Думбов Данил Сергеевич, бакалавр
Уфимский университет науки и технологий
Dumbov Danil Sergeevich
Ufa University of Science and Technology

СРАВНИТЕЛЬНЫЙ АНАЛИЗ СИНХРОННЫХ И АСИНХРОННЫХ МОДЕЛЕЙ ОБРАБОТКИ ЗАПРОСОВ В СЕРВЕРНЫХ ПРИЛОЖЕНИЯХ
COMPARATIVE ANALYSIS OF SYNCHRONOUS AND ASYNCHRONOUS REQUEST PROCESSING MODELS IN SERVER APPLICATIONS

Аннотация. Проведён сравнительный анализ синхронной и асинхронной моделей обработки запросов в серверных приложениях. Рассмотрены особенности блокирующего и неблокирующего ввода-вывода, цикл событий, корутины, использование потоков и процессов. Показано влияние характера нагрузки на задержку, пропускную способность и масштабируемость серверной системы. Сформулированы критерии выбора модели для практических веб-сервисов.

Abstract. Synchronous and asynchronous request processing models in server applications are comparatively analyzed. Blocking and non-blocking I/O, event loops, coroutines, threads, and processes are considered. The influence of workload characteristics on latency, throughput, and scalability is discussed. Criteria for selecting a processing model for practical web services are formulated.

Ключевые слова: Серверные приложения, асинхронность, синхронная обработка, event loop, неблокирующий ввод-вывод, масштабируемость.

Keywords: Server applications, asynchronous processing, synchronous processing, event loop, non-blocking I/O, scalability.

Введение

Современное серверное приложение одновременно обслуживает множество соединений, обращается к базам данных, внешним API, файловым системам и сетевым сервисам. Время выполнения запроса поэтому определяется не только вычислениями процессора, но и ожиданием операций ввода-вывода. При росте числа клиентов способ организации ожидания становится одним из факторов, определяющих масштабируемость. Исторически веб-серверы развивались от схемы «процесс или поток на соединение» к событийным архитектурам, позволяющим одному потоку координировать большое число сетевых операций [1, 2].

В общем виде задержку ответа можно представить как $T = Tq + Tcpu + Tio + Ts$, где Tq – ожидание в очереди, $Tcpu$ – вычислительная часть, Tio – ожидание внешнего ввода-вывода, Ts – сериализация и передача результата. Синхронная и асинхронная модели отличаются прежде всего тем, что происходит с исполнительным контекстом в течение Tio. Цель работы – сопоставить эти модели и определить условия, при которых каждая из них обеспечивает рациональное использование ресурсов.

Синхронная модель обработки запросов

В синхронной модели операция обычно выполняется последовательно: обработчик получает запрос, выполняет необходимые действия и возвращает ответ. Если вызов базы данных или сетевого ресурса является блокирующим, текущий поток не выполняет полезную работу до завершения операции. Простота такого исполнения является важным преимуществом. Последовательный код легче читать, отлаживать и профилировать, а локальное состояние запроса естественно связано с одним стеком вызовов.

Для одновременного обслуживания клиентов синхронные серверы используют несколько процессов или потоков. Операционная система переключает их, пока часть исполнителей ожидает ввод-вывод. Такой подход хорошо работает при умеренной

конкурентности и позволяет задействовать несколько процессорных ядер. Однако каждый поток имеет собственный стек и связан с затратами на планирование. При очень большом количестве одновременно ожидающих соединений растут расходы памяти и переключений контекста. Анализ архитектур высокопроизводительных веб-серверов показывает, что стоимость управления большим числом исполнителей становится существенной при масштабировании [2].

Синхронность не следует отождествлять с низкой производительностью. Если запрос содержит значительный объём CPU-вычислений, отдельный процесс или пул процессов может быть естественным решением. Кроме того, многие библиотеки и драйверы имеют зрелые блокирующие интерфейсы. Поэтому переход к асинхронности оправдан не сам по себе, а только при наличии значительной доли ожидания и высокой конкурентности.

Асинхронная модель и цикл событий

Асинхронный сервер старается не удерживать исполнительный поток во время ожидания сетевой операции. Вместо блокировки задача передаёт управление планировщику и возобновляется после появления события готовности. В Python эта модель формализована в asyncio и основана на цикле событий, задачах и корутинах [3]. Аналогичный событийный принцип используется библиотекой libuv: сетевые дескрипторы работают в неблокирующем режиме, а готовность ввода-вывода отслеживается средствами операционной системы [6].

Корутина сохраняет своё состояние, но не требует отдельного системного потока на каждую операцию. Когда выполняется await для неблокирующего вызова, цикл событий может продолжить другие задачи. Это особенно эффективно для серверов, которые большую часть времени ждут сеть, базу данных или внешний API. Стандарт ASGI закрепляет асинхронный интерфейс между Python-приложением и сервером и позволяет единообразно обслуживать HTTP, WebSocket и другие длительные соединения [4].

