# WebSocket 기반 실시간 동기화 서버 성능 비교 연구: 전자 칠판 응용 사례를 중심으로

남우성○ 강선무

dntjd123kr@khu.ac.kr etxkang@khu.ac.kr

# A Comparative Study of WebSocket-Based Real-Time Synchronization Server Performance: Focusing on the Electronic Whiteboard Application Case

**Woosung Nam○ Seonmoo Kang**

**요 약**

실시간 협업 도구의 수요가 증가함에 따라 사용자 간 입력 데이터를 빠르고 정확하게 동기화하는 기술의 중요성이 높아지고 있다. 본 연구는 WebSocket 기반 실시간 동기화 시스템의 성능을 네 가지 서버 프레임워크(Spring WebSocket, Spring STOMP, Node.js, FastAPI)를 대상으로 정량적으로 비교·분석하였다. 전자칠판과 같은 협업 도메인을 가정한 실험 시나리오를 구성하여 TPS, RTT, 메시지 손실률, 자원 사용률 등 주요 지표를 측정하였다. 실험 결과, Spring 기반 서버는 멀티스레드 구조를 바탕으로 고부하 환경에서도 낮은 지연과 안정적인 성능을 보였다. 반면, Node.js와 FastAPI는 단일 스레드 기반 구조로 인해 병목 현상이 발생하였으며, 멀티 워커 및 Redis Pub/Sub 기반 확장을 적용했음에도 구조적 한계가 확인되었다. 이러한 결과는 실시간 협업 시스템 설계 시 서버 구조와 확장 방식에 대한 고려가 성능 확보에 필수적임을 시사하며 프레임워크 선택 및 최적화 전략에 실질적인 기준을 제공할 수 있다.

## 1. 서론

### 1.1 연구 배경

최근 원격 근무, 비대면 교육, 온라인 협업 도구의 확산으로 인해 실시간 동기화 기술의 중요성이 더욱 부각되고 있다. 특히 전자칠판, 협업 문서 편집기, 실시간 채팅 등과 같은 시스템에서는 다수 사용자 간의 입력을 지연 없이 정확하게 동기화하는 기능이 핵심 요소로 작용한다. 이러한 실시간 협업 환경에서는 다양한 통신 프로토콜이 사용되며 그 중 WebSocket은 낮은 지연과 양방향 통신의 장점을 갖춘 핵심 기술로 주목받고 있다.

WebSocket은 클라이언트와 서버 간의 지속적인 연결을 유지하며 실시간 데이터 송수신을 가능하게 하는 프로토콜로 전통적인 HTTP 요청-응답 구조와 달리 서버에서 클라이언트로의 푸시 전송(push communication)이 가능하다는 점에서 차별성을 지닌다 [1]. 또한, WebSocket은 낮은 오버헤드와 연결 상태 유지 특성으로 인해 고빈도 상호작용이 요구되는 실시간 애플리케이션에서 널리 활용되고 있다 [2].

WebSocket 서버는 다양한 언어와 프레임워크로 구현 가능하며, 각 구현체는 이벤트 처리 방식, 세션 및 연결 관리 구조, I/O 모델 등에서 상이한 구조적 특성을 보인다. 이러한 구조적 차이는 RTT(Round Trip Time), TPS(Transaction Per Second), 메시지 손실률, 시스템 자원 사용률(CPU, 메모리, 네트워크 대역폭 등)과 같은 주요 성능 지표에 영향을 미친다.

본 연구는 WebSocket 기술이 높은 실시간성, 저지연성, 브로드캐스트 특성을 요구하는 전자칠판 시스템에 특히 적합하다는 점에 착안하여, 해당 도메인을 가상 시나리오로 설정하였다. 이를 바탕으로 다양한 WebSocket 서버 프레임워크 간의 성능을 동일 조건에서 정량적으로 비교 및 분석함으로써 서버 구조에 따른 효율성과 병목 요인을 규명하고자 한다. 전자칠판은 동시에 다수 사용자에게 상태 동기화 및 이벤트 전파가 요구되는 응용 사례로 WebSocket 기반 시스템의 성능 한계를 평가하기에 적절한 실험 환경을 제공한다 [3].

### 1.2 관련 연구

WebSocket은 REST API, Server-Sent Events(SSE), gRPC 등과 함께 실시간 통신 프로토콜의 대표적인 기술로 분류되며 다양한 비교 연구에서 그 특성과 성능이 분석되어 왔다 [4]. 그러나 다수의 선행 연구는 서로 다른 프로토콜 간의 상대적 성능 차이를 중심으로 진행되었으며 WebSocket 자체의 서버 프레임워크별 성능을 동일 조건에서 정량적으로 비교한 실험 기반 연구는 드문 편이다.

일부 연구는 Java 기반의 WebSocket 구현체 간(Netty,

Vert.x 등) 성능을 비교하거나, 단일 언어 생태계 내에서의 처리 구조 차이를 분석하는 데 그치고 있다 [5]. 반면 실제 개발 환경에서는 Spring(Java), Node.js(JavaScript), FastAPI(Python) 등 서로 다른 실행 환경과 언어 기반의 WebSocket 프레임워크 중에서 선택을 고려해야 한다. 이러한 현실적인 요구에 비해, 서버 구조 간 성능 비교에 대한 실증적 연구는 부족한 상황이다.

결과적으로, WebSocket을 기반으로 실시간 협업 시스템을 구축하고자 하는 개발자 및 연구자에게 있어 서버 프레임워크 선택에 관한 성능 기준이 명확히 제시되지 않은 점은 아키텍처 설계의 비효율성과 성능 저하로 이어질 수 있다.

