

Георгий Димитриади
**Руководство по созданию
высоконагруженных
систем**



Георгий Димитриади

Руководство по созданию

высоконагруженных систем

<https://litres.ru/74604414>

SelfPub; 2026

Аннотация

Это фундаментальное руководство для инженеров, посвященное проектированию отказоустойчивых ИТ-решений.

В издании последовательно излагаются фундаментальные законы распределенных систем (теоремы CAP и PACELC), принципы обеспечения надежности через дублирование и резервирование, а также методы проведения аварийных учений. Особое внимание уделено неизбежным компромиссам между скоростью, согласованностью данных и стоимостью. Подробно рассмотрены типовые архитектуры — от монолита до микросервисов и Serverless — с анализом их достоинств, недостатков и границ применимости.

Книга учит оценивать совокупную стоимость владения (ТСО) и избегать архитектурных «костылей», закладывая фундамент для создания систем, способных масштабироваться без деградации.

Содержание

Краткое содержание	4
Введение	13
Часть 1. Фундаментальные основы и архитектурные компромиссы	18
Глава 1. Фундаментальные законы распределенных систем	18
Глава 2. Треугольник компромиссов: Скорость, Надежность, Консистентность	25
Конец ознакомительного фрагмента.	31

Георгий Димитриади

Руководство по созданию

высоконагруженных систем

Краткое содержание

Введение

Определение высоконагруженной системы (High-Load). Количественные и качественные критерии: количество запросов в секунду (RPS), объем обрабатываемых данных, количество одновременных соединений.

Целевая аудитория: начинающие инженеры, системные администраторы, студенты технических специальностей.

Структура книги и логика изложения: от фундаментальных ограничений к прикладным паттернам и экономике.

Часть 1. Фундаментальные основы и архитектурные компромиссы

Глава 1. Фундаментальные законы распределенных систем

Теорема CAP (Consistency, Availability, Partition tolerance): Определение согласованности (Consistency), доступности (Availability) и устойчивости к разделению (Partition tolerance). Невозможность одновременного обеспе-

чения всех трех свойств при сетевом разделении.

Теорема PACELC: Расширение CAP. Компромисс между задержкой (Latency) и согласованностью (Consistency) в отсутствие сбоев.

FLP Impossibility (теорема Фишера, Линча и Патерсона): Невозможность достижения консенсуса в асинхронной системе при наличии хотя бы одного отказавшего узла.

Закон Амдала и универсальное уравнение масштабирования: Пределы ускорения системы при распараллеливании задач.

Глава 2. Треугольник компромиссов: Скорость, Надежность, Консистентность

Скорость (Latency vs Throughput): Задержка на один запрос и пропускная способность системы. Влияние очередей на время отклика (теория массового обслуживания).

Надежность (Reliability) и Доступность (Availability): Формальные определения. Коэффициент доступности (например, «три девятки» - 99.9%). Среднее время наработки на отказ (MTBF) и среднее время восстановления (MTTR).

Консистентность данных: Сильная (Strong), причинная (Causal) и конечная (Eventual) согласованность. Проблема чтения собственных записей (Read-Your-Writes).

Практические примеры:

Банковская транзакция: Приоритет - сильная консистентность. Компромисс - увеличение задержки.

Лента социальной сети: Приоритет - низкая задержка и

высокая доступность. Компромисс - конечная согласованность (появление лайка через несколько секунд).

Цена (Cost) как четвертая переменная: Влияние каждого архитектурного решения на капитальные (CAPEX) и операционные (OPEX) затраты.

Глава 3. Цена неверного выбора: Архитектурная деградация и технический долг

Стоимость изменения архитектуры: Экспоненциальный рост сложности при попытке изменить фундаментальный паттерн на поздних этапах (закон Лемана о непрерывной эволюции программ).

«Костыли» (Workarounds) и их влияние на ТСО:
Примеры неверных решений:

Использование реляционной СУБД для хранения сессий (Key-Value) без шардирования. Результат: блокировка таблиц, деградация производительности всей системы.

Применение синхронных межсервисных вызовов в цепочке из десяти компонентов. Результат: каскадные сбои, мультипликация задержки.

Отсутствие идиомпотентности в API. Результат: дублирование платежей при сетевых повторках.

Рефакторинг архитектуры: Стратегии безопасного изменения фундамента (Strangler Fig Pattern, параллельный запуск).