Основное ограничение событийной модели заключается в необходимости кооперативного поведения задач. Если внутри обработчика выполняется длительная блокирующая функция или тяжёлый вычислительный цикл, управление не возвращается event loop и задержка остальных запросов увеличивается. Поэтому CPU-bound работу необходимо выносить в процессы, специализированные workers или отдельные сервисы. Асинхронность уменьшает простои при ожидании, но не создаёт дополнительные вычислительные ресурсы.

Сравнение масштабируемости и задержки

Для I/O-bound нагрузки асинхронная модель позволяет поддерживать большое число одновременно ожидающих операций при сравнительно небольшом числе системных потоков. Это снижает накладные расходы на контекст и делает использование памяти более предсказуемым. Концепция событийной обработки при высокой конкурентности развивалась в архитектурах SEDA и Flash, где внимание уделялось управлению очередями, стадиями и ресурсами под нагрузкой [1, 2].

При этом пропускная способность определяется не только моделью конкурентности. Узким местом может стать база данных, пул соединений, сериализация, внешний API или CPU. Если сервер принимает запросы быстрее, чем зависимый ресурс способен их обрабатывать, возникает очередь и растёт Tq. Асинхронный код способен даже ускорить накопление такой очереди, поскольку дешёво создаёт большое число ожидающих задач. Поэтому необходимы лимиты конкурентности, тайм-ауты и обратное давление.

Для CPU-bound задач преимущество event loop уменьшается. Длительное вычисление занимает поток цикла событий и блокирует продвижение других корутин. Здесь рациональны multiprocessing, отдельные вычислительные workers или горизонтальное масштабирование. На практике эффективная архитектура часто является гибридной: сетевой слой обрабатывается асинхронно, а тяжёлые операции передаются в ограниченный пул исполнителей.

Критерии выбора модели

Выбор следует начинать с профиля нагрузки. Если обработчик выполняет короткие вычисления и часто ожидает внешние ресурсы, асинхронная модель обеспечивает высокую конкурентность без линейного роста числа потоков. Типичные примеры – API-шлюзы, прокси, сервисы агрегации данных, WebSocket-соединения и приложения с большим числом параллельных сетевых запросов. Если же основное время тратится на вычисления, задача требует параллелизма по ядрам, а не только неблокирующего ожидания.

Вторым критерием является экосистема зависимостей. Полезный эффект исчезает, если асинхронный обработчик регулярно вызывает блокирующий драйвер. База данных, HTTP-клиент и другие критические библиотеки должны поддерживать неблокирующий интерфейс или выполняться вне event loop. Третьим критерием является сложность сопровождения: ошибки тайм-аутов, отмены задач, ограничений параллелизма и управления соединениями должны рассматриваться как часть архитектуры, а не как детали реализации.

Таким образом, корректная постановка вопроса состоит не в выборе «синхронно или асинхронно» для всего приложения, а в разделении типов работы. Сетевое ожидание целесообразно мультиплексировать, CPU-нагрузку – ограниченно параллелить, а доступ к дефицитным ресурсам – защищать семафорами, пулами и очередями. Такая декомпозиция позволяет получать преимущества событийной модели без перегрузки зависимых подсистем.

Заключение

Синхронная обработка остаётся рациональной благодаря простоте, предсказуемому исполнению и удобству работы с вычислительными задачами. Асинхронная модель эффективна при высокой конкурентности и преобладании I/O-bound операций, поскольку позволяет использовать время ожидания для продвижения других запросов. Её основой являются неблокирующий ввод-вывод, цикл событий и кооперативное переключение задач [3, 6].

Максимальная эффективность достигается при согласовании модели конкурентности с характером нагрузки. Для современных серверных систем практичным является гибридный подход: асинхронный сетевой контур, ограниченные пулы для блокирующих или CPU-bound операций, контроль очередей и тайм-аутов. Такой подход повышает масштабируемость и одновременно сохраняет управляемость архитектуры.

Список литературы:

1. Welsh M., Culler D., Brewer E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services // ACM SIGOPS Operating Systems Review. 2001. Vol. 35, No. 5. P. 230–243.

2. Pai V. S., Druschel P., Zwaenepoel W. Flash: An Efficient and Portable Web Server // Proceedings of the 1999 USENIX Annual Technical Conference. 1999. P. 199–212.

3. van Rossum G. PEP 3156 – Asynchronous IO Support Rebooted: the asyncio Module // Python Enhancement Proposals. 2012.

4. ASGI Documentation. ASGI (Asynchronous Server Gateway Interface) Specification. Version 3.0. 2019.

5. Python Software Foundation. asyncio – Asynchronous I/O // Python Documentation.

6. libuv Project. Design Overview // libuv Documentation