Distributed Storage System

Primary-Based Remote-Write Architecture

제출일	2024.11.27	전공	컴퓨터공학과
과목	분산처리 (2분반	학번	32224332
담당교수	남재현 교수님	이름	조남웅

목차

1. 프로젝트 개요

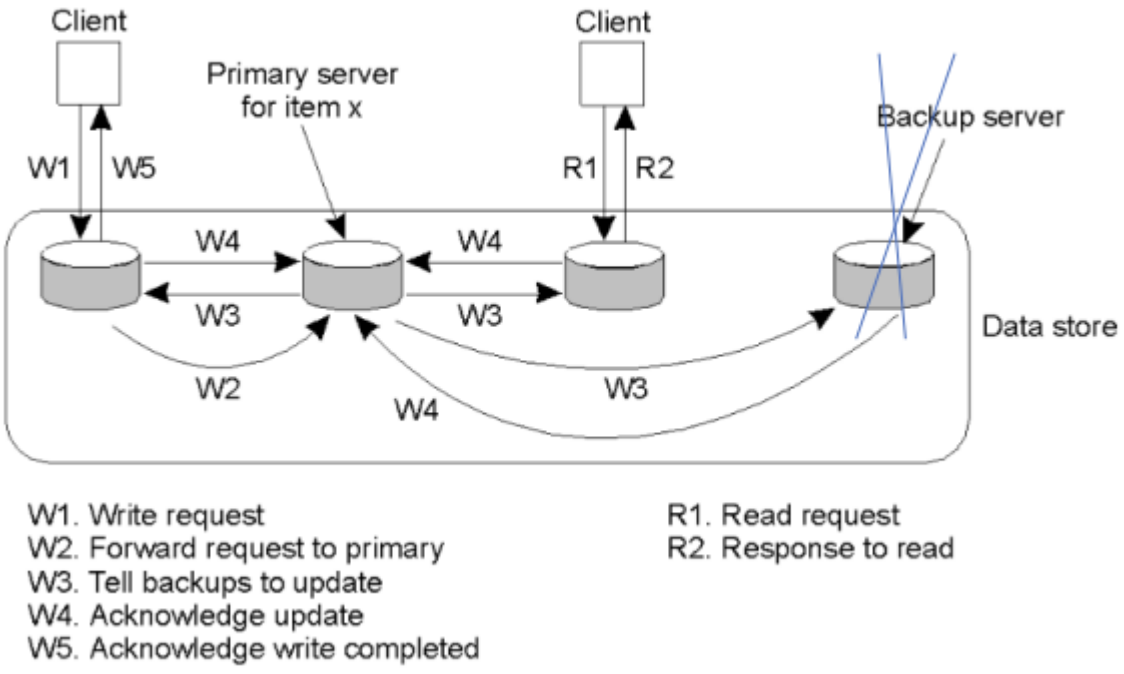
1.1. 설계 목적

본 프로젝트인 Distribution Storage의 설계 목적은 분산 스토리지 환경에서 데이터 일관성을 유지하며 확장가능한 아키텍처 설계를 목적으로 삼았다. 앞선 프로젝트인 API,TCP, UDP 서버들을 활용하여 분산 스토리지들의 클라이언트로서 사용했으며, 분산 스토리지 내에서 데이터 일관성을 유지하는 방법에 대해 연구하였다. 뿐만 아니라 앞으로의 확장 가능성도 염두하여 유연하면서 안전한 분산 스토리지를 목적으로 설계 하였다.

1.2. 요구 사항

프로젝트의 요구사항은 총 3가지로 구성되어있다. 첫번째 요구사항은, Primary-Based Remote-Write 아키텍처를 바탕으로 Primary Storage를 통해 모든 로컬 스토리지와 동기화를 구현하는 것이다. 따라서 전체적인 시스템을 Primary-Based Remote-Write를 기반으로 설계 하였으며 여기서 가장 중요한 가정은 Primary Storage는 서비스에 대한 장애없이 영원히 기능한다는 가정이다. 이 가정을 염두하여 각각의 로컬 스토리지를 구현하였다.

다음 그림은 제공된 요구사항인 Primary-Based Remote-Write 의 구조 및 동작에 대한 그림이다. 이 부분은 2장 시스템 구조 및 아키텍처 부분에 자세히 설명되어있다.



두번째 요구사항은 각 스토리지의 독립적인 프로토콜 지원이다. 이 요구사항은 각각의 로컬 스토리지가 존재하는 이유에 뒷받침하는 근거이기도 하다. 프라이머리 스토리지는 독립적인 3개의 프로토콜을

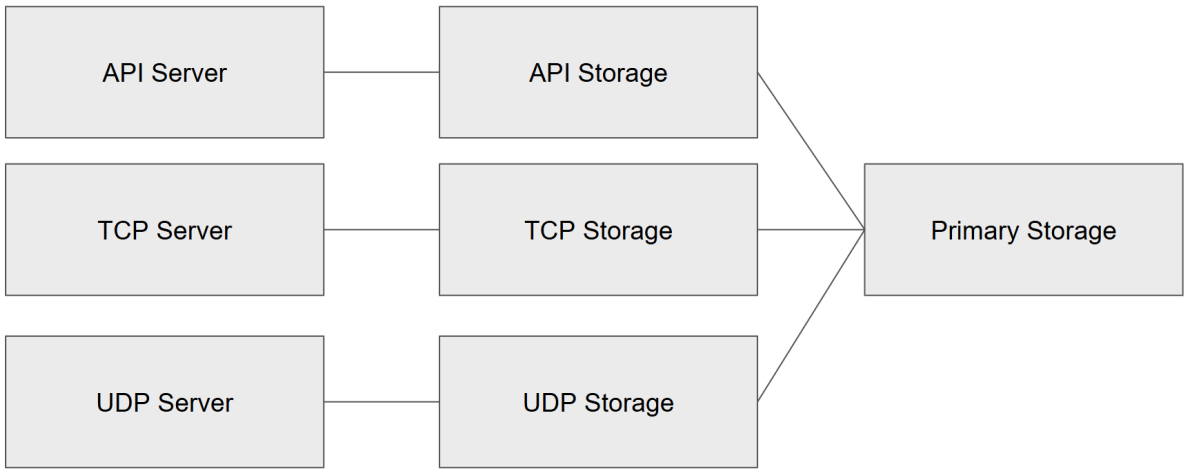
사용하는 각각의 로컬 스토리지와 통신할수 있어야 한다. 따라서 이번 Distribution Storage를 구현할 때 사용한 3개의 독립적인 프로토콜로 UDP,TCP,HTTP를 사용하였다. API 스토리지는 HTTP를 사용하여 프라이머리 스토리지와 통신하였고, TCP 스토리지는 클라이언트와의 통신은 TCP지만, 프라이머리 스토리지와 통신할때는 HTTP를 사용하였다. 여기서도 코드 재사용성을 위해 완전한 HTTP 프로토콜 그대로 사용한 것이 아닌 직접적인 파싱으로 통신하였다. 이 부분은 컴포넌트 상세 설명의 각 로컬 스토리지 구현을 설명하는 3장에서 자세히 설명되어있다.

마지막으로 세번째 요구사항은 로컬 스토리지 장애 시에도 시스템 가용성 확보에 대한 부분이다. 이것의 의미는 전체 시스템의 안정성과 지속성을 유지하는 핵심 설계 목표를 의미한다. 이는 분산 시스템의 가장 중요한 특징 중 하나로, 단일 장애점(Single Point of Failure)을 제거하고 클라이언트 경험을 향상시키는 데 중요한 역할을 한다. 따라서 이 요구사항을 충족하기위해 Primary Storage는 장애를 감지하고 활성화된 스토리지와만 동기화하여 시스템 가용성을 유지하고. 장애 복구 시 누락 데이터를 재동기화하여 데이터 일관성을 보장하며, 서비스 중단 없이 요청을 처리할 수 있도록 설계했다.

2. 시스템 구조 및 아키텍처

High-level 구조,Primary-Based Remote-Write 아키텍처 원리,시스템 동작 흐름

2.1. High-level 구조 (component Diagram)



위 그림은 각각 분산 스토리지의 전체적인 구조를 나타내는 다이어그램이다. 이 다이어그램을 바탕으로 중앙 관리와 데이터 분산, 모듈 통합 및 확장성, 데이터 흐름과 요청 처리의 이렇게 3가지 관점으로 구조를 설명할 수 있다.

