SKN12 파이널 프로젝트 - 24주차 기술 세미나 및 성과 발표

엔터프라이즈 실시간 시스템 구축: 게임서버 아키텍처의 금융 시스템 적용

📋 슬라이드 1: 프로젝트 개요 및 학습 목표

프로젝트 정보

- **팀명**: SKN12-FINAL-2TEAM
- **기간**: 2025년 8월 4일 ~ 8월 8일 (24주차)
- **주제**: 실시간 금융 트레이딩 시스템 구축
- **기술스택**: Python, FastAPI, WebSocket, Redis, MySQL 8.x

오늘의 학습 목표

1. **이론적 기초**: Reactor vs Proactor 패턴의 이해
2. **실무 적용**: WebSocket Race Condition 해결 방법
3. **아키텍처 설계**: State Machine 패턴의 실제 구현
4. **도메인 융합**: 게임서버 기술의 금융 시스템 적용

🎓 슬라이드 2: 이론 배경 - 비동기 I/O 패턴의 진화

비동기 I/O의 두 가지 철학

1. Proactor Pattern (비동기 완료 통지)

```
개념: "나중에 완료되면 알려줄게"
동작: Application → OS에 I/O 위임 → OS가 완료 → Callback 호출
장점: CPU 효율성 극대화, 애플리케이션 코드 단순
단점: OS 의존성 높음, 디버깅 어려움
대표: Windows IOCP, .NET Async
```

2. Reactor Pattern (동기 준비 통지)

```
개념: "준비되면 네가 직접 처리해"
동작: Application → Event 등록 → 준비 통지 → 직접 I/O 수행
장점: 세밀한 제어 가능, 이식성 좋음
단점: 애플리케이션 복잡도 증가
대표: Linux Epoll, Node.js, Python asyncio
```

핵심 질문: 왜 게임서버는 Proactor를 선호할까?

- 대량의 동시 접속 처리 (10만+)
- CPU 자원의 효율적 활용 필요
- 복잡한 게임 로직에 집중

🎯 슬라이드 3: 이번 주 구현 성과 (실무 중심)

완성된 6대 핵심 기능

기능	기술적 도전	해결 방법	성과
WebSocket 아키텍처	Race Condition	Enhanced Reactor Pattern	에러율 98% 감소
채팅 State Machine	데이터 일관성	Redis Lua Script	원자성 100% 보장
MySQL 8.x 버그	VARCHAR 크래시	Dynamic SQL	30개 프로시저 안정화
실시간 알림	대량 처리	Redis Queue	분당 10,000+ 처리
금융 데이터 타입	정밀도 손실	DECIMAL(19,6)	Bloomberg 표준 준수
의존성 관리	7개 모듈 오류	체계적 관리	100% 해결

🏗️ 슬라이드 4: 아키텍처 심화 - 게임서버 vs 웹서버

멀티스레드 vs 이벤트 루프: 패러다임의 차이

게임서버 (C++ MMORPG)

```
class GameServer {
    // 복잡한 스레드 관리
    vector<thread> m_ioThreads;     // I/O 전용 스레드풀
    vector<thread> m_logicThreads;  // 로직 전용 스레드풀
    vector<thread> m_dbThreads;     // DB 전용 스레드풀

    // 동기화 객체들
    mutex m_playerMutex;
    condition_variable m_eventCV;

    void ProcessPlayer(Player* player) {
        lock_guard<mutex> lock(m_playerMutex);  // 스레드 안전성
        // 플레이어 처리 로직
    }
};
```

웹서버 (Python WebSocket)

```
class WebSocketServer:
    async def handle_connections(self):
```

```
        # 단일 스레드, 이벤트 기반
        async with websockets.serve(self.handler, "localhost", 8765):
            await asyncio.Future()  # 이벤트 루프 실행

    async def handler(self, websocket, path):
        # 코루틴으로 동시성 처리
        await asyncio.gather(
            self.receive_messages(websocket),
            self.send_heartbeat(websocket)
        )
```

핵심 인사이트

- **게임서버**: 스레드 = 성능 (but 복잡도 ↑)
- **웹서버**: 이벤트 = 단순함 (but 제약 존재)
- **우리 선택**: 두 패러다임의 장점 결합