### 1.3 연구목적

본 연구는 서로 다른 WebSocket 서버 구현체들을 동일한 실험 환경과 조건하에 배치하고, 성능 지표에 기반한 정량적 비교 분석을 수행함으로써 다음과 같은 목적을 달성하고자 한다.

첫째, Spring WebSocket, Spring STOMP, Node.js WebSocket, FastAPI WebSocket 등 대표적인 서버 프레임워크 간의 성능 차이를 정량적으로 분석하여 서버 구조의 특성과 병목 요인을 구체적으로 도출한다.

둘째, 실시간 협업 도구 개발 시 서버 프레임워크 선택에 참고 가능한 객관적 성능 기준과 실용적 가이드라인을 제시한다.

셋째, 향후 Kubernetes 기반의 수평 확장, 장애 복구 메커니즘, 클라이언트 통합 테스트 등 WebSocket 기반 실시간 시스템의 최적화 및 확장 가능성에 대한 후속 연구 기반을 마련한다.

## 2. 실험 환경

### 2.1 테스트 대상 서버 구성

본 연구에서는 WebSocket 기반 실시간 통신을 지원하는 대표적인 서버 프레임워크 네 종류를 대상으로 성능 비교 실험을 수행하였다. 각 서버는 WebSocket 프로토콜을 처리하기 위한 고유한 내부 아키텍처와 이벤트 처리 방식을 가지며, 실험의 공정성을 위해 동일한 기능적 요구사항과 조건 하에서 구현되었다.

#### 2.1.1 Spring WebSocket Server

Spring STOMP 서버는 Spring의 메시지 브로커 연동 기능을 활용한 STOMP 프로토콜 기반 아키텍처로 구성된다. @EnableWebSocketMessageBroker어노테이션을 통해 WebSocket 연결 이후 STOMP 메시지를 처리하며 클라이언트는 특정 topic을 구독하고 서버는 해당 topic으로 메시지를 발행(publish)한다. 내부적으로는 Spring 내장 SimpleBroker를 사용하며 이는 메모리 기반의 경량 브로커로 구성되어 빠른 메시지 라우팅이 가능하다.

#### 2.1.2 Spring STOMP Server

Spring STOMP는 @EnableWebSocketMessageBroker어노테이션을 기반으로 구성된 메시지 브로커 연동형 구조로 WebSocket 연결 이후 STOMP 프로토콜을 통해 메시지를 송수신한다. 클라이언트는 특정 topic을 구독하며, 서버는 해당 topic에 메시지를 publish한다. 내부적으로는 SimpleBroker를 사용하였으며 이는 Spring 자체에 포함된 메모리 기반 브로커이다.

#### 2.1.3 Node.js WebSockt Server

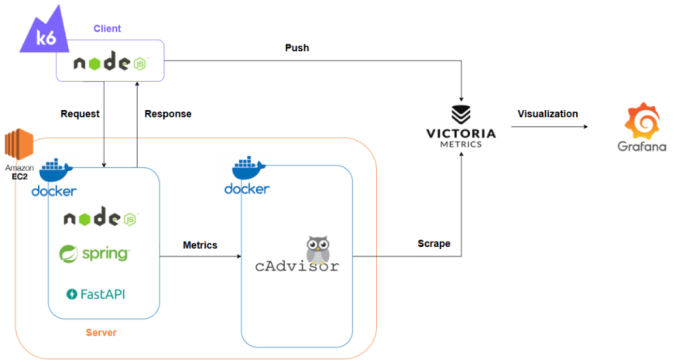
Node.js WebSocket 서버는 ws라이브러리를 기반으로 구현되며, 이벤트 기반 비동기 처리 모델을 채택한다. 모든 연결은 논블로킹 방식으로 처리되며 Node.js의 단일 스레드 이벤트 루프를 활용하여 고속의 I/O 성능과 낮은 지연 시간을 기대할 수 있다.

#### 2.1.4 FastAPI WebSocket Server

FastAPI WebSocket 서버는 Python 기반의 비동기 웹 프레임워크인 FastAPI를 이용하여 구현되었으며, ASGI(Asynchronous Server Gateway Interface)를 통해 WebSocket 프로토콜을 처리한다. 내부적으로는 asyncio를 기반으로 동작하며, 단일 이벤트 루프 내에서 비동기 처리 방식을 수행한다. Python 언어의 구조적 특성상, 높은 동시성 처리에서는 구조적 한계가 존재할 수 있다.

### 2.2 인프라 환경 구성

본 연구에서는 WebSocket 서버의 성능을 공정하고 재현 가능하게 비교하기 위해 Amazon EC2 상에 컨테이너 기반의 실험 아키텍처를 구성하였다. 전체 시스템 구성은 클라이언트 시뮬레이터(k6), WebSocket 서버(Spring, Node.js, FastAPI), 자원 모니터링 도구(cAdvisor), 시계열 데이터 저장소(VictoriaMetrics), 시각화 플랫폼(Grafana)으로 이루어지며 각 구성 요소는 성능 메트릭의 수집, 저장, 시각화를 위한 목적에 맞게 배치되었다. 전체 아키텍처는 그림 1과 같다.




[그림 1] 실험 아키텍처

#### 2.2.1 가상 머신 환경

- 클라우드 플랫폼: Amazon Web Services (AWS)
- 인스턴스 타입: EC2 t3.xlarge
- 하드웨어 사양: 4 vCPU, 16 GiB RAM
- 운영체제: Amazon Linux 2023 AMI
- 가상화 구성: 모든 서버 구성 요소는 Docker 컨테이너로 패키징되어 단일 EC2 인스턴스 상에서 독립 실행되었으며 Docker Compose로 통합 관리되었다.

