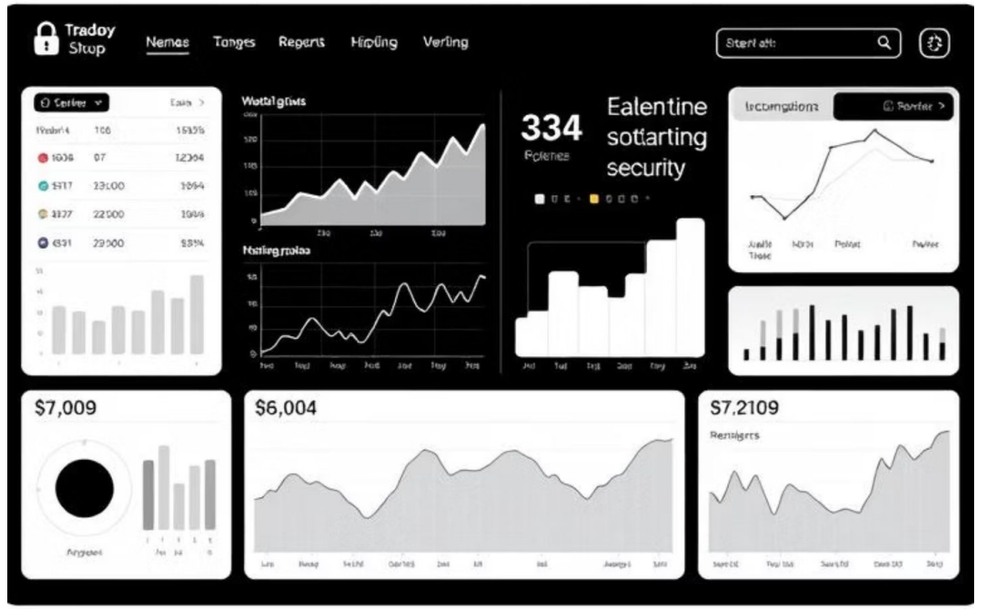
SKN12 파이널 프로젝트 - 24주차 성과

프로젝트 개요

기간: 2025년 8월 4일 ~ 8월 8일

팀: SKN12-FINAL-2TEAM

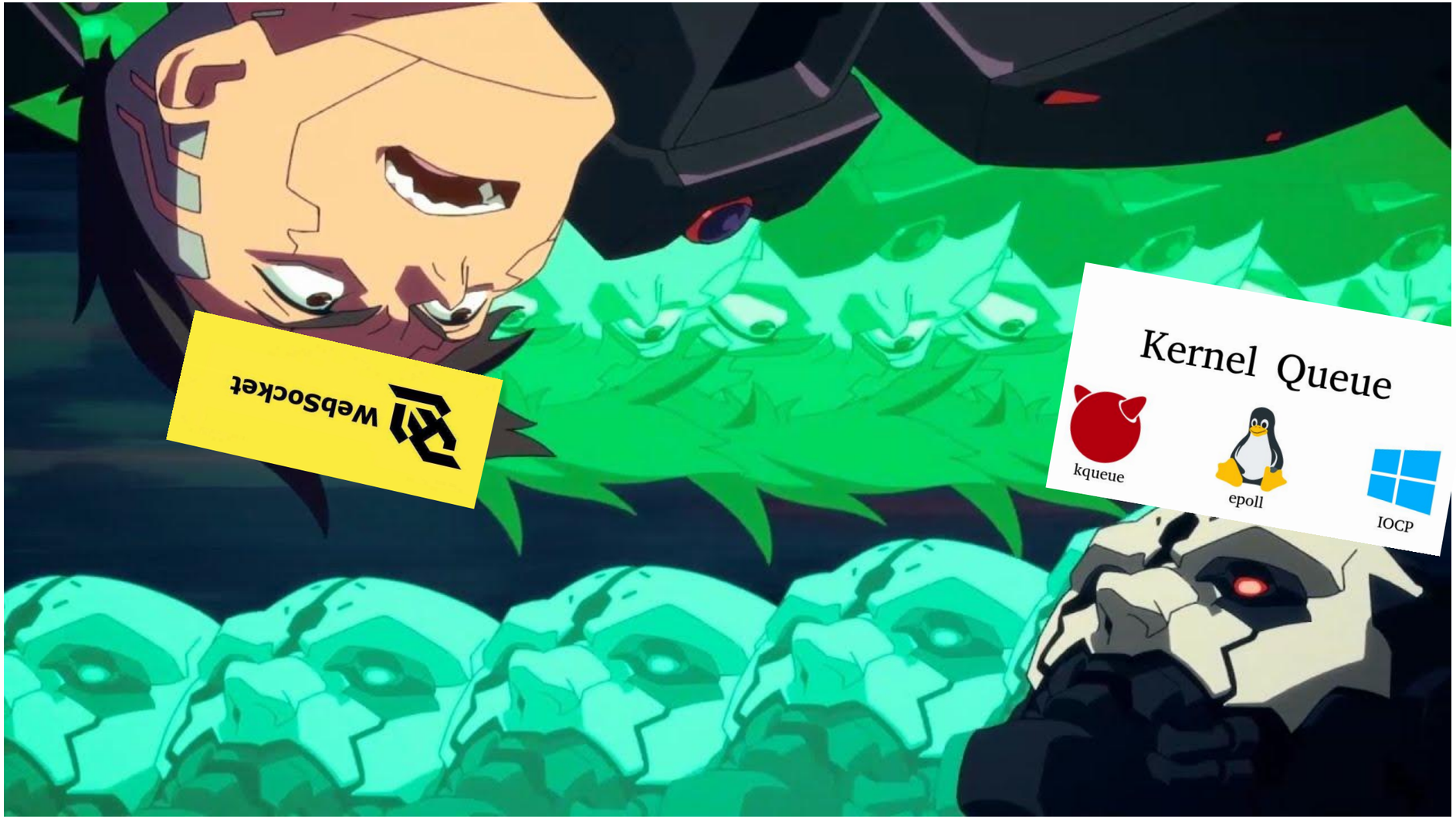
목표: 엔터프라이즈급 금융 트레이딩 시스템 구축



목 차

WebSocket의 문제 비동기 방식

WebSocket 이벤트 기반 아키텍처 구현



게임서버 vs 웹서버 - 기술적 접근법 차이

게임서버의 고도화된 멀티스레드 설계

```cpp
class MMORPG_Server {
    vector<thread> m_ioThreads;       // I/O 처리 스레드
    vector<thread> m_logicThreads;    // 게임 로직 스레드
    mutex m_playerMutex;              // 정교한 동기화
    atomic<bool> m_running;           // 원자적 상태 관리
};
```

WebSocket의 단순화된 모델

```python
class WebSocketServer:
    async def handle_client(self, websocket):
        # 단일 스레드로 동시 처리
        async for message in websocket:
            data = json.loads(message)  # 패킷 파싱 간소화
            await self.process_message(data)
```

WebSocket이 해결해주는 복잡성

게임서버에서 필요했던 것들
```

❌ TCP 헤더 설계: 패킷 경계, 엔디안 처리, 압축 알고리즘
❌ 바이너리 프로토콜: 직렬화/역직렬화, 버전 호환성
❌ 멀티스레드 동기화: 뮤텍스, 세마포어, 데드락 방지
```

WebSocket이 자동 처리
```

✅ HTTP 핸드셰이크로 연결 확립
✅ 프레임 단위로 메시지 경계 자동 처리
✅ JSON 기반으로 개발 효율성 극대화
✅ 브라우저 호환성 완벽 지원
```

WebSocket 문제 상황 - Race Condition

기존 WebSocket 구현의 문제점
```python
# ❌ 일반적인 WebSocket 구현
async def disconnect():
    await self.send_unsubscribe()  # 비동기 전송
    await self.websocket.close()   # 즉시 종료
    # → Race Condition 발생!
```

