# Архитектура Распределенных Систем Enterprise-Уровня: Теоретический Базис Управления Состоянием и Конкурентностью

## 1. Фундаментальная Трансформация: От Монолитного Игрового Цикла к Распределенной Асинхронности

### 1.1. Онтологический разрыв между Unity и Enterprise.NET

Переход инженера из среды разработки интерактивных приложений реального времени (Unity) в экосистему корпоративных бэкенд-систем (.NET Enterprise) представляет собой не просто смену технологического стека, но глубокую онтологическую трансформацию восприятия времени, пространства памяти и управления состоянием. В парадигме GameDev разработчик оперирует в рамках детерминированного, циклически замкнутого времени, диктуемого кадровой частотой. Метод Update() является метрономом этого мира, гарантирующим, что состояние вселенной игры обновляется последовательно и предсказуемо. В этой модели вся совокупность данных — от координат игрока до состояния инвентаря — находится в едином адресном пространстве оперативной памяти (Heap), доступ к которому осуществляется практически мгновенно и, что критически важно, часто без необходимости сложной синхронизации, поскольку основная логика выполняется в главном потоке (Main Thread).2

В противовес этому, архитектура Enterprise Backend, особенно при проектировании систем бронирования ресурсов (Meeting Room Booking System), погружает разработчика в хаотичный мир распределенной асинхронности. Здесь время больше не является линейным и локальным; оно становится относительным понятием, зависящим от сетевых задержек, дрейфа часов на серверах и географического расположения узлов.4 Концепция "основного потока" исчезает, уступая место пулу потоков (ThreadPool), который обслуживает тысячи конкурентных запросов одновременно. В этой среде переменная в памяти перестает быть надежным хранилищем состояния, так как сам процесс, в котором она живет, может быть уничтожен оркестратором (Kubernetes) и перезапущен на другом физическом узле в любой момент времени. Это требует перехода от мышления категориями *Stateful* (состояние в памяти) к *Stateless* (отсутствие состояния), где приложение является лишь эфемерным вычислительным узлом, а "истина" хранится во внешних, надежных системах — базах данных и распределенных кэшах.5

### 1.2. Проблема масштабируемости и управления состоянием

В монолитных приложениях, к которым относится большинство проектов на Unity, масштабирование достигается вертикально — увеличением мощности процессора или объема памяти конкретной машины. Однако в корпоративных системах стандартом является горизонтальное масштабирование. Когда нагрузка возрастает, система автоматически запускает новые экземпляры сервиса бронирования. Это создает фундаментальную проблему синхронизации: как гарантировать, что два экземпляра сервиса, запущенные на разных серверах и не имеющие общей памяти, не забронируют одну и ту же комнату для двух разных пользователей одновременно? Традиционные примитивы синхронизации языка C#, такие как lock, Monitor, Mutex или SemaphoreSlim, которые эффективно работают в Unity для координации потоков внутри одного процесса, становятся абсолютно бесполезными в распределенной среде. Блокировка, установленная на сервере А, невидима для сервера Б, что открывает двери для критических ошибок целостности данных, известных как состояния гонки (Race Conditions).1

Ниже приведена сравнительная таблица, иллюстрирующая ключевые различия в моделях управления состоянием:

| **Характеристика** | **Unity (GameDev Model)** | **Enterprise.NET Backend (Target Model)** | **Архитектурное следствие** |
| --- | --- | --- | --- |
| **Жизненный цикл** | Синхронный, Frame-based (цикл Update). | Асинхронный, Request-Response pipeline. | Переход на async/await для всех операций ввода-вывода (I/O) для освобождения потоков ThreadPool. |
| **Управление памятью** | Ручное или полуавтоматическое управление для избежания GC-пауз. | Интенсивное использование GC (Server GC mode), короткоживущие объекты. | Необходимость минимизации аллокаций в горячих путях (Hot Paths) для снижения давления на GC. |
| **Локальность данных** | Локальная память (In-Memory). Доступ за наносекунды. | Распределенные хранилища (DB, Redis). Доступ за миллисекунды. | Внедрение паттернов кэширования для компенсации сетевых задержек. |
| **Конкурентность** | Однопоточная логика (Main Thread) + Coroutines. | Многопоточная, высокая конкуренция за ресурсы. | Отказ от lock(obj) в пользу распределенных блокировок (Redlock).9 |
| **Масштабирование** | Вертикальное (оптимизация железа клиента). | Горизонтальное (Stateless Instances). | Вынос состояния сессии и блокировок в распределенный кэш (Redis).10 |

## 2. Анатомия Состояния Гонки: Механика Разрушения Целостности Данных

### 2.1. Феноменология "Двойного Бронирования" (Double Booking)

Центральной проблемой, которую предстоит решить в рамках третьего дня обучения, является предотвращение создания конфликтующих бронирований. В наивной реализации алгоритма бронирования разработчик часто следует паттерну "Check-Then-Act" (Проверь-Затем-Действуй). Процесс выглядит логично:

1. Сервис получает запрос на бронирование.
2. Сервис проверяет в базе данных, свободна ли комната на указанное время.
3. Если комната свободна, сервис записывает новую бронь.