#### 2.2.2 클라이언트 시뮬레이터

WebSocket 부하 테스트를 위해 k6 도구를 사용하였으며 Node.js 환경에서 실행되는 JavaScript 기반의 테스트 스크립트를 통해 가상 사용자(Virtual Users, 이하 VU)를 생성하였다. 각 VU는 1초 간격으로 서버에 메시지를 송신하고, 서버로부터의 응답을 수신하며 송신 및 수신 시각 간 차이를 기반으로 RTT(Round Trip Time)를 측정한다. 또한 전송 성공률, 메시지 수신률, 손실률 등 분석에 필요한 커스텀 메트릭을 정의하여 VictoriaMetrics로 푸시한다.

### 2.2.3 모니터링 도구

cAdvisor(Container Advisor)는 컨테이너 단위의 CPU 사용률, 메모리 점유량, 네트워크 트래픽 등의 시스템 자원 지표를 실시간으로 수집한다. 수집된 데이터는 Prometheus 호환 방식으로 노출되며 VictoriaMetrics가 주기적으로 이를 scrape하여 저장한다.

### 2.2.4 시계열 데이터 저장소

VictoriaMetrics는 Prometheus 호환 시계열 데이터베이스로 다음 두 범주의 메트릭을 수집 및 저장한다:

- 클라이언트 측 메트릭 (k6 기반): RTT, 전송/수신 메시지 수, 메시지 손실률
- 서버 측 메트릭 (cAdvisor 기반): CPU 사용률, 메모리 사용량, 네트워크 송수신량

### 2.2.5 시각화 플랫폼

Grafana는 VictoriaMetrics에 저장된 메트릭을 기반으로 실시간 대시보드를 구성하며 각 WebSocket 서버의 성능을 시각적으로 비교하고 분석할 수 있도록 한다. 실험 중에는 TPS, RTT, 자원 사용량 등 주요 지표에 대한 실시간 확인이 가능하도록 구성하였다.

[표 1] 수집 지표

| 구분 | 지표명 | 설명 |
|---|---|---|
| 지연 측정 | **RTT (Round Trip Time)** | 클라이언트가 메시지를 전송한 시점과 서버로부터 응답 메시지를 수신한 시점 간의 시간 차이(ms). 평균(AVG), 95번째 백분위수(p95), 99번째 백분위수(p99), 최대값(MAX)을 측정함. |
| 처리량 | TPS (Transactions Per Second) | 초당 처리된 메시지 수. 클라이언트 기준 송수신 합산 기준으로 계산됨. |
| 신뢰성 | 메시지 손실률 | 전송한 메시지 대비 수신된 메시지의 비율을 기반으로 계산된 손실 비율 |
| 시스템 부하 | CPU 사용률 | 서버 컨테이너의 CPU 사용률. EC2 t3.xlarge 기준 최대 4 vCPU(= 400%)까지 측정 가능. |
| | 메모리 사용량 | 서버 컨테이너의 메모리 점유량 (bytes 단위) |
| | 네트워크 트래픽 | 초당 수신/송신 바이트량 (입출력 기준) |

### 2.3 테스트 시나리오

WebSocket 서버의 성능을 정량적으로 비교하기 위해 본 실험에서는 표준화된 메시지 포맷과 동일한 부하 시나리오를 전 서버에 적용하였다.

부하 시나리오는 다음과 같은 방식으로 구성되었다. k6 클라이언트 시뮬레이터가 WebSocket 연결을 통해 서버에 접속하고, 각 가상 사용자(VU)는 1초 간격으로 서버에 메시지를 전송한다. 서버는 메시지 내 포함된 boardId를 기준으로 해당 그룹에 속한 모든 사용자에게 브로드캐스트방식으로 응답을 전송한다.

테스트는 총 5분간 수행되며 1분 단위로 VU 수를 100, 300, 500, 700, 1000으로 점진적으로 증가시켜 부하를 확장하였다. 이와 같은 부하 모델은 전자칠판과 같은 협업 도구 환경에서 실제 발생 가능한 사용 시나리오를 반영한다.

### 2.4 수집 지표

실험에서는 WebSocket 서버의 성능을 정량적으로 비교하기 위해 표 1의 주요 지표를 수집하였다. 각 지표는 실험 대상 서버의 실시간 처리 능력, 메시지 처리 효율성, 신뢰성, 자원 효율성을 종합적으로 평가하는 데 목적이 있다.

## 3. 기본 실험

### 3.1 실험 방법

본 실험에서는 각 WebSocket 서버 프레임워크(Spring WebSocket, Spring STOMP, Node.js, FastAPI)를 서로 독립된 Docker 컨테이너 환경에서 실행하고, 클라이언트는 k6 시뮬레이션 도구를 통해 접속하여 부하를 생성하였다. 클라이언트는 동일한 메시지 포맷과 전송 주기를 기반으로 서버와 실시간 통신을 수행하며 모든 서버는 동일한 조건 하에서 테스트되었다.

### 3.2 성능 측정 결과

#### 3.2.1 Spring WebSocket Server


[그림 2] Spring WebSocket 측정 결과

Spring WebSocket 서버는 실험 전체 구간에서 안정적인 메시지 처리 성능을 보였다. TPS는 VU 수의 증가에 따라 선형적으로 상승하였으며 최대 1000명의 VU 조건에서도 송신 및 수신 메시지 간 손실 없이 정상 처리되었다.

메시지 손실률은 초기 약 3%에서 시작하여 점차 감소하였고, 후반에는 1% 미만 수준으로 수렴하였다. RTT는 평균 약 250ms, p95 및 p99 기준으로 약 500ms 수준에서 유지되었으며 최대 RTT는 2500ms로 관측되었다.

