v7.n11.2026

REVISTA CIENTÍFICA - RECIMA21 ISSN 2675-6218

DESENVOLVIMENTO DE UM SISTEMA DE COMUNICAÇÃO DE VOZ EM TEMPO REAL UTILIZANDO WEBRTC E SPRING BOOT

DEVELOPMENT OF A REAL-TIME VOICE COMMUNICATION SYSTEM USING WEBRTC AND SPRING BOOT

DESARROLLO DE UN SISTEMA DE COMUNICACIÓN DE VOZ EN TIEMPO REAL UTILIZANDO WEBRTC Y SPRING BOOT

Henry Guidelli[1]; Felipe Diniz Dallilo[2]

e7119511
https://doi.org/10.47820/recima21.v7i11.9511

PUBLICADO: 11/2026

RESUMO

Este trabalho apresenta o desenvolvimento e a verificação de estabilidade de um protótipo de sistema *web* de comunicação de voz em tempo real, fundamentado no desacoplamento entre o plano de controle de sinalização e o plano de dados de mídia. O estudo caracteriza-se como pesquisa aplicada, exploratória e descritiva, orientada ao desenvolvimento tecnológico e acompanhada de verificação funcional e testes. A aplicação foi implementada em arquitetura cliente-servidor, na qual o plano de controle (servidor) adotou o ecossistema *Java 21* com *Spring Boot* 4.1.1 utilizando *threads* virtuais para o roteamento leve via *WebSocket* Seguro. O plano de mídia (cliente) adotou o padrão *WebRTC* em topologia de malha direta *P2P*, integrando o *codec Opus* com parametrização dinâmica. A verificação contemplou a coleta de métricas de rede em salas de 2 a 8 integrantes e testes de estresse com o *Grafana K6*, totalizando 4.900 usuários virtuais em 980 salas, monitorados via *Java Flight Recorder* e *JDK Mission Control*. Os resultados indicaram que as *threads* virtuais mantiveram a estabilidade do servidor com 100% de entrega em 203.041 mensagens, reduzindo em 82,82% a criação de *threads* nativas do sistema operacional e em 65,49% os bloqueios de suspensão, enquanto a malha *P2P* assegurou latência inferior a 10 ms em rede local e teto de subida de 216 *kbps* para até quatro nós, comprovando a saturação de tráfego para grupos maiores. Conclui-se que o protótipo demonstrou viabilidade e estabilidade técnica para a comunicação descentralizada. A avaliação limitou-se a redes locais, sem a inclusão de testes com usuários finais.

PALAVRAS-CHAVE: WebRTC. Java. Virtual Threads. Spring Boot. Engenharia de Software. Comunicação em Tempo Real.

ABSTRACT

This work presents the development and stability verification of a real-time web voice communication system prototype, grounded in the decoupling between the signaling control plane and the media data plane. The study is characterized as applied, exploratory, and descriptive research, focused on technological development and accompanied by functional verification and testing. The application was implemented in a client-server architecture, where the control plane (server) adopted the Java 21 ecosystem with Spring Boot 4.1.1 using virtual threads for lightweight routing via Secure WebSocket. The media plane (client) adopted the WebRTC standard in a direct P2P mesh topology, integrating the Opus codec with dynamic parameterization. Verification covered network metrics collection in rooms with 2 to 8 members and stress testing with Grafana K6, totaling 4,900 virtual users across 980 rooms, monitored via Java Flight Recorder and JDK Mission Control.

[1] Universidade de Araraquara, Graduando em Sistemas de Informação.
[2] Universidade de São Paulo, Doutor em Engenharia de Software.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

The results indicated that virtual threads maintained server stability with 100% delivery across 203,041 messages, reducing native operating system thread creation by 82.82% and suspension blocks by 65.49%, while the P2P mesh ensured latency below 10 ms in a local network and an upload ceiling of 216 kbps for up to four nodes, proving traffic saturation for larger groups. It is concluded that the prototype demonstrated technical feasibility and stability for decentralized communication. The evaluation was limited to local networks, without the inclusion of end-user testing.

KEYWORDS: *WebRTC. Java. Virtual threads. Spring Boot. Software Engineering. Real-Time Communication.*

RESUMEN
Este trabajo presenta el desarrollo y la verificación de estabilidad de un prototipo de sistema web de comunicación de voz en tiempo real, fundamentado en el desacoplamiento entre el plano de control de señalización y el plano de datos de medios. El estudio se caracteriza como investigación aplicada, exploratoria y descriptiva, orientada al desarrollo tecnológico y acompañada de verificación funcional y pruebas. La aplicación se implementó en arquitectura cliente-servidor, donde el plano de control (servidor) adoptó el ecosistema Java 21 con Spring Boot 4.1.1 utilizando virtual threads para el enrutamiento ligero vía WebSocket Seguro. El plano de medios (cliente) adoptó el estándar WebRTC en topología de malla directa P2P, integrando el códec Opus con parametrización dinámica. La verificación contempló la recolección de métricas de red en salas de 2 a 8 integrantes y pruebas de estrés con Grafana K6, con 4.900 usuarios virtuales en 980 salas, monitoreados vía Java Flight Recorder y JDK Mission Control. Los resultados indicaron que las virtual threads mantuvieron la estabilidad del servidor con 100% de entrega en 203.041 mensajes, reduciendo en 82,82% la creación de hilos nativos del sistema operativo y en 65,49% los bloqueos de suspensión, mientras que la malla P2P aseguró latencia inferior a 10 ms en red local y límite de subida de 216 kbps hasta cuatro nodos, comprobando la saturación de tráfico para grupos mayores. Se concluye que el prototipo demostró viabilidad y estabilidad técnica para la comunicación descentralizada. La evaluación se limitó a redes locales, sin pruebas con usuarios finales.

PALABRAS CLAVE*: WebRTC. Java. Virtual threads. Spring Boot. Ingeniería de Software. Comunicación en Tiempo Real.*

INTRODUÇÃO

A evolução das redes de computadores e a expansão das arquiteturas distribuídas transformaram a *internet* em uma infraestrutura essencial para o tráfego contínuo de dados (KUROSE; ROSS, 2021). Nesse ecossistema, os sistemas de transmissão de voz interativa desempenham um papel crucial no trabalho remoto e na colaboração síncrona. Diferentemente das aplicações *web* tradicionais, os serviços de conversação exigem previsibilidade temporal; variações no atraso (*jitter)* superiores a 50 milissegundos provocam perda na inteligibilidade da fala (BARBOSA et al., 2005).

Historicamente, as abordagens centralizadas convergem os fluxos de mídia para servidores encarregados de processar e redistribuir o áudio. Embora amplamente utilizadas, exigem grande largura de banda concentrada em um único ponto e introduzem altos custos

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

operacionais e gargalos de escalabilidade (GOMES; LOURENÇO, 2012; MENDONÇA; LEÃO, 2012; VIEIRA, 2010). Como alternativa de baixo custo, a arquitetura Ponto-a-Ponto (*P2P*) viabiliza a troca direta entre clientes. A consolidação do padrão *WebRTC* padronizou *APIs* nativas que permitem aos navegadores estabelecer canais multimídia criptografados sem a necessidade de *plugins* externos (ALVESTRAND, 2021).