Часть 2. Обеспечение надежности и отказоустойчивости

Глава 4. Дублирование, избыточность и модели резервирования

N+1, N+2, 2N, 2N+1: Модели избыточности на уровне аппаратного и программного обеспечения. Расчет вероятности отказа системы.

Холодный резерв (Cold Standby): Инфраструктура развернута, но не запущена. Время восстановления (RTO) - часы. Стоимость хранения - низкая. Пример: резервный сервер базы данных, требующий ручного поднятия из бэкапа.

Теплый резерв (Warm Standby): Инфраструктура запущена, данные реплицируются с задержкой. RTO - десятки минут. Пример: Slave-реплика БД, на которую нужно переключить трафик приложения.

Горячий резерв (Hot Standby): Active-Passive или Active-Active конфигурации с синхронной репликацией. RTO - секунды. Автоматическое переключение (failover). Стоимость - двукратная и выше.

Глава 5. Аварийные учения и культура надежности (SRE-подход)

Error Budget (Бюджет ошибок): Допустимое время простоя, вытекающее из SLA (Service Level Agreement).

Chaos Engineering (Хаос-инжиниринг): Принудительное внесение сбоев в работающую систему для проверки гипотез о ее устойчивости. Инструментарий (например, имитация отказа сетевого узла или задержки диска).

GameDay (Аварийные учения): Сценарное моделиро-

вание отказов (пожар в дата-центре, деградация провайдера CDN). Роли участников дежурной смены. Протоколы коммуникации во время инцидента.

Post-mortem (Разбор полетов): Структура документа. Культура поиска системных причин (blameless culture), а не виновных.

Глава 6. Организация дежурной смены (On-call)

Уровни эскалации: Первичная линия (L1), инженеры сопровождения (L2), разработчики и архитекторы (L3).

Ротация и предотвращение выгорания: Графики дежурств (например, 12/24/72), компенсационные механизмы.

Runbooks (Инструкции по эксплуатации): Формализация действий при типовых сбоях. Требование к исполнимости инструкции без привлечения автора.

Мониторинг и алертинг: Signal-to-noise ratio. Пороги срабатывания (thresholds) и динамические алерты (anomaly detection) для минимизации ложных срабатываний.

Часть 3. Типовые архитектуры высоконагруженных систем

Глава 7. Монолитная архитектура (Monolith)

Устройство: Единое адресное пространство, общая база данных, синхронное исполнение в рамках одного процесса.

Достоинства: Простота разработки и отладки на старте, сильная консистентность данных (ACID), низкие накладные расходы на межпроцессное взаимодействие (IPC).

Недостатки: Масштабирование только вертикальное

(Scale-Up), ограничение по ресурсам одной машины, высокая связанность (tight coupling), невозможность независимо деплоя компонентов.

Пример: Информационная система небольшого регионального банка на ранней стадии.

Глава 8. Многоуровневая архитектура (N-Tier)

Устройство: Физическое или логическое разделение на уровни: Presentation (клиент), Business Logic (прикладной слой), Data Access (доступ к данным), Data Storage (хранилище).

Достоинства: Изоляция уровней, возможность независимого масштабирования прикладного слоя и слоя данных.

Недостатки: Риск превращения в «болото» (Big Ball of Mud) при нарушении границ слоев, задержки на сетевых переходах между уровнями.

Глава 9. Сервис-ориентированная архитектура (SOA) и Микросервисная архитектура (Microservices)

Устройство: Декомпозиция на независимые сервисы со своими базами данных. Синхронное (REST, gRPC) и асинхронное (Message Queue) взаимодействие.

Достоинства: Технологическая гетерогенность (каждый сервис на своем стеке), независимые циклы разработки и деплоя, горизонтальное масштабирование (Scale-Out) отдельных функций.

Недостатки: Экспоненциальный рост сложности распределенного взаимодействия, проблемы распределенных

транзакций (Saga, 2PC), сетевая задержка, сложность отладки, дублирование данных.

Пример: Экосистема крупного маркетплейса (отдельные сервисы каталога, корзины, рекомендаций, биллинга).

Глава 10. Событийно-ориентированная архитектура (Event-Driven Architecture) и брокеры сообщений

Устройство: Асинхронное взаимодействие через брокеры (Apache Kafka, RabbitMQ). Продюсеры публикуют события, консьюмеры подписываются на них.