Однако в высоконагруженной распределенной системе между шагом 2 (Чтение) и шагом 3 (Запись) существует временной зазор. Этот интервал, измеряемый миллисекундами, является "окном уязвимости". Если два конкурирующих запроса (например, от Алисы и Боба) попадают в это окно, происходит катастрофический сбой бизнес-логики. Оба запроса считывают состояние базы данных как "Свободно", и оба сервиса принимают решение о записи, что приводит к появлению двух валидных записей на одно и то же время.11

Рассмотрим хронологию событий на микроуровне:

* **T0:** Алиса инициирует запрос на бронирование комнаты 101 на 14:00.
* **T1:** Боб инициирует запрос на бронирование комнаты 101 на 14:00.
* **T2 (Сервер А):** Обрабатывает запрос Алисы. Выполняет SELECT count(\*) FROM bookings WHERE room\_id=101 AND time='14:00'. Результат: 0.
* **T3 (Сервер Б):** Обрабатывает запрос Боба. Выполняет SELECT count(\*) FROM bookings WHERE room\_id=101 AND time='14:00'. Результат: 0 (так как транзакция Алисы еще не завершена).
* **T4 (Сервер А):** Принимает решение о доступности и выполняет INSERT INTO bookings....
* **T5 (Сервер Б):** Принимает решение о доступности и выполняет INSERT INTO bookings....
* **T6:** Обе транзакции фиксируются (Commit). В базе данных существуют две записи на одно время.

Этот сценарий демонстрирует классическое состояние гонки типа "Read-Modify-Write", которое невозможно устранить простым увеличением производительности кода.10

### 2.2. Ограниченность Транзакционной Изоляции Баз Данных

Часто возникает вопрос: почему стандартные механизмы транзакций реляционных баз данных (ACID) не предотвращают эту ситуацию? Ответ кроется в уровнях изоляции транзакций. По умолчанию, большинство баз данных (включая PostgreSQL и SQL Server) используют уровень изоляции Read Committed. Этот уровень гарантирует, что транзакция увидит только зафиксированные изменения других транзакций, но он не защищает от *фантомных чтений* (Phantom Reads), когда новая строка, удовлетворяющая условию выборки, добавляется другой транзакцией в процессе выполнения текущей.13

Для предотвращения двойного бронирования исключительно средствами базы данных потребовался бы уровень изоляции Serializable. При этом уровне СУБД фактически выполняет транзакции последовательно, что гарантирует полную целостность. Однако, плата за это — катастрофическое падение производительности и высокая вероятность возникновения взаимных блокировок (Deadlocks) на уровне базы данных при высокой конкурентной нагрузке.14 Блокировка всей таблицы или диапазона строк превращает базу данных в узкое место (Bottleneck) всей системы, лишая архитектуру возможности масштабирования.16

### 2.3. Стратегии Конкурентного Доступа: Оптимизм против Пессимизма

В теории распределенных систем существуют два доминирующих подхода к решению проблемы гонки: Оптимистичная блокировка (Optimistic Locking) и Пессимистичная блокировка (Pessimistic Locking).

Оптимистичная блокировка исходит из предпосылки, что конфликты являются редким исключением. Система позволяет множеству пользователей одновременно читать и пытаться изменить данные, но перед финальной записью происходит проверка версии данных. В экосистеме.NET и Entity Framework Core это реализуется через токены конкурентности (Concurrency Tokens), такие как RowVersion (в SQL Server) или скрытое системное поле xmin в PostgreSQL.17 При попытке обновления EF Core генерирует SQL-запрос вида:

UPDATE Bookings SET User='Bob' WHERE Id=1 AND Version=5.

