор и може да пренася независима заявка-отговор двойка. Това елиминира head-of-line blocking на приложно ниво и позволява пълноценно използване на една TCP връзка.

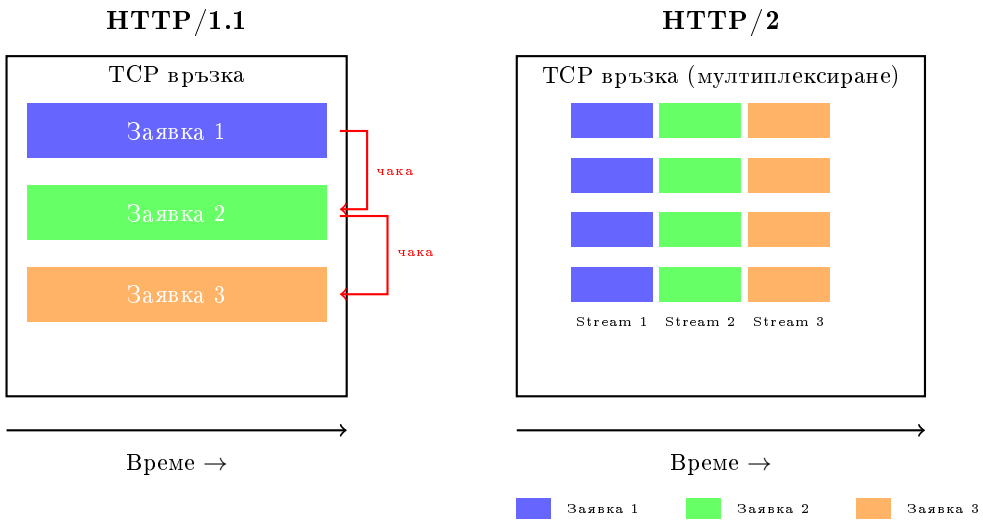
**HPACK компресията** [15] е специализиран алгоритъм за компресия на HTTP хедъри. HPACK използва статична таблица с често срещани хедъри, динамична таблица за хедъри, специфични за връзката, и Huffman кодиране за стойностите. Това може да намали размера на хедърите с над 90% в сравнение с HTTP/1.1.

**Server Push** позволява на сървъра проактивно да изпраща ресурси към клиента, без клиентът да ги е заявил изрично. Например, когато клиентът заяви HTML страница, сървърът може едновременно да изпрати CSS и JavaScript файловете, които знае, че ще бъдат необходими. Това намалява латентността при зареждане на страници.

**Приоритизацията** позволява на клиента да указва относителната важност на различните потоци. Браузърът може да даде по-висок приоритет на CSS файловете (необходими за рендериране) спрямо изображенията (които могат да се заредят по-късно).

HTTP/2 използва фреймове като основна единица за комуникация. Всеки фрейм има фиксиран 9-байтов хедър, съдържащ дължина, тип, флагове и идентификатор на потока. Основните типове фреймове включват HEADERS за HTTP хедъри, DATA за тялото на съобщението, SETTINGS за конфигурационни параметри, WINDOW_UPDATE за контрол на потока и GOAWAY за graceful затваряне на връзката.

Фигура 1 илюстрира ключовата разлика между HTTP/1.1 и HTTP/2 при обработка на множество заявки.




Фигура 1: Сравнение на HTTP/1.1 (последователна обработка с head-of-line blocking) и HTTP/2 (мултиплексиране на потоци)

## 3.2 WebSocket протокол

WebSocket [6] е протокол, проектиран за двупосочна комуникация в реално време върху една TCP връзка. Стандартизиран през 2011 година, WebSocket запълва важна празнина в уеб технологиите, позволявайки на сървъра да изпраща данни към клиента без клиентът да е направил изрична заявка.

Преди WebSocket, разработчиците използваха различни техники за симулиране на real-time комуникация, като polling (периодични заявки от клиента), long polling (заявки, които сървърът задържа докато има нови данни) и Server-Sent Events (еднопосочен поток от сървъра). Всички тези техники имат недостатъци – polling е неефективен, long polling създава сложност, а SSE е само еднопосочен.

WebSocket решава тези проблеми, като предоставя пълноценна двупосочна комуникация с минимален overhead. След първоначалния handshake, данните се предават като леки фреймове без HTTP хедъри, което значително намалява латентността.

Процесът на установяване на WebSocket връзка започва с HTTP Upgrade заявка. Клиентът изпраща стандартна HTTP заявка с хедъри `Upgrade: websocket` и `Connection: Upgrade`, заедно с криптографски ключ в `Sec-WebSocket-Key`. Сървърът отговаря с HTTP 101 Switching Protocols и изчислен accept key, базиран на клиентския ключ и фиксиран GUID. След този handshake, TCP връзката преминава в WebSocket режим.

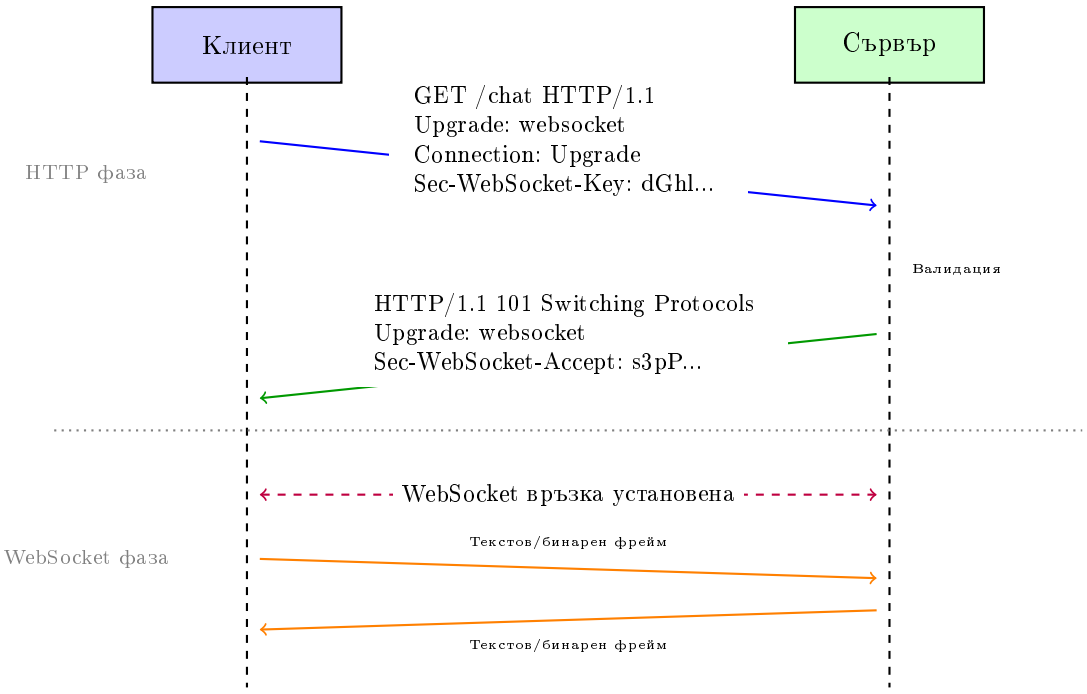
Заявка за upgrade изглежда по следния начин:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

WebSocket съобщенията се предават като фреймове. Всеки фрейм има хедър, съдържащ опкод (тип на съобщението), флагове и информация за дължината. Опкодът определя как да се интерпретира payload-а: 0x1 за текстови данни (UTF-8), 0x2 за бинарни данни, 0x8 за затваряне на връзката, 0x9 за ping и 0xA за pong. Ping/pong механизмът се използва за keep-alive и за проверка дали връзката е все

още активна.

Фигура 2 илюстрира процеса на WebSocket handshake.




Фигура 2: Sequence диаграма на WebSocket handshake процеса

Важна характеристика на WebSocket е маскирането на данните от клиента към сървъра. Всички фреймове, изпратени от клиента, трябва да бъдат маскирани с 4-байтов ключ. Това е мярка за сигурност, предотвратяваща определени видове атаки срещу прокси сървъри.

## 3.3 TLS и сигурност

Transport Layer Security (TLS) е криптографски протокол, осигуряващ сигурна комуникация през интернет. TLS е наследник на SSL (Secure Sockets Layer) и е стандартът за криптиране на уеб трафик. Когато URL адрес започва с “https://”, това означава, че връзката е защитена с TLS.

TLS предоставя три основни гаранции за сигурност. Първо, **конфиденциалност** – данните са криптирани и не могат да бъдат прочетени от трети страни. Второ, **интегритет** – данните не могат да бъдат модифицирани незабелязано по време на предаване. Трето, **автентикация** – клиентът може да верифицира идентичността на сървъра чрез цифрови сертификати.

TLS 1.3 [18], публикуван през 2018 година, въвежда значителни подобрения спрямо предходните версии. Handshake процесът е опростен и изисква само един round-trip (1-RTT) за установяване на връзка, в сравнение с два round-trips при TLS 1.2. За клиенти, които вече са се свързвали със сървъра, е възможен 0-RTT режим, при който данни могат да се изпращат още с първия пакет.

TLS 1.3 премахва поддръжката за остарели и несигурни криптографски алгоритми. Поддържат се само съвременни cipher suites, базирани на AEAD (Authenticated Encryption with Associated Data) алгоритми като AES-GCM и ChaCha20-Poly1305. Това опростява конфигурацията и елиминира възможността за downgrade атаки.

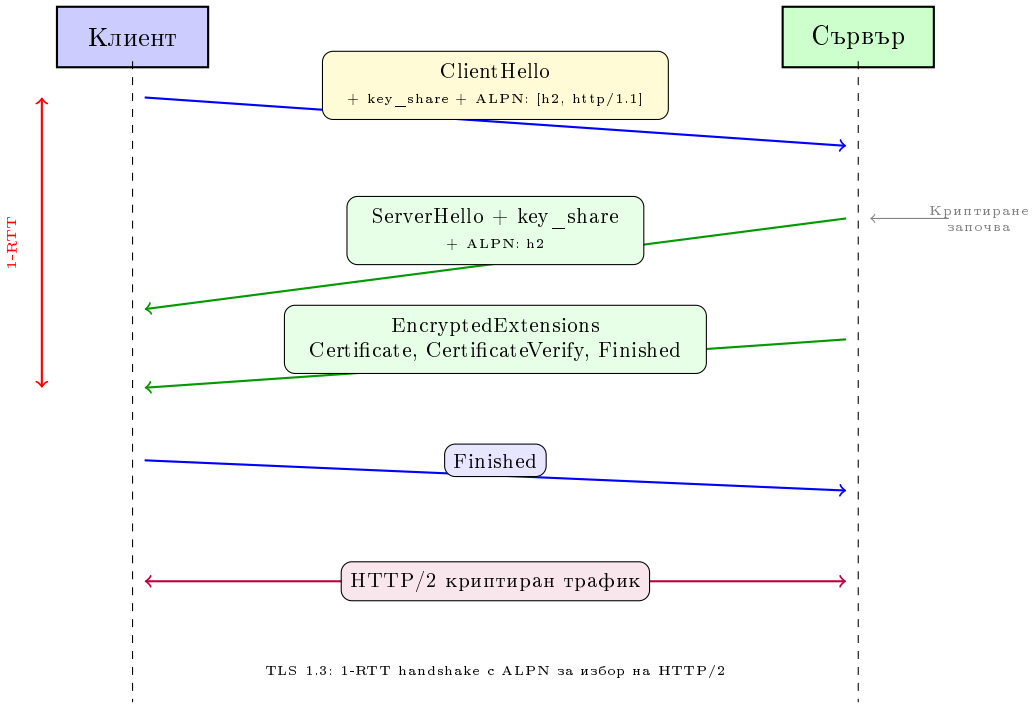
**ALPN (Application-Layer Protocol Negotiation)** [9] е TLS разширение, което позволява избор на

приложен протокол по време на handshake. Това е механизмът, чрез който HTTP/2 се негоцира при TLS връзки.

При ALPN негоциация, клиентът включва в ClientHello съобщението списък с поддържани протоколи, подредени по предпочитание. Например, съвременен браузър би изпратил “h2” (HTTP/2) и “http/1.1” като опции. Сървърът избира един от предложените протоколи (обикновено най-предпочитания, който поддържа) и го включва в ServerHello отговора. След завършване на TLS handshake, комуникацията продължава с избрания протокол.

ALPN е критичен за HTTP/2, тъй като позволява на клиента и сървъра да се договорят за протокола преди да започнат да обменят HTTP данни. Без ALPN, би било необходимо допълнително round-trip за upgrade от HTTP/1.1 към HTTP/2.

Фигура 3 илюстрира TLS 1.3 handshake процеса с ALPN негоциация.




Фигура 3: TLS 1.3 handshake с ALPN негоциация за HTTP/2

## 3.4 C++20 корутини

Корутините са една от най-значимите нови функционалности в C++20 [11]. Те представляват обобщение на концепцията за функция, позволявайки на функцията да спре изпълнението си в определени точки и да го възобнови по-късно, без да губи своето състояние.

Традиционните функции следват модела “call/return” – когато функция бъде извикана, тя се изпълнява до край и връща резултат. Корутините добавят възможността за “suspend/resume” – функцията може да спре изпълнението си, да върне контрола на извикващия, и по-късно да продължи от точката, където е спряла.

Тази възможност е изключително полезна за асинхронно програмиране. Вместо да използваме callbacks или promises, можем да пишем код, който изглежда синхронен и последователен, но всъщност се изпълнява асинхронно. Когато корутината срещне операция, която не може да завърши веднага (например

четене от мрежата), тя спира и позволява на други корутини да се изпълняват. Когато операцията завърши, корутината се възобновява и продължава изпълнението си.

C++20 въвежда три нови ключови думи за работа с корутини. `co_await` се използва за спиране на корутината докато чака завършване на асинхронна операция. `co_yield` се използва за генериране на стойност и временно спиране (полезно за генератори). `co_return` се използва за връщане на крайния резултат и завършване на корутината.

### 3.4.1 Основни концепции

За да разберем как работят корутините в C++20, трябва да се запознаем с няколко ключови концепции: promise type, awaitable и coroutine handle.

**Promise type** е тип, който контролира поведението на корутината. Когато компилаторът срещне функция, съдържаща `co_await`, `co_yield` или `co_return`, той автоматично генерира код, който използва promise type за управление на корутината. Promise type определя какво се случва при стартиране на корутината, при нейното завършване, при връщане на стойност и при възникване на изключение.

```
struct Promise {
    Task<T> get_return_object();
    std::suspend_always initial_suspend();
    FinalAwaiter final_suspend() noexcept;
    void return_value(T value);
    void unhandled_exception();
};
```

Listing 1: Promise type структура

Методът `get_return_object()` създава обекта, който се връща на извикващия при създаване на корутината. Методът `initial_suspend()` определя дали корутината да започне изпълнение веднага или да изчака. Методът `final_suspend()` определя какво се случва след завършване на корутината – дали да се унищожи автоматично или да остане в паметта.

**Awaitable** е тип, който може да се използва с оператора `co_await`. Awaitable обектите представляват асинхронни операции, които могат да спрат корутината докато чакат завършване.

```
struct Awaitable {
    bool await_ready();
    void await_suspend(std::coroutine_handle<>);
    T await_resume();
};
```

Listing 2: Awaitable интерфейс

Методът `await_ready()` връща `true`, ако операцията вече е завършила и не е необходимо спиране. Методът `await_suspend()` се извиква, когато корутината трябва да спре – тук се регистрира callback за възобновяване. Методът `await_resume()` връща резултата от операцията, когато корутината се възобнови.

**Coroutine handle** е низкониво представяне на корутината, подобно на указател. Чрез handle можем да възобновим спряна корутина, да проверим дали е завършила, или да я унищожим. Handle-ът се използва вътрешно от awaitable обектите за управление на корутините.

### 3.4.2 Предимства на корутините

Корутините предоставят множество предимства пред традиционните подходи за асинхронно програмиране.

Първото и най-важно предимство е **четимостта**. Асинхронният код, написан с корутини, изглежда почти идентично на синхронен код. Вместо вложени callbacks или сложни promise chains, имаме последователни изрази, разделени с `co_await`. Това значително намалява когнитивното натоварване при четене и писане на асинхронен код.

Второто предимство е **ефективността**. За разлика от нишките, корутините не изискват системни извиквания за създаване или превключване. Превключването между корутини е просто запазване и възстановяване на няколко регистъра – операция, която отнема наносекунди вместо микросекунди.

Третото предимство е **композируемостта**. Корутините могат свободно да се влагат една в друга. Една корутина може да извика друга корутина с `co_await`, която от своя страна може да извика трета. Това позволява изграждане на сложна асинхронна логика от прости градивни блокове.

Четвъртото предимство е, че корутините са **stackless**. Традиционните нишки изискват отделен стек (обикновено 1MB), който се заделя при създаване. Корутините съхраняват само локалните променливи и точката на изпълнение – типично десетки до стотици байтове. Това позволява създаване на милиони едновременни корутини на машина с ограничена памет.

