비동기 이벤트 게임 서버

- 프로젝트 주소(github): https://github.com/qorzi/asio_server

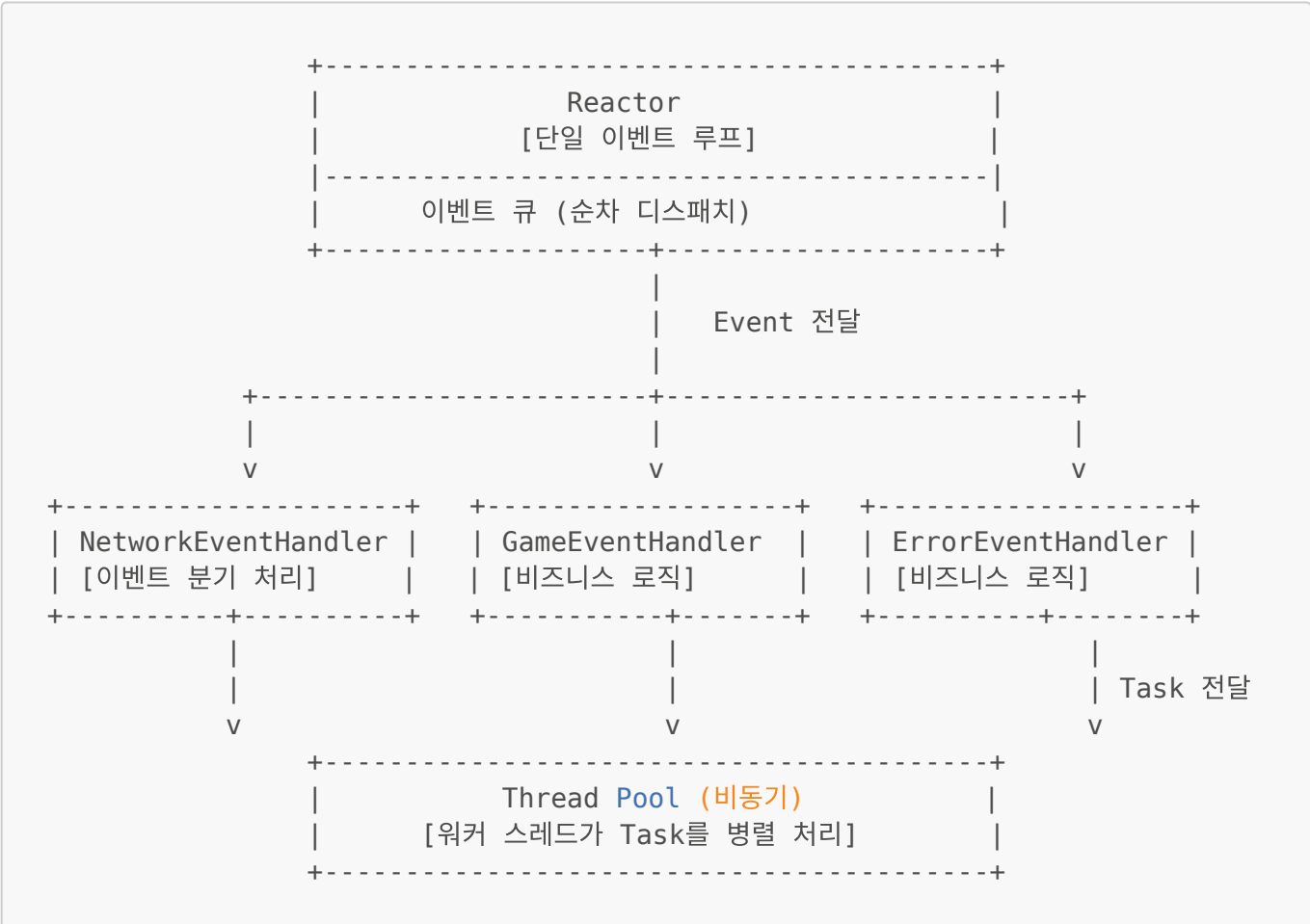
설계 목표

- 대규모 동시 사용자 환경에서 안정적이고 확장 가능한 비동기 이벤트 기반의 게임 서버 구축
- React 패턴 기반의 단일 이벤트 루프를 통해, 앱 내 모든 이벤트를 관리
- 멀티 스레드 및 스레드 풀을 통해, 다양한 이벤트 작업을 동시에 효율적으로 처리하도록 구성

기술 스택

- **C++ 17, Shell, Python:** 프로그래밍 언어
- **Boost.Asio:** 소켓 통신, 타이머 등 OS 기능 활용
- **CMake:** 빌드 도구
- **gtest:** 테스트 도구

서버 구조



- **Reactor**
    - 단일 이벤트 루프를 통해 이벤트 큐에 쌓인 이벤트를 순차적으로 디스패치하며, 각 이벤트를 `NetworkEventHandler`, `GameEventHandler`, `ErrorEventHandler` 등으로 전달
- **Handler**
    - 각 핸들러는 받은 이벤트에 따라 내부 비즈니스 로직을 분기하고, 필요한 Task를 Thread Pool로 전달하여 비동기/병렬 처리를 실행
- **Thread Pool**

