

WebSocket

WebSocket

📌 HTTP 1.1의 한계

✗ 기존 HTTP의 문제점

HTTP 1.1 이하에서는 다음과 같은 제약사항이 있다:

- **단방향 통신**: 클라이언트의 요청 없이 서버에서 메시지를 보낼 수 없음
- **실시간성 부족**: 메시지를 받았을 때 즉시 감지하지 못함
- **요청-응답 패턴**: 항상 클라이언트가 먼저 요청해야 함

⌚ Polling: HTTP의 해결 시도

💡 Polling이란?

Polling은 HTTP에서 실시간성을 구현하기 위한 방법이다.

Client ————— "새 메시지 있나요?" —————> Server
————— <———— "없습니다"

Client ————— "새 메시지 있나요?" —————> Server
————— <———— "네, 있습니다!"

⌚ 동작 과정

1. 클라이언트가 주기적으로 서버에 요청 전송
2. 서버는 업데이트 여부를 확인하여 응답
3. 업데이트가 있으면 데이터 전송, 없으면 빈 응답

⚠️ Polling의 문제점

문제점	설명	영향
지연 발생	요청 주기만큼의 지연 발생	실시간성 저하 ⏳
불필요한 요청	업데이트가 없어도 계속 요청	트래픽 낭비 📈
서버 부담	지속적인 요청 처리	리소스 낭비 💸

🚀 양방향 통신의 필요성

VS HTTP/2 vs WebSocket

📡 HTTP/2의 한계

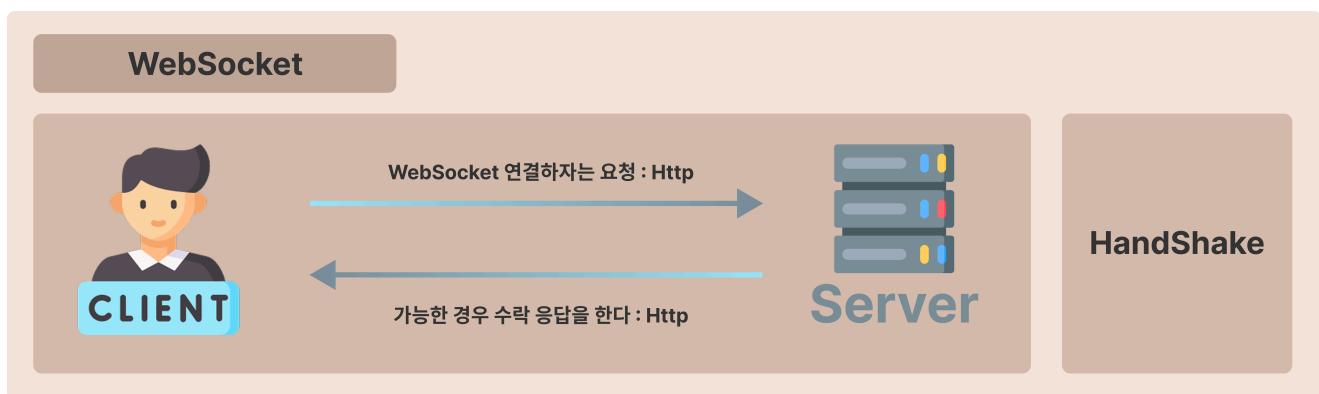
- **설계 목적:** 장시간 양방향 통신을 위해 설계되지 않음
- **Server Push:** 가능하지만 클라이언트의 초기 요청에 대한 응답의 일부
- **브라우저 제한:** 특히 브라우저 사용에는 아직 제한된 부분이 많음

⚡ WebSocket의 등장

| 양방향 실시간 통신을 위한 전용 프로토콜이 필요하게 되었다.

🌐 WebSocket 동작 원리

🔗 연결 설정



| 연결이 설정되면 클라이언트와 서버는 HTTP가 아닌 WebSocket 프로토콜을 사용하여 소통한다.