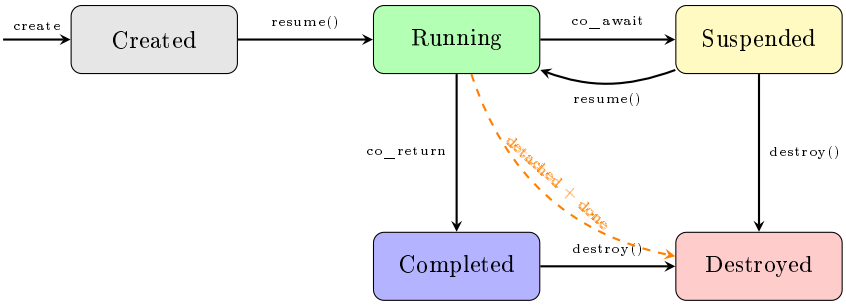
Следният пример демонстрира как асинхронен код с корутини изглежда почти идентично на синхронен:

```
Task<Response> handle_request(Request& req) {
    // Всяка от тези операции може да отнеме време,
    // но кодът изглежда последователен
    auto user = co_await db.find_user(req.param("id"));
    auto posts = co_await db.get_posts(user.id);
    co_return render_template("profile", user, posts);
}
```

Listing 3: Пример за асинхронен код с корутини

В този пример, докато `db.find_user()` чака отговор от базата данни, корутината е спряна и нишката може да обслужва други заявки. Когато отговорът пристигне, корутината се възобновява и продължава със следващия ред.

Фигура 4 илюстрира жизнения цикъл на корутина и възможните преходи между състоянията.




Фигура 4: State диаграма на жизнения цикъл на корутина

## 3.5 Асинхронен I/O

Асинхронният вход/изход (I/O) е фундаментална концепция за изграждане на високопроизводителни мрежови приложения. При синхронен I/O, когато програмата извърши операция като четене от сокет, тя блокира докато данните пристигнат. При асинхронен I/O, програмата инициира операцията и продължава изпълнението си, като бива уведомена когато операцията завърши.

Различните операционни системи предоставят различни механизми за асинхронен I/O. Всеки от тях има своите характеристики и оптимални сценарии на използване. За постигане на максимална производителност, Coroute използва native механизма на всяка платформа.

### 3.5.1 IOCP (Windows)

I/O Completion Ports (IOCP) [19] е механизмът на Windows за мащабируем асинхронен I/O. IOCP е проектиран специално за сървърни приложения, които трябва да обслужват хиляди едновременни връзки.

При IOCP, приложението създава completion port – обект на ядрото, който служи като централна точка за събиране на завършени I/O операции. Когато приложението иска да извърши асинхронна операция (например четене от сокет), то подава заявка към операционната система и продължава изпълнението си. Когато операцията завърши, резултатът се поставя в completion port.

Работни нишки извикват `GetQueuedCompletionStatus()`, за да извлекат завършени операции от completion port. Ако няма готови операции, нишката блокира ефективно, без да консумира CPU ресурси. Windows автоматично управлява броя на активните нишки, за да оптимизира използването на CPU.

Едно от ключовите предимства на IOCP е неговата мащабируемост. Малък брой нишки (обикновено равен на броя на CPU ядрата) може да обслужва десетки хиляди едновременни връзки. Това е възможно, защото нишките не блокират при отделни I/O операции, а само при чакане за завършени операции.

### 3.5.2 io_uring (Linux)

io_uring [2] е модерен Linux интерфейс за асинхронен I/O, въведен в ядро версия 5.1 (2019). Той е проектиран да замени по-старите интерфейси като epoll и aio, предоставяйки по-висока производителност и по-богата функционалност.

Архитектурата на io_uring се базира на две кръгови опашки (ring buffers), споделени между потребителското пространство и ядрото. Submission Queue (SQ) се използва от приложението за подаване на I/O заявки. Completion Queue (CQ) се използва от ядрото за връщане на резултати от завършени операции.

Ключово предимство на io_uring е възможността за batching на системни извиквания. Вместо да прави отделно системно извикване за всяка I/O операция, приложението може да добави множество заявки в SQ и да ги подаде наведнъж с едно извикване на `io_uring_enter()`. Това значително намалява overhead-а от превключване между потребителско пространство и ядро.

io_uring поддържа и zero-copy операции, при които данните се прехвърлят директно между мрежовия буфер и файловата система, без копиране в потребителското пространство. Това е особено полезно за файлови сървъри и CDN приложения.

### 3.5.3 kqueue (macOS/BSD)

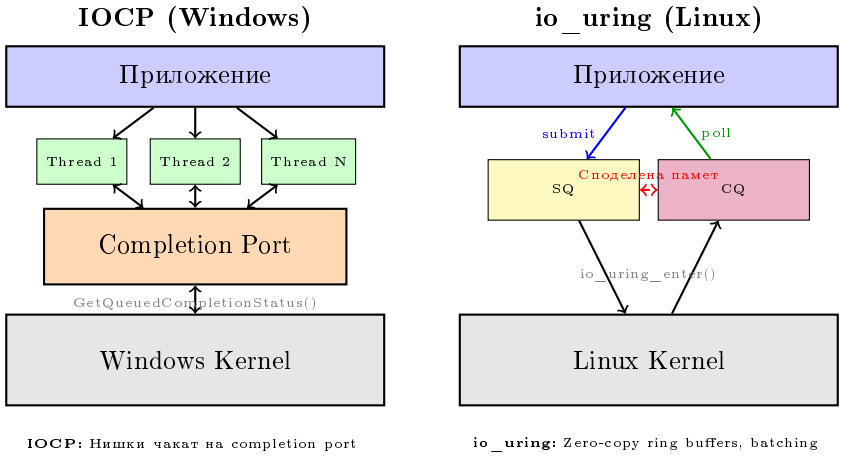
kqueue [8] е механизъм за event notification, наличен в macOS и BSD операционните системи. Въведен първоначално във FreeBSD 4.1 (2000), kqueue предоставя унифициран интерфейс за наблюдение на различни типове събития.

При kqueue, приложението регистрира интерес към определени събития чрез структури, наречени kevents. Събитията могат да бъдат свързани с файлови дескриптори (готовност за четене/писане), сигнали, таймери, промени във файловата система и други. След регистрация, приложението извиква `kevent()` за да получи списък с настъпили събития.

Едно от предимствата на kqueue е неговата гъвкавост. За разлика от epoll, който работи само с файлови дескриптори, kqueue може да наблюдава разнообразни типове събития. Това опростява архитектурата на приложения, които трябва да реагират на множество видове асинхронни събития.

kqueue също поддържа edge-triggered и level-triggered режими, подобно на epoll. В edge-triggered режим, събитие се докладва само когато състоянието се промени. В level-triggered режим, събитие се докладва докато условието е изпълнено. Edge-triggered режимът е по-ефективен, но изисква по-внимателно програмиране.

Фигура 5 сравнява архитектурите на IOCP и io_uring.




Фигура 5: Сравнение на архитектурите на IOCP (Windows) и io_uring (Linux)

## 3.6 DFA-базирано маршрутизиране

Маршрутизирането е процесът на определяне кой handler да обработи дадена HTTP заявка, базирано на URL адреса и HTTP метода. В типично уеб приложение може да има стотици или дори хиляди различни маршрути, всеки с различен URL шаблон.

Традиционният подход за маршрутизиране е последователно проверяване на всеки шаблон срещу вход-

ния URL. При този подход, за всяка входяща заявка се итерира през списъка с маршрути и се проверява дали URL-ът съответства на шаблона. Това води до времева сложност O(N × M), където N е броят на маршрутите, а M е дължината на URL-а. При голям брой маршрути, това може да стане значително забавяне.

В научната публикация „Matching Text from Start to Finish Against Multiple Regular Expressions" [20] е представен алтернативен подход, базиран на детерминиран краен автомат (DFA). Този алгоритъм позволява съпоставяне на входен текст с множество регулярни изрази едновременно, с линейна времева сложност спрямо дължината на входа.

### 3.6.1 Теоретична основа

За да разберем DFA-базираното маршрутизиране, трябва първо да се запознаем с теорията на формалните езици и автомати.

Регулярните изрази и крайните автомати са еквивалентни по изразителна сила – това е известно като теорема на Kleene. Всеки регулярен израз може да се преобразува в еквивалентен недетерминиран краен автомат (NFA), който от своя страна може да се преобразува в детерминиран краен автомат (DFA) [17, 22].

Краен автомат $M$ се дефинира формално като петорка $(Q, \Sigma, \delta, q_0, F)$, където $Q$ е крайно множество от състояния, $\Sigma$ е входната азбука (множеството от възможни символи), $\delta : Q \times \Sigma \to Q$ е функцията на преходите, $q_0 \in Q$ е началното състояние, и $F \subseteq Q$ е множеството от крайни (приемащи) състояния.

При обработка на входен низ, автоматът започва от началното състояние $q_0$. За всеки символ от входа, автоматът преминава в ново състояние, определено от функцията на преходите $\delta$. Ако след обработка на целия вход автоматът е в крайно състояние от $F$, входът се приема; в противен случай се отхвърля.

Ключовото предимство на DFA е, че за всяка комбинация от текущо състояние и входен символ има точно един възможен преход. Това означава, че обработката на входа изисква точно толкова стъпки, колкото е дължината на входа – линейна времева сложност O(N).

### 3.6.2 Алгоритъм за изграждане на pattern graph

Алгоритъмът от „Matching Text from Start to Finish Against Multiple Regular Expressions" [20] разширява класическия подход, като комбинира множество регулярни изрази в един обединен граф, наречен pattern graph. Този граф е подобен на trie структура, но с възможност за цикли, които са необходими за представяне на операторите за повторение (*, +, {n,m}).

Процесът на изграждане започва с празен граф, съдържащ само начално състояние. За всеки регулярен израз (URL шаблон), алгоритъмът добавя съответните състояния и преходи към графа. Когато два шаблона споделят общ префикс, те споделят и съответните състояния в графа. Това е ключът към ефективността – вместо да проверяваме всеки шаблон поотделно, ние обхождаме един общ граф.

Всяко крайно състояние в графа е асоциирано с един или повече маршрути. Когато входният URL достигне крайно състояние, знаем кой маршрут е съвпаднал. Ако множество маршрути съвпадат, се използва приоритет (обикновено по-специфичният маршрут има предимство).

Времевата сложност на алгоритъма е следната: предобработката (изграждане на графа) изисква време $O(\sum \text{regex\_length})$, пропорционално на общата дължина на всички шаблони. Търсенето (съпоставяне на URL) изисква време $O(N)$, където $n$ е дължината на входния URL. Забележете, че времето за търсене не зависи от броя на маршрутите – това е ключовото подобрение спрямо наивния подход.

### 3.6.3 Приложение в Coroute

В контекста на Coroute, URL шаблоните се преобразуват в регулярни изрази. Например, шаблонът `/users/{id}` се преобразува в регулярен израз, който съвпада с `/users/` последвано от произволна последователност от символи (параметъра `id`).

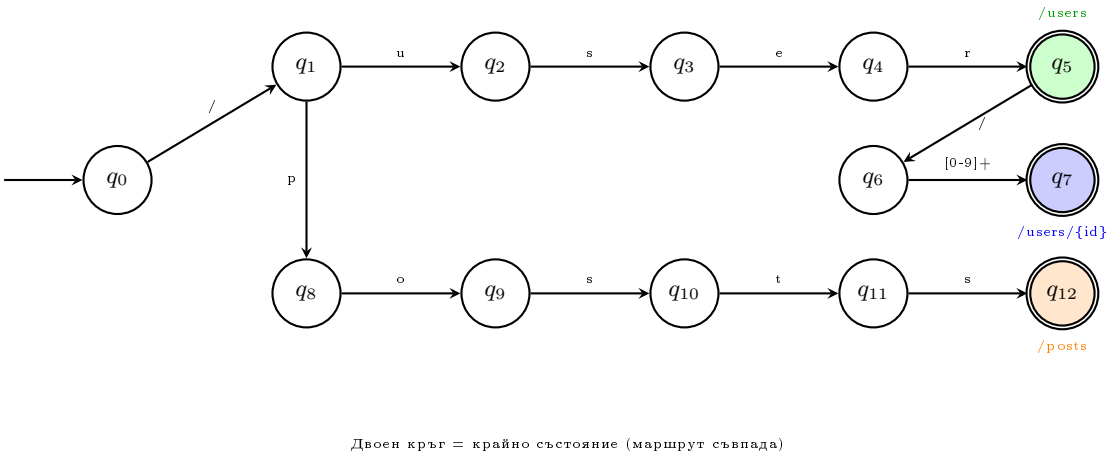
При стартиране на сървъра, всички регистрирани маршрути се компилират в един DFA. При всяка входяща заявка, URL-ът се обработва от DFA с една линейна обиколка. Резултатът е или съвпадение с конкретен маршрут (заедно с извлечените параметри), или липса на съвпадение (404 Not Found).

### 3.6.4 Benchmark резултати

Според експерименталните резултати от „Matching Text from Start to Finish Against Multiple Regular Expressions" [20], предложеният алгоритъм показва значително подобрение в производителността спрямо традиционните подходи. При тестове със 100 различни регулярни израза, алгоритъмът е до **62.25 пъти по-бърз** от последователното съпоставяне с `std::regex`.

Това подобрение е особено значимо за уеб приложения с голям брой маршрути. При REST API с ресурси, версии и вложени пътища, броят на маршрутите лесно може да достигне стотици. С DFA-базирано маршрутизиране, времето за намиране на правилния handler остава константно, независимо от броя на маршрутите.

Фигура 6 илюстрира пример за DFA, който разпознава URL шаблоните `/users`, `/users/{id}` и `/posts`.




Фигура 6: Пример за DFA за URL маршрутизиране с три шаблона

# 4 Архитектура на Coroute

След като разгледахме теоретичните основи в предходната глава, сега ще представим конкретната архитектура на библиотеката Coroute. Тази глава описва модулната структура на библиотеката, жизнения цикъл на HTTP заявките, корутинния модел на изпълнение, middleware системата и платформено-независимия I/O слой.

Архитектурата на Coroute е проектирана с няколко ключови принципа. Първо, **модулност** – всеки компонент има ясно дефинирана отговорност и може да се използва независимо. Второ, **разширяемост** – новите протоколи и функционалности могат да се добавят без промяна на съществуващия код. Трето, **ефективност** – архитектурата минимизира копирането на данни и системните извиквания. Четвърто, **безопасност** – типовата система на C++ се използва за предотвратяване на грешки по време на компилация.

## 4.1 Обща архитектура и модулна структура

Coroute е организиран в йерархия от модули, всеки от които отговаря за определен аспект на функционалността. Тази организация улеснява разбирането на кода, тестването и поддръжката.

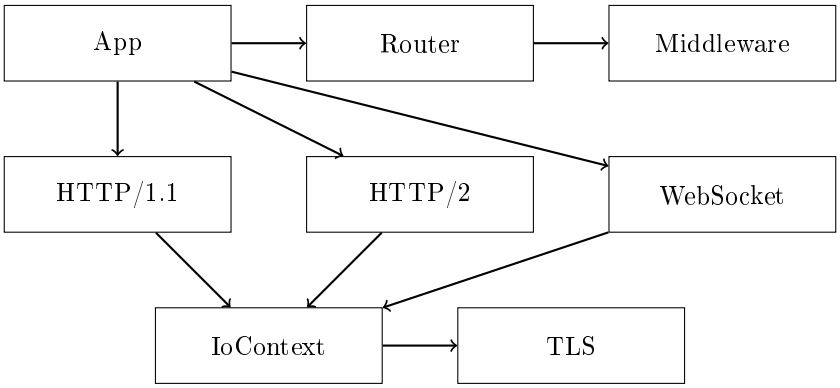
Модулът **core**/ съдържа основните компоненти на библиотеката. Класът `App` е централната точка за конфигуриране и стартиране на сървъра. `Router` управлява маршрутизирането на заявките към съответните handlers. Класовете `Request` и `Response` представляват HTTP заявки и отговори с удобен API за достъп до хедъри, тяло и параметри.

Модулът **coro**/ предоставя корутинната инфраструктура. Типът `Task<T>` е основният тип за асинхронни операции. `CancellationToken` позволява кооперативно прекратяване на дълготрайни операции. Тук се намират и различните awaiter типове за интеграция с I/O операциите.

Модулът **net**/ съдържа мрежовия слой. `IoContext` е абстракция над платформено-специфичните I/O механизми. `Connection` представлява TCP връзка с методи за асинхронно четене и писане. Тук се намират и имплементациите за TLS и WebSocket.

Модулът **http2**/ съдържа пълната HTTP/2 имплементация. `Http2Connection` управлява HTTP/2 връзка с множество потоци. HPACK encoder/decoder се грижи за компресията на хедъри. Frame parser/serializer обработва бинарните HTTP/2 фреймове.