Если к моменту обновления Алиса уже изменила запись и версия стала 6, запрос затронет 0 строк, и будет выброшено исключение DbUpdateConcurrencyException.19 Оптимистичная блокировка идеальна для сценариев редактирования существующих сущностей, но она плохо подходит для создания новых записей (проблема нашего кейса с бронированием), так как проверка версии не защищает от вставки новой строки в "пустой" временной слот.21

**Пессимистичная блокировка**, напротив, исходит из предположения о высокой вероятности конфликта. Она блокирует ресурс для эксклюзивного доступа на все время выполнения операции — от чтения до записи. Это предотвращает вмешательство других процессов, заставляя их ждать освобождения ресурса.22 В контексте распределенной системы бронирования, где критически важно предотвратить пересечение временных интервалов (Gap Locking), пессимистичная блокировка является наиболее надежным решением. Однако, поскольку мы не хотим блокировать базу данных, мы выносим ответственность за блокировку на специализированный внешний компонент — менеджер распределенных блокировок (Distributed Lock Manager, DLM), роль которого в нашей архитектуре выполняет Redis.11

## 3. Redis как Инфраструктура Распределенной Координации

### 3.1. Архитектурные преимущества Redis для блокировок

Redis (Remote Dictionary Server) стал де-факто стандартом для реализации распределенных блокировок благодаря уникальному сочетанию характеристик. Во-первых, Redis работает исключительно в оперативной памяти, обеспечивая латентность операций на уровне микросекунд, что критически важно для минимизации времени ожидания блокировки. Во-вторых, Redis является однопоточным (Single-threaded) сервером с точки зрения обработки команд. Это гарантирует строгую сериализацию запросов: Redis физически не может выполнить две команды SET одновременно. Эта атомарность на уровне сервера избавляет разработчика от необходимости беспокоиться о гонках внутри самого механизма блокировки.24

### 3.2. Эволюция примитивов блокировки: От SETNX к атомарному SET

В ранних версиях Redis реализация блокировки требовала выполнения двух команд: SETNX (Set if Not Exists) для создания блокировки и EXPIRE для установки времени жизни (TTL) во избежание вечной блокировки при падении клиента. Однако этот подход был уязвим: если клиентское приложение падало (например, из-за отключения питания или OOM Killer) в микросекундный интервал *после* выполнения SETNX, но *до* выполнения EXPIRE, ключ оставался в Redis навсегда без тайм-аута, блокируя ресурс бесконечно (Deadlock).26

Современный стандарт (начиная с Redis 2.6.12) использует расширенный синтаксис команды SET, который объединяет создание ключа и установку TTL в одну атомарную транзакцию.

Команда выглядит следующим образом:

SET lock:room:101 unique\_token\_value NX PX 30000

Где:

* lock:room:101 — ключ ресурса.
* unique\_token\_value — уникальный идентификатор (GUID) текущего запроса/потока.
* NX — аргумент, гарантирующий запись только если ключ не существует.
* PX 30000 — установка времени жизни ключа (Expiry) в 30 000 миллисекунд.

Этот примитив обеспечивает базовую гарантию безопасности: в любой момент времени только один клиент может успешно выполнить эту команду и получить "право" на выполнение критической секции кода.25

### 3.3. Концепция "Аренды" (Lease) и проблема асинхронного времени

Важно осознавать терминологическое и концептуальное различие: то, что мы называем "распределенной блокировкой", в академической литературе корректнее называть "арендой" (Lease).30 В отличие от локальной блокировки (lock), которая действует до явного освобождения, аренда имеет ограниченный срок жизни (TTL). Это вводит фундаментальный риск: если выполнение бизнес-логики (сохранение брони в БД) займет больше времени, чем длится аренда (например, из-за "Stop-the-world" паузы сборщика мусора.NET или сетевых задержек), Redis автоматически удалит ключ блокировки. Другой клиент сможет получить новую блокировку, и оба процесса окажутся в критической секции одновременно, нарушая инвариант взаимного исключения.8

Для минимизации этого риска в Enterprise-системах применяются техники автоматического продления аренды (Lock Extension или Heartbeating), когда фоновый процесс периодически обновляет TTL активной блокировки, пока основной процесс выполняет работу. Однако даже с продлением остаются теоретические векторы атаки, связанные с дрейфом часов и приостановкой процессов, которые подробно обсуждаются критиками алгоритмов на базе Redis.8

## 4. Алгоритм Redlock: Надежность в Ненадежной Среде

### 4.1. Ограничения одиночного инстанса и проблема SPOF

Использование одного сервера Redis создает единую точку отказа (Single Point of Failure - SPOF). Если инстанс Redis недоступен, вся система бронирования останавливается. Решение "в лоб" — использовать Master-Slave репликацию с Sentinel — не работает для блокировок из-за асинхронной природы репликации в Redis. Возможен сценарий, когда Master подтверждает запись блокировки клиенту А, но падает до того, как успевает передать данные на Slave. Slave повышается до Master, не имея информации о блокировке, и выдает ту же блокировку клиенту Б. Результат — нарушение взаимного исключения.24

### 4.2. Механика и Математика Redlock

Для решения этой проблемы Сальваторе Санфилиппо (Antirez) разработал алгоритм **Redlock**. Алгоритм предполагает использование N (обычно 5) полностью независимых экземпляров Redis, расположенных на разных физических машинах, чтобы минимизировать вероятность одновременного отказа.