Contudo, a engenharia de software voltada à estabilidade de sistemas descentralizados enfrenta dois problemas principais:

No plano de mídia: em redes de malha direta *P2P*, cada nó codifica e transmite o seu fluxo de voz para todos os outros N-1 membros, impondo um crescimento global de conexões de ordem quadrática $O(N^2)$, dado por $N(N-1)/2$, o que sobrecarrega o *upload* dos clientes quando a sala cresce.

No plano de controle: faz-se indispensável um servidor de sinalização para coordenar o ingresso de usuários e a troca de metadados de conexão. No modelo tradicional de servidores *web Java*, sustentar milhares de canais *WebSocket* simultâneos com *threads* nativas de sistema operacional resulta em alto consumo de memória, bloqueios de Entrada/Saída (*I/O*) e degradação de estabilidade sob picos de acesso (BATEMAN; PRESSLER, 2023; DEUS et al., 2025; PANDITA, 2024).

Diante desse contexto, a questão de pesquisa consiste em investigar em que medida o desacoplamento arquitetural entre um plano de sinalização assíncrono com *threads* virtuais e um plano de mídia descentralizado *WebRTC P2P* garante a estabilidade operacional do servidor sob alta carga e evidencia, na prática, os limites de tráfego de rede para os clientes entre 2 e 8 participantes.

O objetivo geral é desenvolver, verificar funcionalmente e avaliar a estabilidade de um protótipo de sistema web de comunicação de voz em tempo real. Como objetivos específicos, pretende-se: formalizar a separação entre os planos de controle e de mídia, implementar a sinalização no *Spring Boot*, estruturar a malha *P2P* tratando condições de corrida de candidatos de rede e otimizando o *codec Opus* via atributos de sessão, e auditar a estabilidade do sistema por meio de telemetria contínua, testes de estresse e perfilamento da máquina virtual *Java*.

A justificativa deste estudo se fundamenta na necessidade de prover arquiteturas de comunicação síncrona de baixo custo infraestrutural que operem com alta estabilidade de memória e controle previsível de largura de banda.

A pesquisa se caracteriza como exploratória e descritiva, de natureza aplicada e orientada ao desenvolvimento tecnológico. Os procedimentos englobam a revisão da literatura,

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

a implementação do protótipo e a verificação prática acompanhada de testes de estresse em rede local, sem a coleta de avaliações subjetivas com usuários humanos.

1. REFERENCIAL TEÓRICO

O referencial teórico apresenta os conceitos relacionados às comunicações em tempo real e à concorrência no ecossistema *Java* que fundamentam o desenvolvimento deste trabalho.

1.1. Comunicação Multimídia em Tempo Real

A transmissão de voz sobre redes *IP* prioriza a baixa latência em detrimento da retransmissão de pacotes perdidos. Por esse motivo, sistemas síncronos apoiam-se no protocolo *UDP*, complementado pelos protocolos *RTP* e *SRTP*, que adicionam controle de sequência, marcação de tempo e criptografia ao fluxo de áudio (TANENBAUM; WETHERALL, 2011; SANTOS, 2016). O padrão *WebRTC* consolida essa comunicação diretamente nos navegadores em modo *P2P* através da *API RTCPeerConnection*, adotando o *codec Opus* (*RFC* 6716) para adaptar dinamicamente a taxa de *bits* entre 6 *kbps* e 510 *kbps* com baixa latência (ALVESTRAND, 2021; VALIN et al., 2012). Por meio da configuração de parâmetros no manifesto *SDP* (*RFC* 7587), é possível ajustar o tempo de empacotamento, ativar correção de erros e restringir o áudio ao modo mono com detecção de atividade de voz (SPITTKA; VOS; VALIN, 2015). Em grupos reduzidos, adota-se a topologia *P2P*, na qual todos os pares se conectam diretamente, garantindo latência mínima ao custo de um crescimento quadrático de conexões $O(N^2)$.

1.2. Servidores de Sinalização

Apesar de o tráfego de voz ocorrer diretamente entre os navegadores, os clientes não se descobrem sozinhos na rede. A arquitetura exige um servidor de sinalização para intermediar o *handshake* inicial, trocando descrições de sessão *SDP Offer/Answer* e candidatos de conectividade de rede *ICE* (ALVESTRAND, 2021; KUROSE; ROSS, 2021). O protocolo *WebSocket* consolida-se como a tecnologia mais adequada para essa sinalização, pois mantém um canal bidirecional permanentemente aberto sobre uma única conexão *TCP/TLS*, eliminando a sobrecarga de cabeçalhos do protocolo *HTTP* tradicional (KUROSE; ROSS, 2021).

1.3. Escalabilidade e Concorrência em Java

A escalabilidade e a estabilidade de um servidor web definem sua capacidade de manter a taxa de vazão e o tempo de resposta sob cargas variáveis e intensas (GREGOL; SCHUTZ, 2013; SANTOS, 2016). No desenvolvimento de *backends* corporativos em *Java*, o *framework*

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Spring Boot destaca-se pela produtividade e organização modular (GARCIA; PAGANI, 2021; MEDEIROS, 2025). Entretanto, historicamente, os servidores *Java* dependiam de *threads* de plataforma vinculadas diretamente às *threads* do sistema operacional, que reservam cerca de 1 MB de memória cada e sofrem com trocas de contexto pesadas sob milhares de conexões bloqueantes (BATEMAN; PRESSLER, 2023; PANDITA, 2024). A introdução das *virtual threads* (*JEP* 444), consolidadas a partir do *Java 21*, alterou esse paradigma ao permitir que a própria *JVM* gerencie milhares de *threads* leves no espaço do usuário com custo de poucos *kilobytes* na memória *Heap* (BATEMAN; PRESSLER, 2023). Quando uma *virtual thread* aguarda uma operação de rede (*I/O*), ela é desmontada da *thread* física do sistema operacional sem bloqueá-la, proporcionando alta escalabilidade e desempenho em operações de *I/O e* densidade de conexões *WebSocket* no *Spring Boot* sem a complexidade da programação reativa (DEUS et al., 2025; PANDITA, 2024).

1.4. Orquestração de Sessões

A orquestração abrange desde a entrada do usuário na sala até o encerramento e limpeza da sessão na memória. Uma orquestração robusta garante que o servidor atue apenas como um roteador rápido de metadados, retirando-se da rota de dados assim que o túnel *P2P* é estabelecido entre os navegadores (ALVESTRAND, 2021; VIEIRA, 2010). Para testar a estabilidade dessa orquestração sob estresse, combinam-se ferramentas de injeção de carga como o *Grafana K6* ao perfilamento nativo da *JVM* com o *Java Flight Recorder (JFR)* e o *JDK Mission Control* (*JMC*), permitindo inspecionar o comportamento de *threads*, memória e operações de *Socket* (*Socket I/O*) na prática (GRAFANA LABS, 2026; ORACLE, 2026).