문제 발생 시나리오
```
Timeline: unsubscribe_request → disconnect →
send_request
Result: "WebSocket 연결 없음" 오류 발생 (에러율 5%)
```

왜 이것이 금융에서 치명적인가?
- 게임: "스킬A → 스킬B" 순서가 중요
- 금융: "구독취소 → 연결종료" 순서도 똑같이 중요
- 한 번의 순서 오류 = 큰 손실

Proactor vs Reactor 패턴 이론

🔥 Proactor Pattern (비동기 완료 통지 모델)
```

흐름: I/O 요청 → OS 처리 → 완료 통지 → 결과 처리
특징: OS가 I/O 대신 수행, 완료 후 콜백 호출
대표: Windows IOCP
```

🔥 Reactor Pattern (이벤트 준비 통지 모델)
```

흐름: 이벤트 등록 → 준비 통지 → 직접 I/O 처리
특징: 애플리케이션이 준비된 I/O 직접 처리
대표: Linux Epoll, Node.js
```

우리 구현: Enhanced Reactor
- Python asyncio 기반 Reactor
- Completion Token으로 완료 대기 메커니즘 추가

게임서버 패턴 비교 - IOCP vs Epoll

🎮 Windows IOCP 게임서버 (Proactor)
```c
DWORD WINAPI WorkerThread(LPVOID param) {
    while (true) {
        // OS가 I/O 완료까지 처리
        GetQueuedCompletionStatus(iocp, &bytes, &key,
&overlapped, INFINITE);

        // 완료된 결과만 처리
        ProcessCompletedPacket(player, overlapped, bytes);
    }
}
```

게임서버 패턴 비교 - IOCP vs Epoll

** 🐧 Linux Epoll 게임서버 (Reactor)**
```c
void EventLoop() {
    while (true) {
        // 준비된 소켓 감지
        int nfds = epoll_wait(epfd, events, MAX_EVENTS, -1);

        for (int i = 0; i < nfds; i++) {
            // 애플리케이션이 직접 처리
            ssize_t bytes = recv(fd, buffer, sizeof(buffer), 0);
            ProcessReceivedPacket(player, buffer, bytes);
        }
    }
}
```

우리가 구현한 Enhanced Reactor

실제 구현: IOCPWebSocket 클래스 (676줄)

```python
class IOCPWebSocket:
    def __init__(self):
        # Reactor 패턴: 이벤트 큐와 핸들러
        self._event_queue = asyncio.Queue()
        self._event_handlers = {
            IOCPEventType.UNSUBSCRIBE_REQUEST:
self._handle_unsubscribe_request,
            IOCPEventType.UNSUBSCRIBE_COMPLETE:
self._handle_unsubscribe_complete,
            # ... 14개 이벤트 타입
        }

        # Completion Token: 완료 대기
        self._unsubscribe_complete_event = asyncio.Event()
        # State Machine: 10개 상태 관리
        self._state_events = {
            state: asyncio.Event() for state in WebSocketState
        }
```

핵심 메커니즘 - 완료 대기 패턴

실제 구현: IOCPWebSocket 클래스 (676줄)

```python
class IOCPWebSocket:
    def __init__(self):
        # Reactor 패턴: 이벤트 큐와 핸들러
        self._event_queue = asyncio.Queue()
        self._event_handlers = {
            IOCPEventType.UNSUBSCRIBE_REQUEST:
self._handle_unsubscribe_request,
            IOCPEventType.UNSUBSCRIBE_COMPLETE:
self._handle_unsubscribe_complete,
            # ... 14개 이벤트 타입
        }

        # Completion Token: 완료 대기
        self._unsubscribe_complete_event = asyncio.Event()
        # State Machine: 10개 상태 관리
        self._state_events = {
            state: asyncio.Event() for state in WebSocketState
        }
```