Достоинства: Слабая связанность (decoupling), высокая отказоустойчивость (очередь буферизует всплески нагрузки), возможность переобработки событий при сбоях.

Недостатки: Проблема идемпотентности консьюмеров, сложность мониторинга end-to-end задержек, эффект «протекающего ведра» при неправильной настройке политик удаления старых сообщений.

Глава 11. Serverless-архитектура (FaaS)

Устройство: Исполнение кода в виде функций без управления серверами (AWS Lambda, Yandex Cloud Functions). Оплата за время выполнения.

Достоинства: Нулевое обслуживание инфраструктуры, автоматическое масштабирование до нуля и до пиковых нагрузок.

Недостатки: Проблема «холодного старта» (Cold Start), ограничение по времени выполнения (таймауты), риск привязки к поставщику (vendor lock-in), сложность локальной

разработки.

Часть 4. Экономика архитектуры и эксплуатация

Глава 12. Оценка совокупной стоимости владения (Total Cost of Ownership, TCO)

Прямые затраты (Direct Costs):

CAPEX: Стоимость закупки серверов, сетевого оборудования, лицензий на ПО.

OPEX: Стоимость облачных ресурсов (CPU, RAM, Storage, Egress-трафик), подписок на SaaS.

Косвенные затраты (Indirect Costs):

Стоимость персонала (ФОТ инженеров, дежурных смен, администраторов).

Стоимость простоев (Downtime Cost): упущенная выгода в минуту простоя.

Стоимость изменения (Cost of Change): время, затрачиваемое на рефакторинг «костылей».

Модель расчета: Формула $TCO = CAPEX + (OPEX_{\text{ресурсы}} + OPEX_{\text{персонал}} + \text{Риск_простоев}) * \text{Срок_жизни_системы}$.

Пример расчета: Сравнение TCO монолитного приложения на собственных серверах (On-premise) и микросервисного приложения в публичном облаке на горизонте 5 лет.

Глава 13. Инструментарий и жизненный цикл высоконагруженной системы

Наблюдаемость (Observability) против Мониторинга: Метрики, логи, трейсинг (OpenTelemetry). Понятие «Зо-

лотых сигналов» (Latency, Traffic, Errors, Saturation).

Автоматизация инфраструктуры: Infrastructure as Code (Terraform, Ansible), непрерывная интеграция и доставка (CI/CD).

Жизненный цикл: От проектирования (Design Review) и нагрузочного тестирования до вывода из эксплуатации (Decommissioning).

Заключение

Алгоритм выбора архитектуры для новой системы: от бизнес-требований (SLA, бюджет, скорость вывода на рынок) к техническому решению.

Чек-лист для проверки архитектурного решения на типовые ошибки.

Глоссарий

Сводная таблица определений всех использованных терминов (RPS, CAP, MTTR, Sharding, Idempotency и др.) с отсылками к главам, где они подробно разбираются.

Введение

Ежегодный объем генерируемых человечеством данных измеряется в зеттабайтах, а количество активных устройств, подключенных к сети, исчисляется десятками миллиардов. В этих условиях способность информационной системы корректно функционировать под высокой нагрузкой перестала быть прерогативой технологических гигантов и стала базовой необходимостью для любого цифрового продукта. Банковские транзакции, системы бронирования авиабилетов, государственные порталы услуг и платформы электронной коммерции ежедневно сталкиваются с пиковыми нагрузками, измеряемыми десятками и сотнями тысяч запросов в секунду. Отказ системы в момент коммерческой активности или социальной значимости события влечет за собой не только прямые финансовые убытки, но и необратимую потерю репутации.

Данная книга представляет собой систематическое руководство по проектированию, разработке и эксплуатации высоконагруженных систем. Под высоконагруженной системой (High-Load) в рамках настоящего издания понимается программно-аппаратный комплекс, для которого стандартных методов вертикального масштабирования (увеличения ресурсов одного сервера: тактовой частоты процессора, объема оперативной памяти) недостаточно для удовлетворения

заданных показателей производительности и доступности.

Целевой аудиторией работы являются начинающие специалисты в области информационных технологий: инженеры-программисты, системные администраторы, аналитики и студенты старших курсов технических специальностей. Текст намеренно лишен метафор, бытовых аналогий и упрощений, искажающих физическую или логическую суть процессов. Приоритет отдается формальному, научному стилю изложения, опирающемуся на математический аппарат теории массового обслуживания, теории графов и принципы распределенных вычислений. Вместе с тем, учитывая начальный уровень подготовки читателя, все ключевые термины - от «идемпотентности» до «консенсуса» - вводятся с четкими определениями и формальными критериями их применимости.