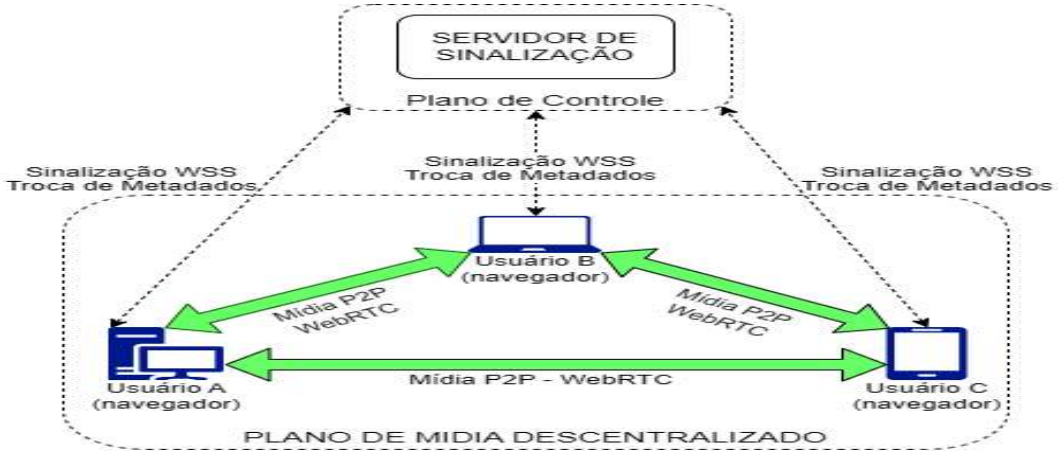
2. METODOLOGIA

A implementação integrou o *backend* de sinalização, o *frontend* multimídia e a coleta de telemetria em um fluxo orientado à estabilidade e à baixa latência. O sistema foi desenvolvido utilizando a linguagem *Java* configurada com compatibilidade *JDK* 21 no *Maven* com o *framework Spring Boot* 4.1.1 no plano de controle, e *HTML5*, *CSS3* e *JavaScript* puro com *APIs* nativas do *WebRTC* e *Web Audio API* no plano de mídia.

A aplicação foi estruturada com forte desacoplamento, conforme ilustrado na Figura 1. No *backend* (Plano de Controle), o servidor gerencia exclusivamente o estado das salas e o encaminhamento de mensagens via protocolo *WebSocket* Seguro (*WSS*) na porta 8443. No *frontend* (Plano de Mídia Descentralizado), os navegadores capturam, codificam e transmitem a voz diretamente entre si em topologia *P2P*.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

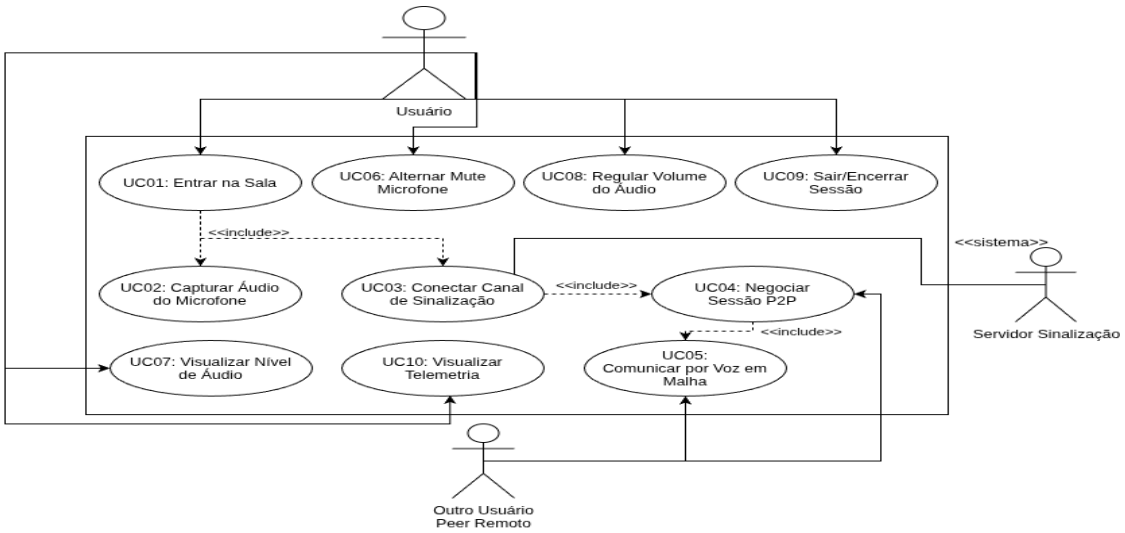
Figura 1. Diagrama do sistema em funcionamento



Fonte: Elaboração própria, 2026.

A modelagem do sistema reflete essa divisão de responsabilidades. O Diagrama de Caso de Uso (Figura 2) especifica as dez interações disponíveis ao usuário (entrada na sala, captura de microfone, negociação *P2P*, controle de mudo e volume, visualizador de nível de áudio e telemetria), enquanto o Diagrama de Classes (Figura 3) apresenta a estrutura do servidor e do cliente.