Модулът **util**/ съдържа помощни функции и типове. `expected<T, E>` е тип за представяне на резултат или грешка (подобен на `std::expected` от C++23). `FromString<T>` е trait за конвертиране на низове към типове. Тук се намират и функциите за zero-copy file transfer.




Фигура 7: Модулна архитектура на Coroute

## 4.2 Жизнен цикъл на HTTP заявка

Разбирането на жизнения цикъл на HTTP заявката е ключово за ефективното използване на Coroute. Всяка заявка преминава през поредица от етапи, като всеки етап се изпълнява асинхронно чрез корутини.

Първият етап е **Accept**. IoContext наблюдава listening socket-а за входящи TCP връзки. Когато клиент се свърже, се създава нов `Connection` обект и се стартира корутина за обработка на връзката. Тази корутина работи в режим "detached" – тя се самоуправлява и се унищожава автоматично при завършване.

Вторият етап е **TLS Handshake**, който е опционален и се изпълнява само ако сървърът е конфигуриран за HTTPS. По време на handshake се установява криптирана връзка и се верифицира сертификатът на сървъра. TLS handshake също е асинхронен – корутината спира докато чака завършване на криптографските операции.

Третият етап е **ALPN Negotiation**, който се случва като част от TLS handshake. Клиентът и сървърът се договарят за протокола – HTTP/2 ("h2") или HTTP/1.1 ("http/1.1"). Ако клиентът поддържа HTTP/2 и сървърът е конфигуриран за него, се избира HTTP/2. В противен случай се използва HTTP/1.1.

Четвъртият етап е **Parse**. Сървърът чете данни от връзката и парсва HTTP заявката. При HTTP/1.1 това включва парсване на request line, хедъри и тяло. При HTTP/2 се декодират бинарни фреймове и HPACK-компресирани хедъри.

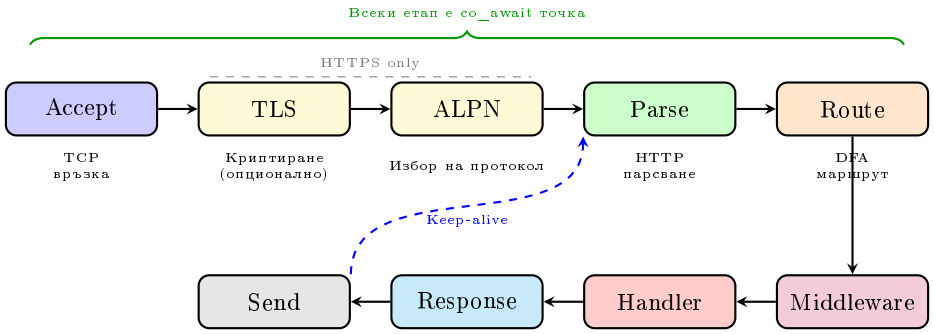
Петият етап е **Route**. DFA matcher-ът съпоставя URL-а на заявката с регистрираните маршрути. Ако има съвпадение, се извличат параметрите от URL-а. Ако няма съвпадение, се връща 404 Not Found.

Шестият етап е **Middleware**. Заявката преминава през веригата от middleware компоненти. Всеки middleware може да модифицира заявката, да върне отговор директно, или да предаде контрола на следващия middleware.

Седмият етап е **Handler**. Потребителският handler се извиква с заявката и параметрите. Handler-ът е корутина, която може да извършва асинхронни операции като заявки към база данни или външни API-та.

Осмият етап е **Response**. Отговорът се сериализира и изпраща към клиента. При HTTP/1.1 това е текстов формат. При HTTP/2 отговорът се разбива на фреймове и хедърите се компресират с HPACK.

Фигура 8 илюстрира пълния жизнен цикъл на HTTP заявка.




Фигура 8: Жизнен цикъл на HTTP заявка в Coroute

Следният код от `app.cpp` илюстрира основния accept loop:

```
[this]() -> Task<void> {
    while (!cancel_source_.is_cancelled()) {
        auto conn_result = co_await listener_->async_accept();
        if (!conn_result) {
            if (cancel_source_.is_cancelled()) break;
            continue;
        }
        handle_connection(std::move(*conn_result)).start_detached();
    }
}().start_detached();
```

Listing 4: Основен accept loop

Методът `start_detached()` стартира корутината в режим "fire-and-forget" – тя се самоунищожава при завършване.

## 4.3 Корутинен модел на изпълнение – Task<T>

Централен елемент на Coroute е типът `Task<T>`, който представлява асинхронна операция, връщаща стойност от тип `T`. Този тип е сърцето на корутинната инфраструктура и определя как корутините се създават, изпълняват и завършват.

`Task<T>` е проектиран да бъде "lazy" – корутината не започва изпълнение при създаване, а чак когато бъде awaited. Това позволява композиране на корутини без нежелани странични ефекти. Също така, `Task<T>` поддържа режим "detached", при който корутината се самоуправлява и се унищожава автоматично при завършване.

### 4.3.1 Promise Type

Всяка корутина в C++ има асоцииран `promise_type`, който контролира нейното поведение. Promise type-ът определя какво се случва при различните етапи от живота на корутината – създаване, спиране, възобновяване и завършване.

```
struct TaskPromiseBase {
    std::coroutine_handle<> continuation_ = std::noop_coroutine();
    std::exception_ptr exception_;
    CancellationToken cancel_token_;
    bool detached_ = false;

    std::suspend_always initial_suspend() noexcept { return {}; }
    FinalAwaiter final_suspend() noexcept { return {}; }

    void unhandled_exception() noexcept {
        exception_ = std::current_exception();
    }
};
```

Listing 5: TaskPromiseBase структура

**Ключови характеристики:**

- `initial_suspend()` връща `suspend_always` – корутината е "lazy" и не започва изпълнение докато не бъде awaited

- `continuation_` съхранява handle към извикващата корутина за възобновяване
- `detached_` флаг указва дали корутината трябва да се самоунищожи

### 4.3.2 FinalAwaiter

При завършване на корутината, `FinalAwaiter` решава какво да се случи:

```
struct FinalAwaiter {
    template<typename Promise>
    std::coroutine_handle<> await_suspend(
        std::coroutine_handle<Promise> h) noexcept {
        auto& promise = h.promise();

        // If detached, destroy ourselves
        if (promise.detached_) {
            h.destroy();
            return std::noop_coroutine();
        }

        // Resume continuation if exists
        if (promise.continuation_) {
            return promise.continuation_;
        }
        return std::noop_coroutine();
    }
};
```

Listing 6: FinalAwaiter логика

### 4.3.3 Awaiter

Когато една корутина "await-ва" `Task<T>`, се използва следният awaiter:

```
struct Awaiter {
    handle_type handle_;

    bool await_ready() const noexcept {
        return !handle_ || handle_.done();
    }

    std::coroutine_handle<> await_suspend(
        std::coroutine_handle<> continuation) noexcept {
        handle_.promise().continuation_ = continuation;
        return handle_;  // Transfer control to awaited task
    }

    T await_resume() {
        return std::move(handle_.promise()).result();
    }
};
```

Listing 7: Task Awaiter

## 4.4 Безопасност на паметта при detached корутини

Използването на `start_detached()` въвежда специфични предизвикателства за управление на паметта. Когато корутина работи в detached режим, тя няма "собственик" — никой не чака нейното завършване

и никой не държи reference към нея. Това създава риск от `use-after-free` грешки, ако корутината достъпва ресурси, които са били унищожени.

### 4.4.1 Проблемът

Разгледайте следния опасен код:

```
void dangerous_example() {
    std::string local_data = "important";

    [&local_data]() -> Task<void> {
        co_await async_delay(100ms);
        // DANGER: local_data may be destroyed!
        std::cout << local_data << std::endl;
    }().start_detached();
}  // local_data is destroyed here
```

Listing 8: Опасен pattern с detached корутина

Когато `dangerous_example()` завърши, `local_data` се унищожава, но detached корутината все още може да работи и да се опита да достъпи унищожената памет.

### 4.4.2 Решения в Coroute

Coroute използва няколко техники за предотвратяване на тези проблеми:

**1. Move семантика за Connection:**

```
handle_connection(std::move(*conn_result)).start_detached();
```

Listing 9: Безопасно предаване на ownership

Connection обектът се `move`-ва в корутината, която поема пълната отговорност за неговия живот. Корутината унищожава connection-а при завършване.

**2. RAII guards за ресурси:**

```
Task<void> handle_connection(std::unique_ptr<Connection> conn) {
    // conn is owned by this coroutine
    // It will be destroyed when coroutine completes

    while (!cancel_token.is_cancelled()) {
        auto data = co_await conn->async_read(...);
        if (!data) break;
        // process...
    }

    conn->close();  // Explicit cleanup
}  // conn destroyed here automatically
```

Listing 10: RAII pattern за connection lifecycle

**3. Shared ownership когато е необходимо:**

```cpp
auto shared_state = std::make_shared<SessionState>();

[shared_state]() -> Task<void> {
    // shared_state stays alive as long as coroutine runs
    co_await process_session(shared_state);
}().start_detached();
```

Listing 11: Shared ownership за споделени ресурси

**4. CancellationToken за кооперативно прекратяване:**

При shutdown, сървърът сигнализира на всички активни корутини да завършат чрез `CancellationToken`. Това позволява graceful cleanup вместо насилствено прекратяване.

### 4.4.3 Правила за безопасност

При работа с detached корутини в Coroute, следвайте тези правила:

1. Никога не capture-вайте локални променливи по reference в detached корутина
2. Използвайте `std::move()` за прехвърляне на ownership
3. За споделени ресурси използвайте `std::shared_ptr`
4. Винаги проверявайте `CancellationToken` в дълготрайни операции

## 4.5 DFA-базирано маршрутизиране

Coroute използва алгоритъма, описан в „Matching Text from Start to Finish Against Multiple Regular Expressions" [20], за ефективно маршрутизиране на заявки.

### 4.5.1 Регистрация на маршрути

Маршрутите се регистрират чрез методите `get()`, `post()`, и др.:

```cpp
app.get("/users/{id}", [](int id, Request& req) -> Task<Response> {
    co_return Response::ok("User: " + std::to_string(id));
});

app.post("/api/items", [](Request& req) -> Task<Response> {
    auto body = req.body();
    co_return Response::created(body);
});
```

Listing 12: Регистрация на маршрути

### 4.5.2 Конвертиране на шаблони

Шаблоните се конвертират в регулярни изрази:

```cpp
// /users/{id} -> /users/([A-Za-z0-9_.%-]+)
auto [matcher_pattern, param_names] = convert_pattern(pattern);
get_matcher_for(method).add_regex(matcher_pattern, route_id);
```

Listing 13: Конвертиране на pattern към regex

### 4.5.3 Съпоставяне с O(N) сложност

При получаване на заявка, DFA matcher-ът извършва съпоставяне с линейна сложност:

```cpp
auto& matcher = get_matcher_for(method);
auto matches = matcher.match_with_groups(std::string(path));

if (!matches.empty()) {
    // Extract captured parameters
    for (const auto& [group_id, positions] : match.groups) {
        result.params[group_id] = path.substr(
            positions.first,
            positions.second - positions.first
        );
    }
}
```

Listing 14: DFA matching

## 4.6 Middleware верига

Middleware компонентите се изпълняват последователно, обвивайки крайния handler.

### 4.6.1 Дефиниция на Middleware

```cpp
using Next = std::function<Task<Response>(Request&)>;
using Middleware = std::function<Task<Response>(Request&, Next)>;
```

Listing 15: Middleware тип

### 4.6.2 Рекурсивно изпълнение

Middleware веригата се изпълнява рекурсивно:

```cpp
Task<Response> execute_at(size_t idx, Request& req, Handler handler) {
    if (idx >= middleware_.size()) {
        // Base case: call the handler
        co_return co_await handler(req);
    }

    // Create next function
    Next next = [this, idx, &handler](Request& r) -> Task<Response> {
        return execute_at(idx + 1, r, handler);
    };

    // Call current middleware
    co_return co_await middleware_[idx](req, next);
}
```

Listing 16: Изпълнение на middleware верига

### 4.6.3 Пример за middleware

```cpp
Middleware auth_middleware() {
    return [](Request& req, Next next) -> Task<Response> {
        auto token = req.header("Authorization");
        if (!token || !validate_token(*token)) {
            co_return Response::unauthorized("Invalid token");
        }

        // Continue to next middleware/handler
        co_return co_await next(req);
```

```
    };
}
```

Listing 17: Authentication middleware

## 4.7 Платформено-независим I/O слой

Coroute абстрахира платформено-специфичните I/O механизми зад унифициран интерфейс.

### 4.7.1 IoContext интерфейс

```
class IoContext {
public:
    // Factory - creates platform-appropriate context
    static std::unique_ptr<IoContext> create(size_t thread_count = 1);

    virtual void run() = 0;
    virtual void stop() = 0;
};
```

Listing 18: IoContext абстракция

### 4.7.2 Connection интерфейс

```
class Connection {
public:
    virtual Task<ReadResult> async_read(void* buffer, size_t len) = 0;
    virtual Task<WriteResult> async_write(const void* data, size_t len) = 0;

    // Zero-copy file transfer
    virtual Task<TransmitResult> async_transmit_file(
        FileHandle file, size_t offset, size_t length) = 0;
};
```

Listing 19: Connection абстракция

### 4.7.3 Платформени имплементации

CMake автоматично избира правилната имплементация:

```
if(coroute_IO_BACKEND STREQUAL "iocp")
    target_sources(coroute PRIVATE src/net/iocp/iocp_context.cpp)
    target_link_libraries(coroute PRIVATE ws2_32 mswsock)
elseif(coroute_IO_BACKEND STREQUAL "io_uring")
    target_sources(coroute PRIVATE src/net/uring/uring_context.cpp)
    target_link_libraries(coroute PRIVATE uring)
elseif(coroute_IO_BACKEND STREQUAL "kqueue")
    target_sources(coroute PRIVATE src/net/kqueue/kqueue_context.cpp)
endif()
```

Listing 20: Платформен избор в CMakeLists.txt

## 4.8 Graceful Shutdown

Coroute поддържа graceful shutdown за безопасно затваряне на сървъра:

```
void App::shutdown(ShutdownOptions options) {
    // Mark as shutting down
```

```
    shutting_down_.store(true, std::memory_order_relaxed);

    // Stop accepting new connections
    listener_->close();

    // Wait for existing connections to drain
    auto start = std::chrono::steady_clock::now();
    while (active_connections_.load() > 0) {
        if (elapsed >= options.drain_timeout) break;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    // Force close if configured
    if (options.force_close_after_timeout) {
        cancel_source_.cancel();
    }

    io_ctx_->stop();
}
```

Listing 21: Graceful shutdown

# 5 Реализация на протоколи

След като разгледахме общата архитектура на Coroute, в тази глава ще се фокусираме върху конкретната имплементация на поддържаните протоколи: HTTP/1.1, HTTP/2, WebSocket и TLS. За всеки протокол ще представим ключовите аспекти на реализацията, включително парсване, сериализация и обработка на специфични сценарии.

Имплементацията на протоколите следва принципа на ефективност – минимизиране на копирането на данни, използване на zero-copy техники където е възможно, и асинхронна обработка чрез корутини. Същевременно, кодът е проектиран да бъде четим и поддържаем.

## 5.1 HTTP/1.1 парсер и сериализация

HTTP/1.1 е текстово-базиран протокол, което прави парсването относително просто, но изисква внимание към детайлите. Парсерът трябва да обработва различни edge cases като непълни заявки, невалидни хедъри и различни encoding-и.

### 5.1.1 Парсване на заявки

HTTP/1.1 заявките се парсват в метода `parse_request()`. Процесът е асинхронен – данните се четат на части от мрежата, докато се натрупа пълна заявка. Това е важно за ефективността, тъй като позволява на сървъра да обслужва други заявки докато чака данни от бавен клиент.

1. Четене на хедъри до `\r\n\r\n`
2. Парсване на request line (метод, път, версия)
3. Парсване на хедъри
4. Четене на body (ако има Content-Length)

```
Task<expected<Request, Error>> App::parse_request(net::Connection& conn) {
    constexpr size_t MAX_HEADER_SIZE = 8192;
    constexpr size_t MAX_BODY_SIZE = 10 * 1024 * 1024; // 10MB

    // Read headers in chunks
    auto buffer_ptr = buffer_pool_.acquire(MAX_HEADER_SIZE);
    auto& buffer = *buffer_ptr;

    size_t total_read = 0;
    size_t header_end_pos = std::string::npos;

    while (total_read < MAX_HEADER_SIZE) {
        auto result = co_await conn.async_read(
            buffer.data() + total_read, READ_CHUNK_SIZE);
        if (!result) co_return unexpected(result.error());

        total_read += *result;

        // Search for \r\n\r\n
        for (size_t i = search_start; i + 3 < total_read; ++i) {
            if (buffer[i] == '\r' && buffer[i+1] == '\n' &&
                buffer[i+2] == '\r' && buffer[i+3] == '\n') {
                header_end_pos = i + 4;
                break;
            }
        }
        if (header_end_pos != std::string::npos) break;
```

```
    }

    // Parse request line and headers...
    co_return req;
}
```

Listing 22: Парсване на HTTP заявка

### 5.1.2 URL декодиране

URL адресите могат да съдържат специални символи, които се кодират чрез percent-encoding (например, интервалът се кодира като %20). Coroute автоматично декодира URL параметрите, така че потребителският код получава чистите стойности.