첫번째로 중앙 관리와 데이터 분산의 관점이다. 시스템은 프라이머리 스토리지를 중심으로 동작하며,

로컬 스토리지는 분산 데이터 저장을 담당한다. 프라이머리 스토리지는 클라이언트 요청을 수신하여 데이터를 검증하고 각 로컬 스토리지로 전달하는 중앙 관리자로 설계했다. 로컬 스토리지는 API, TCP, UDP 통신 프로토콜을 기반으로 설계되어 다양한 데이터 입출력을 처리하며, 프라이머리 스토리지와의 동기화를 통해 데이터 일관성을 유지한다. 이러한 구조는 중앙 집중식 제어와 분산 저장의 장점을 결합하여 데이터 관리 효율성을 극대화한다.

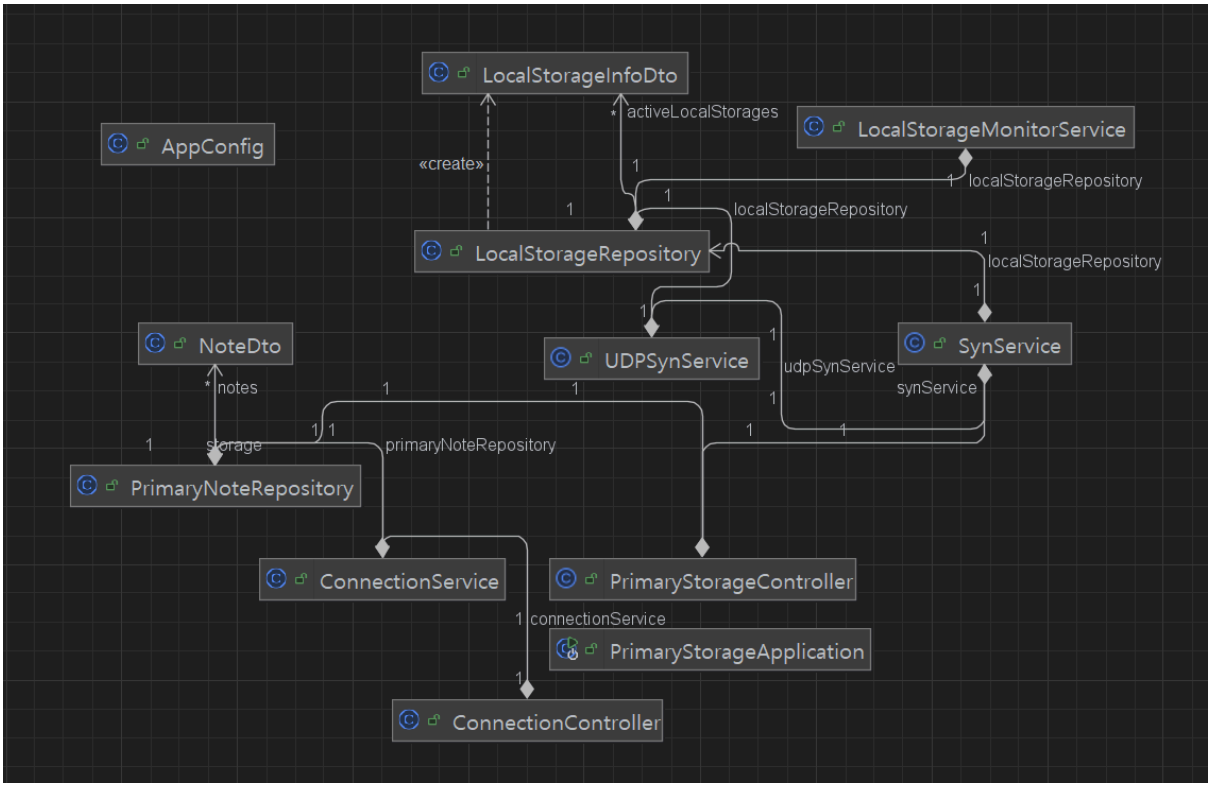
두번째로, 모듈 간 통합 및 확장성의 관점이다.로컬 스토리지는 독립적으로 동작하면서도 프라이머리 스토리지와 긴밀히 연계되어 있다. 프라이머리 스토리지는 상태 확인 및 데이터 전송을 통해 활성 로컬 스토리지를 동적으로 식별하며, 필요한 경우 동기화를 수행한다. 이러한 동작은 `SynService` 및 `UDPSynService`와 같은 모듈로 구현되며, RESTful API 및 UDP 메시지를 활용한다. 각 로컬 스토리지는 독립적인 엔드포인트를 가지며, 특정 장애 발생 시 다른 스토리지가 역할을 대체하여 시스템 가용성을 보장한다. 이 구조는 새로운 스토리지 유형 추가 시에도 손쉽게 확장할 수 있도록 설계했다.

마지막으로, 데이터 흐름과 요청 처리의 관점이다. 시스템의 데이터 흐름은 클라이언트(각 프로토콜별 서버) 요청에서 시작해 프라이머리 스토리지를 거쳐 로컬 스토리지로 전달되는 구조로 이루어진다. 프라이머리 스토리지는 요청을 분석하여 적절한 로컬 스토리지에 전송하며, 로컬 스토리지는 데이터를 처리한 후 결과를 반환한다. UDP 스토리지는 빠른 메시지 전송을, TCP 스토리지는 안정적인 데이터 송수신을, API 스토리지는 HTTP 기반의 접근성을 제공한다. 이러한 멀티 프로토콜 설계는 다양한 클라이언트 요구를 수용하며, 시스템의 유연성과 성능을 향상시킨다.

3. 컴포넌트 구성요소

Primary Storage, API Storage, TCP Storage, UDP Storage 총 4가지 Storage의 구체적인 내용이다.

3.1. Primary Storage



프라이머리 스토리지는 분산 환경에서 데이터의 중앙 집중식 관리를 수행하며, 다양한 로컬 스토리지와 통신하고 데이터를 동기화하는 핵심 역할을 담당한다. 이를 통해 모든 데이터가 일관성을 유지하고, 장애가 발생한 로컬 스토리지의 데이터를 복구하거나 대체할 수 있도록 한다. 프라이머리 스토리지의 구조는 계층적으로 설계되어 있으며, 각 구성 요소는 특정한 역할을 수행하도록 설계되었다. Controller Layer, Service Layer, Repository Layer 총 3가지 Layer에 의해 동작 한다.

첫번째 계층인 Contorller Layer은 `PrimaryStorageController`와 `ConnectionController`로 구성된다. `PrimaryStorageController`는 로컬 스토리지지의 요청을 처리하며, 노트의 CRUD 작업(생성, 읽기, 수정, 삭제)을 프라이머리 레벨에서 수행한다. 로컬 스토리지는 REST API를 통해 이 컨트롤러에 요청을 보내고 응답을 받는다. `ConnectionController`는 로컬 스토리지와의 상태 확인 및 연결을 관리하며, 로컬 스토리지가 활성화되었는지 주기적으로 확인하는 작업을 수행한다.

두번째 계층인 Service Layer은 프라이머리 스토리지의 핵심 로직이 포함되어 있다. `SynService`는 로컬 스토리지와의 데이터 동기화를 담당하며, `UDPSynService`는 UDP 스토리지와의 특화된 통신을 처리한다. `ConnectionService`는 로컬 스토리지와의 네트워크 상태를 유지하며, `LocalStorageMonitorService`는 로컬 스토리지가 정상적으로 작동하는지 주기적으로 점검한다. 이 계층은 프라이머리 스토리지의 안정성과 효율성을 보장하는 데 중요한 역할을 한다.

세번째 계층인 Repository Layer는 프라이머리 스토리지의 데이터 저장소로 노트를 저장하는 `PrimaryNoteRepository`와 로컬 스토리지의 정보를 저장하는 `LocalStorageRepository`를 포함한다. `PrimaryNoteRepository`는 노트 데이터를 관리하며, 클라이언트의 요청에 따라 데이터를 검색하거나 저장한다. `LocalStorageRepository`는 활성화된 로컬 스토리지의 상태와 정보를 추적하며, 동기화 작업에서 필수적인 정보를 제공한다.