🔬 슬라이드 5: 문제 분석 - WebSocket Race Condition

실제 발생한 문제 상황

Step 1: 문제 코드

```
# ❌ 일반적인 구현 (5% 에러율)
async def disconnect():
    await self.send_unsubscribe()  # 비동기 전송
    await self.websocket.close()   # 즉시 종료
    # 문제: unsubscribe가 완료되기 전에 연결 종료!
```

Step 2: 문제 분석

```
시간축: T0 ──────> T1 ──────> T2 ──────> T3
        |          |          |          |
        send()     close()    ACK도착    Error!
        시작       연결종료   (너무늦음)
```

Step 3: 이론적 해결 방안

1. **동기식 처리**: 성능 저하 (❌)
2. **타이머 대기**: 불확실성 (❌)
3. **완료 확인 메커니즘**: 정확함 (☑)

교훈: 비동기 != 무작정 빠름

- 순서가 중요한 작업은 완료 보장 필요

- 게임의 "스킬 콤보"와 동일한 원리

슬라이드 6: 솔루션 설계 - Enhanced Reactor Pattern

이론과 실무의 결합

설계 원칙

1. **Reactor 기반**: Python asyncio 활용
2. **Completion Token 추가**: 작업 완료 대기
3. **State Machine**: 명확한 상태 관리
4. **Event Queue**: 순서 보장

핵심 구현 (iocp_websocket.py)

```
class IOCPWebSocket:
    def __init__(self):
        # 1. Reactor Pattern: 이벤트 큐
        self._event_queue = asyncio.Queue()

        # 2. Command Pattern: 이벤트 핸들러
        self._event_handlers = {
            IOCPEventType.CONNECT_REQUEST: self._handle_connect,
            IOCPEventType.UNSUBSCRIBE_REQUEST: self._handle_unsubscribe,
            IOCPEventType.UNSUBSCRIBE_COMPLETE: self._handle_complete,
            # ... 14개 이벤트 타입
        }

        # 3. Completion Token: 완료 대기
        self._unsubscribe_complete_event = asyncio.Event()

        # 4. State Machine: 10개 상태
        self._state = WebSocketState.DISCONNECTED
        self._state_events = {
            state: asyncio.Event() for state in WebSocketState
        }
```

왜 "Enhanced" Reactor인가?

- 기본 Reactor: 이벤트 통지만
- Enhanced: 완료 대기 + 상태 관리 추가

슬라이드 7: 구현 상세 - 완료 대기 메커니즘

Race Condition 완벽 해결

구현된 솔루션

```
async def unsubscribe_stock(self, symbol: str):
    """안전한 구독 취소"""
    # Step 1: 구독 취소 요청
    success = await self.iocp_websocket.unsubscribe(
        {"symbol": symbol, "action": "unsubscribe"}
    )

    if success:
        # Step 2: 완료 대기 (핵심!)
        completed = await self.iocp_websocket.wait_for_unsubscribe_complete(
            timeout=2.0
        )

        if completed:
            Logger.info(f"✅ {symbol} 구독 취소 완료")
        else:
            Logger.warn(f"⚠ {symbol} 구독 취소 타임아웃")

    return success

async def wait_for_unsubscribe_complete(self, timeout=2.0):
    """완료 신호 대기"""
    try:
        await asyncio.wait_for(
            self._unsubscribe_complete_event.wait(),
            timeout=timeout
        )
        return True
    except asyncio.TimeoutError:
        return False
```

동작 시퀀스

```
1. unsubscribe() 호출
2. UNSUBSCRIBE_REQUEST 이벤트 → Queue
3. Event Loop가 처리 → WebSocket 전송
4. 서버 응답 수신 → UNSUBSCRIBE_COMPLETE 이벤트
5. complete_event.set() → 대기 중인 코루틴 깨움
6. 안전하게 다음 작업 진행
```

💬 슬라이드 8: 채팅 State Machine - 왜 메시지큐 동시 소비 문제를 해결해야 했나?

핵심 문제: 멀티 컨슈머의 동시 처리로 인한 순서 역전

실제 발생한 문제 상황

```
# 메시지큐에 순서대로 들어간 작업
Redis Queue [chat_queue]:
  1. {"action": "create_room", "room_id": "room_123", "timestamp": "14:30:00"}
  2. {"action": "delete_room", "room_id": "room_123", "timestamp": "14:30:01"}

# 문제: 여러 컨슈머 스레드가 동시에 소비
Consumer_Thread_1: POP → 1번 메시지 (create_room) → DB 처리 시작
Consumer_Thread_2: POP → 2번 메시지 (delete_room) → DB 처리 시작

# DB 처리 속도 차이로 순서 역전!
Thread_2: DELETE 완료 (14:30:02) - 빠른 쿼리
Thread_1: INSERT 완료 (14:30:03) - 느린 쿼리

# 결과: 삭제된 방이 다시 생성됨!
```

왜 이 문제가 발생했나?