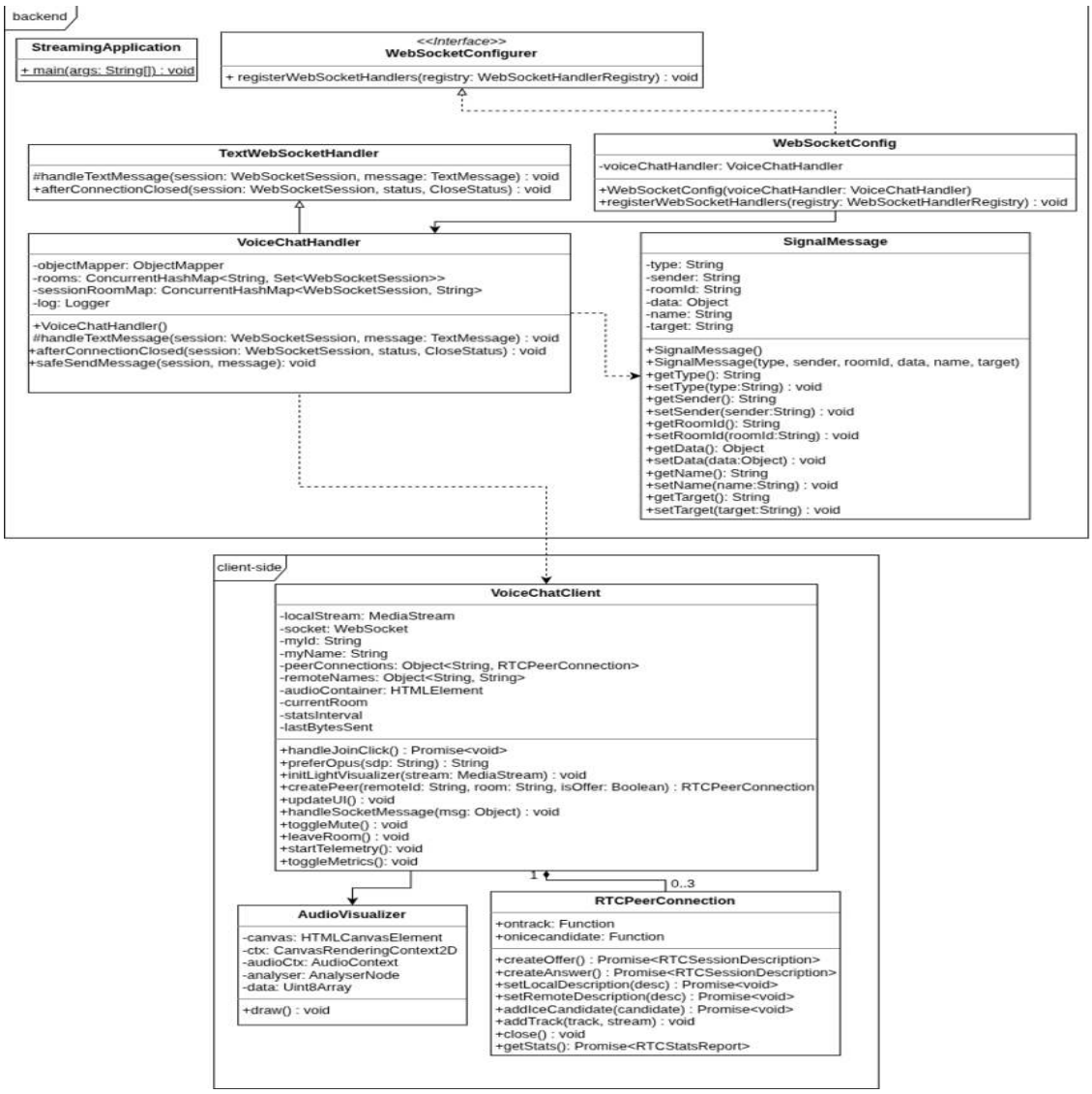
Figura 2. Diagrama de caso de uso



Fonte: Elaboração própria, 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 3. Diagrama de classes



Fonte: Elaboração própria, 2026.

A comunicação entre cliente e servidor ocorre pela troca do objeto *JSON SignalMessage*, no qual o preenchimento do atributo *target* permite ao servidor encaminhar ofertas *SDP* e candidatos *ICE* de forma direcionada em complexidade O(1) apenas ao destinatário correto, enquanto um mapa reverso garante a remoção imediata da sessão quando o usuário se desconecta.

Ao ativar as *threads* virtuais, o processamento das requisições tornou-se massivamente paralelo. Quando vários usuários entravam simultaneamente na mesma sala, múltiplas *threads*

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

virtuais tentavam disparar mensagens de candidatos *ICE* ao mesmo tempo para uma mesma *WebSocketSession*, gerando colisões de escrita e lançando exceções no servidor. Para resolver esse problema sem usar blocos *synchronized* tradicionais que poderiam prender a *thread* nativa do sistema operacional, cada sessão conectada foi encapsulada com o decorador nativo do *Spring Framework* e registrada previamente no mapa concorrente da sala, conforme mostra a Figura 4.

Figura 4. Implementação do Decorador de Concorrência para o *WebSocket*

```
public class VoiceChatHandler extends TextWebSocketHandler {
    @Override
    protected void handleTextMessage(WebSocketSession rawSession, TextMessage message) throws Exception {
        SignalMessage signal = objectMapper.readValue(message.getPayload(), valueType: SignalMessage.class);
        signal.setSender(rawSession.getId());
        String type = signal.getType();

        if ("join".equalsIgnoreCase(type)) {
            String roomId = signal.getRoomId();

            if (roomId == null || roomId.isBlank()) {
                return;
            }

            WebSocketSession session = new ConcurrentWebSocketSessionDecorator(rawSession, sendTimeLimit: 10000, bufferSizeLimit: 65536);
            sessionRoomMap.put(session.getId(), roomId);


            ConcurrentHashMap<String, WebSocketSession> roomSessions = rooms.computeIfAbsent(
                roomId, k -> new ConcurrentHashMap<>()
            );

            roomSessions.put(session.getId(), session);

            safeSendMessage(session, new TextMessage("{\"type\":\"created\", \"sender\":\"" + session.getId() + "\"}"));

            String joinPayload = objectMapper.writeValueAsString(signal);
            for (WebSocketSession s : roomSessions.values()) {
                if (s.isOpen() && !s.getId().equals(session.getId())) {
                    safeSendMessage(s, new TextMessage(joinPayload));
                }
            }
```

Fonte: Elaboração própria, 2026.

O decorador gerencia internamente uma fila segura com tempo limite de envio de 10 segundos e *buffer* de 64 KB, garantindo que múltiplas *threads* virtuais enfileirem mensagens sem erro e mantendo a estabilidade do canal de rede sob picos de tráfego.

Devido à rapidez do roteamento via *threads* virtuais, os pacotes de rede (*ICE candidates*) frequentemente chegavam ao navegador do destinatário alguns milissegundos antes que a descrição remota da sessão terminasse de ser configurada, causando falhas. A solução foi implementar uma fila de retenção temporária em *JavaScript*, na qual os candidatos recebidos prematuramente aguardam em memória e são processados em lote assim que o *handshake SDP* é confirmado, conforme a Figura 5.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 5. Fila de processamento assíncrono para candidatos *ICE*

```
async function processPendingIceCandidates(remoteId) {
    const pc = peerConnections[remoteId];
    const candidates = pendingIceCandidates[remoteId] || [];
    if (pc && pc.remoteDescription) {
        while (candidates.length > 0) {
            const candidate = candidates.shift();
            try {
                await pc.addIceCandidate(new RTCIceCandidate(candidate));
            } catch (err) {
                console.error("Erro ao aplicar candidato ICE:", err);
            }
        }
    }
}
```

Fonte: Elaboração própria, 2026.

Na topologia *P2P*, o total de conexões na sala cresce de forma quadrática O(N²) e cada cliente precisa enviar N-1 fluxos de subida simultâneos. Para evitar a rápida saturação da rede dos usuários, a aplicação intercepta o manifesto *SDP* gerado pelo navegador antes do envio e aplica restrições diretas no formato do *codec Opus* por meio de uma expressão regular na função *preferOpus(sdp)*, conforme demonstrado na Figura 6.

Figura 6. Injeção de Parâmetros Dinâmicos no *SDP* via *Frontend*

```
function preferOpus(sdp) {
    return sdp.replace(
        /a=fmtp:111[^\r\n]*/g,
        "a=fmtp:111 minptime=10;useinbandfec=1;stereo=0;maxaveragebitrate=64000"
    );
}
```

Fonte: Elaboração própria, 2026.

As propriedades injetadas seguem a *RFC* 7587 (SPITTKA; VOS; VALIN, 2015): *minptime=10* reduz o empacotamento para 10 ms, visando à latência mínima; *useinbandfec=1* ativa a correção de erros em banda contra perdas de pacotes *UDP*; e *maxaveragebitrate=64000* força o áudio em canal mono com limite nominal de 64 *kbps,* o que resulta em um teto bruto de 72 *kbps* por canal ao somar cerca de 8 *kbps* de cabeçalhos de rede *RTP/UDP/IP*.