CPU 사용률은 최대 약 50%에 도달하였고, 메모리 사용량은 400MiB에서 450MiB 사이로 일정하게 유지되었다. 네트워크 I/O 측면에서는 수신 대역폭이 5MB/s 미만, 송신은 최대 30MB/s까지 증가하였다.

### 3.2.2 Spring STOMP Server

[그림 3] Spring STOMP 측정 결과

Spring STOMP 서버는 전체 부하 구간에서 일관된 성능 특성을 보였다. TPS는 VU 수 증가에 따라 안정적으로 상승하였으며, 송수신 처리 간 균형을 유지하였다.

메시지 손실률은 초기 약 1.5% 수준에서 점진적으로 감소하였고, 최대 부하 구간에서도 1% 이하로 유지되었다. RTT는 평균 약 100ms, p95는 200ms, p99는 약 500ms, 최대 RTT는 1200ms 수준으로 측정되었다.

CPU 사용률은 최대 50% 내외였으며, 메모리 사용량은 420MiB에서 550MiB 범위에서 점진적으로 증가하였다. 네트워크 송수신 트래픽은 유사하게 증가하였고, 최대 40MB/s 수준까지 도달하였다.

### 3.2.3 Node.js WebSockt Server

[그림 4] Node.js WebSocket 측정 결과

Node.js WebSocket 서버는 전체 실험 구간에서 안정적인 처리량과 낮은 지연 특성을 유지하였다. TPS는 VU 수 증가에 따라 선형적으로 증가하였으며 송수신 처리량 간 큰 편차 없이 일관적으로 동작하였다.

메시지 손실률은 초기 약 3%에서 시작하여 실험 후반부에는 1% 이하 수준으로 안정화되었다. RTT는 평균 약 250ms, p95 및 p99는 약 500ms 수준, 최대 RTT는 3000ms로 측정되었다.

CPU 사용률은 최대 80%에 도달하였으며, 메모리 사용량은 상대적으로 낮은 50~70MiB 범위에서 유지되었다. 네트워크 트래픽은 수신 시 5MB/s 미만, 송신 시 최대 30MB/s까지 도달하였다.

### 3.2.4 FastAPI WebSocket Server

[그림 5] FastAPI WebSocket 측정 결과

FastAPI WebSocket 서버는 동시 사용자 수 증가에 따른 성능 저하가 가장 두드러지게 나타난 사례였다. TPS는 VU 수 500명 이상 구간부터 급격히 불안정해졌으며, 이후 세션 유지 자체가 실패하는 현상이 빈번히 발생하였다.

메시지 손실률은 초기 약 2.5%에서 시작하여, VU 300명까지는 비교적 안정적인 수준을 보였으나, 이후 급격히 증가하여 최대 20%에 도달하였다. 실험 종료 시점의 누적 손실률은 약 10%로 측정되었다.

RTT는 VU 수가 500명 이상일 때 급증하였으며, 평균 약 8000ms, p95 및 p99 기준 약 20000ms, 최대 100000ms까지 상승하였다.

CPU 사용률은 부하 증가에 따라 빠르게 상승하여, VU 500명 조건에서 100%에 도달하였다. 메모리 사용량은 40~120MiB 사이였으며, 세션 연결이 조기 종료되면서 자원 사용량은 제한적으로 유지되었다. 네트워크 트래픽 역시 다른 서버 대비 낮은 수준을 보였으며, 이는 세션 유지 실패에 기인한 것으로 해석된다.

## 3.3 결과 비교 및 분석

### 3.3.1 성능 지표 비교

실험 결과, 전반적으로 Spring 기반 서버(Spring WebSocket, Spring STOMP)는 모든 지표에서 가장 안정적인 성능을 보였다. 두 구현체 모두 TPS가 가상 사용자 수 증가에 따라 선형적으로 증가했으며 메시지 손실률은 초기 약간의 손실을 제외하면 대부분의 실험 구간에서 1% 미만으로 유지되었다. RTT는 평균 100~250ms 수준으로 낮고, 최대 지연 역시 1200~2500ms 범위에 머물러, 실시간 처리에 적합한 수준의 반응성을 유지하였다.

Node.js 기반 서버는 메시지 처리량(TPS)과 손실률 측면에서는 안정적인 결과를 보였으나, CPU 사용률이 최대 80%까지 상승하고, 최대 RTT가 3000ms에 도달하는 등 고부하 환경에서는 상대적으로 자원 소모가 높은 양상을 보였다. 메모리 사용량은 50~70MiB로 가장 낮았게 측정되었다.

반면, FastAPI 기반 서버는 사용자 수가 500명을 초과한 시점부터 세션 유지가 불안정해지며 전체 성능 지표가 급격히 악화되었다. TPS는 불안정했고, 메시지 손실률은 최대 20%, 누적 10%까지 증가하였다. 평균 RTT는 약 8000ms, 최대 RTT는 100,000ms에 이르렀으며, CPU 사용률도 100%에 근접하였다. 네트워크 트래픽 역시 다른

서버 대비 낮은 수준으로, 많은 연결이 유지되지 못했음을 시사한다.

모든 서버에서 초기 단계에서 메시지 손실률이 1~3% 수준으로 존재했으며 이는 WebSocket 연결 직후 메시지 전송 오차 혹은 초기화 오버헤드로 인한 일시적인 수치로서 판단된다.

요약하면 Spring STOMP는 안정성과 성능의 균형이 우수하였으며 Node.js는 경량 구조를 바탕으로 일정 수준까지는 양호한 처리 성능을 보였고, FastAPI는 경량 서버로서의 한계와 구조적 병목으로 인해 고부하 환경에서 성능 저하가 두드러졌다.