Процедура получения блокировки клиентом выглядит следующим образом:

1. Клиент фиксирует текущее время с точностью до миллисекунд ($T\_{start}$).
2. Клиент пытается последовательно получить блокировку (используя SET... NX PX) во всех $N$ инстансах. При этом используется короткий тайм-аут соединения (5-50 мс), чтобы не ждать ответа от упавших узлов.
3. Клиент подсчитывает количество успешных блокировок и время, затраченное на весь процесс: $T\_{elapsed} = T\_{now} - T\_{start}$.
4. Блокировка считается успешно полученной, если выполнены два условия:
   * Блокировка получена в большинстве инстансов (Quorum = $N/2 + 1$, т.е. 3 из 5).
   * $T\_{elapsed}$ меньше времени валидности блокировки (TTL).
5. Фактическое время, оставшееся у клиента на выполнение операции, равно $TTL - T\_{elapsed} - ClockDrift$ (поправка на рассинхронизацию часов).
6. Если блокировка не получена (не набран кворум или вышло время), клиент должен немедленно попытаться разблокировать все инстансы (даже те, где он не получил подтверждения, на случай потери пакета ответа).24

### 4.3. Дискуссия о корректности: Клеппман против Санфилиппо

Вокруг алгоритма Redlock существует значимая теоретическая дискуссия. Мартин Клеппман, исследователь распределенных систем, подверг критике Redlock, указав, что он опирается на предположение о синхронизированных часах серверов. Спонтанный скачок времени (например, NTP-синхронизация) на серверах Redis может привести к преждевременному истечению ключей и нарушению гарантий кворума. Клеппман предложил использовать **Fencing Tokens** (Ограждающие токены) — монотонно возрастающие числа, выдаваемые сервисом блокировок. Хранилище (БД) должно проверять этот токен при каждой записи и отвергать транзакции со старыми токенами.8

Однако, в ответном анализе Санфилиппо и сообщество разработчиков пришли к выводу, что для большинства прикладных задач (включая системы бронирования) гарантий Redlock достаточно. Реализация Fencing Tokens требует глубокой интеграции с базой данных (изменение схемы для хранения токенов), что часто избыточно. Риск "Split Brain" в Redlock существует, но он крайне мал по сравнению с рисками, возникающими при использовании одного инстанса Redis.4 Для третьего дня обучения мы принимаем Redlock как оптимальный баланс между надежностью (CP-системы) и доступностью/производительностью (AP-системы).

### 4.4. Безопасное освобождение через Lua-скриптинг

Критически важным аспектом является механизм освобождения блокировки. Клиент не может просто отправить команду DEL key. Если клиент А владел блокировкой, но "завис", и время TTL истекло, блокировку мог получить клиент Б. Если после этого клиент А "очнется" и отправит DEL, он удалит блокировку клиента Б, оставив систему незащищенной.

Для предотвращения этого используется Lua-скрипт, который выполняется на сервере Redis атомарно:

Lua

if redis.call("get", KEYS) == ARGV then  
 return redis.call("del", KEYS)  
else  
 return 0  
end

Скрипт удаляет ключ только в том случае, если его текущее значение совпадает с уникальным токеном, который клиент сгенерировал при получении блокировки. Это гарантирует, что клиент может снять только "свою" блокировку.12

## 5. Инженерная Реализация в Экосистеме.NET 8

### 5.1. Интеграция библиотеки RedLock.net

Для реализации алгоритма в.NET 8 не рекомендуется писать низкоуровневый код поверх драйвера StackExchange.Redis, так как это чревато ошибками в логике кворума и ретраев. Отраслевым стандартом является использование библиотеки **RedLock.net** (автор Sam Cook), которая корректно реализует алгоритм, включая поддержку асинхронности и автоматическое продление блокировок.24

Ключевым паттерном использования является идиома IDisposable через блок using. Это гарантирует, что попытка освобождения блокировки произойдет даже в случае возникновения исключения внутри критической секции.