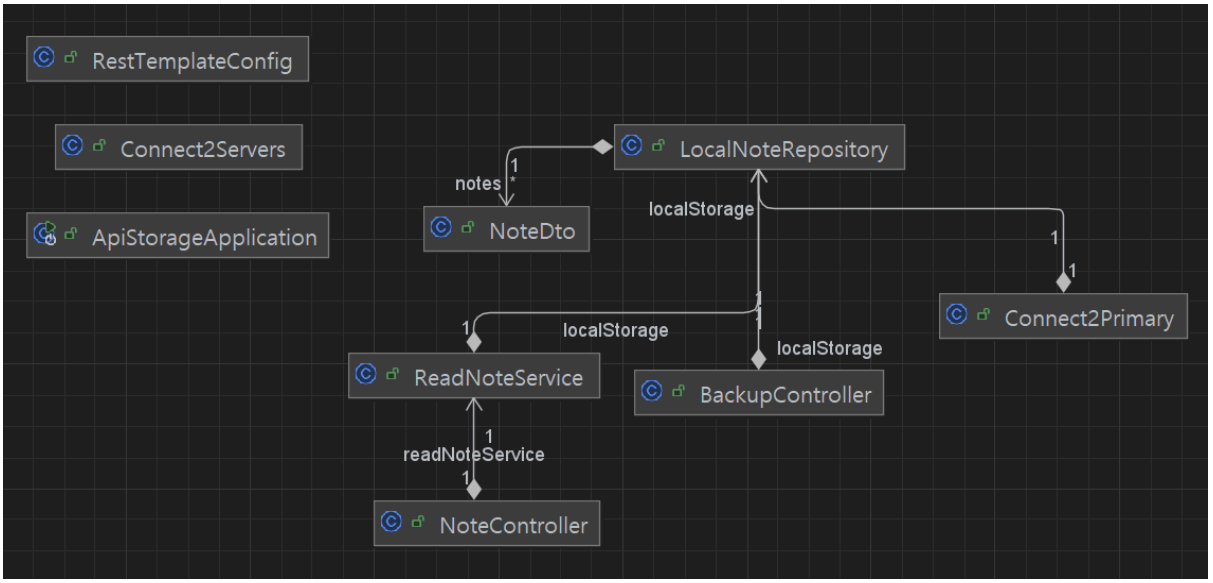
프라이머리 스토리지는 Spring Boot 프레임워크를 기반으로 개발되었으며, RESTful API를 활용해 클라이언트와의 통신을 구현한다. Spring의 `@Service`, `@Repository`, `@Controller`와 같은 어노테이션을 활용해 계층별로 역할을 분리하여 코드의 가독성과 유지보수성을 높였다. 프라이머리 스토리지는 로컬 스토리지와의 통신 시 HTTP와 UDP를 모두 지원하며, UDP 스토리지는 별도의 `UDPSynService`를 통해 최적화된 통신 방식으로 처리한다. 또한, 프라이머리 스토리지는 `application.properties` 파일을 통해 포트 번호, 네트워크 설정 등을 관리하며, 개발 및 배포 환경에서의 유연성을 제공한다. Spring의 DI(의존성 주입)를 적극 활용하여 각 컴포넌트를 느슨하게 결합하고, 테스트와 확장성을 고려한 설계를 적용하였다.

프라이머리 스토리지는 모든 데이터 변경의 중앙집중식 처리와 동기화를 통해 데이터 일관성을 유지한다. 로컬 스토리지는 데이터를 추가, 수정, 삭제하거나 조회할 때 프라이머리 스토리지와 직접 통신하므로, 분산 시스템에서 데이터 동기화의 복잡성을 각각의 로컬 스토리지가 직접 처리할 필요가 없다. 이는 시스템의 확장성과 안정성을 동시에 보장한다. 또한, 다양한 프로토콜(HTTP, UDP)을 유연하게 사용함으로써 각 로컬 스토리지의 특징에 맞는 통신 방식을 제공한다. 다음은 프라이머리 스토리지에서의 주요한 통신의 흐름을 단계별로 나타낸 것이다.

1) **로컬 스토리지 요청 처리:** 로컬 스토리지가 데이터를 추가(`POST`), 수정(`PUT`/`PATCH`), 삭제(`DELETE`)하거나 조회(`GET`)하면, 프라이머리 스토리지가 이를 처리한다.
   클라이언트의 요청은 `PrimaryStorageController`를 통해 전달되며, 요청이 적절한 서비스(`SynService`, `ConnectionService`)로 위임된다. 예를 들어, 클라이언트가 `POST` 요청을 보낼 경우, 프라이머리 스토리지는 데이터를 저장한 뒤, 각 로컬 스토리지에 동기화 요청을 보낸다.

2) **로컬 스토리지와의 동기화:** 프라이머리 스토리지는 로컬 스토리지와 통신하여 데이터를 동기화한다. HTTP 기반 스토리지(API 및 TCP 스토리지)에는 `RestTemplate`을 사용해 HTTP 요청을 보낸다.UDP 스토리지에는 `UDPSynService`를 통해 UDP 패킷으로 요청을 전송한다. UDP 스토리지는 UDP 응답을 통해 자신의 상태 및 데이터 처리 결과를 반환한다.이러한 동기화 과정은 `SynService`에서 수행되며, 각 로컬 스토리지가 제대로 활성화되어 있는지 확인한 후 요청이 전달된다. 로컬 스토리지의 비활성화 상태는 동기화 로직에서 무시되므로 시스템 가용성을 유지할 수 있다.
3) **로컬 스토리지 상태 확인:** `LocalStorageMonitorService`는 정기적으로 각 로컬 스토리지의 상태를 확인한다. HTTP 기반 스토리지는 `/connect/status` 엔드포인트를 통해 상태를 확인하고, UDP 스토리지는 UDP 패킷을 전송하여 준비 상태를 응답받는다. 활성화된 스토리지 정보는 `LocalStorageRepository`에 저장되며, 동기화 시에 참조된다.

3.2. API Storage



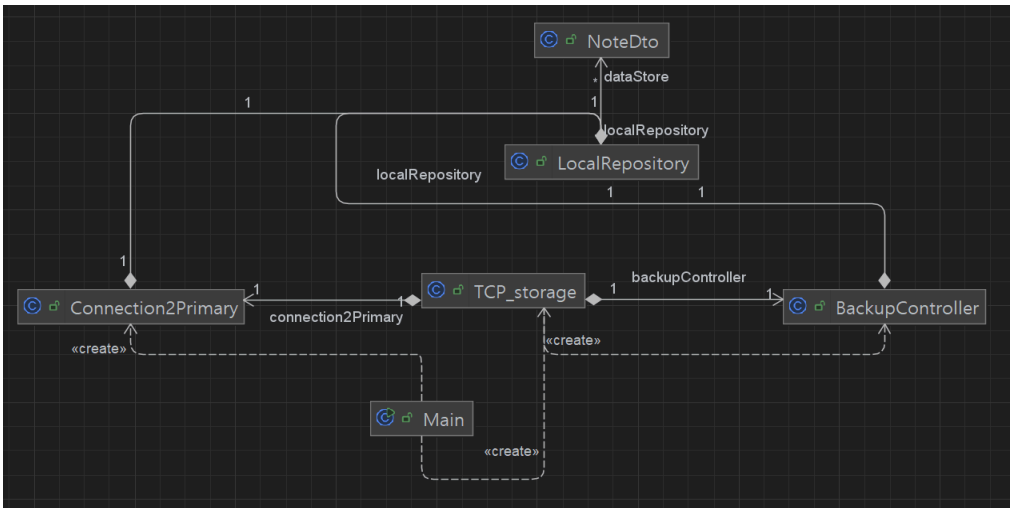
API 스토리지는 클라이언트와의 HTTP 기반 통신을 통해 데이터 요청을 처리한다. 주요 역할은 클라이언트가 요청한 데이터를 저장, 조회, 수정, 삭제하는 것이다. 또한, 데이터의 일관성을 보장하기 위해 프라이머리 스토리지와 주기적으로 동기화하며, 다른 스토리지들과 데이터를 공유한다. 이러한 작업은 다양한 REST 엔드포인트를 통해 처리되며, 클라이언트가 시스템의 다른 구성 요소에 직접 접근하지 않고도 데이터를 관리할 수 있도록 한다. 클라이언트인 API Server는 어떠한 서비스 장애 없이 항상 기능하는것을 가정으로 세운 후 API 스토리지를 구현하였다.