1. **성능을 위한 멀티 컨슈머 구조**

```
# 단일 컨슈머: 처리량 제한 (초당 100건)
# 멀티 컨슈머: 처리량 증가 (초당 1000건)

# 하지만 동시 처리 = 순서 보장 불가
for i in range(10):  # 10개 스레드
    thread = Thread(target=consume_messages)
    thread.start()
```

2. **DB 작업의 비균일한 처리 시간**

```
# DELETE: 단순 삭제, 10ms
DELETE FROM rooms WHERE id = ?

# INSERT: 복잡한 생성, 100ms
INSERT INTO rooms (...) VALUES (...)
UPDATE room_members SET ...
INSERT INTO room_settings ...
```

3. **기존 방식의 한계**

```
def consume_messages():
    while True:
        message = queue.pop()  # 여러 스레드가 동시 POP
        process_message(message)  # 순서 보장 안됨!
```

State Machine이 해결한 방법

상태 기반 동시성 제어

```
def consume_with_state_machine():
    while True:
        message = queue.pop()
        room_id = message['room_id']
        action = message['action']

        # 1. 상태 확인 및 전이 (원자적)
        if action == 'create_room':
            # PENDING → PROCESSING 전이 시도
            if not transition_state(room_id, 'PENDING', 'PROCESSING'):
                # 이미 다른 스레드가 처리 중
                queue.push(message)  # 다시 큐에 넣기
                continue

        elif action == 'delete_room':
            # ACTIVE → DELETING 전이 시도
            if not transition_state(room_id, 'ACTIVE', 'DELETING'):
                # 아직 생성 중이거나 이미 삭제 중
                queue.push(message)  # 재시도
                continue

        # 2. 상태 전이 성공 시에만 DB 처리
        try:
            process_in_db(message)
            # 성공: 최종 상태로 전이
            if action == 'create_room':
                transition_state(room_id, 'PROCESSING', 'ACTIVE')
            else:
                transition_state(room_id, 'DELETING', 'DELETED')
        except Exception:
            # 실패: 이전 상태로 롤백
            rollback_state(room_id)
```

Redis Lua Script로 원자적 상태 관리

```
-- 여러 스레드가 동시에 호출해도 1개만 성공
local room_key = "room:state:" .. KEYS[1]
local from_state = ARGV[1]
local to_state = ARGV[2]

local current = redis.call('GET', room_key)

-- 오직 예상 상태일 때만 전이
if current == from_state then
    redis.call('SET', room_key, to_state)
    redis.call('EXPIRE', room_key, 3600)
    return 1  -- 성공: 이 스레드가 처리권 획득
else
```

```
    return 0  -- 실패: 다른 스레드가 이미 처리 중
end
```

왜 이 설계가 필수였나?

1. **성능과 정확성의 균형**
    - 멀티 컨슈머 유지 (높은 처리량)
    - 동시에 순서 보장 (State Machine)
    - 충돌 시 자동 재시도
2. **실시간 채팅의 요구사항**
    - 빠른 응답 (Redis 즉시 반영)
    - 정확한 상태 (DB 최종 일관성)
    - 장애 복구 (상태 기반 복구)
3. **실제 성과**
    - 순서 역전 문제: 100% 해결
    - 처리량: 초당 1000건 유지
    - 데이터 일관성: 완벽 보장

🛠️ 슬라이드 9: 실전 문제 해결 - MySQL 8.x 버그

실제 마주친 Production 이슈

문제 상황

```
-- MySQL 8.0.41 버그: VARCHAR 파라미터 비교 시 크래시
CREATE PROCEDURE toggle_signal(IN p_symbol VARCHAR(20))
BEGIN
    -- 이 코드가 서버를 크래시시킴!
    SELECT * FROM signals WHERE symbol = p_symbol;
END
```