📊 효율적인 통신



🌟 WebSocket의 장점

- **작은 헤더 크기:** 오버헤드 최소화
- **HTTP보다 효율적:** 불필요한 헤더 정보 제거
- **지속적 연결:** 한 번 연결하면 계속 유지

연결 관리

정상 종료

종료 메시지를 보낼 때까지 연결이 지속된다.



1. 한쪽이 **Close Frame** 전송
2. 다른 쪽이 이를 확인하고 **Close Frame** 응답
3. **연결 종료** 완료

비정상 종료 탐지

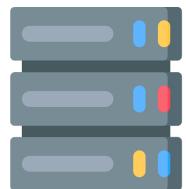
Ping-Pong 메커니즘

- 지정된 시간 동안 메시지가 없으면 확인 패킷 전송
- 주기적으로 Ping-Pong 프레임 교환하여 접속 상태 확인

리소스 효율성

WebSocket은 하나의 연결을 끝까지 유지하고, 그 과정에서 적은 자원만 소요한다. 따라서 Long Polling만큼 서버에 부담을 주지 않는다.

HandShake 과정



CLIENT

Server

HTTP GET Upgrade Request

HTTP 101 Switching Protocol

WebSocket 연결

데이터 주고 받음

7. Application

WebSocket

: TCP Socket 사용한다.