다음은 API 스토리지의 주요 클래스에 관한 내용이다.

1. Controller: `NoteController`는 클라이언트의 요청을 처리하며, 노트 데이터의 CRUD 작업을 위한 다양한 엔드포인트를 제공한다. `BackupController`는 데이터 백업 및 동기화 작업을 관리한다.
2. Service: `ReadNoteService`는 노트 데이터의 조회 로직을 포함하며, 데이터를 효율적으로 검색하고 클라이언트에 전달한다.
3. Connection: `Connect2Primary`와 `Connect2Servers`는 각각 프라이머리 스토리지와 다른 스토리지들과의 네트워크 통신을 관리한다. 이를 통해 API 스토리지는 데이터 동기화 및 공유를 원활하게 수행할 수 있다.
4. Repository: `LocalNoteRepository`는 로컬 저장소로, 클라이언트 요청 처리 중 캐싱된 데이터를 저장하거나, 프라이머리 스토리지와 동기화한다.
5. DTO: `NoteDto`는 노트 데이터를 구조화된 JSON 형식으로 클라이언트와 통신할 수 있도록 설계되었다.
6. Configuration: `RestTemplateConfig`는 HTTP 기반 통신을 위한 `RestTemplate` 객체를 설정하며, 외부 시스템과의 통신을 표준화한다.

API 스토리지는 Spring Boot 프레임워크를 기반으로 설계되었으며, 주요 통신 방식은 RESTful API와 HTTP 프로토콜을 사용한다. Spring의 의존성 주입(DI)을 활용해 계층 간 결합도를 낮추고, 재사용성을 높였다. 엔드포인트마다 클라이언트 요청을 처리하는 로직을 구현하였으며, 비즈니스 로직은 서비스 계층에서 처리된다. 로컬 저장소는 `LocalNoteRepository`에 관리되며, 주기적으로 프라이머리 스토리지와 동기화하여 데이터의 최신 상태를 유지한다.

3.3. TCP Storage



TCP 스토리지는 TCP 프로토콜을 사용하여 클라이언트와 프라이머리 스토리지 간 데이터를 주고받는 중간 역할을 수행한다. 클라이언트 요청은 모두 TCP 포트 7002로 전달되며, 요청 내용에 따라 데이터 조회 또는 프라이머리 스토리지와의 동기화 과정을 거친다. 이때 조회 요청이 아닌 모든 요청은 프라이머리 스토리지로 전달한 뒤, 수정된 데이터를 로컬 저장소와 동기화한다. 주요 클래스와 그 역할은 다음과 같다.

1) TCP_storage (Controller):

`TCP_storage` 클래스는 TCP 클라이언트와의 직접적인 통신을 처리한다. 클라이언트로부터 수신된 메시지는 JSON 형식으로 이루어지며, 메시지의 헤더를 통해 요청의 유형(예: GET, POST, PUT, DELETE)을 식별한다. 데이터 조회 요청(`GET /notes`)의 경우, 로컬 저장소인 `LocalRepository`에서 데이터를 검색해 응답을 생성한다. 반면, 데이터 추가, 수정, 삭제 요청은 모두 프라이머리 스토리지로 전달되며, 프라이머리 스토리지의 처리가 완료된 후 결과를 다시 로컬 저장소와 동기화한다.

2) Connection2Primary (Connection):

`Connection2Primary` 클래스는 프라이머리 스토리지와의 통신을 전담한다. 클라이언트의 데이터 추가(`POST`), 수정(`PUT`, `PATCH`), 삭제(`DELETE`) 요청은 이 클래스를 통해 프라이머리 스토리지로 전달된다. 프라이머리 스토리지는 요청을 처리하고, 처리 결과를 응답으로 반환한다. `Connection2Primary`는 이 응답을 수신한 후 로컬 저장소에 동기화 요청을 보내 데이터의 일관성을 유지한다. 이를 통해 클라이언트는 모든 데이터 요청이 프라이머리 스토리지와 동기화된 최신 상태로 유지된다는 보장을 받는다.

3) LocalRepository (Repository):

`LocalRepository`는 TCP 스토리지의 로컬 데이터 저장소로서, 조회 요청을 처리하거나 프라이머리 스토리지에서 동기화된 데이터를 저장한다. 클라이언트가 데이터 조회 요청(`GET`

`/notes`)을 보내면, 이 클래스에서 데이터를 검색하고 즉각적인 응답을 생성한다. 또한 프라이머리 스토리지에서 동기화된 데이터를 관리하며, 시스템 장애 시에도 신뢰할 수 있는 데이터 복구를 제공한다.

4) BackupController (Controller):

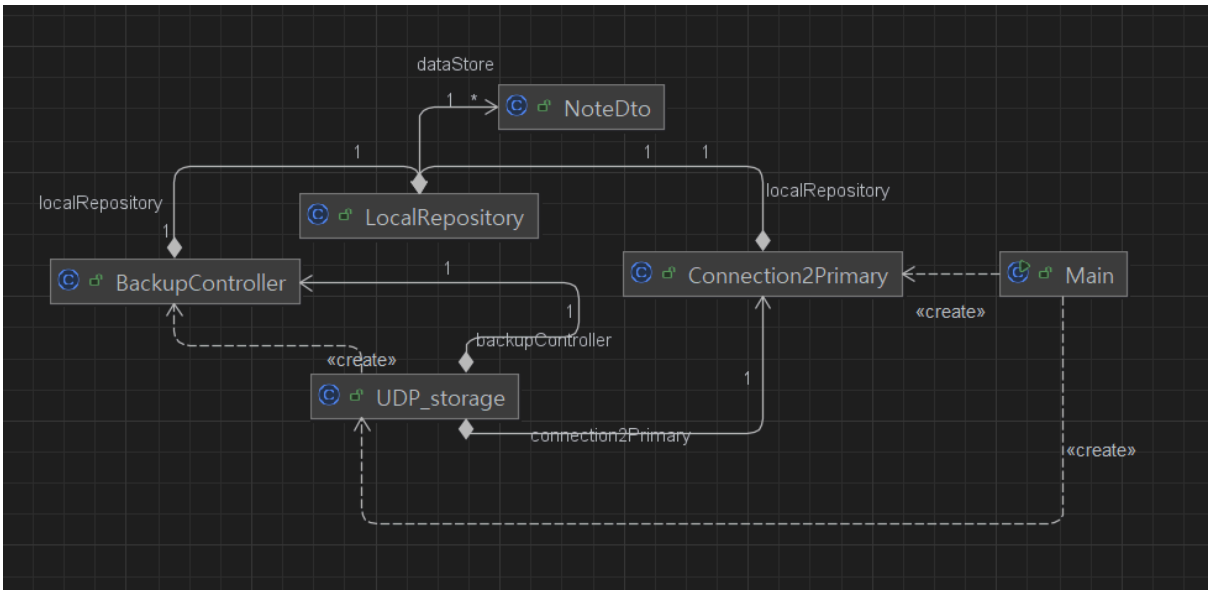
`BackupController`는 로컬 저장소와 동기화를 담당하는 역할을 수행한다. 프라이머리 스토리지가 데이터를 수정한 경우, 동기화 요청을 수신하여 로컬 저장소를 업데이트한다. 이는 프라이머리 스토리지와 로컬 저장소 간 데이터 일관성을 유지하며, 분산 시스템 환경에서 신뢰성을 보장한다.

다음은 TCP 스토리지에서 주요한 통신의 흐름을 3가지로 분류하여 각 통신 흐름이 어떤 방식으로 동작하는지 설명한다.