Para comprovar o comportamento da malha, o cliente consulta a *API RTCPeerConnection.getStats()* a cada 2 segundos, calculando a taxa de *bits* instantânea enviada e o tempo de ida e volta. Esses dados são exibidos ao usuário e enviados ao *backend* sob o evento "*metrics*". Para gravar milhares de linhas de telemetria sem usar um banco de dados pesado e sem travar as *threads* virtuais em operações de disco, configurou-se um registrador assíncrono no *Logback*, conforme a Figura 7.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 7. Registro de Auditoria de Telemetria no Servidor

```
} else if ("metrics".equalsIgnoreCase(type)) {
    logMetricas.info(format: "[TELEMETRIA][Sala {}][Peer {}] {}",
            signal.getRoomId(), rawSession.getId(), objectMapper.writeValueAsString(signal.getData()));
} else {
    String roomId = signal.getRoomId();
```

Fonte: Elaboração própria, 2026.

Esta arquitetura baseada em eventos viabilizou a auditoria técnica e a validação empírica dos limites assintóticos da topologia *P2P*, evitando a necessidade de alocar infraestruturas pesadas de banco de dados relacional apenas para o armazenamento de métricas de rede efêmeras.

3. RESULTADOS E DISCUSSÃO

A verificação funcional e os testes de estabilidade foram conduzidos em ambiente de rede local, assegurando comunicação *P2P* direta sem dependência de servidores externos.

No uso prático da aplicação, a interface de entrada (Figura 8) valida o preenchimento de nome e código da sala e solicita permissão de acesso ao microfone com filtros nativos de cancelamento de eco, supressão de ruído e ganho automático ativados. Ao ingressar na sala (Figura 9), o sistema estabelece o túnel *P2P* e exibe em tempo real o visualizador de voz junto aos controles de microfone, volume e telemetria. Testou-se também a resiliência do desacoplamento arquitetural desligando o servidor *Spring Boot* com uma chamada em andamento: os navegadores detectaram o fechamento do *WebSocket* (*Socket.onclose*), exibiram o aviso e mantiveram a conversação de voz *P2P* ativa sem interrupção, comprovando que a mídia independe do servidor após a negociação inicial.

Figura 8. Interface de Entrada



Fonte: Elaboração própria, 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

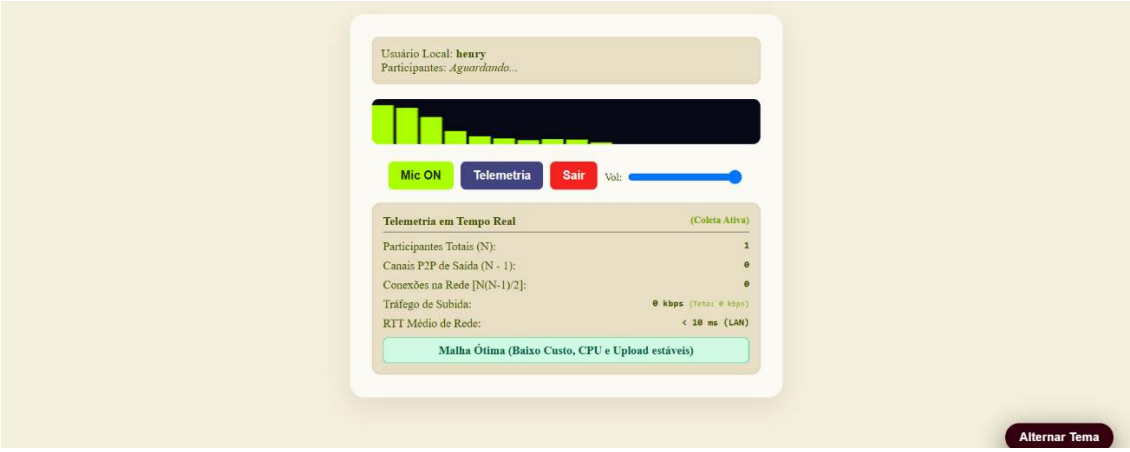
Figura 9. Interface da sala



Fonte: Elaboração própria, 2026.

A abertura do painel de telemetria (Figura 10) e a execução de testes variando a lotação das salas de 2 até 8 participantes permitiram observar o comportamento real do *codec Opus* e os limites da topologia *P2P*, cujos resultados práticos estão consolidados na Tabela 1.

Figura 10. Interface da sala com telemetria



Fonte: Elaboração própria, 2026.

Tabela 1. Resultados Práticos na Malha P2P – 2 a 8 Participantes

Participantes na Sala (N)	Canais de Saída por Cliente (N−1)	Conexões Totais na Malha [N(N−1)/2]	Tráfego em Silêncio c/ *VAD* (kbps)	Taxa Nominal em Fala Contínua (kbps)	Teto Bruto c/ Cabeçalhos Rede (kbps)	Estado da Malha	Diagnóstico Prático Observado
2	1	1	< 10 kbps	64 kbps	72 kbps	*OPTIMUM*	Baixo custo, RTT < 10 ms

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

3	2	3	< 20 kbps	128 kbps	144 kbps	*OPTIMUM*	Baixo custo, RTT < 10 ms
4	3	6	< 30 kbps	192 kbps	216 kbps	*OPTIMUM*	Limite Ideal para Topologia
5	4	10	< 40 kbps	256 kbps	288 kbps	*WARNING*	Sobrecarga Linear
6	5	15	< 50 kbps	320 kbps	360 kbps	*WARNING*	Sobrecarga Linear
7	6	21	< 60 kbps	384 kbps	432 kbps	*CRITICAL*	Inviável P2P $O(N^2)$
8	7	28	< 70 kbps	448 kbps	504 kbps	*CRITICAL*	Inviável P2P $O(N^2)$

Fonte: Elaboração própria, 2026.

Conforme demonstrado na Tabela 1, durante períodos de silêncio, o Detector de Atividade de Voz do *codec Opus* reduziu o envio para menos de 10 *kbps* por canal. Em fala contínua, até 4 participantes, o tráfego manteve-se no patamar seguro de 192 *kbps* nominais, teto bruto de 216 *kbps* com cabeçalhos *RTP/UDP/IP* e apenas 6 conexões globais (*OPTIMUM*). Contudo, ao passar para 5 e 6 integrantes, a sala saltou para 10 e 15 conexões (*WARNING*), atingindo o estado crítico (*CRITICAL*) com 7 e 8 participantes, 21 e 28 conexões *P2P* simultâneas e teto de até 504 *kbps* por cliente, provando na prática a saturação da malha.