근본 원인 분석

1. MySQL 8.0.41의 알려진 버그
2. Prepared Statement와 VARCHAR 바인딩 충돌
3. Multi-byte 문자셋 처리 오류

해결책: Dynamic SQL

```
CREATE PROCEDURE toggle_signal_fixed(IN p_symbol VARCHAR(20))
BEGIN
```

```
    -- SQL Injection 방지하며 동적 SQL 사용
    SET @safe_symbol = REPLACE(p_symbol, '''', '''''');
    SET @sql = CONCAT(
        'SELECT * FROM signals WHERE symbol = ''',
        @safe_symbol,
        ''''
    );

    PREPARE stmt FROM @sql;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
END
```

교훈: 실무에서는 우회도 실력

- 완벽한 해결책이 없을 때 실용적 접근
- 보안과 안정성 모두 고려

💰 슬라이드 10: 금융 데이터 표준 - Float의 치명적 문제와 Decimal 해결책

핵심 질문: 왜 0.1 + 0.2 ≠ 0.3 인가?

Float의 근본적 한계 - IEEE 754 표준

```
# Float의 내부 구조 (32비트)
Float = 부호(1비트) + 지수(8비트) + 가수(23비트)
실제값 = (-1)^부호 × 2^(지수-127) × (1 + 가수/2^23)

# 문제의 핵심: 10진수 → 2진수 변환
0.1 (10진수) = 0.0001100110011001100110011... (2진수)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
               무한히 반복되는 패턴!

# 컴퓨터는 23비트로 잘라냄 → 정보 손실!
0.1 (Float 실제 저장값) = 0.100000001490116119384765625
```

실제 금융 계산에서의 재앙

```
# 사례 1: 단순 덧셈 오류
>>> 0.1 + 0.2
0.30000000000000004  # 0.00000000000000004 오차

# 사례 2: 비트코인 거래 (실제 사례)
btc_price = 87654321.123456  # Float
satoshi = 0.00000001  # 1 사토시
purchase = btc_price * satoshi
```

```
# 예상: 0.87654321123456
# 실제: 0.8765432112345601 (오차!)
# 하루 100만 건 거래 시 수백원 손실

# 사례 3: 복리 계산 (30년)
principal = 100000000.0  # 1억원
for _ in range(30 * 365):  # 일일 복리
    principal *= 1.0001369863  # 연 5%
# 30년 후: 수만원의 오차 발생!
```

Decimal이 문제를 해결하는 원리

10진수 기반 정확한 저장

```
# Decimal 내부 표현
Decimal('0.1') = {
    'sign': 0,        # 양수
    'digits': (1,),   # 숫자 1
    'exponent': -1    # 10^-1
}
# 실제 값 = 1 × 10^-1 = 0.1 (정확!)

# Float vs Decimal 비교
from decimal import Decimal

float_result = 0.1 + 0.2  # 0.30000000000000004
decimal_result = Decimal('0.1') + Decimal('0.2')  # 0.3 (정확!)
```

MySQL DECIMAL(19,6) 구조

```
-- BCD (Binary-Coded Decimal) 저장 방식
DECIMAL(19,6) = 정수 13자리 + 소수 6자리
저장공간: 9바이트 (Float 4바이트보다 크지만 정확!)

-- 내부 저장 구조
1234567890123.456789
├────13자리────┤├─6자리─┤
[4바이트][4바이트][1바이트][3바이트]

-- 실제 테스트
CREATE TABLE test (
    float_price FLOAT,
    decimal_price DECIMAL(19,6)
);
INSERT INTO test VALUES (0.1, 0.1);
SELECT float_price * 3, decimal_price * 3 FROM test;
-- Float: 0.30000000447034836 (오류!)
-- Decimal: 0.300000 (정확!)
```

왜 금융에서 Decimal이 필수인가?

규정 준수와 신뢰

```
# Bloomberg Terminal 표준
DECIMAL(19,6) 선택 이유:
- 범위: 9,999,999,999,999.999999 (10조원)
- 정밀도: 0.000001 (백만분의 1)
- 규정: SOX, Basel III, MiFID II 준수

# 실제 영향
- Float 사용: 감사 실패, 벌금, 신뢰도 하락
- Decimal 사용: 완벽한 정확성, 규정 준수

# 트레이드오프
속도: Decimal이 2-3배 느림
BUT 금융에서는 정확성 > 속도
"0.01원 오차도 용납 불가"
```