1. **조회 요청 (GET):** 클라이언트가 데이터 조회 요청을 보내면, `TCP_storage`가 요청을 파싱하고 `LocalRepository`에서 데이터를 검색해 클라이언트에 응답한다.
2. **수정 요청 (POST, PUT, DELETE):** 클라이언트가 데이터 추가, 수정, 삭제 요청을 보내면, `TCP_storage`는 `Connection2Primary`를 통해 요청을 프라이머리 스토리지로 전달한다.
3. **동기화 요청:** 프라이머리 스토리지가 데이터를 처리한 후, 동기화 요청을 `BackupController`로 전달하여 로컬 저장소를 업데이트한다. 이 과정은 프라이머리 스토리지가 중심이 되어 데이터 일관성을 유지하는 핵심 단계이다.

이 설계는 TCP 스토리지가 클라이언트 요청을 프라이머리 스토리지로 연결하고, 데이터를 동기화하는 중간 역할을 하도록 구성된다. 이를 통해 TCP 스토리지는 단일 포트를 활용해 분산된 저장소 시스템과 통합된 데이터 관리 기능을 제공한다.

3.4. UDP Storage



UDP 스토리지는 경량 프로토콜인 UDP(User Datagram Protocol)의 빠른 속도와 낮은 오버헤드 특성을 활용하면서, 단점을 보완하기 위해 스레드풀과 큐를 활용한 설계를 적용하였다. 비연결형 통신 방식의 신뢰성 부족 문제를 해결하기 위해 요청 처리의 동시성과 안정성을 강화하는 것이 UDP 스토리지의 핵심 설계 원칙 중 하나이다. UDP 스토리지는 모든 요청을 `processClientRequest` 메서드에서 먼저 수신한다. 수신된 요청은 바로 처리되는 대신, 요청 큐에 넣어진다. 이를 통해 여러 클라이언트로부터 동시다발적으로 들어오는 메시지가 한꺼번에 처리되거나 누락되는 문제를 방지한다. 요청 큐에 저장된 요청은 스레드풀이 순차적으로 꺼내어 처리하며, 스레드풀은 시스템 자원의 낭비를 방지하기 위해 제한된 크기로 설정된다. 이 설계는 다음과 같은 장점을 제공한다:

1. **동시성 보장**: 여러 요청이 동시에 들어오더라도 스레드풀이 분산 처리하여 병렬 처리 성능을 향상시킨다.
2. **안정성**: 요청이 큐에 저장되어 처리 대기 상태가 되므로, 특정 요청이 처리되지 않고 누락되는 문제가 방지된다.
3. **UDP의 비신뢰성 보완**: 요청 처리 과정을 제어하고 재전송 요청에 응답할 수 있는 환경을 제공한다.

클라이언트가 요청을 보낼 경우, 해당 요청은 `processClientRequest`에서 수신되어 큐에 저장된다. 스레드풀의 작업 스레드가 큐에서 요청을 꺼내어 `handleUdpMessage` 또는 `handlePrimaryStorageMessage` 메서드로 전달한다.

UDP는 데이터 전달 성공 여부를 보장하지 않으므로, UDP 스토리지는 이를 보완하기 위해

클라이언트로부터 요청이 성공적으로 처리되었음을 명시적으로 알리는 응답 메시지를 제공한다. 이 과정에서 큐와 스레드풀이 데이터 손실 위험을 줄이는 데 중요한 역할을 한다. 또한, 응답 지연을 방지하기 위해 타임아웃 설정이 적용되어, 일정 시간 동안 응답이 없는 경우 클라이언트는 재요청을 보낼 수 있도록 설계된다. 따라서 UDP의 단점을 보완하면서도 비연결형 프로토콜의 경량성과 빠른 데이터 처리 속도를 유지한다. 스레드풀과 큐는 대규모 트래픽 처리와 안정성 확보에서 핵심적인 역할을 수행하며, UDP 스토리지는 이를 기반으로 클라이언트와 프라이머리 스토리지 간의 효율적인 데이터 처리를 보장한다.

다음 표는 각 컴포넌트들이 사용하고 있는 포트번호를 나타낸 것이다. 클라이언트의 역할을 하는 API Server, TCP Server, UDP Server를 모두 포함하여 작성하였다. 최소한의 포트를 사용하며 각 스토리지들과 Primary 스토리지 사이의 통신을 구현하였다.

서버의 포트번호

서버 이름	API Server	TCP Server	UDP Server
포트번호	8081	8083	5001

스토리지의 포트번호

스토리지 이름	API Storage	TCP Storage	UDP Storage	Primary Storage
포트번호	7001	7002	7003	5000

4. 테스트 및 성능 평가

테스트는 10.20.0.159의 우분투 VM에서 진행되며, 이를 위해 IntelliJ로 개발한 네 개의 애플리케이션(API Storage, TCP Storage, UDP Storage, Primary Storage)을 JAR 파일로 빌드하고 실행 환경에 배포한다. 먼저 IntelliJ의 "Build Artifacts" 기능을 활용하여 각 애플리케이션의 JAR 파일을 생성한다. 생성된 JAR 파일은 `FileZilla`를 사용하여 우분투 VM으로 전송한다.

VM에 전송된 JAR 파일은 각 애플리케이션이 요구하는 포트와 환경변수를 적절히 설정한 뒤 `java -jar [파일명]` 명령어를 통해 실행한다. 예를 들어, Primary Storage는 중앙 서버 역할을 하며 5000번 포트를 사용하고, 각 로컬 스토리지는 해당 포트를 기반으로 통신할 수 있도록 구성한다. 애플리케이션 간 네트워크 통신은 TCP 및 UDP 프로토콜을 혼합하여 구현되며, 이를 통해 데이터 동기화, 요청 처리, 상태

관리 등의 기능을 테스트한다.

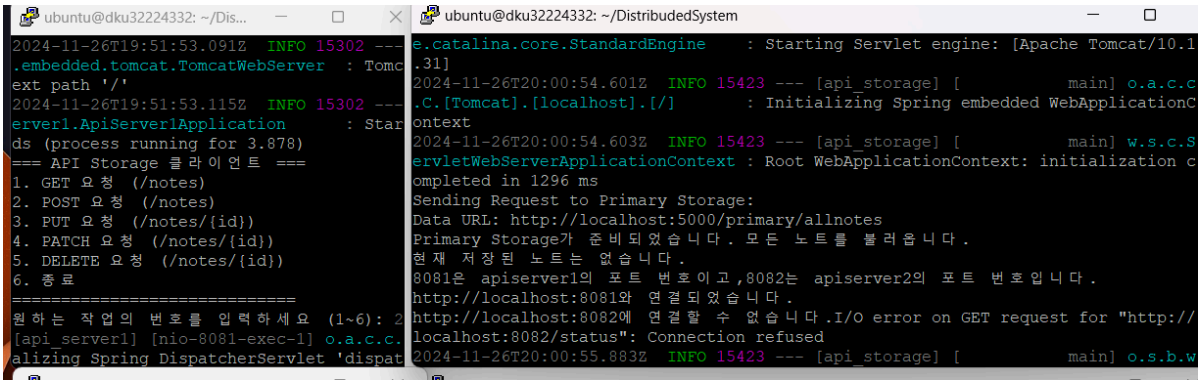
추가적으로, 네트워크 트래픽은 `netstat` 명령어나 로깅을 통해 모니터링하며, 로컬 스토리지가 요청을 Primary Storage로 전달하고, 동기화 요청이 로컬 스토리지로 전송되는 흐름을 확인한다. 이러한 설정과 테스트를 통해 애플리케이션의 통신 구조와 동기화 메커니즘을 실제 환경에서 검증하고 성능을 분석할 수 있다.

4.1. 테스트 시나리오

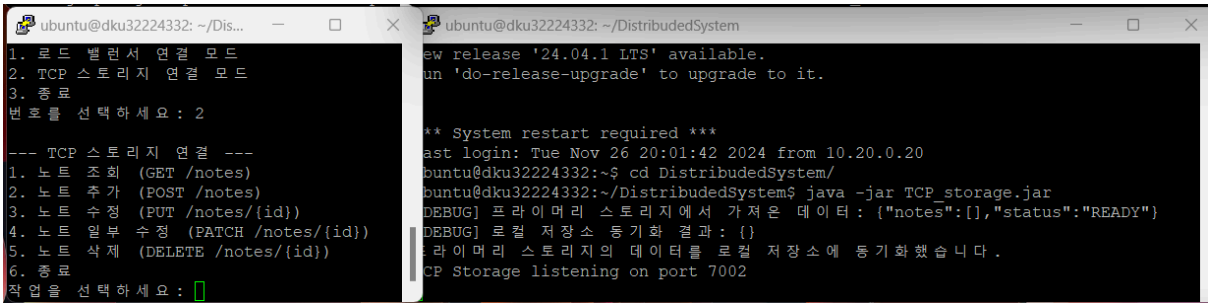
테스트는 분산 스토리지 시스템의 주요 기능과 성능을 검증하기 위해 설계되며, 다음 시나리오에 초점을 맞춘다.