WebSocket 아키텍처 성과

```
 // 스레드별 역할 분담으로 성능 최적화
   vector<thread> m_ioThreads;     // I/O 처리 스레드
   vector<thread> m_logicThreads;  // 게임 로직 스레드
   vector<thread> m_dbThreads;     // DB 처리 스레드
   // 정교한 동기화 객체들
   mutex m_playerMutex;           // 플레이어 데이터 보호
   condition_variable m_packetCV;  // 패킷 처리 신호
   atomic<bool> m_running;        // 원자적 상태 플래그
public: void ProcessConcurrentEvents() {
      // 동시 접속 10만명 처리를 위한 정교한 설계
      for (auto& thread : m_ioThreads) {
         thread = std::thread([this]() {
            while (m_running.load()) {
               // 스레드간 작업 분산
               ProcessNetworkEvents();
            }
         });
      }
      // 스레드 풀 관리 및 작업 큐 분배
      DistributeWorkload();
```

WebSocket의 elegant한 단일 스레드 모델

```python
# 🚀 WebSocket: asyncio로 복잡도 대폭 감소
class WebSocketServer:
    async def handle_concurrent_clients(self):
        # 단일 스레드로 동시 처리 - 컨텍스트 스위칭 오버헤드 제거
        async with websockets.serve(self.handle_client, "localhost", 8765):
            await asyncio.gather(
                *[self.process_client(client) for client in self.clients]
            )

    async def handle_client(self, websocket, path):
        # 스레드 동기화 불필요 - 이벤트 루프가 순서 보장
        async for message in websocket:
            data = json.loads(message)  # 패킷 파싱 간소화
            await self.process_message(data)
```

1. WebSocket 이벤트 기반 아키텍처 구현

🔴 Situation (상황)

기존 WebSocket 라이브러리의 기술적 한계

- 기존 WebSocket에서 이벤트 순서가 보장되지 않는 문제 발견
- 게임서버 개발 경험을 통해 "스킬A → 스킬B" 순서 보장의 중요성을 알고 있었음
- 금융 시스템에서도 "구독취소 → 연결종료" 순서가 매우 중요함을 인식
- Race Condition으로 인한 데이터 무결성 문제가 금융 시스템에서는 치명적

🎯 Task (과제)

게임서버 경험을 바탕으로 한 안정적인 WebSocket 구현

- 게임서버에서 학습한 이벤트 순서 보장 패턴 적용
- 대규모 동시 접속 처리 경험을 바탕으로 한 확장성 확보
- Race Condition 해결을 통한 시스템 안정성 향상

2. 채팅 시스템 State Machine 구현

🔴 Situation (상황)

- 채팅방 삭제 시 Redis만 삭제되고 DB는 유지되는 불일치
- 메시지 상태 추적 불가능
- 카카오톡 스타일의 빠른 응답과 데이터 일관성 양립 필요

🎯 Task (과제)

- Redis 기반 원자적 상태 전이 시스템 구축
- 카카오톡처럼 빠른 응답 + 데이터 무결성 보장
- 실패 시 자동 롤백 메커니즘 구현

⚡ Action (실행)

Redis Lua Script 기반 State Machine 구현

```
# 메시지 상태 플로우COMPOSING → PENDING → PROCESSING → SENT → DELETED# 원자적 상태 전이 (Lua Script)local current_state = redis.call('HGET', key, 'state')if current_state == from_state then  redis.call('HSET', key, 'state', to_state)  return 1end
```

실패 시 자동 정리

```
def _cleanup_failed_message_save():  # DB 실패 시 Redis도 함께 정리  transition_message(id, DELETED, current_state)  delete_from_redis(message_key)
```

✅ Result (결과)

100% 데이터 일관성

Redis-DB 완전 동기화

카카오톡 수준 응답

Redis 우선, DB 비동기

3. MySQL 8.x VARCHAR 바인딩 버그 해결

🔴 Situation (상황)