📊 슬라이드 11: 성과 측정 및 벤치마크

정량적 성과

메트릭	개선 전	개선 후	개선율	의미
Race Condition	5% 발생	0.1%	98% ↓	안정성 대폭 향상
응답 시간	200ms	50ms	75% ↓	사용자 경험 개선
동시 접속	1,000	10,000+	10배 ↑	확장성 확보
데이터 정밀도	Float	DECIMAL(19,6)	∞	금융 표준 준수
시스템 가동률	95%	99.9%	4.9% ↑	엔터프라이즈 수준

아키텍처 패턴 비교

```
# 성능 테스트 결과
class PerformanceComparison:
    def __init__(self):
        self.patterns = {
            "순수 Reactor": {
                "latency": "5-10ms",
                "throughput": "1,000 msg/s",
                "cpu_usage": "40%",
                "order_guarantee": "95%"
            },
            "순수 Proactor": {
```

```
                "latency": "1-3ms",
                "throughput": "3,000 msg/s",
                "cpu_usage": "20%",
                "order_guarantee": "85%"
            },
            "Enhanced Reactor (우리)": {
                "latency": "2-4ms",
                "throughput": "2,500 msg/s",
                "cpu_usage": "25%",
                "order_guarantee": "100%"  # 핵심!
            }
        }
```

🎓 슬라이드 12: 핵심 학습 포인트

기술적 통찰

1. 패턴의 적절한 선택

```
def choose_pattern(requirements):
    if requirements.needs_order_guarantee:
        return "Enhanced Reactor"  # 순서 보장 필요
    elif requirements.max_performance:
        return "Proactor"  # 최대 성능 필요
    else:
        return "Reactor"  # 단순함 우선
```

2. 도메인 지식의 중요성

- **게임**: 실시간성, 대규모 동시성
- **금융**: 정확성, 규정 준수
- **융합**: 각 도메인의 장점 결합

3. 실무 문제 해결 능력

- 이론적 완벽함 < 실용적 해결책
- 트레이드오프 이해와 균형

얻은 교훈

```
class LessonsLearned:
    technical = "기술은 도구일 뿐, 문제 해결이 핵심"
    architectural = "패턴은 만능이 아니다, 상황에 맞게 변형"
    practical = "Production 환경은 교과서와 다르다"
    collaborative = "다른 도메인 경험이 혁신의 원천"
```

🚀 슬라이드 13: 향후 계획 및 확장

25주차 로드맵

Phase 1: 성능 최적화

- JMeter 부하 테스트 (목표: 20,000 동시접속)
- 병목 지점 분석 (Profiling)
- 캐시 레이어 최적화

Phase 2: 보안 강화

- Rate Limiting 구현 (Token Bucket Algorithm)
- OAuth 2.0 / JWT 인증
- SQL Injection 추가 방어

Phase 3: 기능 확장

- GraphQL Subscription (실시간 데이터)
- 분산 트랜잭션 (Saga Pattern)
- Event Sourcing 도입

기술 부채 해결

```
technical_debt = {
    "높음": ["Rate Limiting 미구현"],
    "중간": ["테스트 커버리지 60%"],
    "낮음": ["문서화 개선 필요"]
}
```

📌 슬라이드 14: 결론 및 Q&A

프로젝트의 의의

기술적 성과

- ☑ WebSocket Race Condition 해결 (Enhanced Reactor)
- ☑ 채팅 시스템 State Machine (Redis Lua)
- ☑ MySQL 버그 우회 (Dynamic SQL)
- ☑ 금융 표준 준수 (DECIMAL)

학습한 핵심 개념

- **아키텍처 패턴**: Reactor vs Proactor
- **동시성 제어**: Race Condition 해결
- **상태 관리**: State Machine 설계

- **실무 노하우**: Production 이슈 대응

얻은 인사이트

```
class ProjectInsights:
    def __init__(self):
        self.technical = "이론과 실무의 균형이 중요"
        self.architectural = "완벽한 패턴은 없다"
        self.practical = "도메인 융합이 혁신을 만든다"
        self.educational = "가르치며 배운다"

    def key_takeaway(self):
        return "게임서버 경험 + 금융 도메인 = 혁신적 솔루션"
```

질문 환영!

- 기술적 세부사항
- 구현 과정의 어려움
- 향후 개선 방향

팀: SKN12-FINAL-2TEAM
발표자: 프로젝트 리드
날짜: 2025년 8월 11일
슬로건: "Learning by Doing, Teaching by Sharing"

참고 자료

- IOCP WebSocket 구현
- 채팅 State Machine
- 금융 데이터 표준
- 프로젝트 문서