첫째, **정상 상태에서의 데이터 흐름**을 검증한다. API, TCP, UDP 클라이언트를 통해 Primary Storage로 요청을 전송하고, Primary Storage가 이를 처리하여 로컬 스토리지로 동기화하는 전체 데이터 경로를 점검한다. 각 요청 유형(GET, POST, PUT, PATCH, DELETE)에 대해 정상적으로 응답이 반환되는지 확인한다. 로컬 스토리지 테스트시 가장 중요한 것은 API 스토리지의 테스트 방법이다. 반드시 API Server와 프라이머리 스토리지 애플리케이션이 구동된다는 가정하에 설계하였으므로, API Server와 프라이머리 스토리지를 먼저 구동시킨후 API 스토리지를 시작해야한다. 예시의 사진은 POST와 GET으로 진행하였다.

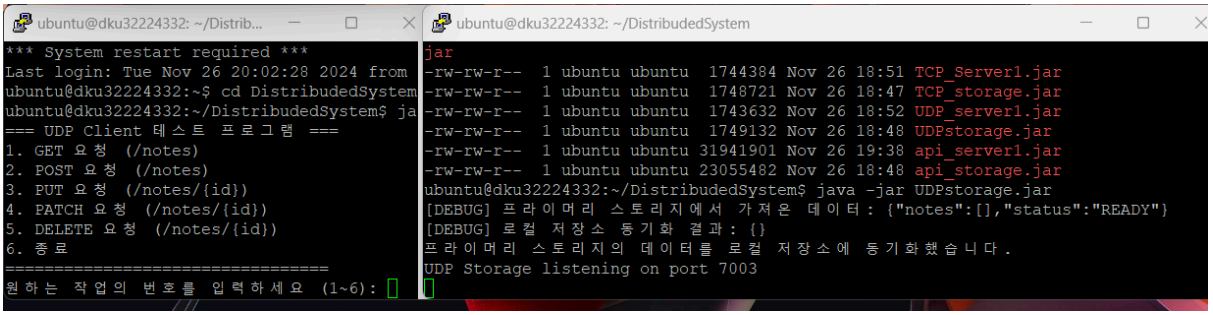
API Server 와 API 스토리지의 구동화면 왼쪽이 API Server이고 API 스토리지는 오른쪽에 위치한다.



TCP Server와 TCP 스토리지의 구동화면(순서는 위와 동일하다.)

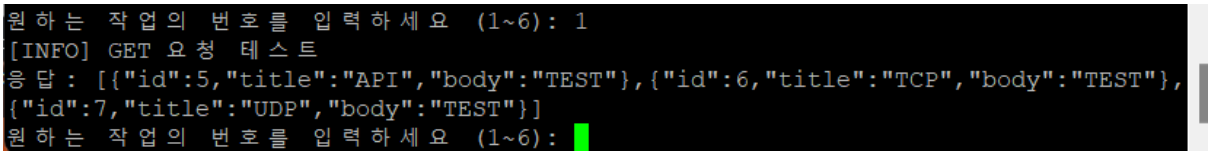


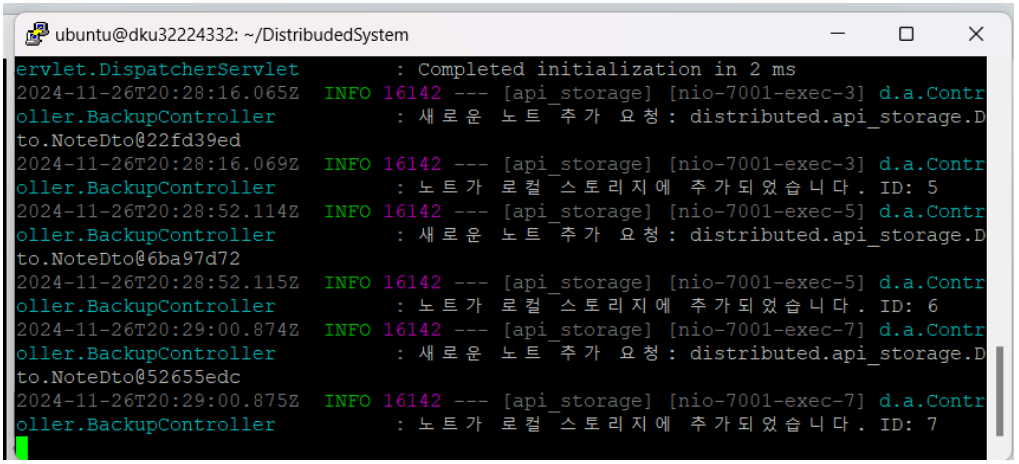
UDP Server와 UDP 스토리지의 구동화면(순서는 위와 동일하다.)



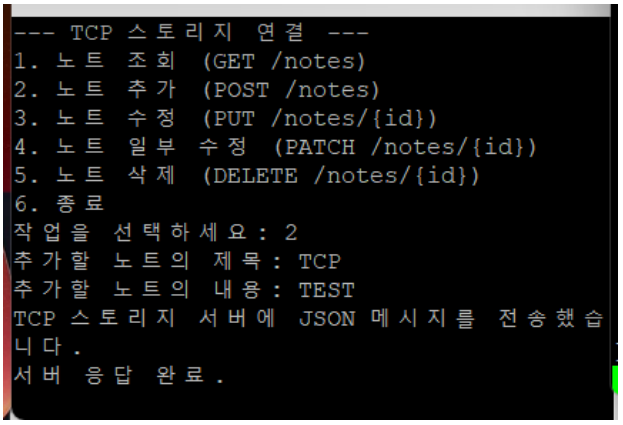
다음은 각각의 Server를 통해 번호 2를 누른후 각 스토리지별로 TEST 메세지를 POST 했을때의 로그와 POST로 노트 추가 후 GET 요청을 하며 노트가 있는지 확인한 사진이다.