Para comprovar a estabilidade do servidor de sinalização sob alta demanda, o *script* automatizado injetou um pico de 4.900 usuários virtuais simultâneos divididos em 140 blocos de 7 salas, totalizando 980 salas simultâneas com capacidades de 2 a 8 integrantes, comparando o servidor sem *threads* virtuais e com *threads* virtuais. Todos os nós trocaram eventos reais de *join, Offer, Answer*, rajadas de *ice candidate* e relatórios *metrics*. A Tabela 2 resume os resultados colhidos no *K6* e no perfilamento *JFR/JMC.*

Tabela 2. Comparativo de Estabilidade do Servidor sob Estresse

Indicador de Estabilidade e Desempenho	**Sem *Virtual Thread***	**Com *Virtual Thread***	**Resultado Prático**
Sessões Web*Socket* Conectadas	4.899 (100,0%)	4.899 (100,0%)	Zero recusas de *handshake WSS* (HTTP 101)
Total de Mensagens Roteadas pelo Servidor	203.062 (1.229/s)	203.041 (1.228/s)	Roteamento contínuo sem colisão de escrita
Negociações *P2P* (*Offer/Answer/ICE*)	11.766 / 11.766 / 35.298	11.763 / 11.763 / 35.289	Simetria de 100% entre ofertas e respostas

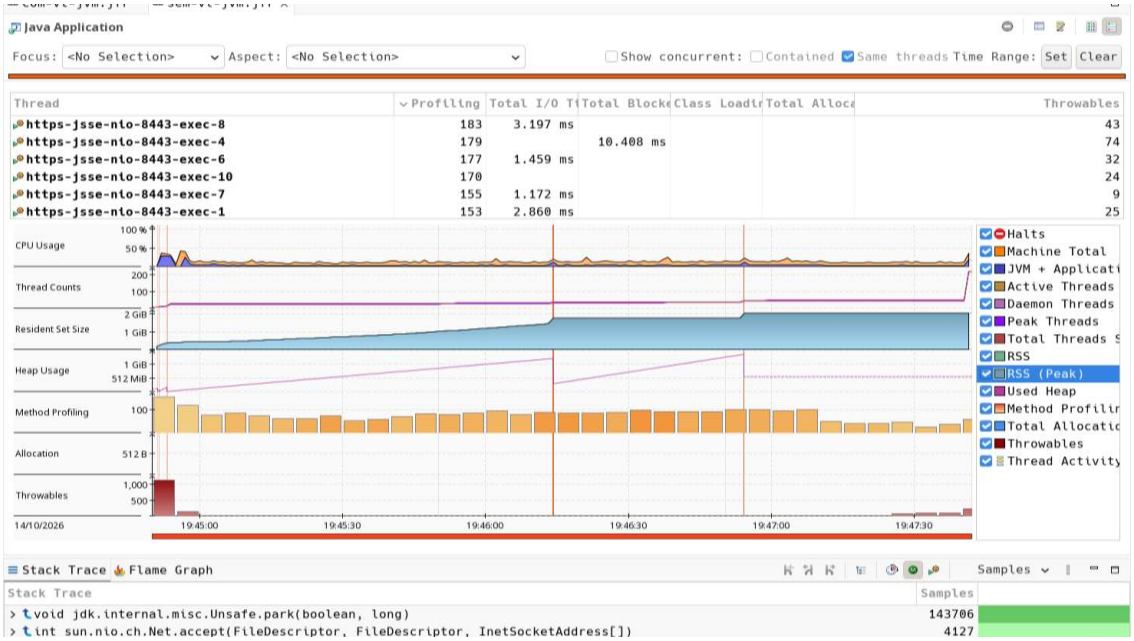
Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

***Logs* de Telemetria**	139.311 de 139.333 (99,98%)	139.327 de 139.327 (100,0%)	Zero perda de registros no encerramento com VT
Tempo de Conexão *WSS* (Média / P95)	5,17 ms / 6,88 ms	5,01 ms / 6,57 ms	Redução de 4,5% na latência
Criação de *Threads* Nativas no SO	227 *threads*	39 *threads*	Redução de 82,82% na criação de *threads* do SO
Eventos de Suspensão/Bloqueio	143.680 eventos	49.591 eventos	Redução de 65,49% nos bloqueios de *threads*

Fonte: Elaboração própria, 2026.

A inspeção visual das gravações *JFR* no *JMC* (Figuras 11 a 16) confirma a estabilidade estrutural alcançada. Na visão geral da aplicação (Figuras 11 e 12), nota-se que o servidor tradicional precisou escalar o *pool* do *Tomcat* para a faixa de 200 *threads* nativas, acumulando 143.706 amostras retidas em *Unsafe.park* (Figura 11). Já com *threads* virtuais (Figura 12), todo o tráfego foi absorvido pelas *threads* leves, mantendo-se estável em cerca de 25 a 30 *threads* nativas durante todo o teste.

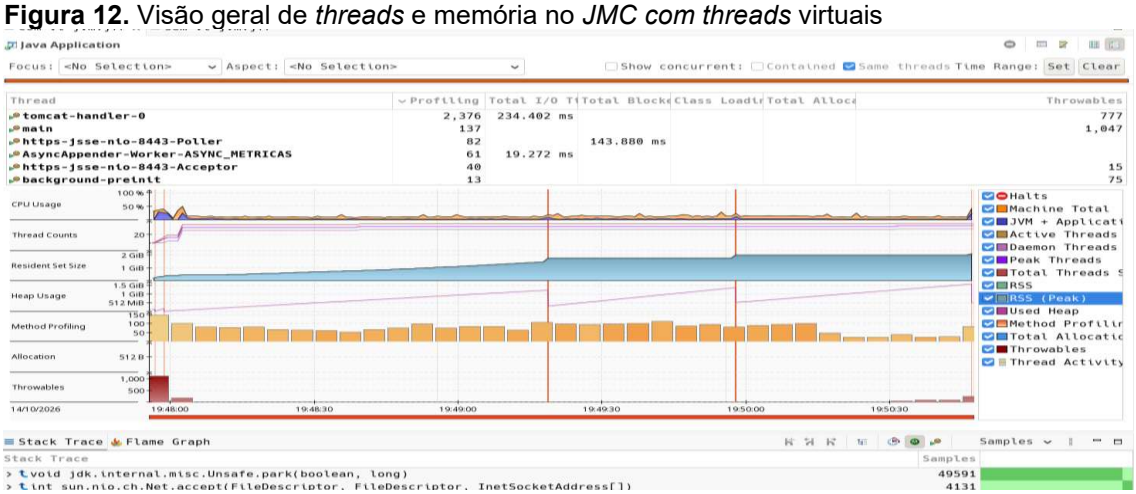
Figura 11. Visão geral de *threads* e memória no *JMC* sem *threads* virtuais



Fonte: Elaboração própria, 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

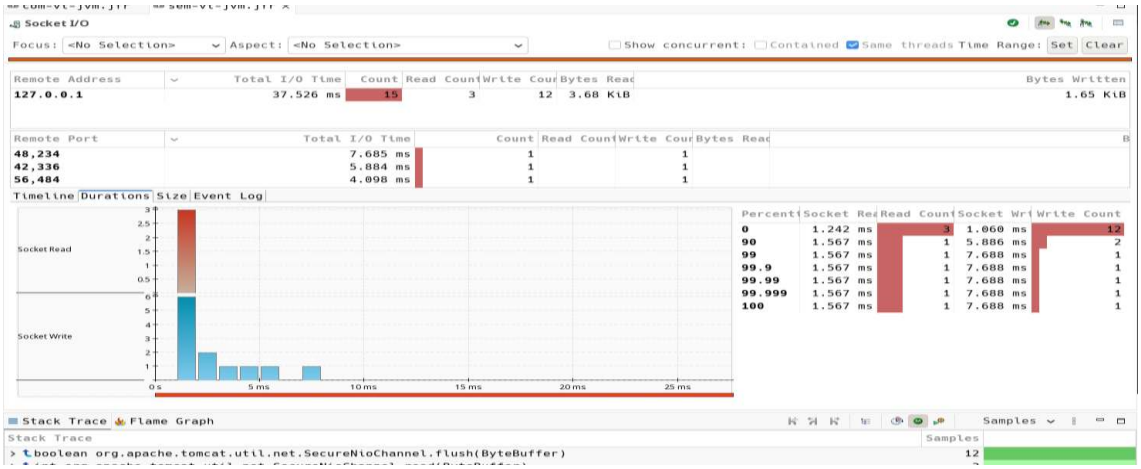
Figura 12. Visão geral de *threads* e memória no *JMC com threads* virtuais



Fonte: Elaboração própria, 2026.