Фундаментальный тезис, который будет доказываться на протяжении всей книги, заключается в том, что архитектура - это не диаграмма из стрелочек и квадратиков, а набор жестких компромиссов. Проектирование высоконагруженной системы - это всегда процесс оптимизации в условиях ограниченных ресурсов при наличии конфликтующих требований. Невозможно одновременно максимизировать скорость отклика (минимизировать латентность), гарантировать абсолютную сохранность и согласованность данных (консистентность), обеспечить стопроцентную доступность и при этом минимизировать стоимость владения (TCO).

В основе этих компромиссов лежат фундаментальные законы информатики. Теорема CAP (Consistency, Availability, Partition Tolerance) математически доказывает невозможность одновременного обеспечения согласованности данных, доступности сервиса и устойчивости к сетевому разделению. Теорема PACELC расширяет этот принцип, указывая на неизбежный выбор между задержкой и консистентностью даже в штатном режиме работы системы, когда сбоев не наблюдается. Понимание этих ограничений переводит дискуссию об архитектуре из плоскости личных предпочтений инженера («мне нравится язык Go и брокеры сообщений») в плоскость инженерного расчета, где выбор паттерна диктуется бизнес-требованиями к допустимому времени простоя (SLA) и скорости обработки транзакций.

Особое внимание в книге уделено стоимости неверного архитектурного выбора. Изменение фундамента системы на поздних этапах ее жизненного цикла подчиняется экспоненциальному росту сложности. Попытки компенсировать изначально неверный паттерн (например, использование реляционной базы данных для задач, требующих горизонтального масштабирования в оперативной памяти) с помощью локальных исправлений - так называемых «костылей» - приводят к деградации кодовой базы, кратному увеличению операционных расходов и формированию критического технического долга. Мы рассмотрим конкретные кейсы, демонстрирующие, как архитектурные ошибки на старте увеличи-

вают совокупную стоимость владения (ТСО) в несколько раз на горизонте пяти лет.

Структура издания выстроена по принципу «от абстрактного к конкретному». В первой части будут рассмотрены фундаментальные ограничения распределенных систем и треугольник компромиссов между скоростью, надежностью и консистентностью. Вторая часть посвящена прикладным методам обеспечения отказоустойчивости: моделям дублирования (холодный, теплый и горячий резерв), организации дежурных смен и культуре аварийных учений (Chaos Engineering). Третья часть содержит детальный разбор типовых архитектурных паттернов - от классического монолита до микросервисных и событийно-ориентированных систем, с анализом их достоинств, недостатков и границ применимости. Завершающая, четвертая часть, посвящена экономике архитектуры: методологии расчета совокупной стоимости владения (ТСО), включающей не только стоимость серверных мощностей, но и затраты на персонал, а также финансовые риски, связанные с простоями.

Цель данной книги - предоставить начинающему специалисту не набор готовых рецептов, а систему координат и инженерный инструментарий. Освоение представленного материала позволит принимать обоснованные архитектурные решения, прогнозировать поведение систем под нагрузкой и проектировать решения, способные к эволюционному развитию без необходимости полной переработки при двукрат-

ном или десятикратном росте нагрузки.

Часть 1. Фундаментальные основы и архитектурные компромиссы

Глава 1. Фундаментальные законы распределенных систем

Проектирование высоконагруженных систем начинается не с выбора языка программирования или типа базы данных, а с понимания физических и математических ограничений, накладываемых законами реального мира. Распределенная система - это набор независимых вычислительных узлов (серверов), взаимодействующих друг с другом исключительно путем обмена сообщениями через сеть. Как только логика приложения покидает пределы одного физического сервера, разработчик сталкивается с ненадежностью каналов связи и асинхронностью процессов. Данная глава посвящена трем столпам, на которых строится теория распределенных вычислений: теореме CAP, теореме PACELC и теореме FLP.

1.1. Теорема CAP (Теорема Брюера)

В 2000 году Эрик Брюер сформулировал гипотезу, которая в 2002 году была математически доказана Сетом Гилбертом и Нэнси Линч. Теорема CAP утверждает, что в любой распределенной системе, где узлы обмениваются данными

по сети, невозможно одновременно обеспечить более двух из трех следующих свойств:

Согласованность (Consistency, C): На любом корректно работающем узле системы чтение данных возвращает результат последнего завершённого обновления. Иными словами, все узлы видят одни и те же данные в один и тот же момент времени. Формально это означает линейризуемость операций.