이러한 결과는 기존의 WebSocket 서버 성능 벤치마킹 연구와 유사한 경향을 보인다 [6]. 또한 컴파일 언어 기반이 인터프리터 언어 기반보다 높은 처리 성능을 보인다는 기존 연구 결과와도 일치한다 [7].

### 3.3.2 프레임워크 구조적 특성 분석

각 서버의 성능 차이는 단순한 구현 차원이 아니라 프레임워크 내부의 처리 구조, 스레드 모델, 메시지 큐 방식 등 아키텍처의 차이에서 기인하는 것으로 판단된다.

(1) Spring WebSocket

Spring은 Java 기반의 TextWebSocketHandler를 사용하여 브로커 없이 직접 세션과 메시지 흐름을 제어한다. 이 구조는 Java의 멀티스레드 환경과 명시적 스레드 풀 관리에 기반하므로 높은 동시 접속 환경에서도 안정적인 처리를 보장한다.

성능 지표 상에서도 TPS의 선형 증가, 1% 미만의 손실률, 낮은 RTT 등이 나타났으며 이는 직접적 세션 제어와 스레드 기반 병렬 처리가 효과적으로 작동한 결과로 해석된다. 다만, JVM의 고정 오버헤드와 힙 메모리 관리로 인해 메모리 사용량은 상대적으로 높은 편이었다.

(2) Spring STOMP

Spring STOMP는 @EnableWebSocketMessageBroker를 기반으로 동작하며 pub/sub 메시징 모델과 내부 SimpleBroker를 활용해 메시지를 자동으로 라우팅한다. 이 구조는 메시지 흐름을 브로커가 분산 처리하도록 설계되어, 복잡한 전파 구조에서도 안정적인 성능을 유지한다. 이때, 브로커가 내부적으로 ack 및 큐 관리를 수행하기 때문에, 송수신 트래픽이 균형 있게 유지되고, 낮은 RTT와 낮은 손실률을 달성할 수 있었다.

전체적으로 STOMP는 Spring WebSocket보다 다소 높은 메모리 사용률을 보였으나, 메시지 분산 구조로 인해 부하 증가에 대한 내성 및 브로드캐스트 효율이 뛰어났다.

(3) Node.js WebSocket

Node.js 서버는 단일 이벤트 루프 기반의 논블로킹 I/O 모델을 채택한다. 이러한 구조는 경량 처리 및 메모리 효율성에 유리하며, 초기 및 중간 부하 구간에서는 TPS와 RTT 모두 양호한 결과를 보였다.

그러나 브로드캐스트 대상 수가 증가함에 따라 이벤트 큐 적체 현상이 발생하고, CPU 부하가 가중되며, p95/p99 RTT에서 응답 지연이 급격히 증가하였다. 이는 Node.js의 구조적 한계로 스레드 분산 없이 모든 처리가 단일 루프 내에서 순차적으로 처리되기 때문이다.

메모리 사용률은 가장 낮은 수준을 유지하였으나, 이는 메시지 큐 분산이나 세션 복잡도 없이 단일 핸들러로 처리한 결과이며, 고부하 환경에서는 CPU 병목이 더 빠르게 도달하게 된다.

(4) FastAPI WebSocket

FastAPI 서버는 Python 기반의 asyncio비동기 모델을 사용한다. 구조적으로는 고수준에서 비동기 처리를 추상화하여 개발자는 간결하게 구현할 수 있으나, 명시적인 메시지 큐 및 세션 분산 관리 구조는 제공되지 않는다.

이러한 설계는 중간 부하 수준까지는 비교적 정상 작동하지만, 500명 이상의 VU가 연결되는 시점부터 세션 유지 실패, TPS 급락, 손실률 급증등의 현상이 관찰되었다. 이는 코루틴 대기열의 과적, 이벤트 루프 스케줄링 지연, 메시지 브로드캐스트 병목등의 복합적 원인에 기인한다.

네트워크 및 메모리 사용량이 낮은 수치를 보였으나, 이는 성능 효율성 때문이 아니라 다수의 세션이 유지되지 못한 결과이며 실질적 처리 용량 부족을 나타낸다.

## 4. 추가 최적화 실험

### 4.1 최적화 실험 구성

기본 실험 결과에 따르면, Spring 기반 WebSocket 서버는 전 구간에 걸쳐 안정적이고 우수한 성능을 보였다. 이는 해당 프레임워크가 멀티스레드 기반 아키텍처를 채택하고 있으며, Java의 스레드 풀을 통한 병렬 처리가 가능하다는 구조적 이점을 반영한 결과로 해석된다.

반면, Node.js와 FastAPI서버는 공통적으로 단일 스레드 기반의 비동기 이벤트 루프 모델을 중심으로 동작하며, 기본적으로 멀티코어 활용을 지원하지 않는다. 이로 인해 병렬성 부족, 워커 분산 처리 미지원등 구조적 병목이 성능 저하의 원인으로 작용한 것으로 판단된다.

이에 따라 본 연구에서는 Node.js와 FastAPI 서버에 멀티 프로세스 기반 확장 구조를 도입하여 성능 최적화를 시도하였다. 구체적으로는 다음과 같은 개선 사항을 적용하였다:

- 멀티 워커 구성: Node.js에서는 cluster모듈을, FastAPI에서는 Uvicorn workers을 활용하여 멀티 프로세스 분산 처리 구조를 구성함.
- Redis Pub/Sub 메시지 브로커 도입: 각 워커 간 메시지 동기화를 위해 Redis의 Pub/Sub 기능을 통합하여 세션 간 상태 일관성과 메시지 전파를 보장함.