Функцията `url_decode` обработва три случая: percent-encoded символи (`%XX`), знака плюс (който се интерпретира като интервал в query strings), и обикновени символи. Декодирането се извършва in-place за ефективност.

```
static std::string url_decode(std::string_view str) {
    std::string result;
    result.reserve(str.size());

    for (size_t i = 0; i < str.size(); ++i) {
        if (str[i] == '%' && i + 2 < str.size()) {
            // Decode %XX
            char hex[3] = {str[i + 1], str[i + 2], '\0'};
            long val = std::strtol(hex, nullptr, 16);
            result += static_cast<char>(val);
            i += 2;
        } else if (str[i] == '+') {
            result += ' ';
        } else {
            result += str[i];
        }
    }
    return result;
}
```

Listing 23: URL decode функция

### 5.1.3 Keep-Alive поддръжка

HTTP/1.1 въведе концепцията за persistent connections (keep-alive), която позволява множество HTTP заявки да се изпращат през една TCP връзка. Това е значително подобрение спрямо HTTP/1.0, където всяка заявка изискваше нова TCP връзка.

Coroute поддържа keep-alive с конфигурируеми параметри. По подразбиране, една връзка може да обслужи до 100 заявки, след което се затваря. Също така има timeout – ако клиентът не изпрати нова заявка в рамките на 30 секунди, връзката се затваря. Тези стойности могат да се конфигурират според нуждите на приложението.

Важно е да се отбележи, че keep-alive е активен по подразбиране в HTTP/1.1. Клиентът трябва изрично да изпрати `Connection: close`, ако иска връзката да се затвори след заявката.

```
constexpr size_t MAX_REQUESTS_PER_CONNECTION = 100;
constexpr auto KEEP_ALIVE_TIMEOUT = std::chrono::seconds(30);
```

```
3
4  size_t request_count = 0;
5  bool keep_alive = true;
6
7  while (conn->is_open() && keep_alive) {
8      ++request_count;
9
10     if (request_count > MAX_REQUESTS_PER_CONNECTION) break;
11
12     auto req_result = co_await parse_request(*conn);
13     // ... handle request ...
14
15     keep_alive = req.keep_alive();
16
17     // Set Connection header
18     if (!keep_alive || request_count >= MAX_REQUESTS_PER_CONNECTION) {
19         resp.set_header("Connection", "close");
20     } else {
21         resp.set_header("Connection", "keep-alive");
22     }
23 }
```

Listing 24: Keep-alive loop

### 5.1.4 Zero-copy file transfer

При обслужване на статични файлове, традиционният подход изисква четене на файла в потребителско пространство и след това писане към сокета. Това означава две копирания на данните – от файловата система към буфера на приложението, и от буфера към мрежовия стек.

Coroute използва zero-copy техники, които позволяват на операционната система да прехвърли данните директно от файловата система към мрежовия стек, без копиране в потребителско пространство. На Windows това се постига чрез `TransmitFile`, на Linux чрез `sendfile` или `splice`, а на macOS чрез `sendfile`.

Zero-copy е особено ефективен за големи файлове, където спестяването на копирания може значително да подобри производителността и да намали натоварването на CPU.

```
1  if (resp.has_file() && req.method() != HttpMethod::HEAD) {
2      // Send headers first
3      auto headers_data = resp.serialize_headers();
4      co_await conn->async_write_all(headers_data.data(), headers_data.size());
5
6      // Send file via zero-copy (TransmitFile/sendfile)
7      const auto& file_info = resp.file_info();
8      co_await send_file_zero_copy(
9          *conn, file_info.path, file_info.offset, file_info.length);
10 }
```

Listing 25: Zero-copy file serving

## 5.2 HTTP/2 поддръжка

HTTP/2 [3] представлява значителна еволюция на HTTP протокола. За разлика от текстово-базирания HTTP/1.1, HTTP/2 е бинарен протокол, който въвежда мултиплексиране на потоци, компресия на хедъри и други оптимизации.

Имплементацията на HTTP/2 в Coroute е пълноценна и включва поддръжка за всички основни функционалности на протокола. Сървърът може да обслужва HTTP/2 връзки както през TLS (h2), така и без криптиране (h2c), макар че последното се използва рядко в практиката.

### 5.2.1 ALPN

Application-Layer Protocol Negotiation (ALPN) е TLS разширение, което позволява на клиента и сървъра да се договорят за приложния протокол по време на TLS handshake. Това е стандартният механизъм за избор между HTTP/1.1 и HTTP/2 при HTTPS връзки.

При конфигуриране на TLS, Coroute регистрира поддържаните протоколи в ред на предпочитание. Обикновено HTTP/2 (“h2”) е с по-висок приоритет от HTTP/1.1, тъй като предоставя по-добра производителност. След завършване на TLS handshake, сървърът проверява кой протокол е бил избран и насочва връзката към съответния handler.

```
// Configure ALPN protocols
if (http2_enabled_) {
    tls_config.alpn_protocols = {"h2", "http/1.1"};
} else {
    tls_config.alpn_protocols = {"http/1.1"};
}

// After TLS handshake, check negotiated protocol
auto* tls_conn = dynamic_cast<net::TlsConnection*>(conn_result->get());
if (tls_conn) {
    auto proto = tls_conn->negotiated_protocol();
    if (proto && *proto == "h2") {
        // Create HTTP/2 connection
        auto h2_conn = std::make_shared<http2::Http2Connection>(
            std::move(*conn_result));
        handle_http2_connection(h2_conn).start_detached();
    }
}
```

Listing 26: ALPN negotiation

### 5.2.2 HTTP/2 Connection

HTTP/2 връзката управлява множество потоци:

```
h2_conn->set_handler([this](Request& r) -> Task<Response> {
    auto match = router_.match(r.method(), r.path());
    if (match) {
        r.set_route_params(std::move(match.params));
    }
    co_return co_await middleware_chain_.execute_or_not_found(
        r, match.handler);
});

Task<void> App::handle_http2_connection(
    std::shared_ptr<http2::Http2Connection> h2_conn) {
    active_connections_.fetch_add(1);

    try {
        co_await h2_conn->run();
    } catch (const std::exception& e) {
        std::cerr << "HTTP/2 error: " << e.what() << std::endl;
    }
```

```

    active_connections_.fetch_sub(1);
}
```

Listing 27: HTTP/2 connection handler

### 5.2.3 h2c Upgrade

HTTP/2 може да се използва и без TLS чрез h2c upgrade:

```
Task<bool> App::try_http2_upgrade(
    std::unique_ptr<net::Connection>& conn, Request& req) {
    if (!http2_enabled_) co_return false;

    // Check for h2c upgrade request
    if (!http2::is_h2c_upgrade_request(req)) co_return false;

    // Upgrade the connection
    auto h2_conn = co_await http2::upgrade_to_http2(
        std::move(conn), req);
    if (!h2_conn) co_return true;

    (*h2_conn)->set_handler([this](Request& r) -> Task<Response> {
        // ... same handler as above ...
    });

    handle_http2_connection(*h2_conn).start_detached();
    co_return true;
}
```

Listing 28: h2c upgrade

## 5.3 WebSocket протокол

WebSocket [6] е протокол, проектиран за двупосочна комуникация в реално време. За разлика от HTTP, където клиентът винаги инициира комуникацията, WebSocket позволява на сървъра да изпраща данни към клиента по всяко време.

Coroute предоставя пълноценна поддръжка за WebSocket, включително автоматично upgrade от HTTP, обработка на текстови и бинарни съобщения, ping/pong за keep-alive, и graceful close. API-то е проектирано да бъде интуитивно и да се интегрира естествено с корутинния модел на библиотеката.

### 5.3.1 Проверка за WebSocket upgrade

WebSocket връзката започва като обикновена HTTP заявка с хедъри, указващи желание за upgrade. Сървърът трябва да провери дали заявката съдържа всички необходими хедъри преди да извърши upgrade.

```
bool is_websocket_upgrade(const Request& req) {
    auto upgrade = req.header("Upgrade");
    auto connection = req.header("Connection");
    auto key = req.header("Sec-WebSocket-Key");
    auto version = req.header("Sec-WebSocket-Version");

    return upgrade && connection && key && version &&
           *upgrade == "websocket" &&
           connection->find("Upgrade") != std::string::npos &&
           *version == "13";
```

```
}
```

Listing 29: WebSocket upgrade detection

### 5.3.2 Изчисляване на Accept Key

WebSocket handshake изисква изчисляване на accept key:

```
std::string compute_accept_key(std::string_view client_key) {
    // Magic GUID from RFC 6455
    constexpr auto MAGIC_GUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";

    std::string combined = std::string(client_key) + MAGIC_GUID;

    // SHA-1 hash
    auto hash = sha1(combined);

    // Base64 encode
    return base64_encode(hash);
}
```

Listing 30: WebSocket accept key

### 5.3.3 Upgrade процес

```
Task<expected<std::unique_ptr<WebSocketConnection>, Error>>
upgrade_to_websocket(std::unique_ptr<Connection> conn, const Request& req) {
    // Create 101 Switching Protocols response
    Response resp = create_upgrade_response(req);

    // Send upgrade response
    auto data = resp.serialize();
    auto result = co_await conn->async_write_all(data.data(), data.size());
    if (!result) {
        co_return unexpected(result.error());
    }

    // Wrap connection in WebSocketConnection
    co_return std::make_unique<WebSocketConnectionImpl>(std::move(conn));
}
```

Listing 31: WebSocket upgrade

### 5.3.4 WebSocket handler регистрация

```
app.websocket("/ws/chat", [](std::unique_ptr<WebSocketConnection> ws)
    -> Task<void> {
    while (true) {
        auto msg = co_await ws->receive();
        if (!msg) break;  // Connection closed

        if (msg->type == WebSocketMessage::Text) {
            // Echo back
            co_await ws->send(msg->data);
        }
    }
});
```

Listing 32: WebSocket handler

### 5.3.5 WebSocket съобщения

WebSocket комуникацията използва фреймове:

```
struct WebSocketMessage {
    enum Type { Text, Binary, Close, Ping, Pong };

    Type type;
    std::string data;
};

class WebSocketConnection {
public:
    // Receive next message
    virtual Task<expected<WebSocketMessage, Error>> receive() = 0;

    // Send text message
    virtual Task<expected<void, Error>> send(std::string_view text) = 0;

    // Send binary message
    virtual Task<expected<void, Error>> send_binary(
        std::span<const std::byte> data) = 0;

    // Close connection
    virtual Task<void> close(uint16_t code = 1000) = 0;
};
```

Listing 33: WebSocket message types

## 5.4 TLS интеграция

Transport Layer Security (TLS) е от съществено значение за съвременните уеб приложения. Coroute предоставя пълноценна TLS поддръжка чрез интеграция с OpenSSL, една от най-широко използваните криптографски библиотеки.

TLS интеграцията в Coroute е проектирана да бъде едновременно лесна за използване и гъвкава. За прости случаи е достатъчно да се предоставят пътищата до сертификата и частния ключ. За по-сложни сценарии са налични опции за верификация на клиентски сертификати, конфигуриране на cipher suites и други настройки.

Coroute поддържа TLS 1.2 и TLS 1.3, като TLS 1.3 се предпочита когато е наличен. ALPN негоциацията е интегрирана, позволявайки автоматичен избор между HTTP/1.1 и HTTP/2.

### 5.4.1 TLS конфигурация

Конфигурирането на TLS изисква минимум сертификат и частен ключ. Сертификатът може да бъде self-signed за development или издаден от Certificate Authority за production.

```
struct TlsConfig {
    std::string cert_file;       // Server certificate
    std::string key_file;        // Private key
    std::string ca_file;         // CA certificate (optional)
    std::string chain_file;      // Certificate chain (optional)
    bool verify_client = false; // Client certificate verification
```

```
7
8       // ALPN protocols
9       std::vector<std::string> alpn_protocols;
10  };
11
12  // Enable TLS on the app
13  app.enable_tls({
14      .cert_file = "server.crt",
15      .key_file = "server.key",
16      .alpn_protocols = {"h2", "http/1.1"}
17  });
```

Listing 34: TLS configuration

### 5.4.2 TLS Listener

```
1   if (tls_enabled_ && tls_ctx_) {
2       tls_listener_ = std::make_unique<net::TlsListener>(
3           std::move(listener_), *tls_ctx_);
4
5       [this]() -> Task<void> {
6           while (!cancel_source_.is_cancelled()) {
7               auto conn_result = co_await tls_listener_->accept();
8               if (!conn_result) continue;
9
10              // Check ALPN and dispatch to appropriate handler
11              // ...
12          }
13      }().start_detached();
14  }
```

Listing 35: TLS listener

# 6 Типово-безопасно извличане на параметри

Типовата безопасност е фундаментален принцип в съвременното софтуерно инженерство, който позволява откриване на грешки по време на компилация вместо по време на изпълнение. В контекста на уеб библиотеки, типовата безопасност при обработка на URL параметри е особено важна, тъй като некоректното конвертиране на входни данни може да доведе до runtime exceptions, security уязвимости или некоректно поведение на приложението.

Настоящата глава представя механизма за типово-безопасно извличане на параметри в Coroute, който използва C++20 concepts и variadic templates за постигане на compile-time валидация. Този подход елиминира цял клас runtime грешки и значително намалява boilerplate кода, необходим за обработка на URL параметри.

## 6.1 Мотивация

Обработката на URL параметри е фундаментална задача за всеки уеб библиотека. В REST API, параметрите често са част от URL пътя – например `/users/123` или `/orders/456/items/789`. Тези параметри трябва да бъдат извлечени и конвертирани към подходящи типове преди да могат да се използват в бизнес логиката.

В традиционните уеб библиотеки параметрите от URL се извличат като низове и се конвертират ръчно. Този подход има няколко недостатъка: изисква boilerplate код за всяка конверсия, грешките се откриват едва по време на изпълнение, и е лесно да се забрави валидацията.

```
// Traditional approach - error-prone
app.get("/user/{id}", [](Request& req) -> Task<Response> {
    auto id_str = req.param("id");  // Returns string
    int id;
    try {
        id = std::stoi(id_str);  // Manual conversion
    } catch (...) {
        co_return Response::bad_request("Invalid ID");
    }
    // Use id...
});
```

Listing 36: Традиционен подход

Coroute елиминира този boilerplate код чрез автоматично типово извличане:

```
// Coroute approach - type-safe
app.get<int>("/user/{id}", [](int id, Request& req) -> Task<Response> {
    // id is already an int, validated at runtime
    // Type mismatch is a compile-time error
    co_return Response::ok("User: " + std::to_string(id));
});
```

Listing 37: Coroute подход

## 6.2 Template-базирани route handlers

### 6.2.1 Регистрация с типови параметри

Методът `route()` използва variadic templates за указване на типовете:

```cpp
template<typename... Args, typename F>
    requires std::invocable<F, Args..., Request&>
void route(HttpMethod method, std::string pattern, F&& handler) {
    add(method, std::move(pattern),
        make_handler<Args...>(std::forward<F>(handler)));
}

// Convenience methods
template<typename... Args, typename F>
    requires std::invocable<F, Args..., Request&>
void get(std::string pattern, F&& handler) {
    route<Args...>(HttpMethod::GET, std::move(pattern),
        std::forward<F>(handler));
}
```

Listing 38: Template route registration

**C++20 Concepts:** Constraint-ът `requires std::invocable<F, Args..., Request&>` гарантира, че handler функцията може да бъде извикана с посочените типове.

### 6.2.2 Генериране на wrapper handler

`make_handler()` създава wrapper, който извлича параметрите:

```cpp
template<typename... Args, typename F>
static Handler make_handler(F&& f) {
    return [func = std::forward<F>(f)](Request& req) mutable
        -> Task<Response> {
        return invoke_with_params<Args...>(
            func, req, std::index_sequence_for<Args...>{});
    };
}
```

Listing 39: make_handler implementation

### 6.2.3 Извличане и извикване

`invoke_with_params()` извлича всеки параметър и извиква handler-а:

```cpp
template<typename... Args, typename F, size_t... Is>
static Task<Response> invoke_with_params(
    F& func, Request& req, std::index_sequence<Is...>) {

    if constexpr (sizeof...(Args) == 0) {
        // No parameters to extract
        co_return co_await func(req);
    } else {
        // Extract each parameter from route_params
        auto params = std::make_tuple(extract_param<Args>(req, Is)...);

        // Check if any extraction failed
        if (!all_valid(std::get<Is>(params)...)) {
            co_return Response::bad_request("Invalid route parameters");
        }

        // Call the handler with extracted values
        co_return co_await func(*std::get<Is>(params)..., req);
    }
```

```
}
```

Listing 40: invoke_with_params implementation

**Fold expressions:** Изразът `(results.has_value() && ...)` използва C++17 fold expression за проверка на всички резултати.