C#

// Концептуальный пример использования в сервисе бронирования  
public async Task<BookingResult> TryCreateBookingAsync(BookingRequest request)  
{  
 // Ключ блокировки должен быть гранулярным: блокируем конкретную комнату  
 var resourceKey = $"lock:room:{request.RoomId}";  
 var expiry = TimeSpan.FromSeconds(30); // Максимальное время удержания  
 var wait = TimeSpan.FromSeconds(10); // Время ожидания (Spin wait)  
 var retry = TimeSpan.FromSeconds(1); // Интервал между попытками  
  
 // Попытка захвата распределенной блокировки  
 await using (var redLock = await \_redLockFactory.CreateLockAsync(resourceKey, expiry, wait, retry))  
 {  
 if (redLock.IsAcquired)  
 {  
 // --- НАЧАЛО КРИТИЧЕСКОЙ СЕКЦИИ ---  
   
 // 1. Повторная проверка бизнес-правил (Double Check)  
 var conflicts = await \_dbContext.Bookings  
 .AnyAsync(b => b.RoomId == request.RoomId && b.Overlaps(request.Time));  
   
 if (conflicts) return BookingResult.Conflict();  
  
 // 2. Атомарное действие  
 \_dbContext.Bookings.Add(new Booking(request));  
 await \_dbContext.SaveChangesAsync();  
   
 return BookingResult.Success();  
   
 // --- КОНЕЦ КРИТИЧЕСКОЙ СЕКЦИИ ---  
 }  
 else  
 {  
 // Не удалось получить лок за время 'wait'.  
 // Это означает высокую конкуренцию или сбой Redis.  
 return BookingResult.SystemBusy();  
 }  
 }   
 // Dispose автоматически вызывает Lua-скрипт разблокировки  
}

### 5.2. Паттерны Внедрения Зависимостей (Dependency Injection)

Правильная конфигурация DI-контейнера критична для производительности. RedLockFactory (как и ConnectionMultiplexer в StackExchange.Redis) должен регистрироваться как **Singleton**. Создание нового соединения с Redis на каждый HTTP-запрос является анти-паттерном, который приводит к исчерпанию портов (Socket Exhaustion) и высокой латентности из-за рукопожатий TCP/TLS.35

В Program.cs конфигурация должна предусматривать список эндпоинтов для всех участвующих инстансов Redis:

C#

builder.Services.AddSingleton<IDistributedLockFactory>(sp =>   
{  
 var endPoints = new List<RedLockEndPoint>  
 {  
 new DnsEndPoint("redis-node-1", 6379),  
 new DnsEndPoint("redis-node-2", 6379),  
 new DnsEndPoint("redis-node-3", 6379)  
 };  
 return RedLockFactory.Create(endPoints);  
});

Также важно корректно обрабатывать завершение работы приложения (IApplicationLifetime), вызывая Dispose у фабрики для корректного закрытия соединений.34

## 6. Высокопроизводительные Архитектуры Кэширования

### 6.1. Паттерн Cache-Aside (Lazy Loading): Теория и Практика

Системы бронирования характеризуются высоким коэффициентом чтения к записи (Read/Write Ratio). Пользователи просматривают расписание сотни раз, прежде чем сделать одно бронирование. Прямые запросы к PostgreSQL для построения календарной сетки, особенно с учетом сложной логики развертывания рекуррентных событий (RRULE), создают недопустимую нагрузку на CPU базы данных. Решением является паттерн **Cache-Aside** (Кэширование "сбоку" или Ленивая загрузка).37

Алгоритм работы Cache-Aside:

1. **Приложение получает запрос на чтение данных.**
2. **Проверка кэша (Redis):** Приложение ищет данные по ключу (например, schedule:room:101:week:42).
3. **Cache Hit:** Если данные найдены, они десериализуются и возвращаются клиенту. База данных не затрагивается.
4. **Cache Miss:** Если данных нет, приложение выполняет "тяжелый" запрос к PostgreSQL.
5. **Заполнение кэша:** Полученные данные сохраняются в Redis с установленным TTL (например, 10 минут).
6. **Ответ:** Данные возвращаются клиенту.