Доступность (Availability, A): Любой корректно работающий запрос к любому корректно работающему узлу системы завершается корректным ответом (успешным выполнением или предсказуемой ошибкой). Система не отказывается от обслуживания запросов.

Устойчивость к разделению (Partition Tolerance, P): Система продолжает функционировать и сохраняет свои свойства (согласованность и/или доступность) даже в условиях сетевого разделения. Сетевое разделение (Partition) - это состояние, при котором группа узлов теряет возможность обмениваться сообщениями с другой группой узлов из-за сбоя коммутаторов, маршрутизаторов или обрыва кабеля, хотя сами узлы остаются работоспособными.

Математическое следствие: Поскольку сбой в сети (Partition) в крупных инфраструктурах является статистической неизбежностью, условие P (Partition Tolerance) в распределённых системах всегда должно быть истинным ($P = \text{true}$). Следовательно, выбор всегда стоит между C и A. Си-

стема может быть либо *CA* (согласованной и доступной, но не устойчивой к разделению - то есть при любом сбое сети она полностью остановится), либо *CP* (согласованной и устойчивой к разделению, но недоступной для записи в момент разделения), либо *AP* (доступной и устойчивой к разделению, но предоставляющей устаревшие данные в момент разделения).

Пример:

CP-система (C + P): Реляционная база данных (например, PostgreSQL) в режиме синхронной репликации. Если мастер-узел теряет связь с репликой, он может заблокировать запись, чтобы не допустить рассинхронизации данных. Согласованность сохраняется, но доступность падает (запросы на запись отклоняются).

AP-система (A + P): Система доменных имен (DNS) или хранилище типа «ключ-значение» (например, Apache Cassandra в типовой конфигурации). Если часть узлов недоступна, система продолжит отвечать на запросы, отдавая те данные, которые есть у доступных узлов. Доступность сохраняется, но данные могут быть неактуальными (нарушение согласованности).

1.2. Теорема PACELC

Теорема CAP описывает поведение системы исключительно в условиях сбоя (наличия разделения). Однако она не дает ответа на вопрос: как ведет себя система в штатном режиме, когда сеть работает стабильно? Для описания этого

компромисса была сформулирована теорема PACELC (разработчик - Дэниел Абади). Ее название представляет собой акроним, описывающий два независимых выбора:

Partition (Разделение): If there is a partition (в случае разделения), как система выбирает между **Availability** (доступностью) и **Consistency** (согласованностью)?

Else (Иначе): In the absence of a partition (в отсутствие разделения), как система выбирает между **Latency** (задержкой) и **Consistency** (согласованностью)?

Теорема утверждает, что даже при идеальной работе сети (P не наступило) разработчик вынужден жертвовать либо скоростью ответа (**Latency**), либо свежестью данных (**Consistency**).

Если система выбирает **C** (согласованность), то перед выполнением операции чтения она обязана убедиться, что все реплики данных синхронизированы. Это требует времени на сетевые запросы и ожидание подтверждений, что увеличивает задержку (**Latency**).

Если система выбирает **L** (минимальную задержку), она отвечает клиенту тем, что лежит в локальной реплике, не дожидаясь подтверждения от остальных узлов. Это обеспечивает мгновенный ответ, но данные могут быть устаревшими.

Пример: Банковская система перевода средств между счетами. Даже в штатном режиме работы сети она выберет **C** (согласованность). Прежде чем списать деньги со счета А, система заблокирует запись и убедится, что все связан-

ные узлы подтверждают состояние счета. Пользователь будет ждать ответа лишние 200–500 миллисекунд (высокая Latency), но баланс всегда будет точным. Лента новостей в социальной сети. Система выберет L (задержку). Запрос пользователя будет направлен к ближайшему серверу, который отдаст наиболее свежие из имеющихся у него данных, не дожидаясь синхронизации со всем глобальным кластером. Пользователь увидит контент мгновенно, но может пропустить лайк, поставленный долей секунды ранее на другом континенте.