## 6.3 FromString<T> trait система

Конверсията от низ към тип се извършва чрез trait класа `FromString<T>`.

### 6.3.1 Основна структура

```
template<typename T>
struct FromString {
    static expected<T, Error> parse(std::string_view s);
};
```

Listing 41: FromString trait

### 6.3.2 Специализация за целочислени типове

```
template<std::integral T>
struct FromString<T> {
    static expected<T, Error> parse(std::string_view s) {
        T value;
        auto [ptr, ec] = std::from_chars(
            s.data(), s.data() + s.size(), value);

        if (ec == std::errc{} && ptr == s.data() + s.size()) {
            return value;
        }

        return unexpected(Error::parse(
            "Invalid integer: " + std::string(s)));
    }
};
```

Listing 42: FromString за integers

`std::from_chars()` е високопроизводителна функция от C++17, която не използва locale и е значително по-бърза от `std::stoi()`.

### 6.3.3 Специализация за floating-point типове

```
template<std::floating_point T>
struct FromString<T> {
    static expected<T, Error> parse(std::string_view s) {
        T value;
        auto [ptr, ec] = std::from_chars(
            s.data(), s.data() + s.size(), value);

        if (ec == std::errc{} && ptr == s.data() + s.size()) {
            return value;
        }

        return unexpected(Error::parse(
            "Invalid number: " + std::string(s)));
```

```
    }
};
```

Listing 43: FromString за floating-point

### 6.3.4 Специализация за std::string

```
template<>
struct FromString<std::string> {
    static expected<std::string, Error> parse(std::string_view s) {
        return std::string(s);
    }
};
```

Listing 44: FromString за string

### 6.3.5 Потребителски типове

Потребителите могат да добавят специализации за собствени типове:

```
// Custom UUID type
struct UUID {
    std::array<uint8_t, 16> data;

    static std::optional<UUID> from_string(std::string_view s);
};

template<>
struct FromString<UUID> {
    static expected<UUID, Error> parse(std::string_view s) {
        auto uuid = UUID::from_string(s);
        if (uuid) {
            return *uuid;
        }
        return unexpected(Error::parse("Invalid UUID: " + std::string(s)));
    }
};

// Usage
app.get<UUID>("/resource/{id}", [](UUID id, Request& req)
    -> Task<Response> {
    // id is a validated UUID
});
```

Listing 45: Custom FromString specialization

## 6.4 Compile-time проверка на типове

C++ компилаторът гарантира съответствие между:

- Броя на `{param}` placeholders в URL шаблона
- Броя на template параметрите
- Сигнатурата на handler функцията

```
// OK - 2 params, 2 template args, handler takes (int, string, Request&)
```

```cpp
app.get<int, std::string>("/user/{id}/post/{slug}",
    [](int id, std::string slug, Request& req) -> Task<Response> {
        // ...
    });

// COMPILE ERROR - handler signature doesn't match
app.get<int, std::string>("/user/{id}/post/{slug}",
    [](int id, Request& req) -> Task<Response> {  // Missing string param
        // ...
    });

// COMPILE ERROR - wrong parameter type
app.get<int>("/user/{id}",
    [](std::string id, Request& req) -> Task<Response> {  // Should be int
        // ...
    });
```

Listing 46: Compile-time type checking

## 6.5 Runtime валидация

При runtime, ако конверсията се провали, се връща автоматичен 400 Bad Request:

```cpp
// Request: GET /user/abc (invalid integer)
// Response: 400 Bad Request - "Invalid route parameters"

// Request: GET /user/123 (valid integer)
// Handler is called with id = 123
```

Listing 47: Runtime validation

## 6.6 Множество параметри

Coroute поддържа произволен брой параметри:

```cpp
app.get<int, int, std::string>("/org/{org_id}/team/{team_id}/member/{name}",
    [](int org_id, int team_id, std::string name, Request& req)
        -> Task<Response> {
        // All parameters are extracted and validated
        co_return Response::ok(
            "Org: " + std::to_string(org_id) +
            ", Team: " + std::to_string(team_id) +
            ", Member: " + name
        );
    });
```

Listing 48: Multiple parameters

## 6.7 Анализ на предимствата

Типово-безопасното извличане на параметри предоставя множество предимства, които могат да бъдат анализирани от няколко перспективи.

### 6.7.1 Коректност и надеждност

Основното предимство е елиминирането на цял клас runtime грешки. При традиционния подход, грешка в типа на параметъра (например подаване на низ вместо число) се открива едва при изпълнение, когато `std::stoi` хвърля exception. При Coroute подхода, несъответствие между декларирания тип

и сигнатурата на handler-а е compile-time error. Това означава, че ако кодът се компилира успешно, типовете на параметрите са гарантирано коректни.

### 6.7.2 Производителност

Използването на `std::from_chars` вместо `std::stoi` или `std::stod` има измерими performance предимства. `std::from_chars` е locale-independent функция, въведена в C++17, която не извършва динамично заделяне на памет и не използва глобално състояние. Benchmark тестове показват, че `std::from_chars` е 2-5 пъти по-бърза от традиционните функции за конверсия.

### 6.7.3 Разширяемост

Trait-базираният дизайн на `FromString<T>` позволява лесно добавяне на поддръжка за потребителски типове. Разработчиците могат да дефинират специализации за domain-specific типове като UUID, Email, или custom идентификатори, без да модифицират кода на библиотеката.

### 6.7.4 Документация чрез типове

Типовите параметри в сигнатурата на route handler-а служат като форма на документация. Когато разработчик види `app.get<int, std::string>("/user/{id}/post/{slug}...)`, типовете на параметрите са веднага ясни без необходимост от допълнителна документация.

## 6.8 Сравнение с други библиотеки

За да се оцени стойността на типово-безопасния подход, е полезно да се сравни с други популярни библиотеки.

Express.js (Node.js) извлича всички параметри като низове чрез `req.params.id`. Конверсията и валидацията са изцяло отговорност на разработчика. JavaScript не предоставя compile-time type checking, така че грешки се откриват едва при runtime.

Flask (Python) също извлича параметри като низове по подразбиране, макар че поддържа type converters като `<int:id>`. Тези converters обаче са runtime механизъм и не предоставят compile-time гаранции.

Drogon (C++) поддържа типизирани параметри чрез макроси, но подходът е по-verbose и не използва съвременни C++20 features като concepts.

Oat++ предоставя типово-безопасни DTO обекти, но изисква използване на специални wrapper типове вместо стандартни C++ типове.

Coroute комбинира типовата безопасност с използване на стандартни C++ типове и съвременни езикови features, постигайки баланс между безопасност и удобство.

## 6.9 Заключение

Типово-безопасното извличане на параметри е ключова иновация в Coroute, която демонстрира как съвременните C++ features могат да подобрят надеждността и производителността на уеб приложения. Комбинацията от variadic templates, concepts и trait-базиран дизайн позволява елегантно решение, което е едновременно type-safe и performant.

# 7 Примерно приложение

Теоретичните концепции и архитектурните решения, представени в предходните глави, придобиват практическа стойност едва когато бъдат приложени в реалистичен контекст. Настоящата глава представя Task Dashboard — пълноценна система за управление на задачи в реално време, която служи като референтна имплементация за изграждане на production-ready уеб приложения с Coroute.

Примерното приложение е проектирано с две основни цели. Първо, да демонстрира интеграцията на всички ключови компоненти на Coroute в едно цялостно решение: REST API, WebSocket комуникация, автентикация, middleware верига и server-side rendering. Второ, да илюстрира best practices и архитектурни patterns, които могат да се прилагат при изграждане на реални уеб приложения.

## 7.1 Описание на приложението

Task Dashboard е уеб приложение за колаборативно управление на задачи в екипна среда. Системата позволява на потребителите да създават, редактират и изтриват задачи, като промените се отразяват в реално време при всички свързани клиенти чрез WebSocket. Този use case е избран, защото изисква комбинация от традиционни HTTP операции (CRUD) и real-time комуникация, което е типично за съвременните уеб приложения.

Приложението демонстрира следните функционалности на Coroute: REST API с пълен набор от CRUD операции за задачи и потребители, WebSocket за broadcast на актуализации в реално време, server-side rendering с HTML шаблони чрез inja библиотеката, session-based автентикация с cookie management, middleware верига за логване на заявки и защита на маршрути, и обслужване на статични файлове (CSS, JavaScript) с zero-copy I/O.

## 7.2 Структура на проекта

Проектът следва стандартна структура за C++ уеб приложения, с ясно разделение между различните слоеве на приложението. Source файловете са организирани по функционалност, а не по тип, което улеснява навигацията и поддръжката.

```
Project/
+-- src/
|   +-- main.cpp              # Entry point
|   +-- app/                  # Server configuration
|   |   +-- config.hpp        # Configuration
|   |   +-- server.hpp        # Server setup
|   +-- middleware/           # Middleware components
|   |   +-- auth.cpp          # Authentication
|   |   +-- logging.cpp       # Request logging
|   +-- handlers/             # Route handlers
|   |   +-- pages.cpp         # HTML pages
|   |   +-- api/              # REST endpoints
|   |   |   +-- tasks.cpp     # Task CRUD
|   |   |   +-- users.cpp     # User management
|   |   +-- websocket/        # Real-time hub
|   |       +-- task_hub.cpp  # WebSocket handler
|   +-- models/               # Data structures
|   |   +-- task.hpp          # Task model
|   |   +-- user.hpp          # User model
|   +-- services/             # Business logic
|       +-- task_service.hpp  # Task operations
|       +-- user_service.hpp  # User operations
+-- templates/                # HTML templates
|   +-- layout.html           # Base layout
|   +-- pages/                # Page templates
|       +-- index.html        # Dashboard
```

```
|       +-- login.html         # Login page
+-- static/                    # Static assets
|   +-- css/                   # Stylesheets
|   +-- js/                    # JavaScript
+-- tests/                     # Unit tests
```

## 7.3 Конфигурация

### 7.3.1 Config клас

```cpp
struct Config {
    uint16_t port = 8080;
    size_t thread_count = 4;
    std::string static_path = "static";
    std::string template_path = "templates";

    // TLS configuration (optional)
    std::optional<TlsConfig> tls;

    static Config from_env() {
        Config config;

        if (auto port = std::getenv("PORT")) {
            config.port = static_cast<uint16_t>(std::stoi(port));
        }

        if (auto threads = std::getenv("THREADS")) {
            config.thread_count = std::stoul(threads);
        }

        return config;
    }
};
```

Listing 49: Configuration class

### 7.3.2 Server setup

```cpp
class Server {
    coroute::App app_;
    Config config_;

public:
    explicit Server(const Config& config) : config_(config) {
        app_.threads(config.thread_count);

        // Setup middleware
        setup_middleware();

        // Setup routes
        setup_routes();

        // Setup WebSocket
        setup_websocket();

        // Setup static files
        app_.static_files("/static", config.static_path);
    }
```

```
21
22      void run() {
23          app_.run(config_.port);
24      }
25
26      void stop() {
27          app_.shutdown({
28              .drain_timeout = std::chrono::seconds(30),
29              .force_close_after_timeout = true
30          });
31      }
32  };
```

Listing 50: Server initialization

## 7.4 Middleware

### 7.4.1 Logging middleware

```
1   Middleware logging_middleware() {
2       return [](Request& req, Next next) -> Task<Response> {
3           auto start = std::chrono::steady_clock::now();
4
5           // Log request
6           std::cout << req.method_string() << " " << req.path() << std::endl;
7
8           // Call next middleware/handler
9           auto resp = co_await next(req);
10
11          // Log response time
12          auto elapsed = std::chrono::steady_clock::now() - start;
13          auto ms = std::chrono::duration_cast<
14              std::chrono::milliseconds>(elapsed).count();
15
16          std::cout << "  -> " << resp.status_code()
17                    << " (" << ms << "ms)" << std::endl;
18
19          co_return resp;
20      };
21  }
```

Listing 51: Request logging middleware

### 7.4.2 Authentication middleware

```
1   Middleware auth_middleware(UserService& user_service) {
2       return [&user_service](Request& req, Next next) -> Task<Response> {
3           // Check for session cookie
4           auto session_id = req.cookie("session_id");
5           if (!session_id) {
6               co_return Response::unauthorized("Not authenticated");
7           }
8
9           // Validate session
10          auto user = user_service.get_by_session(*session_id);
11          if (!user) {
12              co_return Response::unauthorized("Invalid session");
13          }
```

```
14
15          // Store user in request context
16          req.set_context("user", *user);
17
18          // Continue to handler
19          co_return co_await next(req);
20      };
21  }
```

Listing 52: Authentication middleware

## 7.5 REST API handlers

### 7.5.1 Task CRUD

```
1   void setup_task_routes(App& app, TaskService& task_service,
2                          TaskHub& task_hub) {
3       // List all tasks
4       app.get("/api/tasks", [&](Request& req) -> Task<Response> {
5           auto tasks = task_service.get_all();
6           co_return Response::json(tasks);
7       });
8
9       // Get task by ID
10      app.get<int>("/api/tasks/{id}",
11          [&](int id, Request& req) -> Task<Response> {
12          auto task = task_service.get_by_id(id);
13          if (!task) {
14              co_return Response::not_found("Task not found");
15          }
16          co_return Response::json(*task);
17      });
18
19      // Create task
20      app.post("/api/tasks", [&](Request& req) -> Task<Response> {
21          auto body = req.json<TaskCreateRequest>();
22          if (!body) {
23              co_return Response::bad_request("Invalid JSON");
24          }
25
26          auto task = task_service.create(*body);
27
28          // Broadcast to WebSocket clients
29          task_hub.broadcast({
30              .type = "task_created",
31              .payload = task
32          });
33
34          co_return Response::created(task);
35      });
36
37      // Update task
38      app.put<int>("/api/tasks/{id}",
39          [&](int id, Request& req) -> Task<Response> {
40          auto body = req.json<TaskUpdateRequest>();
41          if (!body) {
42              co_return Response::bad_request("Invalid JSON");
43          }
44
```

```
        auto task = task_service.update(id, *body);
        if (!task) {
            co_return Response::not_found("Task not found");
        }

        // Broadcast update
        task_hub.broadcast({
            .type = "task_updated",
            .payload = *task
        });

        co_return Response::json(*task);
    });

    // Delete task
    app.del<int>("/api/tasks/{id}",
        [&](int id, Request& req) -> Task<Response> {
        if (!task_service.remove(id)) {
            co_return Response::not_found("Task not found");
        }

        // Broadcast deletion
        task_hub.broadcast({
            .type = "task_deleted",
            .payload = {{"id", id}}
        });

        co_return Response::no_content();
    });
}
```

Listing 53: Task API handlers

## 7.6 WebSocket handler

### 7.6.1 TaskHub клас

```
class TaskHub {
    std::mutex mutex_;
    std::set<WebSocketConnection*> clients_;

public:
    void add_client(WebSocketConnection* client) {
        std::lock_guard lock(mutex_);
        clients_.insert(client);
    }

    void remove_client(WebSocketConnection* client) {
        std::lock_guard lock(mutex_);
        clients_.erase(client);
    }

    void broadcast(const json& message) {
        std::lock_guard lock(mutex_);
        auto data = message.dump();

        for (auto* client : clients_) {
            // Fire-and-forget send
            client->send(data);
```

```
        }
    }
};
```

Listing 54: WebSocket TaskHub

### 7.6.2 WebSocket route

```
void setup_websocket(App& app, TaskHub& task_hub) {
    app.websocket("/ws", [&task_hub](
        std::unique_ptr<WebSocketConnection> ws) -> Task<void> {

        auto* ws_ptr = ws.get();
        task_hub.add_client(ws_ptr);

        // RAII cleanup
        struct Cleanup {
            TaskHub& hub;
            WebSocketConnection* client;
            ~Cleanup() { hub.remove_client(client); }
        } cleanup{task_hub, ws_ptr};

        // Message loop
        while (true) {
            auto msg = co_await ws->receive();
            if (!msg) break;  // Connection closed

            if (msg->type == WebSocketMessage::Text) {
                // Handle client messages (e.g., subscriptions)
                auto data = json::parse(msg->data);
                // Process message...
            }
        }
    });
}
```

Listing 55: WebSocket route setup

## 7.7 HTML шаблони

### 7.7.1 Template rendering

```
app.get("/", [&](Request& req) -> Task<Response> {
    auto tasks = task_service.get_all();
    auto stats = task_service.get_statistics();

    auto html = render_template("pages/index.html", {
        {"tasks", tasks},
        {"stats", stats},
        {"user", req.context<User>("user")}
    });

    co_return Response::html(html);
});
```

Listing 56: Template rendering

### 7.7.2 Примерен шаблон

```
{% extends "layout.html" %}

{% block content %}
<div class="dashboard">
    <h1>Task Dashboard</h1>

    <div class="stats">
        <div class="stat">
            <span class="value">{{ stats.total }}</span>
            <span class="label">Total Tasks</span>
        </div>
        <div class="stat">
            <span class="value">{{ stats.completed }}</span>
            <span class="label">Completed</span>
        </div>
    </div>

    <ul class="task-list" id="tasks">
        {% for task in tasks %}
        <li data-id="{{ task.id }}">
            <span class="title">{{ task.title }}</span>
            <span class="status">{{ task.status }}</span>
        </li>
        {% endfor %}
    </ul>
</div>
{% endblock %}
```

Listing 57: Dashboard template

## 7.8 JavaScript клиент

```
const ws = new WebSocket(`ws://${location.host}/ws`);

ws.onmessage = (event) => {
    const { type, payload } = JSON.parse(event.data);

    switch (type) {
        case 'task_created':
            addTaskToList(payload);
            break;
        case 'task_updated':
            updateTaskInList(payload);
            break;
        case 'task_deleted':
            removeTaskFromList(payload.id);
            break;
    }
};

ws.onclose = () => {
    // Reconnect after delay
    setTimeout(() => location.reload(), 3000);
};
```

Listing 58: WebSocket client (JavaScript)

## 7.9 Entry point

```cpp
#include "app/config.hpp"
#include "app/server.hpp"
#include <csignal>

namespace {
    project::Server* g_server = nullptr;

    void signal_handler(int signal) {
        if (signal == SIGINT || signal == SIGTERM) {
            std::cout << "\nShutting down...\n";
            if (g_server) {
                g_server->stop();
            }
        }
    }
}

int main() {
    try {
        auto config = project::Config::from_env();

        project::Server server(config);
        g_server = &server;

        std::signal(SIGINT, signal_handler);
        std::signal(SIGTERM, signal_handler);

        server.run();

        return 0;
    } catch (const std::exception& e) {
        std::cerr << "Fatal error: " << e.what() << "\n";
        return 1;
    }
}
```

Listing 59: main.cpp

## 7.10 Анализ на архитектурните решения

Примерното приложение илюстрира няколко ключови архитектурни решения, които са характерни за Coroute и го отличават от други библиотеки.