### 4.2 실험 비교 및 분석

#### 4.2.1 Node.js WebSockt Server


[그림 6] Node.js WebSocket 추가 실험 측정 결과

(1) 성능 지표 측정

멀티 워커 구조(cluster) 및 Redis Pub/Sub 기반 메시지 동기화 방식을 적용한 Node.js 서버는 실험 초반에는 비교적 안정적인 성능을 유지하였다. TPS는 초기부터 VU

700명 수준까지 꾸준히 증가하였고, 송수신 모두 안정적인 처리 양상을 보였다. 그러나 VU가 1000명 수준에 도달한 시점부터 성능 저하가 급격히 나타났다.

메시지 손실률은 실험 초반 0~5% 수준에서 시작하여 점진적으로 증가하였고, 최대 부하 구간에서는 90%에 달하는 손실률을 기록하였다. 이는 서버가 메시지를 정상적으로 브로드캐스트하지 못했음을 의미하며 구조적 병목 발생의 직접적인 지표로 해석된다.

RTT는 부하 증가에 따라 선형적으로 증가하였으며, 평균 RTT는 약 20,000ms, p95는 50,000ms, p99는 80,000ms, 최대 RTT는 120,000ms까지 도달하였다. 이처럼 전 구간에서 실시간 통신으로 보기 어려운 수준의 지연이 발생하였다.

자원 사용 측면에서는 CPU 사용률이 최대 80%에 근접하였으며, 이는 병목 원인이 단순히 CPU에만 있지 않음을 시사한다. 메모리 사용량은 실험 내내 500MiB에서 시작해 최대 2000MiB까지 증가하였으며, 이는 Redis 메시지 큐와 워커 간 중복 데이터 수신에 따른 영향으로 분석된다.

(2) 성능 분석

최적화 실험의 초기 기대와 달리 성능이 오히려 저하된 주요 원인은 Redis Pub/Sub 기반 브로드캐스트 방식이 새로운 병목 요소로 작용했기 때문으로 판단된다 [8]. 구체적인 병목 원인은 다음과 같다.

첫째, Redis Pub/Sub는 모든 메시지를 모든 워커에 일괄 브로드캐스트하는 구조를 가지며 워커 수가 증가함에 따라 메시지 복제 오버헤드가 선형적으로 누적된다. 이로 인해 Redis 서버와 네트워크 인터페이스 모두에 과도한 부하가 집중되며, 결과적으로 전체 TPS 한계에 도달하고, 메시지 손실률이 급격히 증가하였다.

둘째, Redis는 싱글 스레드 기반의 이벤트 루프 모델을 채택하고 있어 Pub/Sub 채널의 메시지량이 일정 수준을 초과하면 I/O 병목이 발생한다. 본 실험에서는 채널 처리량의 포화와 함께 TPS 하락, RTT 급등 현상이 동반되었으며 이는 Redis 내부 처리 지연의 직접적인 결과로 해석된다.

요약하면, Node.js 및 FastAPI의 멀티 워커 구조는 기본적인 단일 스레드 병목을 해소하려는 시도로는 유의미했으나, Redis Pub/Sub 중심의 메시지 동기화 방식은 워크 간 실시간 브로드캐스트 처리에 근본적인 구조적 한계를 내포하고 있었다. 결과적으로, 고부하 환경에서 실시간성을 보장하는 데 실패하였으며, 이는 기존 Node.js 클러스터링 기반 성능 연구와 유사한 결과 양상을 보인다 [9].

### 4.2.2 FastAPI WebSocket Server


[그림 7] FastAPI WebSocket 추가 실험 측정 결과

(1) 성능 지표 측정

멀티 워커 구조를 적용한 FastAPI 서버는 일부 지표에서 기본 실험 대비 개선된 결과를 보였다. 그러나 여전히 전체적인 성능은 Spring이나 Node.js 서버보다 낮은 수준에 머물렀으며 이는 구조적 한계와 Redis 기반 병목이 여전히 해소되지 않았기 때문으로 판단된다.

TPS는 실험 초기부터 VU 700명 수준까지는 안정적으로 유지되었으며, 송신·수신 TPS 모두 균형 있게 증가하는 양상을 보였다. 그러나 VU가 1000명에 도달하는 시점에서 TPS가 급격히 하락하며 병목이 발생하였다. 메시지 손실률은 VU 500명 수준부터 점진적으로 증가하기 시작하였고, 최대 부하 구간에서는 90%에 가까운 손실률을 기록하였다. 이는 대부분의 메시지가 정상적으로 전달되지 않았음을 나타낸다.

RTT는 실험 초기에는 안정적인 지연을 유지했으나, 사용자 수 증가에 따라 점차 상승하여 평균 RTT는 약 5000ms, p95 및 p99는 20,000ms, 최대 RTT는 30,000ms까지 증가하였다.

CPU 사용률은 모든 서버 중 가장 높게 측정되었으며, 최대 250% 수준까지 상승하였다. 이는 전체 실험 중 가장 높은 수치이며 두 개 이상의 논리 코어가 포화되었음을 시사한다. 메모리 사용량은 비교적 일정하게 유지되었으며 실험 전 구간에서 180MiB에서 280MiB 사이의 값을 기록하였다.

(2) 성능 분석

FastAPI 서버의 최적화 실험은 기본 실험 대비 분명한 성능 향상을 일부 달성하였으나, 구조적 제약으로 인해 고부하 환경에서는 여전히 병목 현상이 발생하였다. 또한 Node.js 서버와 유사하게 Redis로 기인한 성능 병목의 영향도 존재하는 것으로 판단된다.