Этот подход обеспечивает надежность: если Redis упадет, приложение продолжит работать, обращаясь напрямую к БД (хоть и медленнее), в отличие от паттернов Read-Through, где кэш является обязательным посредником.38

### 6.2. Проблема Грочущего Стада (Thundering Herd) и методы минимизации

Классической уязвимостью Cache-Aside является проблема "Грочущего Стада" (также известная как Cache Stampede). Представим, что кэш популярной комнаты истекает (TTL expire). В ту же секунду приходят 1000 одновременных запросов. Все они получают Cache Miss. Все 1000 потоков одновременно идут в базу данных с тяжелым запросом. Это может мгновенно обрушить базу данных ("Avalanche").40

Для защиты применяются следующие стратегии:

1. **Jitter (Дрожание TTL):** К стандартному времени жизни кэша (например, 10 минут) добавляется случайное значение (например, +/- 60 секунд). Это предотвращает одновременное истечение множества ключей, "размазывая" нагрузку на БД по времени.43
2. **Mutex on Cache Miss:** При получении Cache Miss поток не бежит сразу в БД, а пытается взять локальную (в памяти) или распределенную блокировку на обновление этого конкретного ключа. Первый поток, взявший лок, идет в БД и обновляет кэш. Остальные 999 потоков ждут, а затем читают уже обновленное значение из Redis. Библиотеки вроде FusionCache или LazyCache реализуют этот механизм ("Cache Stampede Protection") из коробки.44

### 6.3. Инвалидация против Обновления (Write-Through vs. Invalidation)

При изменении данных (создании брони) возникает дилемма: обновить значение в кэше или удалить его?

В распределенных системах предпочтительна стратегия Инвалидации (Удаления).

Рассмотрим сценарий обновления (Write-Through) с состоянием гонки:

1. Процесс А меняет данные в БД на v1.
2. Процесс Б меняет данные в БД на v2.
3. Процесс Б обновляет кэш значением v2.
4. Процесс А (из-за задержки сети) обновляет кэш значением v1.  
   Итог: В БД лежит актуальное v2, а в кэше — устаревшее v1. Данные рассинхронизированы.

При стратегии удаления (redis.DEL(key)) порядок прихода команд не так критичен. Даже если удаления перепутаются, кэш просто останется пустым, и следующий запрос на чтение подтянет актуальные данные из БД. Это соответствует принципу "Eventual Consistency" (Согласованность в конечном счете).45

## 7. Защитная Инженерия: Идемпотентность и Наблюдаемость

### 7.1. Идемпотентность API (Idempotency)

В ненадежных сетях клиент может не получить ответ от сервера из-за тайм-аута, хотя операция на сервере прошла успешно. Если клиент (или Retry Policy) повторит POST-запрос на создание брони, это может привести к дублированию данных или ложной ошибке конфликта.

Для решения этой проблемы внедряется механизм идемпотентности:

1. Клиент генерирует уникальный Idempotency-Key (UUID) и отправляет его в заголовке запроса.
2. Сервер (через Middleware) проверяет наличие этого ключа в Redis.
3. Если ключ найден, сервер возвращает сохраненный результат (HTTP статус и тело ответа) без повторного выполнения бизнес-логики.
4. Если ключ не найден, сервер выполняет операцию, и перед возвратом ответа атомарно сохраняет ключ и результат в Redis с TTL (например, 24 часа).  
   Это гарантирует, что операция f(x) выполняется ровно один раз, независимо от количества вызовов f(x).48

### 7.2. Наблюдаемость (Observability) через OpenTelemetry

Для управления сложной распределенной системой "черного ящика" (каким является скомпилированный код в продакшене) необходима полная наблюдаемость. В рамках.NET 8 стандартом является использование OpenTelemetry.

Мы фокусируемся на метриках по методу RED (Rate, Errors, Duration):

* **Гистограммы (Histograms):** Критически важны для измерения латентности распределенных блокировок (lock\_acquisition\_duration). Мы должны следить не за средним временем (которое скрывает выбросы), а за перцентилями P95 и P99 (время, в которое укладываются 99% запросов).
* **Счетчики (Counters):** Для отслеживания бизнес-событий, таких как bookings\_created\_total, и технических событий, таких как redlock\_failures\_total. Резкий рост отказов в получении блокировки является ранним индикатором проблем с производительностью Redis или сетью.1

## 8. Заключение

Третий день обучения знаменует собой переход от написания кода, выполняющего функции, к проектированию систем, обеспечивающих гарантии. Мы деконструировали иллюзию надежности локальной памяти и заменили её на явные механизмы распределенной координации. Использование Redis и алгоритма Redlock позволяет достичь компромисса между строгой согласованностью данных (Consistency) и доступностью системы (Availability) в условиях возможного разделения сети (Partition Tolerance). Внедрение паттернов Cache-Aside с защитой от Stampede и механизмов идемпотентности превращает хрупкое приложение в надежный инженерный продукт Enterprise-уровня. Эти знания формируют фундамент компетенций Senior Backend разработчика, способного аргументированно выбирать архитектурные решения для высоконагруженных сред.