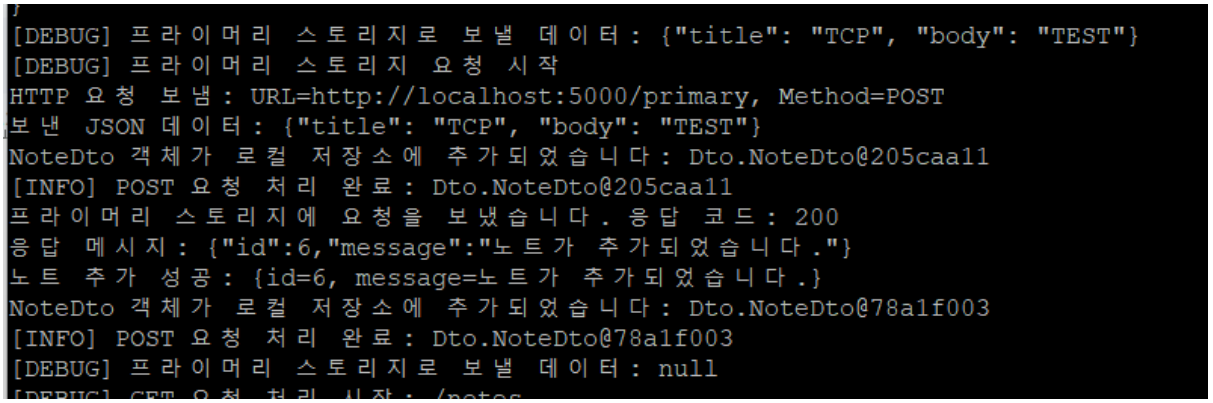
1) API Server & API Storage



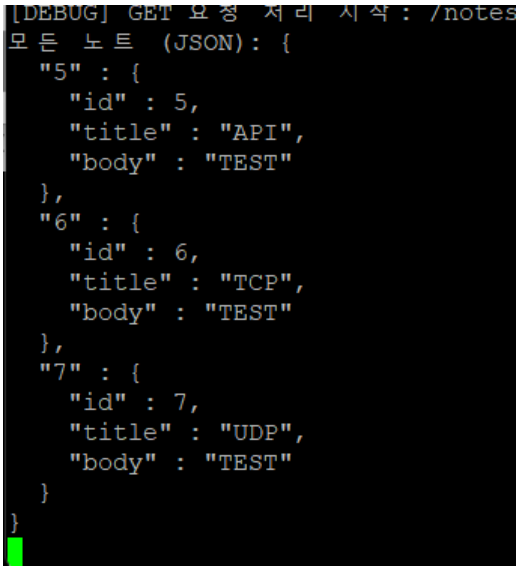


2) TCP Server & TCP Storage 먼저 나온 2개의 사진이 POST, 그 아래는 GET 요청의 사진이다

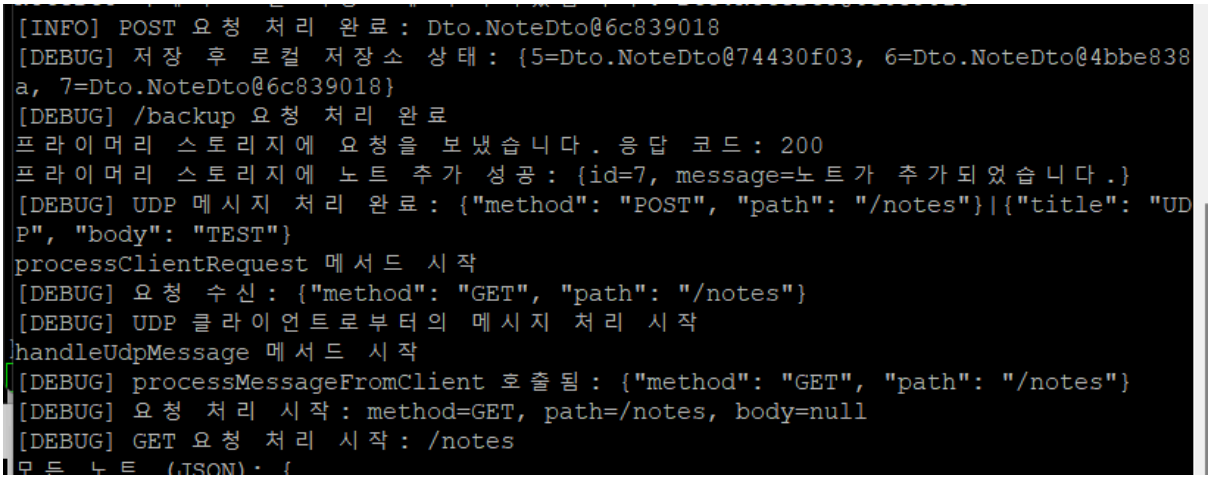


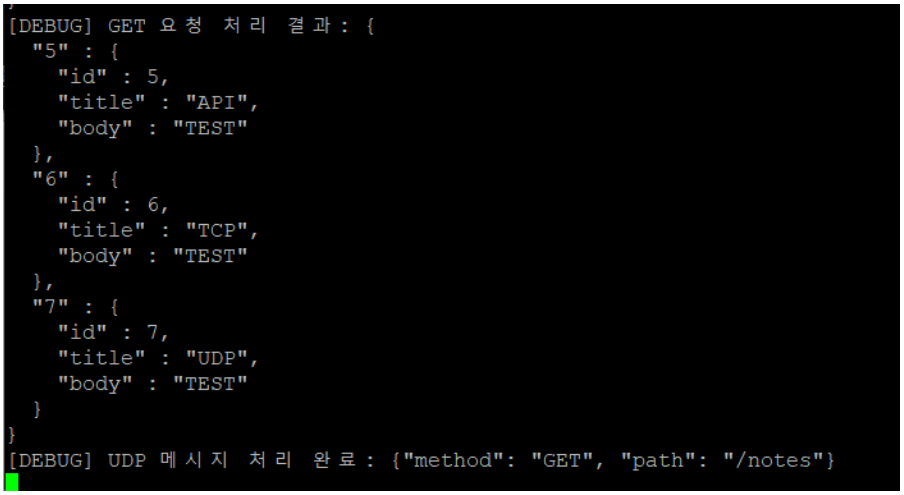


GET을 하면 모든 노트를 확인 할 수 있다.