: 따라서 데이터의 순서와 신뢰성 보장한다

6. Presentation

5. Session

4. Transport

TCP Socket, UUP Socket

3. Network

2. Data

1. Physical



클라이언트 요청

```
GET /chat HTTP/1.1
HOST: example.com
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

🔑 핵심 헤더 분석

헤더	의미
Upgrade: WebSocket	HTTP 연결을 WebSocket으로 업그레이드 요청
Connection: Upgrade	연결 업그레이드 지시
Sec-WebSocket-Key	클라이언트가 생성한 랜덤 값 (Base64 인코딩)

보안 키 검증 과정

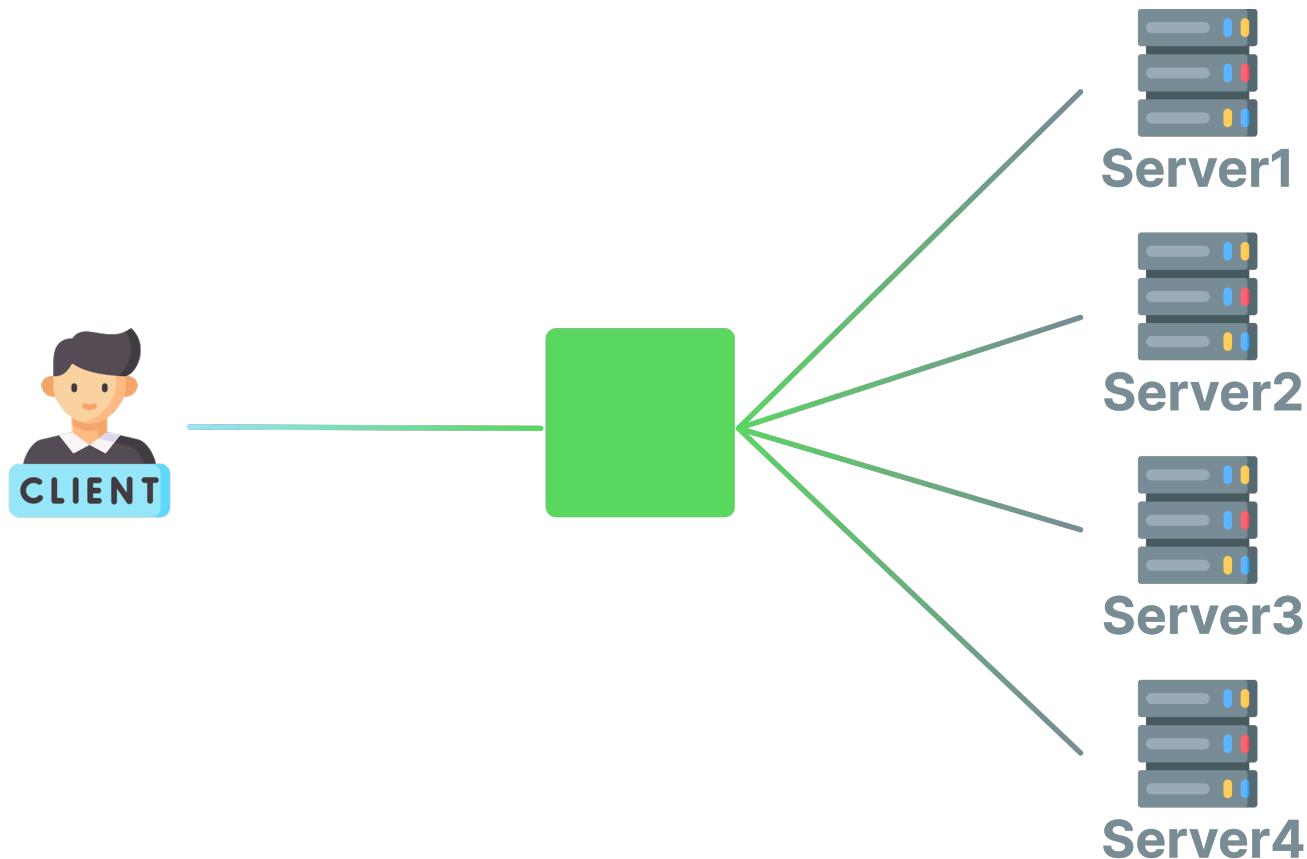
검증 단계

1. 서버가 클라이언트 키 수신
2. GUID 문자열을 키에 이어 붙임
3. SHA-1 해시로 계산
4. Base64로 인코딩하여 응답

WebSocket 한계 및 극복방안

복잡한 서버 설계

로드 밸런싱 문제



문제점: 서버 여러 대가 클라이언트 요청을 나눠서 받는 환경에서 WebSocket은 특정 서버와의 지속적인 연결 안에서만 동작한다.

해결방안

- **Sticky Session:** 한 서버와 통신을 시작하면 계속 그 서버로만 데이터 전송
- 전용 로드 밸런서 사용:
 - NGINX
 - HAProxy
 - AWS ELB

- 기타 WebSocket 처리 가능한 로드 밸런서

2 메시지 크기 제한

제약사항

- **브라우저별 제한**: 각기 다른 메시지 크기 제약
- **서버 환경별 제한**: 서버 설정에 따른 제약
- **네트워크 환경**: 네트워크 인프라의 제약

해결방안

- **데이터 분할**: 대용량 데이터를 작은 단위로 분할하여 전송
- **다른 프로토콜 사용**: 파일 전송 등에는 HTTP 사용
- **압축 사용**: 데이터 압축으로 크기 최적화

3 보안 문제

WS 프로토콜의 한계

WS (기본 WebSocket 프로토콜)는 통신이 암호화되지 않는다.

보안 강화 방법

- **SSL/TLS 인증서 획득**
- **WSS 프로토콜 사용** (`wss:// URL`)
- **추가 인증 계층 구현**

WS (비암호화) → `wss://example.com/socket`

WSS (암호화) → `wss://example.com/socket`

WebSocket vs 다른 기술 비교

특징	WebSocket	HTTP Polling	Server-Sent Events
양방향 통신	✓	✗	✗ (단방향)
실시간성	⚡ 높음	⌚ 낮음	⚡ 높음
서버 부담	🟢 낮음	🔴 높음	🟡 보통
브라우저 지원	✓ 우수	✓ 완벽	✓ 우수
구현 복잡도	🟡 보통	🟢 쉬움	🟢 쉬움



마무리

WebSocket은 실시간 양방향 통신이 필요한 현대 웹 애플리케이션에서 필수적인 기술이다.

🎯 핵심 요약

- **실시간성:** HTTP Polling의 한계를 극복
- **효율성:** 지속적 연결로 오버헤드 최소화
- **양방향:** 클라이언트와 서버 모두 언제든 메시지 전송 가능

⚡ 성공적인 도입을 위한 고려사항

1. **로드 밸런싱 전략 수립**
2. **보안 (WSS) 적용**
3. **메시지 크기 관리**
4. **연결 상태 모니터링**

[gRPC](#)