#### Источники

1. Неделя 14
2. API compatibility levels for .NET - Unity - Manual, дата последнего обращения: декабря 6, 2025, <https://docs.unity3d.com/6000.2/Documentation/Manual/dotnet-profile-support.html>
3. Why I Think We Shouldn't Be Recommending Unity To New C# Developers - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/csharp/comments/1hdanoi/why_i_think_we_shouldnt_be_recommending_unity_to/>
4. Is Redlock safe? -
5. Analysis of Monolithic and Distributed Systems - Learn System Design - GeeksforGeeks, дата последнего обращения: декабря 6, 2025, <https://www.geeksforgeeks.org/system-design/analysis-of-monolithic-and-distributed-systems-learn-system-design/>
6. Stateful vs Stateless: Choosing the Right Backend Architecture - F22 Labs, дата последнего обращения: декабря 6, 2025, <https://www.f22labs.com/blogs/stateful-vs-stateless-choosing-the-right-backend-architecture/>
7. Stateful vs. Stateless Architecture - GeeksforGeeks, дата последнего обращения: декабря 6, 2025, <https://www.geeksforgeeks.org/system-design/stateful-vs-stateless-architecture/>
8. How to do distributed locking - Martin Kleppmann, дата последнего обращения: декабря 6, 2025, <https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html>
9. can someone explain why we ditched monoliths for microservices? like... what was the reason fr? - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/SoftwareEngineering/comments/1k2ppy9/can_someone_explain_why_we_ditched_monoliths_for/>
10. Race condition in Distributed Systems - Software Engineering Stack Exchange, дата последнего обращения: декабря 6, 2025, <https://softwareengineering.stackexchange.com/questions/403297/race-condition-in-distributed-systems>
11. Handling the Double-Booking Problem in Databases - Adam Djellouli, дата последнего обращения: декабря 6, 2025, <https://adamdjellouli.com/articles/databases_notes/07_concurrency_control/04_double_booking_problem>
12. Hands-on Preventing Database Race Conditions with Redis | by Miftahul Huda | Medium, дата последнего обращения: декабря 6, 2025, <https://iniakunhuda.medium.com/hands-on-preventing-database-race-conditions-with-redis-2c94453c1e47>
13. Optimistic vs. Pessimistic Locking - Vlad Mihalcea, дата последнего обращения: декабря 6, 2025, <https://vladmihalcea.com/optimistic-vs-pessimistic-locking/>
14. Pessimistic vs Optimistic Locking - ByteByteGo, дата последнего обращения: декабря 6, 2025, <https://bytebytego.com/guides/pessimistic-vs-optimistic-locking/>
15. Optimism vs Pessimism in Distributed Systems - Marc's Blog - brooker.co.za, дата последнего обращения: декабря 6, 2025, <https://brooker.co.za/blog/2023/10/18/optimism.html>
16. Implementation Principles and Best Practices of Distributed Lock - Alibaba Cloud, дата последнего обращения: декабря 6, 2025, <https://www.alibabacloud.com/blog/implementation-principles-and-best-practices-of-distributed-lock_600811>
17. Optimistic Locking in .NET - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@imaanmzr/optimistic-locking-in-net-bd677916ef60>
18. Concurrency Tokens | Npgsql Documentation, дата последнего обращения: декабря 6, 2025, <https://www.npgsql.org/efcore/modeling/concurrency.html>
19. Solving Race Conditions With EF Core Optimistic Locking, дата последнего обращения: декабря 6, 2025, <https://www.milanjovanovic.tech/blog/solving-race-conditions-with-ef-core-optimistic-locking>
20. Handling Concurrency Conflicts - EF Core - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/ef/core/saving/concurrency>
21. Optimistic vs. Pessimistic locking - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/129329/optimistic-vs-pessimistic-locking>
22. Optimistic vs. Pessimistic Locking in Databases | by Ivan Duhov | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@duhov/optimistic-vs-pessimistic-locking-in-databases-c32a52aeadfe>
23. Pessimistic Locking Vs. Optimistic Locking - Modern Treasury, дата последнего обращения: декабря 6, 2025, <https://www.moderntreasury.com/learn/pessimistic-locking-vs-optimistic-locking>
24. Distributed Locks with Redis | Docs, дата последнего обращения: декабря 6, 2025, <https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/>
25. How to implement distributed locks with Redis? - Tencent Cloud, дата последнего обращения: декабря 6, 2025, <https://www.tencentcloud.com/techpedia/126127>
26. SETNX | Docs - Redis, дата последнего обращения: декабря 6, 2025, <https://redis.io/docs/latest/commands/setnx/>