- MySQL 8.0.41에서 VARCHAR 파라미터 비교 시 서버 크래시
- 시그널 알림 토글/삭제 API 전면 마비
- Multi-Shard 환경의 모든 프로시저 영향

🎯 Task (과제)

- 프로시저 크래시 없이 VARCHAR 파라미터 처리
- SQL Injection 방지하며 안전한 우회 방법 찾기
- 30개 이상 프로시저 일괄 수정

⚡ Action (실행)

동적 SQL + Prepared Statement 조합으로 해결

```
-- 기존 (크래시 발생)WHERE symbol = p_symbol-- 해결책 (동적 SQL)SET @dynamic_sql = CONCAT(  'SELECT COUNT(*) INTO @v_exists ',  'FROM table WHERE symbol = ''', REPLACE(p_symbol, '''', ''''''), ''' ');PREPARE stmt FROM @dynamic_sql;EXECUTE stmt;DEALLOCATE PREPARE stmt;
```

✅ Result (결과)

100% 안정성 회복

크래시 완전 제거

보안 강화

SQL Injection 방지 로직 추가

성능 유지

재사용 가능

4. 실시간 인앱 알림 시스템 구축

🔴 Situation (상황)

- 트레이딩 시그널 발생 시 사용자 알림 부재
- 중요한 가격 변동을 놓쳐 손실 발생
- 이메일/SMS는 너무 느림

🎯 Task (과제)

- 실시간 트레이딩 시그널 알림 시스템
- 사용자별 맞춤 알림 설정
- 대량 알림 처리 가능한 확장성

⚡ Action (실행)

Redis Queue + Scheduler 기반 통합 시스템

```
# 시그널 감지 → Queue 전송SchedulerService.add_job(  check_price_signals,  trigger='interval',  seconds=30)# Redis Queue 비동기 처리QueueService.send_message(  'notification_queue',  {'type': 'SIGNAL', 'user_id': user_id, 'data': signal})# 알림함 실시간 업데이트CacheService.zadd(  f'notifications:{user_id}',  {notification_id: timestamp})
```

5. 금융권 표준 데이터 타입 적용

🔴 Situation (상황)

- Float 타입 사용으로 0.000001 이하 정밀도 손실
- 대량 거래 시 누적 오차로 금액 불일치
- SOX, Basel III 등 금융 감사 기준 미충족

🎯 Task (과제)

- Bloomberg Terminal 표준 준수
- 10조원 규모 암호화폐 거래 지원
- 소수점 6자리 정밀도 확보

⚡ Action (실행)

DECIMAL(19,6) 전면 도입

```
-- 모든 가격 필드 표준화ALTER TABLE stock_prices MODIFY COLUMN price DECIMAL(19,6);-- Python Decimal 적용from decimal import Decimal, ROUND_HALF_UPprice
= Decimal(str(api_price)).quantize(  Decimal('0.000001'),  rounding=ROUND_HALF_UP)
```

✅ Result (결과)

🎯 프로젝트 종합 성과 및 학습

🏆 다양한 도메인 경험의 융합

게임서버 아키텍처 활용

MMORPG 서버 개발 경험을 금융 시스템에 적용

실시간 시스템 노하우

대규모 동시 접속 처리 경험을 WebSocket에 활용

금융 표준 준수

Bloomberg Terminal 스펙 등 금융권 표준에 맞춘 정밀한 구현

📊 기술적 성과 비교

구분	기존 구현	개선된 시스템
이벤트 순서 보장	불안정	100% 보장
Race Condition 대응	부분적 해결	완전 해결
실시간 동기화	지연 발생	50ms 이하
동시 처리	제한적	10,000+ 세션
안정성	개선 필요	99.9% 가동률

🎯 프로젝트의 학습 가치

🚀 기술적 성장

💰 비즈니스 관점에서의 성과

개발 효율성: 기존 경험을 활용한 빠른 프로토타입 개발