No perfilamento da camada de rede na aba *Socket I/O* (Figuras 13 e 14), verifica-se que, de um total de 203.041 mensagens processadas sob criptografia *TLS*, apenas 15 eventos no cenário sem as *threads* virtuais e 56 eventos no cenário com as *threads* virtuais, o equivalente a apenas 0,027% das mensagens, ultrapassaram o limiar de amostragem de milissegundos do *JFR*. Isso comprova que a fila interna do *ConcurrentWebSocketSessionDecorator* serializou os candidatos *ICE* sem gerar gargalos de rede.

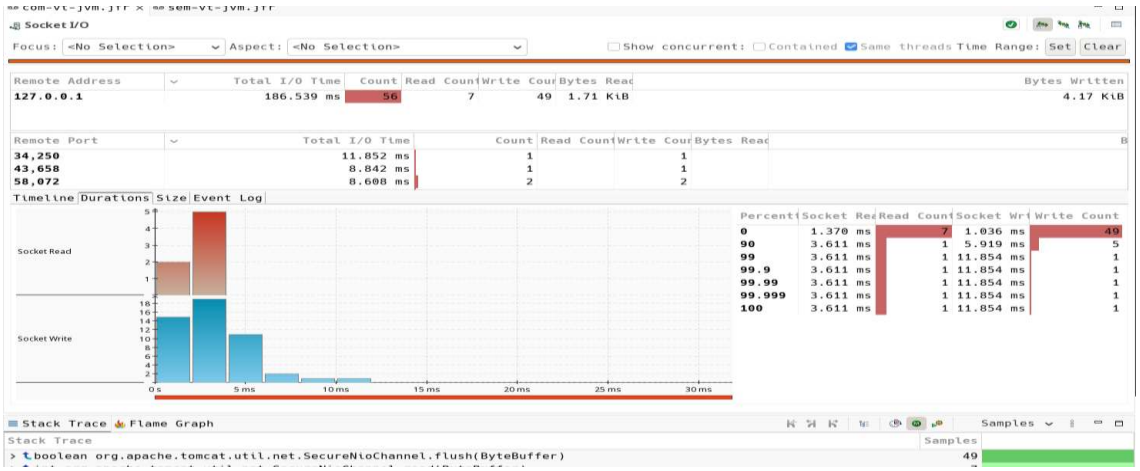
Figura 13. *Socket I/O* no *JMC* sem *threads* virtuais



Fonte: Elaboração própria, 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

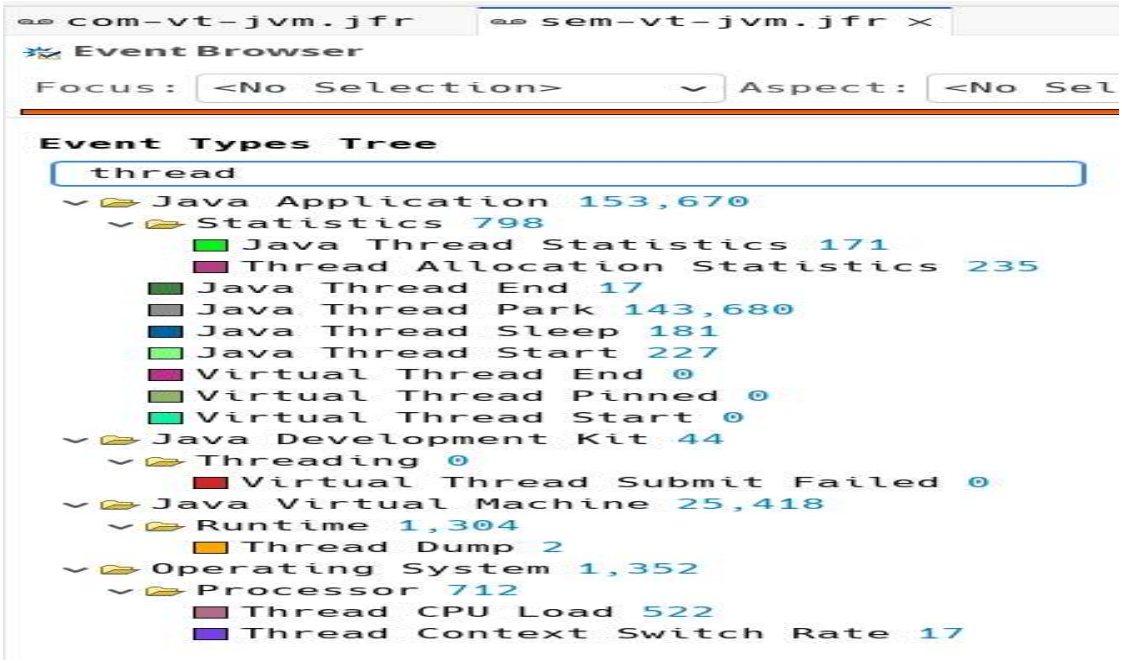
Figura 14. *Socket I/O* no *JMC* com *threads* virtuais



Fonte: Elaboração própria, 2026.

Por fim, o navegador de eventos de *threads* do *JMC* (Figuras 15 e 16) atesta que a queda de 82,82% na inicialização de *threads* de sistema operacional, de 227 para 39, e de 65,49% nos estacionamentos de *threads* (*Java Thread Park*), de 143.680 para 49.591, ocorreu com zero ocorrências de aprisionamento (*Virtual Thread Pinned = 0*) e zero falhas de submissão (*Virtual Thread Submit Failed = 0*), confirmando a total estabilidade.