27. The Best Way to Implement Distributed Locks Using Redis: Avoiding Common Pitfalls, дата последнего обращения: декабря 6, 2025, <https://dilipcoder.medium.com/the-best-way-to-implement-distributed-locks-using-redis-avoiding-common-pitfalls-1e4cff5231b0>
28. Creating Distributed Lock With Redis In .NET Core - C# Corner, дата последнего обращения: декабря 6, 2025, <https://www.c-sharpcorner.com/article/creating-distributed-lock-with-redis-in-net-core/>
29. .NET Apps: Redis Distributed Locking - AnimatLabs, дата последнего обращения: декабря 6, 2025, <https://animatlabs.com/technical/.net/apps/redis-distribulted-locking/>
30. Locks, leases, fencing tokens, FizzBee! - Surfing Complexity, дата последнего обращения: декабря 6, 2025, <https://surfingcomplexity.blog/2025/03/03/locks-leases-fencing-tokens-fizzbee/>
31. Implementing Distributed Locks with Redis Delving into SETNX, Redlock, and Their Controversies | Leapcell, дата последнего обращения: декабря 6, 2025, <https://leapcell.io/blog/implementing-distributed-locks-with-redis-delving-into-setnx-redlock-and-their-controversies>
32. Distributed Locks with Redis (2014) - Hacker News, дата последнего обращения: декабря 6, 2025, <https://news.ycombinator.com/item?id=41292477>
33. Is Redlock safe? -
34. RedLock.net/README.md at master · samcook/RedLock.net · GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/samcook/RedLock.net/blob/master/README.md>
35. Dependency injection guidelines - .NET | Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection-guidelines>
36. RedLockFactory as a Singleton · Issue #54 · samcook/RedLock.net - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/samcook/RedLock.net/issues/54>
37. Cache-Aside Pattern - GeeksforGeeks, дата последнего обращения: декабря 6, 2025, <https://www.geeksforgeeks.org/system-design/cache-aside-pattern/>
38. Caching patterns - Database Caching Strategies Using Redis - AWS Documentation, дата последнего обращения: декабря 6, 2025, <https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html>
39. Cache-Aside Pattern - Azure Architecture Center - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside>
40. Dealing with Cache Stampede and Thundering Herd - System Design Interview Guide, дата последнего обращения: декабря 6, 2025, <https://bugfree.ai/knowledge-hub/dealing-with-cache-stampede-and-thundering-herd>
41. A Complete Beginner Guide for Cache Penetration, Stampede, and Avalanche, дата последнего обращения: декабря 6, 2025, <https://philosophyotaku.medium.com/a-complete-beginner-guide-for-cache-penetration-stampede-avalanche-ecadd7f16009>
42. Thundering Herd/Cache Stampede - Distributed Computing Musings, дата последнего обращения: декабря 6, 2025, <https://distributed-computing-musings.com/2021/12/thundering-herd-cache-stampede/>
43. Caching Strategy: Cache-Aside Pattern - EnjoyAlgorithms, дата последнего обращения: декабря 6, 2025, <https://www.enjoyalgorithms.com/blog/cache-aside-caching-strategy/>
44. Redis as LRU cache race condition - database - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/28886711/redis-as-lru-cache-race-condition>
45. Cache-Aside Explained: The Pattern Every Backend Developer Should Know | by Anderson Kuntz Meurer | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@andersonmeurerr/cache-aside-explained-the-pattern-every-backend-developer-should-know-3e0cbda93f26>
46. Possibility of stale data in cache-aside pattern - Software Engineering Stack Exchange, дата последнего обращения: декабря 6, 2025, <https://softwareengineering.stackexchange.com/questions/334028/possibility-of-stale-data-in-cache-aside-pattern>
47. Cache Invalidation vs. Expiration: Best Practices - Daily.dev, дата последнего обращения: декабря 6, 2025, <https://daily.dev/blog/cache-invalidation-vs-expiration-best-practices>
48. A Hitchhiker's Guide to Caching Patterns - Hazelcast, дата последнего обращения: декабря 6, 2025, <https://hazelcast.com/blog/a-hitchhikers-guide-to-caching-patterns/>
49. Idempotency in a Distributed System | by Sameer Ahmed - Medium, дата последнего обращения: декабря 6, 2025, <https://sameerahmed56.medium.com/idempotency-in-a-distributed-system-df67fbd93b49>
50. Airbnb Idempotency - Avoiding Double Payments in a Distributed Payments System, дата последнего обращения: декабря 6, 2025, <https://www.geeksforgeeks.org/system-design/airbnb-idempotency-avoiding-double-payments-in-a-distributed-payments-system/>