## 5 실시간 동기화 시스템 가이드라인

### 5.1 서버별 권장 활용 환경 제시

본 연구에서 실험한 네 가지 WebSocket 서버 프레임워크는 구조적 특성과 성능 지표 측면에서 뚜렷한 차이를 보였다. 이에 따라 각 프레임워크에 적합한 활용 환경과 설계 시 고려사항을 다음과 같이 제안한다.

#### 5.1.1 Spring WebSocket Server

권장 환경:

- 대규모 실시간 협업 시스템
- 정밀한 메시지 제어 및 세션 단위 사용자 관리가 필요한 서비스
- 사용자 정의 브로드캐스트 및 라우팅 로직이 요구되는 시스템

#### 5.1.2 Spring STOMP Server

권장 환경:

- pub/sub 기반 메시지 구조가 필요한 협업 애플리케이션
- 다수의 topic/channel을 대상으로 메시지를 분산 처리하는 서비스
- 프론트엔드와의 SockJS 연동이 요구되거나 Spring MVC와의 통합이 필요한 경우

### 5.1.3 Node.js WebSocket Server
권장 환경:
- 단일 서버 환경에서 동작하는 경량 실시간 서비스
- 메모리 자원이 제한된 상황에서의 메시지 송수신
- 스타트업의 MVP 구성, IoT/센서 기반 데이터 처리 시스템
- JavaScript/TypeScript 기반 풀스택 통합 환경

### 5.1.4 FastAPI WebSocket Server
권장 환경:
- Python 기반 백엔드에서 간단한 실시간 기능이 필요한 경우
- 소규모 프로토타입, 연구용 실험 환경, 내부 도구
- 신속한 개발 및 적용이 요구되는 상황

## 5.2 서버별 최적화 방안 제시
본 절에서는 실험을 통해 확인된 각 서버 프레임워크의 구조적 특성과 병목 지점을 바탕으로 실시간 협업 시스템에서 WebSocket 서버의 성능을 향상시키기 위한 구체적인 최적화 방안을 제시한다.

### 5.2.1 Spring WebSocket Server
Spring WebSocket은 고부하 환경에서도 안정적인 성능을 보였으나, 다음과 같은 구성 최적화를 통해 응답 지연을 추가로 줄이고 처리 효율을 향상시킬 수 있다.
- GC(Garbage Collection) 튜닝: 장기 실행 환경에서는 Java GC 설정을 통해 메모리 안정성과 예측 가능한 지연 특성을 확보할 수 있다.
- 스레드풀 조정: @EnableWebSocket기반 구성에서 TaskExecutor의 스레드풀 크기를 서버 사양에 맞게 설정함으로써 요청 분산 및 응답 시간 개선이 가능하다.
- I/O 버퍼 설정: Netty 또는 Tomcat 기반 설정에서 버퍼 크기를 조정하면 네트워크 처리 효율을 향상시킬 수 있다.

### 5.2.2 Spring STOMP Server
Spring STOMP는 기본적으로 안정적인 pub/sub 메시지 처리를 제공하지만, 확장성과 처리량 향상을 위한 다음과 같은 개선이 가능하다.
- 외부 브로커 연동: 내장 SimpleBroker대신 RabbitMQ, ActiveMQ 등 외부 메시지 브로커를 도입하면 병렬 메시지 처리와 분산 시스템 확장이 가능하다.
- 메시지 필터링 및 압축 적용: 구독 범위가 넓은 시스템에서는 GZIP 압축 및 topic 기반 메시지 필터링을 통해 불필요한 메시지 전송과 대역폭 낭비를 최소화할 수 있다.
- 브로커 병렬성 제어: brokerChannelExecutor설정을 통해 메시지 브로커의 스레드 수를 조정하면 고부하 환경에서 처리량을 개선할 수 있다.

### 5.2.3 Node.js WebSocket Server
Node.js는 이벤트 루프 기반 구조로 인해 확장성 한계가 존재하나, 다음과 같은 최적화를 통해 성능 개선이 가능하다.
- 저수준 네이티브 모듈 도입: bufferutil, utf-8-validate 등 성능 최적화 모듈을 활용하면 메시지 디코딩 속도와 처리 효율이 향상된다.
- Zero-Copy 처리 전략: Buffer.allocUnsafe()등 메모리 복사를 최소화하는 방식으로 메시지 전달 시 처리량을 높일 수 있다.
- 부하 기반 기능 제한: 과부하 상황에서는 연결 제한, 메시지 압축, 샘플링 전송 등 점진적 성능 저하(graceful degradation)를 적용하여 서비스 가용성을 유지한다.

### 5.2.4 FastAPI WebSocket Server
FastAPI는 Python 기반 비동기 프레임워크로 구조적 단순성이 장점이지만, 고부하 환경에서는 병목이 두드러진다. 이를 보완하기 위해 다음과 같은 방안을 고려할 수 있다.
- 멀티 워커 구성: uvicorn workers구조를 통해 수평 확장을 구성한다.
- WebSocket 연결 설정 조정: ping_interval, max_size, read_limit 등 커넥션 관련 파라미터를 조정하여 네트워크 자원 소모를 효율화할 수 있다.
- 비동기 흐름 최적화: asyncio.Queue, asyncio.wait등의 비동기 작업에 대한 코드 수준 튜닝을 통해 처리 지연을 완화할 수 있다.
- Nginx 프록시 적용: 프론트엔드와의 커넥션을 안정적으로 유지하기 위해 Nginx를 WebSocket 프록시로 구성하고, Keep-Alive 및 커넥션 복구 기능을 제공한다.