  작업 큐에 쌓인 작업을 각 워커 스레드들에게 분배하여 각 Task를 병렬로 처리

이벤트 중심 설계

- **단일 이벤트 구조:** 네트워크, 게임, 에러 등의 이벤트를 주/서브 타입으로 통합 관리

```
enum class MainEventType : uint16_t {
    NETWORK = 1,
    GAME    = 2,
    ERROR   = 3,
    // ... etc
};

enum class NetworkSubType : uint16_t {
    JOIN  = 101,
    LEFT  = 102,
    CLOSE = 103,
    // ... etc
};

enum class GameSubType : uint16_t {
    ROOM_CREATE          = 201, // 방 생성
    GAME_COUNTDOWN       = 202, // 대기화면 후, 카운트다운
    GAME_START           = 203, // 카운트다운=0 → 게임 시작
    // ... etc
};

struct Event {
    MainEventType main_type;      // NETWORK or GAME 등
    uint16_t sub_type;            // 각 메인 타입에 따른 상세 서브 타입
        // ... etc
};
```

- **핸들러 분기:** 각 이벤트는 `NetworkEventHandler`나 `GameEventHandler`처럼 전용 핸들러로 전달되어 서브 타입에 따라 처리

```
void GameEventHandler::handle_event(const Event& event)
{
        ...
    switch (static_cast<GameSubType>(event.sub_type)) {
    case GameSubType::ROOM_CREATE:
        handle_room_create(event);
        break;
    case GameSubType::GAME_COUNTDOWN:
        handle_game_countdown(event);
        break;
    case GameSubType::GAME_START:
        handle_game_start(event);
        break;
        ...
    }
}
```

리액터 패턴 기반 이벤트 루프

- **단일 이벤트 루프:** 이벤트 큐에 들어온 작업들을 순차적(동기적)으로 디스패치
- **일관된 흐름 제어**: 하나의 중앙 루프에서 모든 이벤트를 관리하므로 전체 시스템의 동작 흐름을 쉽게 추적

비동기 및 스레드 처리

- **작업 큐와 스레드 풀 활용**: 이벤트 디스패치 후, 각 작업이 스레드풀 내 워커 스레드에서 비동기(병렬)로 실행되어, 대규모 동시 사용자 상황에서도 높은 성능 보장

상태 저장소 역할의 게임 관련 객체

- **상태 저장소(혹은 데이터 리포지토리) 패턴**: **GameManager** 및 그 하위 객체들(예: Room, Map, GameResult 등)은 단순한 get/set 메서드와 스레드 안전(lock)을 통해 데이터를 저장하고 관리함
- **유지보수와 확장성:** 비지니스 로직(핸들러)과 상태 관리를 명확하게 분리

핵심 장점

- **유지보수 용이성**:
  - 이벤트(네트워크, 게임 등)와 관련 비즈니스 로직이 각각 전용 핸들러에 분리되어 있어, 기능 수정 및 확장이 독립적으로 이루어질 수 있음
  - GameManager 이하의 상태 저장소 구조 덕분에 전체 상태 관리가 단순화됨
- **확장성**:
  - 새로운 이벤트 타입이나 핸들러를 추가하기 쉽고, 리액터 패턴과 스레드풀 덕분에 대규모 동시 사용자 환경에서도 안정적/확장 가능함
- **일관된 흐름 및 고성능**:
  - 앱 내 모든 이벤트가 단일 이벤트 루프로 자연스럽게 반영되어 전체 시스템 동작을 한 곳에서 추적 가능
  - 스레드풀을 통한 비동기 작업 처리 덕분에 각 이벤트 처리 작업이 병렬로 수행되어 응답 시간이 향상됨

통신 구조

- 앱 내부의 이벤트 처리 메커니즘의 일부를 차용한 자체 프로토콜로 TCP 통신

데이터 구조

```
+--------------------------+-----------------------------------+
|       고정 헤더            |              가변 바디               |
+--------------------------+-----------------------------------+
| MainEventType (2 bytes) | JSON 데이터 (가변 길이)              |
| SubEventType (2 bytes)  | - 8바이트 정렬 패딩 처리              |
| Body Length (4 bytes)   |                                   |
+--------------------------+-----------------------------------+
```

- **고정 헤더 (8 byte)**
  - **MainEventType** (2 byte)**:** 이벤트 주 분류
  - **SubEventType** (2 byte)**:** 이벤트 보조 분류
  - **Body Length** (4 byte)**:** 가변 바디의 실제 데이터 크기 (패딩 제외)
- **가변 바디** (가변 길이)
  - **JSON 포맷 데이터:** 이벤트의 구체적인 세부 정보를 담은 JSON 형식의 데이터
  - **8바이트 정렬:** 가변 바디는 8바이트 정렬(padded to 8-byte alignment) 되도록 패딩 처리됨

프로토콜 설계 및 통합 배경

- **내부 이벤트 구조의 활용:**
  - 기존 앱 내에서 이미 사용 중인 이벤트 구조(주/서브 타입 기반)를 그대로 활용하여, 네트워크 통신에도 일관되게 적용
  - 이를 통해, 내부 이벤트 핸들러와 네트워크 통신, 클라이언트에서의 이벤트 구조를 공유하도록 구성하여, 코드 일관성과 유지보수성 향상
- **고정 헤더와 가변 바디 장점:**
  - **고정 헤더:** 일정한 크기를 유지하므로 TCP 스트림 상에서 먼저 읽어 이벤트의 유형과 데이터 길이를 파악, 안정적인 파싱 및 오류 처리를 가능하게 함
  - **가변 바디:** JSON 데이터 형태로 다양한 이벤트 파라미터(예: 플레이어 정보, 게임 상태, 에러 메시지 등)를 구조화하여 전달하며, 8바이트 정렬 패딩으로 데이터 정렬이 보장됨
- **TCP 통신과의 통합:**
  - 고정된 헤더와 정렬된 가변 바디 덕분에, 일정 크기의 패킷 단위로 데이터를 처리하여 파싱 오류 최소화 및 에러 검출에 강점을 부여