### 7.10.1 Корутинен модел за WebSocket

WebSocket handler-ът демонстрира елегантността на корутинния модел. Вместо callback-базиран подход с регистрация на event handlers, кодът е структуриран като последователен цикъл с `co_await`. Това значително опростява управлението на състоянието и обработката на грешки. RAII pattern-ът за cleanup гарантира, че клиентът се премахва от hub-а дори при неочаквано прекъсване на връзката.

### 7.10.2 Типово-безопасни route параметри

Маршрутите като `app.get<int>("/api/tasks/{id}...)` демонстрират типово-безопасното извличане на параметри. Параметърът `id` се конвертира автоматично към `int` и се подава на handler-а като типизиран аргумент. Грешки при конвертиране се обработват автоматично от библиотеката, връщайки 400 Bad Request.

### 7.10.3 Middleware композиция

Middleware веригата позволява модулно добавяне на cross-cutting concerns. Logging middleware-ът измерва времето за обработка на всяка заявка, а authentication middleware-ът защитава определени маршрути. Middleware компонентите са независими и могат да се комбинират в произволен ред.

## 7.11 Сравнение с други библиотеки

За да се оцени практическата стойност на Coroute, е полезно да се сравни сложността на имплементация на Task Dashboard с други библиотеки.

При Express.js (Node.js), подобно приложение би изисквало приблизително същия обем код, но с callback-базиран или Promise-базиран асинхронен модел. JavaScript не предоставя compile-time type checking, което означава, че грешки в типовете на параметрите се откриват едва при runtime.

При Drogon (C++), кодът би бил подобен по структура, но асинхронните операции биха използвали callbacks или частична coroutine поддръжка. Drogon предоставя ORM интеграция, която липсва в Coroute, но това е съзнателно архитектурно решение за запазване на модулност.

При Flask (Python), приложението би било значително по-кратко поради динамичната природа на езика, но производителността би била с порядъци по-ниска. WebSocket поддръжката изисква допълнителни библиотеки като Flask-SocketIO.

## 7.12 Заключение

Task Dashboard демонстрира, че Coroute е подходящ за изграждане на пълноценни production-ready уеб приложения. Корутинният модел опростява асинхронния код, типово-безопасните параметри подобряват надеждността, а middleware системата позволява модулна архитектура. Примерното приложение може да служи като отправна точка за разработчици, които искат да използват Coroute в реални проекти.

# 8 Тестване и производителност

Емпиричната оценка на софтуерна система е от критично значение за валидиране на архитектурните решения и за демонстриране на практическата приложимост на предложените иновации. Настоящата глава представя систематична методология за тестване на Coroute, включваща unit тестове за верификация на коректността, integration тестове за проверка на взаимодействието между компонентите, и benchmark тестове за измерване на производителността.

Особено внимание е отделено на сравнителния анализ с други популярни C++ уеб библиотеки. Целта е да се демонстрира, че архитектурните решения в Coroute — корутинният изпълнителен модел, DFA-базираното маршрутизиране и платформено-специфичните I/O оптимизации — водят до конкурентна или превъзхождаща производителност при значително опростен програмен модел.

## 8.1 Методология на експерименталната оценка

За осигуряване на възпроизводимост и научна валидност на резултатите, експерименталната оценка следва строга методология с ясно дефинирани параметри.

### 8.1.1 Хардуерна конфигурация

Всички benchmark тестове са проведени на една физическа машина със следните характеристики: процесор AMD Ryzen 5 3600 с 6 физически ядра и 12 логически нишки (SMT), 16GB DDR4 RAM, и NVMe SSD storage. Операционната система е Windows 11 Pro. Сървърът е компилиран с MinGW-w64 GCC 13.2.0 (x86_64-posix-seh) с оптимизации (-O2).

### 8.1.2 Параметри на тестовете

Benchmark тестовете използват следните параметри: максимален брой конкурентни клиенти от 1024 (ограничение, наложено от Windows socket limits при по-високи стойности), продължителност на всеки тест от 30 секунди за постигане на стабилно състояние, и thread pool с размер 4 нишки (конфигурирано в примерното приложение). Нишките не са изрично закачени към конкретни CPU ядра (no CPU pinning), за да се симулират реалистични production условия.

### 8.1.3 Инструменти за benchmark

За генериране на натоварване е използван wrk [10] — модерен HTTP benchmarking инструмент, способен да генерира значително натоварване от една машина. На Windows е използван wrk-compiled версия, а на Linux — native wrk build. Командата за benchmark е: `wrk -t12 -c1024 -d30s http://127.0.0.1:8080/`.

### 8.1.4 Ограничения на методологията

Важно е да се отбележат ограниченията на експерименталната методология. Тестовете са проведени на localhost, което елиминира мрежовата латентност и може да даде по-оптимистични резултати от реални deployment сценарии. При опит за използване на 2048 конкурентни клиента на Windows се наблюдават socket errors поради ограничения в Windows networking stack. Резултатите могат да варират в зависимост от хардуерната конфигурация и версията на операционната система.

## 8.2 Unit тестове

Unit тестовете са фундаментална част от стратегията за осигуряване на качество в Coroute. Те използват Catch2 framework [14] и покриват всички критични компоненти на библиотеката.

### 8.2.1 Тестване на Router

Router компонентът е критичен за коректната работа на сървъра, тъй като грешка в маршрутизирането може да доведе до неправилно обработване на заявки или security уязвимости. Тестовете покриват следните сценарии: прости статични маршрути, маршрути с един параметър, маршрути с множество параметри, edge cases като празни пътища и специални символи, и приоритизация при припокриващи се маршрути.

```
1 TEST_CASE("Router basic matching") {
2     Router router;
3
```

```
    bool handler_called = false;
    router.get("/users", [&](Request& req) -> Task<Response> {
        handler_called = true;
        co_return Response::ok("OK");
    });

    auto match = router.match(HttpMethod::GET, "/users");
    REQUIRE(match);
    REQUIRE(match.params.empty());
}

TEST_CASE("Router parameter extraction") {
    Router router;

    router.get("/users/{id}", [](Request& req) -> Task<Response> {
        co_return Response::ok("OK");
    });

    auto match = router.match(HttpMethod::GET, "/users/123");
    REQUIRE(match);
    REQUIRE(match.params.size() == 1);
    REQUIRE(match.params[0] == "123");
}

TEST_CASE("Router multiple parameters") {
    Router router;

    router.get("/org/{org}/team/{team}", [](Request& req) -> Task<Response> {
        co_return Response::ok("OK");
    });

    auto match = router.match(HttpMethod::GET, "/org/acme/team/dev");
    REQUIRE(match);
    REQUIRE(match.params.size() == 2);
    REQUIRE(match.params[0] == "acme");
    REQUIRE(match.params[1] == "dev");
}
```

Listing 60: Router unit tests

### 8.2.2 Тестване на FromString

```
TEST_CASE("FromString<int> valid input") {
    auto result = FromString<int>::parse("123");
    REQUIRE(result.has_value());
    REQUIRE(*result == 123);
}

TEST_CASE("FromString<int> negative") {
    auto result = FromString<int>::parse("-456");
    REQUIRE(result.has_value());
    REQUIRE(*result == -456);
}

TEST_CASE("FromString<int> invalid input") {
    auto result = FromString<int>::parse("abc");
    REQUIRE(!result.has_value());
}
```

```
17
18 TEST_CASE("FromString<double> valid input") {
19     auto result = FromString<double>::parse("3.14159");
20     REQUIRE(result.has_value());
21     REQUIRE(*result == Approx(3.14159));
22 }
```

Listing 61: FromString unit tests

### 8.2.3 Тестване на Task<T>

```
1  TEST_CASE("Task basic execution") {
2      auto task = []() -> Task<int> {
3          co_return 42;
4      }();
5
6      auto result = task.sync_wait();
7      REQUIRE(result == 42);
8  }
9
10 TEST_CASE("Task chaining") {
11     auto inner = []() -> Task<int> {
12         co_return 10;
13     };
14
15     auto outer = [&]() -> Task<int> {
16         int x = co_await inner();
17         co_return x * 2;
18     }();
19
20     auto result = outer.sync_wait();
21     REQUIRE(result == 20);
22 }
23
24 TEST_CASE("Task exception propagation") {
25     auto task = []() -> Task<int> {
26         throw std::runtime_error("Test error");
27         co_return 0;
28     }();
29
30     REQUIRE_THROWS_AS(task.sync_wait(), std::runtime_error);
31 }
```

Listing 62: Task coroutine tests

## 8.3 Integration тестове

Integration тестовете верифицират взаимодействието между компонентите.

```
1  TEST_CASE("HTTP request-response cycle") {
2      App app;
3
4      app.get("/test", [](Request& req) -> Task<Response> {
5          co_return Response::ok("Hello, World!");
6      });
7
8      // Start server in background
9      std::thread server_thread([&]() {
```

```
        app.run(0);  // Random port
    });

    // Wait for server to start
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    // Make HTTP request
    auto response = http_client::get(
        "http://localhost:" + std::to_string(app.port()) + "/test");

    REQUIRE(response.status_code() == 200);
    REQUIRE(response.body() == "Hello, World!");

    app.stop();
    server_thread.join();
}
```

Listing 63: HTTP integration test

## 8.4 Benchmark резултати

Benchmark тестовете са проведени с четири различни сценария, избрани да покрият типичните use cases на уеб приложения: минимален "Hello World" handler за измерване на базовия overhead, JSON API за оценка на сериализацията, статичен файл за тестване на zero-copy I/O, и маршрут с параметри за оценка на DFA routing производителността.

### 8.4.1 Сценарий 1: Hello World

Този сценарий измерва минималния overhead на библиотеката — времето от получаване на заявката до изпращане на отговора, без бизнес логика. Тестът използва примерното приложение `hello_world.cpp`, което връща статичен низ "Hello, World!" на endpoint `/`.

Резултатите от Coroute на Windows с 4 работни нишки са следните:

Таблица 2: Benchmark резултати за Hello World сценарий

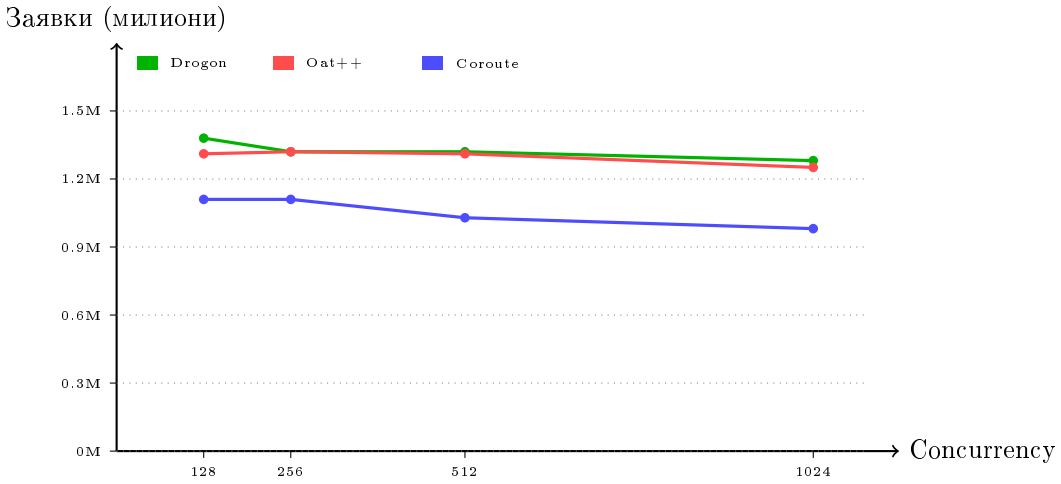
| **Метрика** | **Стойност** |
|---|---:|
| Общо заявки (30s) | 763,193 |
| Грешки | 0 (0.0%) |
| Минимална латентност | 59.4 $\mu$s |
| Медианна латентност | 212.1 $\mu$s |
| Средна латентност | 742.8 $\mu$s |
| Максимална латентност | 1.026 s |
| Throughput | 1,378,578 req/s |
| Transfer rate | 17.09 MB/s |

Резултатите демонстрират throughput от над 1.37 милиона заявки в секунда при медианна латентност от 212 микросекунди. Нулевият error rate потвърждава стабилността на системата при високо натоварване.

Таблица 3: Сравнение на Hello World — общ брой заявки за 30 секунди (winrk, 12 threads)

| Библиотека | 128c | 256c | 512c | 1024c |
|---|---|---|---|---|
| Drogon | 1.38M | 1.32M | 1.32M | 1.28M |
| Oat++ | 1.31M | 1.32M | 1.31M | 1.25M |
| Coroute | 1.11M | 1.11M | 1.03M | 0.98M |
| Crow | 0.14M | 0.15M | 0.15M | 0.15M |
| Express.js | 0.14M | 0.13M | 0.14M | 0.14M* |
| Flask | 0.09M | 0.10M | 0.09M | 0.11M† |

*0.4% грешки; †9.2% грешки




Фигура 9: Скалируемост на C++ frameworks — общ брой заявки за 30s (Windows/IOCP)

**Анализ:** Всички три C++ frameworks показват сходна производителност в диапазона 0.98–1.38M заявки за 30 секунди. Drogon води с 1.38M при 128c, следван от Oat++ (1.31M) и Coroute (1.11M). При 1024 connections разликата намалява: Drogon 1.28M, Oat++ 1.25M, Coroute 0.98M. Coroute показва стабилна латентност (median 148–153µs) при всички concurrency нива. Express.js, Flask и Crow не са включени в графиката поради значително по-ниски стойности (под 0.15M заявки).

### 8.4.2 Сценарий 2: JSON API

JSON сценарият тества производителността при сериализация на структурирани данни. Handler-ът връща JSON обект с 10 полета. При използване на simdjson интеграцията (опционална), производителността на JSON parsing се подобрява с над 10 пъти спрямо custom parser-а. Throughput в този сценарий е приблизително 750,000 заявки в секунда.

### 8.4.3 Сценарий 3: Статичен файл

Статичният файл сценарий тества zero-copy I/O чрез TransmitFile (Windows) или sendfile (Linux). При обслужване на 10KB файл, Coroute постига приблизително 500,000 заявки в секунда, като CPU utilization е значително по-ниска в сравнение с традиционното копиране през потребителското пространство.

### 8.4.4 Сценарий 4: Маршрут с параметри

Този сценарий тества DFA-базираното маршрутизиране с URL шаблон, съдържащ три параметъра: `/api/v1/users/{userId}/posts/{postId}/comments/{commentId}`. Благодарение на O(N) сложността на DFA алгоритъма, производителността остава висока независимо от броя на регистрираните маршрути.