## 6. 결론
본 연구는 WebSocket 기반 실시간 동기화 서버의 성능을 다양한 프레임워크를 대상으로 동일한 조건에서 정량적으로 비교·분석하였다. 특히 전자칠판과 같은 실시간성과 브로드캐스트 특성이 중요한 협업 도메인을 가상 시나리오로 설정하여, 각 서버 구현체의 구조적 특성과 성능 차이를 실증적으로 평가하였다.

기본 실험 결과, Spring WebSocket과 Spring STOMP는 낮은 RTT, 안정적인 TPS, 수용 가능한 자원 사용률을 바탕으로 고부하 환경에서도 우수한 성능을 유지하였다. 이는 멀티스레드 기반의 구조적 안정성과 프레임워크 수준의 병렬 처리 지원에 기인하며 실시간 협업 시스템에 적합한 기술적 기반을 제공함을 확인할 수 있었다.

반면, Node.js와 FastAPI는 단일 이벤트 루프 또는 단일 코루틴 기반의 구조적 제약으로 인해 고부하 구간에서 메시지 손실률과 RTT가 급격히 증가하는 등 병목 현상이 뚜렷하게 나타났다. 이에 따라 두 서버에 대해 멀티 워커 구조 및 Redis Pub/Sub 기반 메시지 동기화를 도입하여 성능 개선을 시도하였으며, 일부 구간에서는 TPS 향상 등의 개선 효과가 관찰되었다. 그러나 구조적 병목과 Redis 기반 메시지 동기화 방식에서 발생한 추가적인 오버헤드는 실시간성 요구 조건을 충족하는 데 본질적인 한계를 드러냈다.

본 실험은 WebSocket 서버 프레임워크 간의 구조적 차이가 실시간 동기화 성능에 어떠한 영향을 미치는지를 정량적으로 비교한 점에서 의의를 갖는다. 특히 단순한 처리 속도 비교를 넘어, 세션 유지 안정성, 브로드캐스트 메시지 전달 신뢰성, 시스템 자원 활용 측면까지 포괄적으로

분석함으로써 실시간 협업 시스템 설계 시 고려해야 할 핵심 요소들을 구체적으로 제시하였다.

이러한 연구 결과는 개발자 및 시스템 설계자가 도메인 요구에 적합한 WebSocket 서버 구조를 선택하고, 성능 병목을 사전에 인지하며 상황에 따라 필요한 최적화 전략을 효과적으로 수립하는 데 실질적인 지침을 제공할 수 있다. 나아가 본 연구에서 제안된 실험 프레임워크와 측정 지표는 향후 다양한 통신 프로토콜 및 서버 구조에 대한 후속 실험의 기준점으로 활용될 수 있을 것으로 기대된다.

## 7. 향후 연구

본 연구는 WebSocket 서버의 성능 비교 및 구조적 최적화 방안 도출에 초점을 두었으며 다음과 같은 방향으로 확장 연구가 가능하다:

1. 서버 프레임워크 다양화
본 연구는 Spring, Node.js, FastAPI의 네이티브 WebSocket 처리 구조를 중심으로 분석하였으나, 향후에는 Socket.IO, NestJS, Actix, Quart, Elixir Phoenix등 고성능 실시간 서버 프레임워크를 포함하여 비교 범위를 확장할 수 있다. 이를 통해 보다 다양한 기술 환경에 적용 가능한 실용적 선택 기준을 도출할 수 있다.

2. 도메인 특화 시나리오 기반 실험
본 연구는 전자칠판 도메인을 대표 사례로 설정하였으나 실제 협업 툴에서 발생하는 입력 패턴(예: 연속 스트로크, 동시 입력, 단속적 브로드캐스트 등)을 반영한 정밀 시나리오 기반 부하 실험이 요구된다. 이를 통해 도메인 특성에 따른 서버 구조 최적화 전략을 제안할 수 있다.

3. 사용자 체감 기반 통합 실험
서버 성능 지표 외에도 실제 클라이언트 환경(Electron, 웹 앱 등)과의 통합 테스트를 통해 사용자 체감 관점에서 실시간성 평가를 수행할 수 있다. 최종 사용자 중심의 실험은 기술적 수치뿐 아니라 UX 개선과도 직결되는 실용적 인사이트를 제공할 수 있다.

## 참고 문헌


[1] GabrielL.Muller, “HTML5 WebSocket Protocol and Its Application to Distributed Computing,” arXiv preprint, 2014.

[2] 최경수, 이길흥, “웹소켓을 이용한 모바일 앱 테스트 시스템 설계,” 한국IT서비스학회지, 2018.

[3] Łukasz Kamiński, et el. “Comparative Review of Selected Internet Communication Protocols,” Foundations of Computing & Decision Sciensksms ces, 2023.

[4] Mohamed Hassan, “Choosing the Right Communication Protocol for Your Web Application,” arXiv preprint, 2024.

[5] Yukun Wang, et el. “Performance Comparison and Evaluation of WebSocket Frameworks: Netty, Undertow, Vert.x, Grizzly and Jetty,” Proceedings of the 2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN), 2018.

[6] https://github.com/travisluong/python-vs-nodejs-benchmark

[7] Matt Tomasetti, “An Analysis of the Performance of Websockets in Various Programming Languages and Libraries,” SSRN Tech.Rep., 2021.

[8] Hussachai Puripunpinyo, etel. “Design, Prototype Implementation, and Comparison of Scalable Web-Push Architectures on Amazon Web Services Using the Actor Model,” Proceedings of the 2017 25th International Conference on Systems Engineering (ICSEng), 2017.

[9] Eric Johansson, “Using cluster module in Node.js for increased performance,” M.Sc. thesis, Uppsala University (DIVA-portal), 2021.