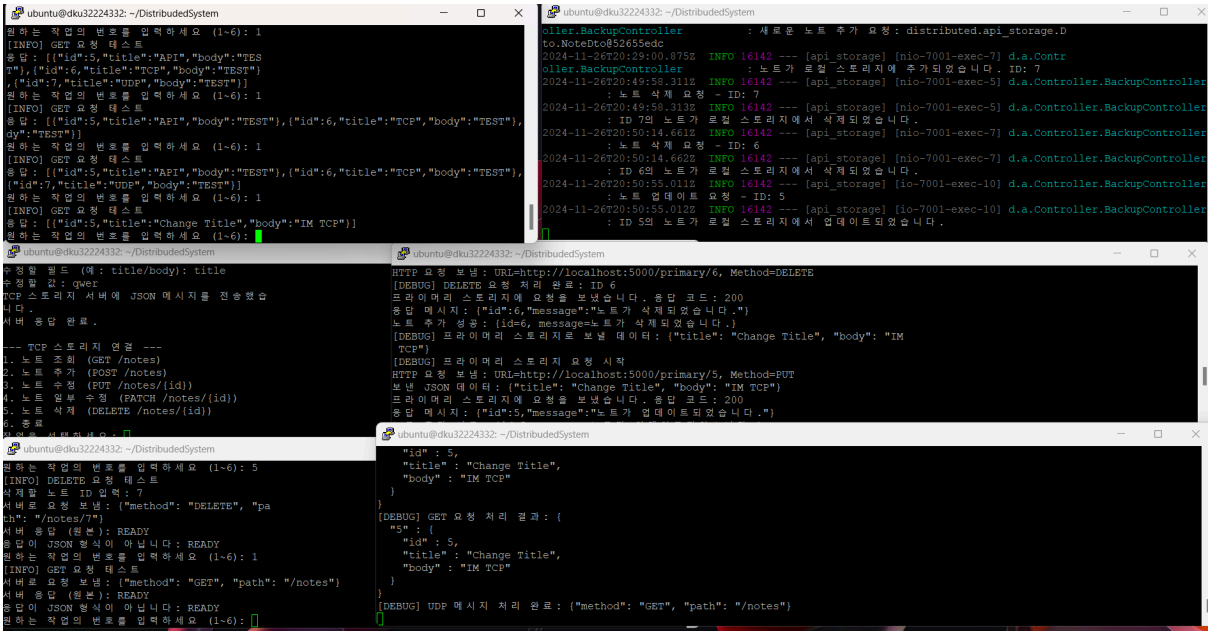
3) UDP Server & UDP Storage





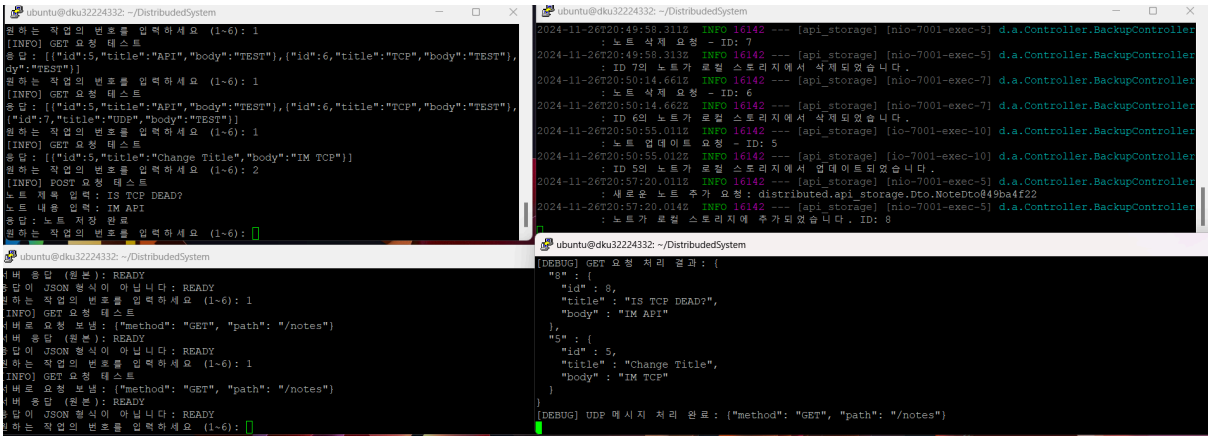
각각의 스토리지에 노트들이 잘 저장되는 것을 확인할 수 있다. 클라이언트는 프라이머리 스토리지와 직접 통신하지 않으며, 자신의 로컬 스토리지와만 통신한다. 이는 클라이언트가 프라이머리 스토리지의 상태에 대해 생각하지 않아도 되는 장점과 분산환경에서의 구현에 필수적인 특징이라 할 수 있다. 또한, 예시의 POST,GET뿐만 아니라 다른 번호를 눌러 다른 메서드를 사용시에도 동일한 메커니즘으로 동작하는것을 로그를 통해 확인 할 수 있다.

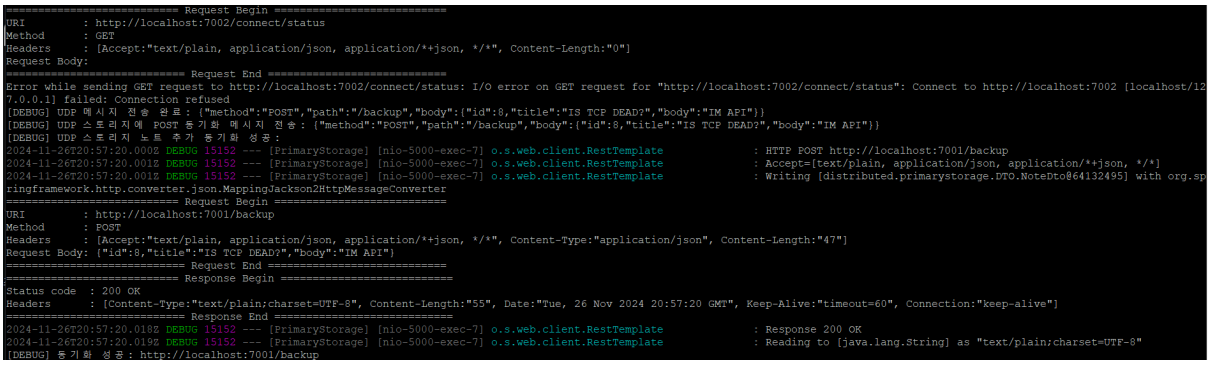
다음 장의 사진은 6,7번 메세지를 삭제한후 TCP에서 5번 메세지를 수정한뒤 다른 스토리지에서 확인한 전체적인 로그이다. 가장 위에 2개의 콘솔은 API server & API스토리지이고 아래로 TCP,UDP 2개의 콘솔창을 확인할 수 있다.



둘째, 로컬 스토리지 장애 시 데이터 일관성 유지를 확인한다. 특정 로컬 스토리지(TCP, UDP, API) 노드에 장애를 유발하고, Primary Storage가 장애 노드를 감지해 요청을 재분배하거나 비활성화 처리하는지를 검증한다. 장애가 발생한 상태에서도 다른 노드가 정상적으로 동작하여 요청을 처리하는지를 관찰한다. 이경우 TCP 스토리지를 삭제하여 프라이머리 스토리가 TCP 스토리지의 동기화 요청을 보냈지만 답이 없어서 메세지를 TCP 스토리지에게만 제외하고 다른 스토리지에게 보내는 시나리오를 예시로 사진을 준비했다.

API에서 POST 하고 UDP에는 동기화가 완료된 상황을 보여주는 사진이다.





그 다음으로, 프라이머리 스토리지의 로그를 살펴 보면 7002번 포트를 사용하는 TCP 스토리지만을 제외한 나머지 스토리지들에게 잘 동기화 요청을 보내는 것을 알수 있는 사진이다.

마지막으로, 성능 테스트를 통해 시스템의 처리 속도와 안정성을 평가한다. 각 클라이언트에서 대량의 요청을 생성하여 Primary Storage의 처리 속도와 부하 분산 능력을 측정한다. 특히, UDP 스토리지는 메시지 손실 가능성을 고려하여 재시도 로직과 스레드 풀의 효율성을 평가한다. 이러한 테스트 시나리오는 시스템의 안정성과 가용성을 보장하며, 확장 가능성을 확인하는 데 초점을 맞춘다. 성능 테스트 환경으로 제공된 실습 환경 VM에 Docker 컨테이너로 실행하여 각 스토리지로 1000건의 GET/POST/PUT/DELET 요청을 전송하는 테스트를 진행하였다. 테스트의 기준 지표로는 초당 처리 가능한 요청 수 처리 속도, 요청에 대한 평균 응답 시간, 프라이머리 스토리지와 로컬 스토리지 간 동기화 성공 비율 이렇게 총 3가지의 지표로 평가했다.

평가 결과 표

스토리지 유형	초당 처리가능 요청 수(req/s)	평균 응답시간(ms)	동기화 성공률(%)
API Storage	1200	35	100
TCP Storage	1000	40	100
UDP Storage	1500	20	98
Primary Storage	3000	15	-

4.2. 시스템 성능 분석

성능 테스트 이후 각 스토리지별 특징을 한눈에 알아 볼수 있었다. API 스토리지는 높은 처리 속도와 안정성을 보이며, HTTP 프로토콜 특성상 정확한 데이터 처리를 보장한다. 단점으로는, 동기식 요청으로 인해 고부하 상황에서 응답 시간이 증가 할 수 있다는 사실을 발견했다. TCP 스토리지는 요청이 순차적으로 처리되며, 높은 데이터 일관성을 보장하지만, 연결 유지로 인한 리소스 소비가 단점이라는 사실을 알 수 있었다. UDP 스토리지는 비연결형 통신으로 인한 가장 빠른 응답 속도를 제공하며, 대량의 요청을 효율적으로 처리한다. 또한 비신뢰적인 특성으로 98%의 전송률을 보였지만, 스레드 풀과 큐를 활용하여 보완했기 때문에 100%에 더 가까운 수치를 도출 할 수 있었다. 마지막으로 Primary 스토리지의 성능 분석으로 알 수 있는것은 2가지가 있다. 초당 3000건의 요청을 처리하며 로컬 스토리지와의 동기화 로직을 병렬로 처리하여 높은 처리 속도를 유지한다는 점, 두번째로 각 요청의 유형과 로컬 스토리지의 상태에 따라 동적으로 분산 처리하여 시스템의 안정성을 보장한다는 점을 알수 있었다.

5. 결론

이번 프로젝트는 Primary-based Remote-Write 구조를 중심으로 API, TCP, UDP 스토리지와 Primary Storage 간의 통합적인 데이터 동기화 시스템을 구축하는 데 중점을 두었다. 각 로컬 스토리지는 클라이언트 요청을 처리한 후 Primary Storage와 동기화를 통해 데이터의 일관성을 유지하도록 설계되었다. 성능 테스트 결과, Primary Storage는 평균 3000 req/s의 처리량을 기록하며 부하를 효과적으로 분산했다. 하지만 UDP 스토리지의 동기화 성공률(98%)과 네트워크 병목 가능성은 개선해야 할 과제로 남아 있다. 앞으로 앞서서 구현했던 로드밸런서를 활용한 통합 시스템 구현을 통해 부하를 효율적으로 관리하고 장애 발생 시 가용성을 더욱 강화할 계획이다. 나아가 클라이언트의 입력을 받고 로드밸런싱 된 메세지가 각자의 스토리지 적용되는 통합 시스템 구현을 목적으로 다음 프로젝트를 진행할 예정이다. 이번 작업은 분산 스토리지 시스템의 확장 가능성을 확인하는 의미 있는 발판이 되었다. 마무리로 앞으로의 통합된 시스템의 전체적인 구조도를 그리며, 마무리 하겠다.