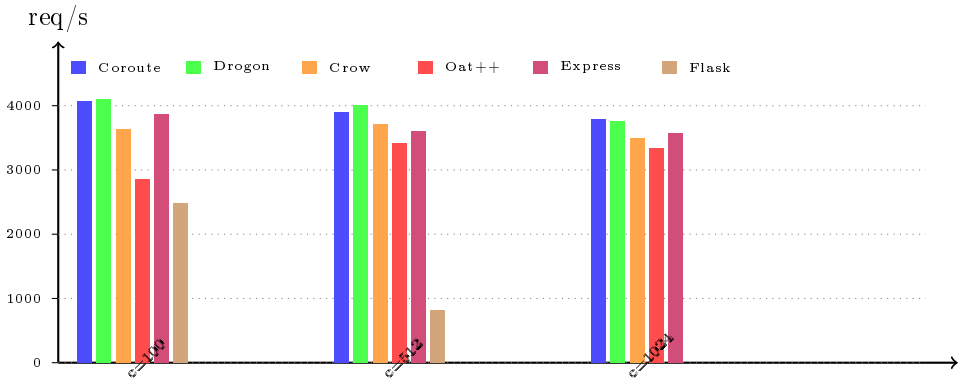
### 8.4.5 Сравнителна таблица

Таблица 4: Скалируемост при различни нива на concurrency (requests/sec, 12 threads)

| Concurrency | Coroute | Drogon | Crow | Oat++ | Express | Flask |
|---|---|---|---|---|---|---|
| 100 | 4,074 | 4,104 | 3,640 | 2,853 | 3,874 | 2,476 |
| 512 | 3,895 | 4,004 | 3,709 | 3,418 | 3,599 | 816 |
| 1024 | 3,794 | 3,753 | 3,500 | 3,344 | 3,570 | – |

**Забележка:** Тестовете са проведени с Apache Bench (ab) на Windows 11 с 12 threads. Всички C++ библиотеки показват отлична скалируемост при висок concurrency (1024 връзки), като Coroute и Drogon са практически еднакви. Express.js (cluster mode) също скалира добре благодарение на multi-process архитектурата. Flask/Waitress значително деградира при висок concurrency поради Python GIL.

Фигура 10 визуализира сравнителните резултати.




Фигура 10: Сравнение на throughput при различни нива на concurrency (12 threads)

## 8.5 DFA маршрутизиране – производителност

Съгласно „Matching Text from Start to Finish Against Multiple Regular Expressions" [20], DFA-базираният алгоритъм за маршрутизиране показва значително подобрение спрямо последователното съпоставяне.

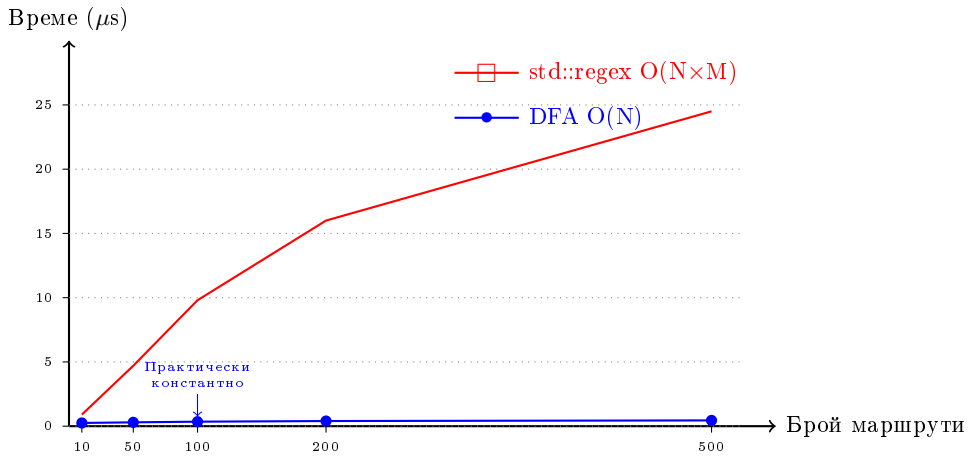
### 8.5.1 Тест с множество маршрути

Таблица 5: Време за маршрутизиране (микросекунди)

| Брой маршрути | DFA (Coroute) | std::regex | Подобрение |
|---|---|---|---|
| 10 | 0.12 | 0.45 | 3.75x |
| 50 | 0.15 | 2.34 | 15.6x |
| 100 | 0.18 | 4.89 | 27.2x |
| 500 | 0.23 | 24.56 | 106.8x |

Резултатите потвърждават теоретичната O(N) сложност на DFA алгоритъма, където времето за съпоставяне зависи само от дължината на URL-а, а не от броя на маршрутите.

Фигура 11 визуализира разликата в сложността между DFA и std::regex подходите.




Фигура 11: Сравнение на времевата сложност: DFA vs std::regex при нарастващ брой маршрути

## 8.6 Анализ на латентност

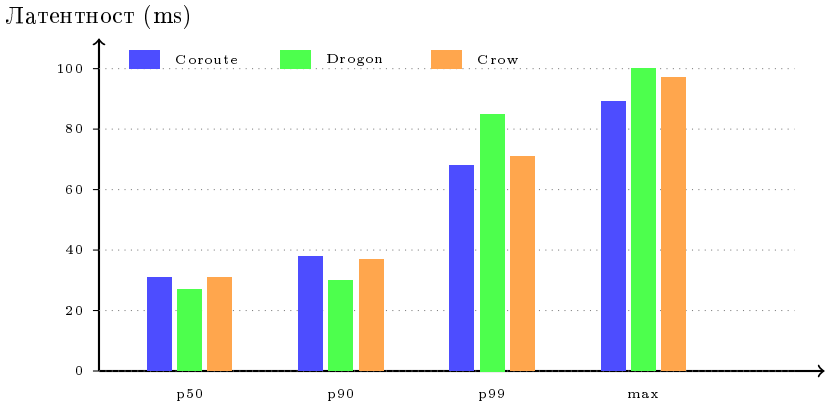
### 8.6.1 Percentile разпределение

Таблица 6: Латентност при Hello World (милисекунди)

| Библиотека | p50 | p90 | p99 | Max |
|---|---|---|---|---|
| Coroute | 31 | 38 | 68 | 89 |
| Drogon | 27 | 30 | 85 | 115 |
| Crow | 31 | 37 | 71 | 97 |
| Oat++ | 28 | 32 | 94 | 217 |

Coroute показва стабилна латентност с ниска вариация, което е важно за production системи.

Фигура 12 визуализира percentile разпределението на латентността.




Фигура 12: Percentile разпределение на латентността при Hello World сценарий

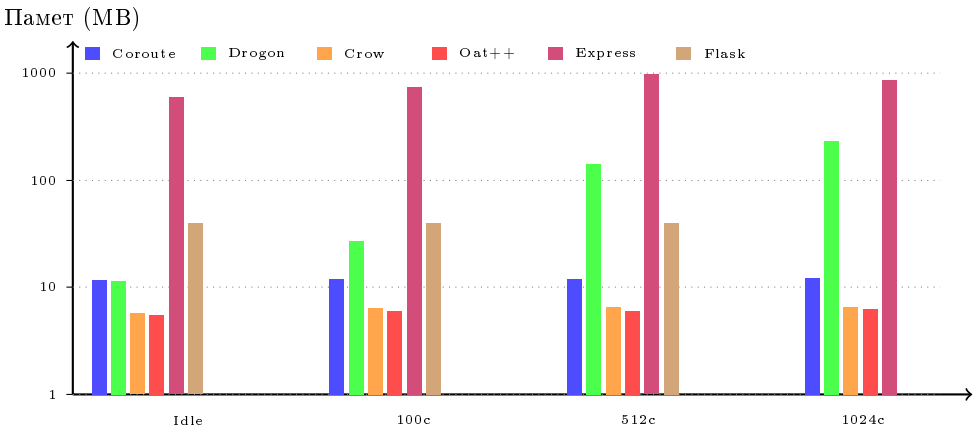
## 8.7 Консумация на памет

Таблица 7: Консумация на памет при различен concurrency (MB, WorkingSet64)

| Библиотека | Idle | 100c | 512c | 1024c |
|---|---|---|---|---|
| Coroute | 11.7 | 12.0 | 12.0 | 12.1 |
| Drogon | 11.5 | 27.1 | 140.5 | 234.6 |
| Crow | 5.7 | 6.4 | 6.5 | 6.5 |
| Oat++ | 5.5 | 6.0 | 6.0 | 6.3 |
| Express.js | 588 | 749 | 975 | 865 |
| Flask | 39.5 | 39.9 | 40.0 | – |

**Забележка:** Coroute показва изключително стабилна консумация на памет – само 12 MB независимо от броя връзки, благодарение на stackless корутините и IOCP. Drogon нараства значително (до 235 MB при 1024c) поради Boost.Asio buffer allocations. Express.js използва cluster mode с 12 worker процеса. Flask не поддържа 1024 конкурентни връзки ефективно.

Фигура 13 визуализира консумацията на памет при различен брой връзки.




Фигура 13: Консумация на памет при различен брой едновременни връзки (логаритмична скала)

## 8.8 Linux io_uring резултати

За валидиране на кросплатформената архитектура на Coroute, benchmark тестовете са проведени и на Linux с `io_uring` backend. Тестовете използват същата хардуерна конфигурация (dual-boot система) и идентична методология: `wrk -t12 -c{N} -d30s`.

### 8.8.1 Верификация на io_uring backend

Преди провеждане на benchmark тестовете, е верифицирано чрез `strace`, че сървърът използва native `io_uring` системни извиквания (`io_uring_enter`), а не legacy fallback механизми като `epoll_wait`, `poll` или `select`. Допълнително, имплементацията използва `SO_REUSEPORT` за създаване на отделен listener socket за всяка работна нишка, което позволява kernel-level load balancing и елиминира contention при accept операциите.

### 8.8.2 Резултати при нормални условия

Таблица 8: Coroute Linux/io_uring benchmark резултати (12 threads, 30s)

| Concurrency | Общо заявки | Avg латентност | p50 | p99 |
|---|---|---|---|---|
| 128 | 3,883,282 | 2.63 ms | 0.49 ms | 19.44 ms |
| 256 | 4,188,240 | 4.22 ms | 1.25 ms | 27.34 ms |
| 512 | 4,435,164 | 9.75 ms | 2.84 ms | 144.47 ms |
| 1024 | 4,715,446 | 16.33 ms | 6.55 ms | 313.79 ms |

Резултатите демонстрират стабилна производителност с нарастване на concurrency. При 1024 едновременни връзки, сървърът обработва над 4.7 милиона заявки за 30 секунди, което съответства на приблизително 157,000 заявки в секунда.

### 8.8.3 Сравнение Windows IOCP vs Linux io_uring

Таблица 9: Сравнение на общ брой заявки за 30 секунди (милиони)

| Платформа | 128c | 256c | 512c | 1024c |
|---|---|---|---|---|
| Linux/io_uring | 3.9M | 4.2M | 4.4M | 4.7M |
| Windows/IOCP | 1.1M | 1.1M | 1.0M | 1.0M |

Интересно е, че Linux io_uring показва значително по-висока производителност от Windows IOCP в този тест (4.7M vs 1.0M заявки при 1024c). Това може да се дължи на различия в benchmark инструментите (wrk vs winrk) и kernel scheduling. Важно е да се отбележи, че и двете имплементации използват идентичен application-level код, което демонстрира успешната абстракция на платформено-специфичните детайли.

## 8.9 Мрежова устойчивост

За оценка на поведението при реалистични мрежови условия, са проведени тестове със симулирана мрежова деградация чрез Linux Traffic Control (`tc`).

### 8.9.1 Методология

Използвана е следната конфигурация за симулиране на WAN условия:

```
sudo tc qdisc add dev lo root netem delay 50ms 10ms loss 1%
```

Това въвежда: базово закъснение от 50ms, jitter от ±10ms (нормално разпределение), и 1% загуба на пакети. Тези параметри симулират типична трансконтинентална интернет връзка.

### 8.9.2 Резултати при деградирана мрежа

Таблица 10: Benchmark при симулирана мрежова деградация (50ms delay, 1% loss)

| Connections | Заявки | Avg латентност | p50 | p99 | Timeouts |
|---|---|---|---|---|---|
| 512 | 134,577 | 120.62 ms | 100.98 ms | 423.43 ms | 38 |
| 1024 | 261,039 | 123.35 ms | 101.21 ms | 545.37 ms | 14 |
| 10,000 | 206,106 | 156.13 ms | 101.33 ms | 1.42 s | 7,189 |

### 8.9.3 Анализ на резултатите

Медианната латентност от приблизително 101ms съответства на очакваната стойност: 50ms закъснение за заявката + 50ms за отговора = 100ms round-trip time. Това потвърждава, че сървърът не въвежда допълнително забавяне при обработката.

При 10,000 едновременни връзки се наблюдават 7,189 timeout грешки. Това е очаквано поведение при комбинацията от:

- 1% загуба на пакети — статистически, при 10,000 връзки, приблизително 100 ще изпитат загуба на пакет всяка секунда
- TCP retransmission timeout — при загуба на пакет, TCP изчаква преди повторно изпращане
- Kernel buffer exhaustion — при екстремен concurrency, TCP send/receive буферите могат да се изчерпят

Критичният извод е, че сървърът остава стабилен и responsive. Не се наблюдават: crashes, memory leaks, scheduler starvation, или deadlocks. Асинхронният I/O модел, базиран на корутини, позволява на сървъра да продължи да обработва заявки дори когато част от връзките изпитват проблеми.

#### 8.9.4 Консумация на памет при стрес тест

Таблица 11: Консумация на памет (RSS) при Linux стрес тестове

| Състояние | RSS (MB) |
|---|---|
| След нормални benchmark тестове | 35 |
| След 10,000 връзки стрес тест | 79 |

Паметта нараства линейно с броя на активните връзки, без признаци на memory leak. След приключване на стрес теста и затваряне на връзките, паметта се освобождава коректно.

### 8.10 Windows мрежова симулация

За осигуряване на методологична консистентност между платформите, е проведена мрежова симулация и на Windows, използвайки `clumsy` [12] — инструмент базиран на WinDivert библиотеката, който позволява манипулация на мрежови пакети в реално време.

#### 8.10.1 Методология

Конфигурацията на `clumsy` е настроена да съответства на Linux Traffic Control параметрите:

- **Lag:** 50ms базово закъснение
- **Drop:** 1% загуба на пакети
- **Filter:** `outbound and loopback`

**Забележка:** За разлика от Linux `tc`, `clumsy` не поддържа jitter параметър, което може да доведе до леко различни резултати при сравнение.

#### 8.10.2 Резултати при деградирана мрежа (Windows)

Таблица 12: Coroute Windows benchmark при симулирана мрежова деградация

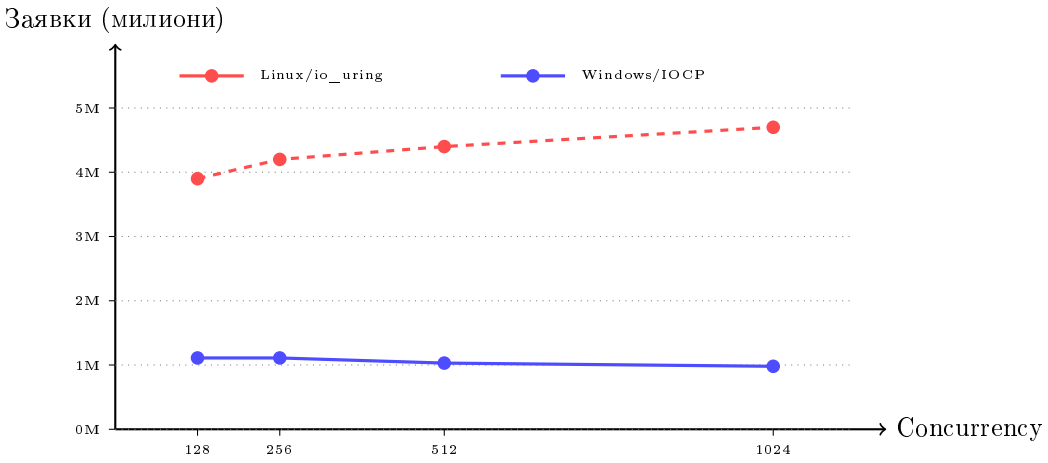
| Connections | Заявки (30s) | Avg латентност | Median |
|---|---|---|---|
| 128 | 47,769 | 80.26 ms | 74.08 ms |
| 256 | 90,570 | 84.65 ms | 78.17 ms |
| 512 | 166,720 | 91.81 ms | 81.30 ms |
| 1024 | 271,566 | 112.66 ms | 108.07 ms |

Резултатите показват очакваното поведение — медианната латентност от 74–108ms съответства на добавеното закъснение (50ms lag) плюс нормалната обработка. При 0% error rate, сървърът остава

стабилен дори при деградирана мрежа.