1.3. Теорема FLP Impossibility (Невозможность консенсуса)

В 1985 году Майкл Фишер, Нэнси Линч и Майкл Патерсон доказали фундаментальный результат, касающийся достижению согласия (консенсуса) в распределенной системе. Теорема гласит: в чисто асинхронной системе, где нет верхнего предела на время доставки сообщения и нет единых часов, достижение консенсуса невозможно, если хотя бы один узел может отказать (замолчать, уйти в офлайн или работать бесконечно медленно).

Пояснение для начинающего: Консенсус - это процесс, в котором несколько узлов должны договориться о едином значении (например, какой сервер будет мастером, или была ли транзакция зафиксирована). Проблема заключается в неопределенности. Если узел отправил сообщение и долго не получает ответа, он не может понять причину: сообщение

потерялось в сети, получатель его еще не обработал, или получатель навсегда вышел из строя? Ожидание ответа может длиться вечно. Любая попытка установить таймаут (время ожидания) в чисто асинхронной модели является лишь эвристикой, а не строгим доказательством отказа узла.

Практическое следствие: Чтобы обойти это ограничение, все реальные высоконагруженные системы вводят элементы синхронности или частичной синхронности. Это реализуется через:

Сетевые таймауты: Мы соглашаемся с тем, что если ответа нет N миллисекунд, мы считаем узел отказавшим (осознанно идем на риск ошибочного суждения).

Алгоритмы консенсуса с лидером (Leader-based consensus): Использование протоколов вроде Paxos или Raft. Система выбирает один узел (лидера), который временно становится источником истины. Это снижает количество необходимых сетевых взаимодействий для принятия решения и позволяет системе прогрессировать, пока большинство узлов (кворум) доступно.

Резюме

Проектирование архитектуры начинается с фиксации требований бизнеса к параметрам из теорем CAP и PACELC.

Если бизнес требует абсолютной точности финансовых данных, архитектура будет строиться вокруг **C** (Consistency), принося в жертву **A** (Availability) при сбоях и **L** (Latency) в штатном режиме.

Если бизнес требует непрерывности обслуживания миллионов пользователей (например, лента контента), архитектура будет строиться вокруг **A** (Availability) и **L** (Latency), принимая риски временного нарушения согласованности **C** (Consistency).

Понимание теоремы FLP предупреждает инженера о том, что любая распределенная логика, требующая согласия узлов, должна включать в себя механизмы обработки неопределенности, таймауты и кворумы. Игнорирование этих фундаментальных законов на этапе проектирования неизбежно приводит к созданию систем, которые ведут себя непредсказуемо при первой же сетевой задержке или отказе одного сервера.

Глава 2. Треугольник компромиссов: Скорость, Надежность, Консистентность

В предыдущей главе были рассмотрены фундаментальные теоретические ограничения распределенных систем. Настоящая глава переводит эти абстракции в плоскость прикладного проектирования. Любая высоконагруженная система функционирует в условиях дефицита ресурсов и физических ограничений (пропускная способность сети, скорость вращения шпинделей дисков или время доступа к ячейкам памяти). Следовательно, архитектура представляет собой непрерывный процесс поиска локального оптимума на многомерном графике конфликтующих требований. Основными осями этого графика являются скорость (производительность), надежность (доступность) и консистентность (целостность данных). Стоимостные характеристики интегрируются в эту модель как масштабирующие коэффициенты.

2.1. Скорость: Латентность и Пропускная способность

В инженерной практике под скоростью работы системы понимают две независимые метрики: задержку (Latency) и пропускную способность (Throughput).

Латентность (Latency): Временной интервал между моментом инициации запроса клиентом и моментом получения им финального байта ответа. Измеряется в единицах времени (миллисекунды ms, микросекунды mks). Латентность является аддитивной величиной. В распределенной системе общая задержка L_{total} представляет собой сумму задержек на каждом этапе: $L_{total} = L_{network} + L_{queue} + L_{processing} + L_{disk}$ где $L_{network}$ - время передачи пакета по сети, L_{queue} - время ожидания в очереди планировщика, $L_{processing}$ - время работы CPU, L_{disk} - время ввода-вывода.

Пропускная способность (Throughput): Количество единиц работы (запросов, транзакций, мегабайт), которые система способна обработать за единицу времени. Измеряется в запросах в секунду (RPS - Requests Per Second) или битах в секунду (bps).