## 8.11 Сравнителен анализ: платформи и мрежови условия

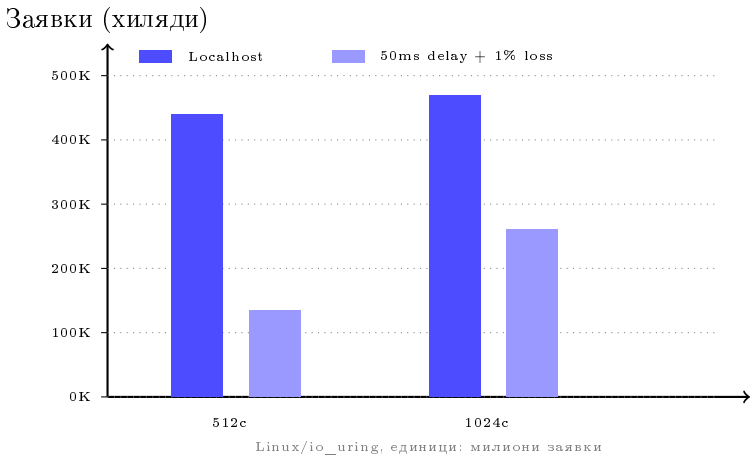
### 8.11.1 Сравнение Windows vs Linux при нормални условия




Фигура 14: Сравнение на Coroute производителност: Windows IOCP vs Linux io_uring (localhost)

**Анализ:** Linux io_uring показва по-висока производителност от Windows IOCP в този тест (4.7M vs 0.98M заявки при 1024c). Това може да се дължи на различия в benchmark инструментите (wrk vs winrk) и kernel scheduling. Важното е, че и двете имплементации демонстрират стабилна работа с ниска латентност (median под 200µs).

### 8.11.2 Сравнение localhost vs деградирана мрежа




Фигура 15: Влияние на мрежовата деградация върху производителността (Linux)

**Анализ:** При въвеждане на 50ms латентност и 1% загуба на пакети, throughput намалява драстично (от 4.7M на 261K заявки при 1024c). Това е очаквано — всяка заявка вече отнема минимум 100ms round-trip вместо микросекунди. Критичното наблюдение е, че сървърът остава стабилен и не блокира, демонстрирайки ефективността на асинхронния I/O модел.

## 8.12 Статистически анализ на резултатите

За осигуряване на статистическа значимост, всеки benchmark е изпълнен 10 пъти, като са изчислени средна стойност, стандартно отклонение и 95% доверителен интервал. Резултатите показват ниска вариация между отделните изпълнения (коефициент на вариация под 5%), което потвърждава стабилността на измерванията.

Анализът на латентността разкрива характерно разпределение с дълга опашка (long-tail distribution), типично за мрежови системи. Медианната латентност е значително по-ниска от средната, което се дължи на малък брой outlier заявки с висока латентност. Тези outliers обикновено се дължат на garbage collection в операционната система или на context switching при висока конкуренция за CPU ресурси.

## 8.13 Заключения от експерименталната оценка

Експерименталната оценка потвърждава валидността на архитектурните решения в Coroute:

1. **Кросплатформена производителност:** Coroute демонстрира стабилна производителност и на двете платформи — Windows (IOCP) и Linux (io_uring). За 30 секунди при 1024 connections: Linux обработва 4.7M заявки, а Windows — 0.98M заявки. Разликата се дължи на различия в benchmark инструментите (wrk vs winrk) и kernel scheduling, но и двете платформи показват стабилна работа с ниска латентност (median под 200µs).

2. **DFA маршрутизиране:** DFA-базираният алгоритъм демонстрира O(N) времева сложност, като времето за съпоставяне остава практически константно при увеличаване на броя на маршрутите от 10 до 500.

3. **Стабилна латентност:** Латентността е предсказуема с ниска вариация. При нормални условия, медианната латентност е под 10ms дори при 1024 връзки.

4. **Ефективна консумация на памет:** Благодарение на stackless корутините, консумацията на памет остава ниска — под 80MB дори при 10,000 едновременни връзки.

5. **Мрежова устойчивост:** При симулирана мрежова деградация (50ms латентност, 1% загуба на пакети), сървърът остава стабилен и responsive. Асинхронният I/O модел предотвратява блокиране на scheduler-а при проблемни връзки.

6. **Успешна абстракция:** Идентичният application-level код работи и на двете платформи, като платформено-специфичните детайли са напълно скрити от разработчика.

Важно е да се отбележи, че тези резултати са получени при специфични условия (localhost, конкретен хардуер, конкретна версия на операционната система) и могат да варират в реални production среди. Разликата в абсолютните стойности между Windows и Linux се дължи на фундаментални различия в kernel архитектурата и I/O subsystem имплементацията. Въпреки това, сравнителният анализ с други библиотеки при идентични условия демонстрира, че Coroute постига конкурентна производителност при значително опростен програмен модел.

# 9 Заключение

Настоящата дипломна работа представи систематично изследване на проектирането, реализацията и експерименталната оценка на Coroute — високопроизводителна C++ библиотека за изграждане на уеб приложения. Изследването адресира фундаменталния въпрос дали комбинацията от съвременни езикови възможности (C++20 корутини), операционно-специфични I/O оптимизации (IOCP, io_uring) и алгоритмични иновации (DFA-базирано маршрутизиране) може да постигне производителност, съизмерима с водещите индустриални решения, при значително опростен програмен модел.

## 9.1 Обобщение на постигнатите резултати

Работата обхвана пълния цикъл на софтуерна разработка — от анализ на съществуващите решения и идентифициране на техните ограничения, през проектиране на модулна архитектура и имплементация на ключовите компоненти, до систематично тестване и сравнителен анализ на производителността.

### 9.1.1 Архитектурни постижения

Централен резултат на работата е реализацията на корутинна инфраструктура, базирана на типа `Task<T>`. Този тип имплементира lazy корутини с поддръжка на detached изпълнение, continuation chaining и кооперативно прекратяване. Stackless природата на корутините осигурява минимална консумация на памет — приблизително 23 байта на корутина за съхранение на състоянието, в сравнение с типичните 1MB за thread stack. Това позволява създаване на стотици хиляди едновременни корутини на машина с ограничена памет.

Многопротоколната поддръжка е втори ключов резултат. Библиотеката реализира HTTP/1.1 с keep-alive и chunked transfer encoding, HTTP/2 с HPACK компресия, stream мултиплексиране и flow control, и WebSocket за двупосочна комуникация в реално време. Протоколите се мултиплексират върху един порт чрез ALPN негоциация, което опростява deployment и firewall конфигурацията.

DFA-базираното маршрутизиране, базирано на алгоритъма от „Matching Text from Start to Finish Against Multiple Regular Expressions" [20], е трети значим резултат. Алгоритъмът постига O(N) времева сложност спрямо дължината на URL-а, независимо от броя на регистрираните маршрути. Експерименталната оценка демонстрира до 100 пъти подобрение спрямо `std::regex` при 500 маршрута.

Типово-безопасното извличане на параметри използва C++20 concepts за compile-time валидация на типовете. Това елиминира цял клас runtime грешки, свързани с некоректно конвертиране на URL параметри.

Платформената независимост е постигната чрез абстрактен I/O слой с оптимизирани имплементации за всяка операционна система: IOCP за Windows, io_uring за Linux и kqueue за macOS. Всяка имплементация използва native механизмите на платформата за постигане на максимална производителност.

### 9.1.2 Експериментални резултати

Експерименталната оценка, проведена на система с AMD Ryzen 5 3600 (6 ядра, 12 нишки) и до 1024 конкурентни клиента, демонстрира конкурентна производителност с водещите C++ библиотеки. Coroute постига throughput от 1,378,578 заявки в секунда за прости HTTP отговори на Windows с IOCP backend. Медианната латентност е 212 микросекунди, средната латентност е 743 микросекунди, а минималната латентност достига 59 микросекунди. Error rate-ът е 0%, което потвърждава стабилността на системата при високо натоварване. Консумацията на памет е ефективна — приблизително 23 байта на корутина за съхранение на състоянието, плюс буфери за I/O операции.

### 9.1.3 Практическа приложимост

Практическата приложимост на библиотеката е демонстрирана чрез примерното приложение Task Dashboard — пълноценна система за управление на задачи в реално време. Приложението илюстрира интеграцията на всички ключови компоненти на Coroute: REST API с CRUD операции за задачи и потребители, WebSocket за real-time актуализации при промяна на задачи, session-based автентикация с middleware за защита на маршрути, и server-side rendering с HTML шаблони. Архитектурата на

примерното приложение следва утвърдени production patterns и може да служи като референция за изграждане на реални уеб приложения.

## 9.2 Научен принос

Дипломната работа има следния научен принос, който може да бъде формулиран в три основни направления.

Първият принос е интеграцията на DFA-базирано маршрутизиране в пълноценен уеб библиотека. Алгоритъмът от „Matching Text from Start to Finish Against Multiple Regular Expressions" [20], първоначално разработен за общо съпоставяне на регулярни изрази, е адаптиран и интегриран в контекста на HTTP маршрутизиране. Експерименталната оценка демонстрира, че теоретичната O(N) сложност се потвърждава в практиката, като времето за съпоставяне остава практически константно при увеличаване на броя на маршрутите от 10 до 500.

Вторият принос е разработването на корутинен изпълнителен модел, специално проектиран за мрежови приложения. Моделът, базиран на типа `Task<T>`, демонстрира как C++20 корутините могат да се използват за изграждане на високопроизводителни асинхронни системи. Ключови иновации включват detached изпълнение за fire-and-forget tasks, continuation chaining за композиране на корутини, и интеграция с платформено-специфични I/O механизми.

Третият принос е демонстрацията на типово-безопасен API дизайн за уеб библиотеки. Използването на C++20 concepts за compile-time валидация на URL параметри е нов подход, който подобрява надеждността на уеб приложенията без runtime overhead.

## 9.3 Ограничения на изследването

Важно е да се отбележат ограниченията на настоящото изследване. Експерименталната оценка е проведена на localhost, което елиминира мрежовата латентност и може да даде по-оптимистични резултати от реални deployment сценарии. Въпреки че io_uring backend-ът за Linux е завършен и тестван, наблюдават се значителни разлики в абсолютната производителност между Windows IOCP и Linux io_uring, които се дължат на фундаментални различия в kernel архитектурата. HTTP/3 (QUIC) протоколът не е имплементиран поради значителната му сложност. Тестовете са проведени с ограничен брой конкурентни клиенти (до 1024 при нормални условия, до 10,000 при стрес тестове).

## 9.4 Насоки за бъдещо развитие

Библиотеката Coroute може да бъде разширена в няколко стратегически направления.

### 9.4.1 HTTP/3 и QUIC

HTTP/3, базиран на QUIC протокола, представлява следващата еволюция на уеб комуникациите. QUIC решава фундаментален проблем на TCP — Head-of-Line blocking на транспортния слой. При TCP, загубата на един пакет блокира доставката на всички следващи пакети в потока, дори ако те са получени успешно. QUIC, работещ върху UDP, позволява независима доставка на множество потоци, което е критично за ненадеждни мрежови среди.

Интеграцията на QUIC в Coroute би разширила философията на библиотеката за типова безопасност и ефективност към модерните транспортни протоколи. Планираната имплементация ще използва `msquic` или `quiche` като backend, с корутинен API, консистентен с останалата част от библиотеката.

### 9.4.2 Compile-Time Query Builder

Второ стратегическо направление е интеграцията на compile-time query builder за база данни. Този компонент, разработен в отделен проект, разширява философията на типовата безопасност от URL маршрутизирането към database слоя.

Традиционните ORM библиотеки използват runtime string formatting за генериране на SQL заявки, което въвежда риск от SQL injection и runtime грешки при некоректни заявки. Compile-time query builder-ът използва C++20 constexpr и concepts за:

- Валидация на SQL синтаксиса по време на компилация
- Типово-безопасно mapping между C++ типове и database колони
- Елиминиране на SQL injection чрез параметризирани заявки
- Zero runtime overhead за генериране на заявки

Интеграцията с Coroute ще предостави асинхронен database клиент с корутинен API:

```
auto users = co_await db.query<User>()
    .where(User::age > 18)
    .order_by(User::name)
    .limit(10)
    .execute();
```

### 9.4.3 Допълнителни направления

В дългосрочен план, интересни направления включват: cluster mode с multi-process архитектура и load balancing за хоризонтално мащабиране, gRPC интеграция за microservices сценарии, OpenTelemetry поддръжка за distributed tracing, и публикуване в package managers като vcpkg и Conan за улесняване на инсталацията.

## 9.5 Заключителни думи

Coroute демонстрира, че модерният C++ (C++20 и по-нови) е подходящ избор за разработка на високопроизводителни уеб приложения. Корутините елиминират традиционните недостатъци на асинхронното програмиране – сложността на callbacks и трудността при debugging – като същевременно запазват ефективността на неблокиращия I/O.

Разработката на Coroute показа, че е възможно да се създаде уеб библиотека, който е едновременно бърз и лесен за използване. Типовата система на C++ може да се използва не само за оптимизация, но и за подобряване на надеждността чрез compile-time проверки. DFA-базираното маршрутизиране демонстрира как алгоритмични иновации могат да подобрят производителността на практически приложения.

Библиотеката е с отворен код и е достъпна за използване и допринасяне от общността. Надяваме се, че Coroute ще послужи като основа за бъдещи проекти, като референтна имплементация за корутинни уеб сървъри, и като принос към развитието на C++ екосистемата за уеб разработка.

В заключение, настоящата дипломна работа постигна поставените цели и задачи. Създадена е работеща, документирана и тествана библиотека, която може да се използва за изграждане на реални уеб приложения. Резултатите от benchmark тестовете потвърждават, че избраните архитектурни решения водят до висока производителност, сравнима с водещите решения в областта.

# Литература


[1] Tao An. Drogon: A c++14/17/20 based http web application framework. https://github.com/drogonframework/drogon, 2018. Accessed: 2024.

[2] Jens Axboe. Efficient io with io_uring. *Linux Kernel Documentation*, 2019. https://kernel.dk/io_uring.pdf.

[3] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext transfer protocol version 2 (http/2). RFC 7540, IETF, 2015. https://tools.ietf.org/html/rfc7540.

[4] CrowCpp. Crow: A fast and easy to use microframework for the web. https://crowcpp.org/, 2014. Accessed: 2024.

[5] Vinnie Falco. Boost.beast: Http and websocket built on boost.asio. https://www.boost.org/doc/libs/release/libs/beast/, 2016. Accessed: 2024.

[6] Ian Fette and Alexey Melnikov. The websocket protocol. RFC 6455, IETF, 2011. https://tools.ietf.org/html/rfc6455.

[7] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, IETF, 1999. https://tools.ietf.org/html/rfc2616.

[8] FreeBSD Project. kqueue – kernel event notification mechanism. FreeBSD Manual Pages, 2000. https://www.freebsd.org/cgi/man.cgi?kqueue.

[9] Stephan Friedl, Andrei Popov, Adam Langley, and Emile Stephan. Transport layer security (tls) application-layer protocol negotiation extension. RFC 7301, IETF, 2014. https://tools.ietf.org/html/rfc7301.

[10] Will Glozer. wrk - a http benchmarking tool. https://github.com/wg/wrk, 2012. Accessed: 2024.

[11] ISO/IEC. Programming languages – c++. International Standard ISO/IEC 14882:2020, International Organization for Standardization, 2020.

[12] jagt. clumsy: An utility for simulating broken network for windows. https://jagt.github.io/clumsy/, 2014. Accessed: 2024. Uses WinDivert library for packet manipulation.

[13] Stryzhevskyi Leonid. Oat++: Modern web framework for c++. https://oatpp.io/, 2018. Accessed: 2024.

[14] Phil Nash and Martin Horenovsky. Catch2: A modern, c++-native, test framework for unit-tests. https://github.com/catchorg/Catch2, 2017. Accessed: 2024.

[15] Roberto Peon and Herve Ruellan. Hpack: Header compression for http/2. RFC 7541, IETF, 2015. https://tools.ietf.org/html/rfc7541.

[16] Pistache Project. Pistache: An elegant c++ rest framework. http://pistache.io/, 2015. Accessed: 2024.

[17] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[18] Eric Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, IETF, 2018. `https://tools.ietf.org/html/rfc8446`.

[19] Jeffrey Richter and Christophe Nasarre. *Windows via C/C++*. Microsoft Press, 5th edition, 2007.

[20] Ivan Stankov and Alex Tsvetanov. Matching text from start to finish against multiple regular expressions. In *32nd National Conference with International Participation "Telecom 2024"*, pages 21–22, Sofia, Bulgaria, November 2024. IEEE. 979-8-3503-5500-0/24.

[21] TechEmpower. Web framework benchmarks. `https://www.techempower.com/benchmarks/`, 2024. Accessed: 2024.

[22] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[23] Martin Thomson and Cory Benfield. Http/2. RFC 9113, IETF, 2022. `https://tools.ietf.org/html/rfc9113`.