Компромисс: Стремление к минимальной латентности часто вступает в конфликт с высокой пропускной способностью. Обработка запросов по одному (First-In-First-Out) минимизирует задержку для каждого конкретного пользователя, но неэффективно использует процессорное время на переключение контекста. Группировка запросов (Batching) позволяет накопить 1000 запросов и выполнить одну крупную операцию ввода-вывода, тем самым колоссально увеличивая пропускную способность. Однако каждый из 1000 пользователей будет вынужден ждать, пока накопится пакет,

что кратко увеличивает их индивидуальную латентность.

2.2. Надежность и Доступность

В строгом инженерном смысле понятия «надежность» и «доступность» не являются синонимами.

Надежность (Reliability): Вероятность того, что система будет выполнять свою функцию в течение заданного промежутка времени при определенных условиях. Характеризуется показателем MTBF (Mean Time Between Failures - среднее время наработки на отказ). Надежная система ломается редко.

Доступность (Availability): Доля времени, в течение которого система находится в работоспособном состоянии и способна принимать запросы. Характеризуется коэффициентом доступности A : $A = \text{MTBF} / (\text{MTBF} + \text{MTTR})$, где MTTR (Mean Time To Repair) - среднее время восстановления.

Система может быть высокодоступной, но ненадежной. Пример: скрипт, который перезапускает упавший сервис каждую секунду. MTTR составляет 1 секунду, поэтому доступность стремится к 100%, однако надежность крайне низка, так как сервис постоянно падает (MTBF мал).

Компромисс: Повышение доступности требует введения избыточности (резервирования), что подробно рассматривается в Главе 4. Каждая дополнительная копия сервера или базы данных увеличивает капитальные и операционные затраты, а также сложность синхронизации, но снижает MTTR

и минимизирует единую точку отказа (SPOF - Single Point of Failure).

2.3. Консистентность данных

Консистентность определяет, насколько данные в разных частях системы соответствуют друг другу и реальности.

Сильная консистентность (Strong Consistency): После завершения операции записи все последующие операции чтения во всей системе гарантированно вернут новое значение. Это свойство требует синхронной репликации и блокировок, что напрямую увеличивает латентность.

Конечная консистентность (Eventual Consistency): Если в систему не поступают новые обновления, через некоторое (не гарантированное) время все реплики данных придут к единому значению. В момент сразу после записи чтение может вернуть как старое, так и новое значение. Это позволяет достичь минимальной латентности записи.

Компромисс: Выбор между сильной и конечной консистентностью - это прямая реализация теоремы PACELC. *Пример:* В системе аутентификации (проверка логина и пароля) необходима сильная консистентность. Если пользователь сменил пароль, он должен иметь возможность войти с новым паролем на любом сервере немедленно. Использование кэша с конечной консистентностью здесь недопустимо, так как создаст дыру в безопасности. *Пример:* В системе подсчета просмотров видеоролика допустима конечная консистентность. Если пользователь нажал на кнопку

«лайк», а счетчик обновился через 2 секунды, это не является критическим отказом системы. Это позволяет агрегировать миллионы лайков в памяти и сбрасывать их на диск раз в несколько секунд, обеспечивая колоссальную пропускную способность.

2.4. Практические кейсы столкновения параметров

Кейс 1: Фронтальная система обработки платежей

Требования: Нулевая потеря транзакций, защита от двойного списания.

Выбор: Приоритет - **Консистентность (C)** и **Надежность (Reliability)**.

Реализация: Использование распределенных транзакций (например, паттерн Saga или протокол двухфазного коммита 2PC). Каждая запись подтверждается минимум двумя узлами хранилища.

Цена компромисса: Высокая латентность (пользователь ждет подтверждения 1–2 секунды) и низкая доступность при сбое одного из узлов подтверждения (транзакция отклоняется). Стоимость владения (ТСО) высока из-за необходимости поддержки синхронно работающих кластеров.

Кейс 2: Глобальный сервис микроблогинга (лента новостей)

Требования: Мгновенная публикация контента, поддержка миллионов одновременных сессий.

Выбор: Приоритет - **Латентность (L)** и **Доступность (A)**.

Реализация: Асинхронная запись в очередь (Message Queue), последующая фоновая сборка ленты из кэша (например, в Redis). Используется конечная консистентность.

Цена компромисса: Пользователь видит свой пост мгновенно, но его подписчики могут увидеть его с задержкой до 5–10 секунд. В случае сбоя одного из серверов кэша лента части пользователей может не обновиться, но общая работоспособность системы сохранится.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.