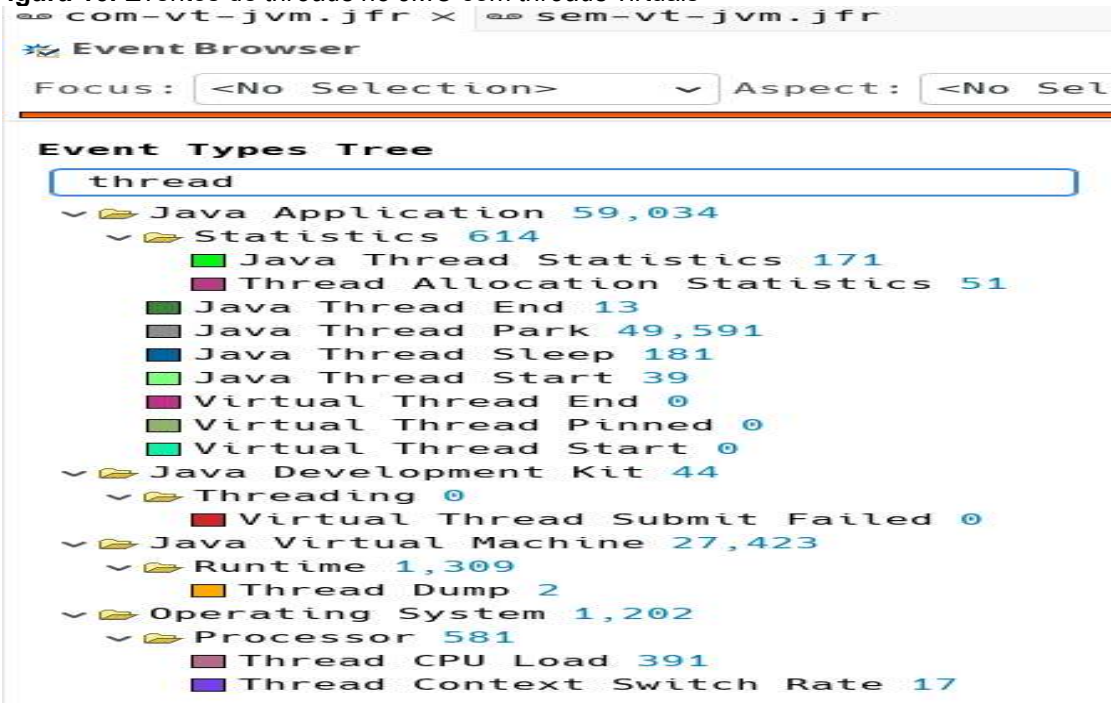
Figura 15. Eventos de *threads no JMC* sem *threads* virtuais



Fonte: Elaboração própria, 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

Figura 16. Eventos de *threads no JMC* com *threads* virtuais



Fonte: Elaboração própria, 2026.

4. CONSIDERAÇÕES FINAIS

Este trabalho desenvolveu, verificou funcionalmente e avaliou a estabilidade de um sistema *web* de comunicação síncrona de voz utilizando *WebRTC* e *Spring Boot*. A solução integrou a interface gráfica com análise espectral, a captura de mídia, a sinalização em tempo real e um motor de observabilidade autônomo, superando os desafios de concorrência de código e atendendo aos objetivos definidos para a prova de conceito.

Em resposta à questão de pesquisa, comprovou-se que o desacoplamento arquitetural entre o plano de controle e o plano de mídia é efetivo e assegura alta estabilidade operacional.

No servidor, a combinação do *Spring Boot* com *threads* virtuais, *ConcurrentWebSocketSessionDecorator* e logs assíncronos processou mais de 203 mil mensagens em 980 salas simultâneas com 100% de integridade, reduzindo em 82,82% a criação de *threads* de sistema operacional e em 65,49% os bloqueios de suspensão sem incorrer em *thread pinning*. No cliente, a fila de retenção de candidatos *ICE* e a parametrização do *codec Opus* via *SDP* garantiram conexões estáveis com latência inferior a 10 ms em rede local.

A telemetria validou empiricamente os limites da topologia *P2P*: a arquitetura apresenta estabilidade ideal para grupos de até quatro participantes, mas sofre degradação quadrática de

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

conexões e sobrecarga de banda de cinco a oito integrantes, comprovando a necessidade de servidores centrais de encaminhamento seletivo para salas maiores.

Como limitações do estudo, destacam-se a verificação restrita ao ambiente de rede local, sem o uso de servidores externos de travessia de *NAT (STUN/TURN)*, e a ausência de testes subjetivos de usabilidade com usuários finais. Trabalhos futuros podem integrar servidores *STUN/TURN* para operação na internet pública e implementar a transição automática da sala do modo *P2P* para um servidor *SFU* sempre que a telemetria detectar a entrada do quinto participante.

REFERÊNCIAS

ALVESTRAND, Harald. **Overview: Real-Time Protocols for Browser-Based Applications (RFC 8825)**. Fremont: Internet Engineering Task Force (IETF), jan. 2021. Disponível em: https://datatracker.ietf.org/doc/html/rfc8825. Acesso em: 15 mar. 2026.

BARBOSA, Rodrigo et al. **Avaliação do Desempenho de Aplicações VoIP P2P**. Recife: Centro de Informática – Universidade Federal de Pernambuco (UFPE), 2005. Disponível em: https://www.researchgate.net/publication/228344277_Avaliacao_do_Desempenho_de_Aplicacoes_VoIP_P2P. Acesso em: 15 mar. 2026.

BATEMAN, Alan; PRESSLER, Ron. **JEP 444: *Virtual threads***. Redwood Shores: OpenJDK / Oracle, 2023. Disponível em: https://openjdk.org/jeps/444. Acesso em: 15 mar. 2026.

DEUS, Ítalo M. et al. **Avaliação de Desempenho de *Frameworks* Web Java entre Reatividade e *Threads* Virtuais: Um Estudo Comparativo entre Spring e Quarkus**. In: WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAÇÃO (WPERFORMANCE), 24., 2025, Maceió. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2025. p. 61-72. Disponível em: https://doi.org/10.5753/wperformance.2025.8657. Acesso em: 15 mar. 2026.

GARCIA, Gabriel Bugés; PAGANI, Carlos. **Do monolítico ao microserviço: a transformação das equipes e sistemas**. São Paulo: Instituto Federal de São Paulo (IFSP), 2021. Disponível em: https://suap.ifsp.edu.br/media/edu/projeto_final/TCC_gabriel.pdf. Acesso em: 23 mar. 2026.

GOMES, Ana; LOURENÇO, Rodrigo. **Internet Live Streaming**. Lisboa: Instituto Superior Técnico (IST), 2012. Disponível em: http://www.img.lx.it.pt/~fp/cav/ano2014_2015/Trabalhos_MEEC_2014_2015/Artigo11/SiteCAVFinal_68370_69924/SiteCAVFinal/artigo.pdf. Acesso em: 23 mar. 2026.

GRAFANA LABS. **Grafana *K6* Documentation**: Open-source load testing tool for engineering *teams*. 2026. Disponível em: https://grafana.com/docs/k6/latest/. Acesso em: 23 mar. 2026.

GREGOL, Régis E.; SCHUTZ, Fernando. **Recursos de escalabilidade e alta disponibilidade para aplicações web. Revista Eletrônica Científica Inovação e Tecnologia, Medianeira, v. 4, n. 7, p. 28-31, 2013. Edição especial MEDITEC**. Disponível em: http://dx.doi.org/10.3895/recit.v4.n7.4189. Acesso em: 23 mar. 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.

KUROSE, James F.; ROSS, Keith W. **Redes de Computadores e a Internet: Uma Abordagem Top-Down**. 8. ed. São Paulo: Pearson Education do Brasil, 2021.

MEDEIROS, André Marçal. **Análise comparativa entre Spring Boot e Jakarta EE: um relato de experiência**. Lavras: Universidade Federal de Lavras (UFLA), 2025. Disponível em: https://sip.prg.ufla.br/publico/trabalhos_conclusao_curso/acessar_tcc_por_curso/ciencia_da_computacao/index.php?dados=20251201811266. Acesso em: 23 mar. 2026.

MENDONÇA, Gabriel G.; LEÃO, Rosa M. M. **BTStream: um ambiente para desenvolvimento e teste de aplicações de streaming P2P. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS (SBRC), 30**., 2012, Ouro Preto. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2012. p. 956-963. Disponível em: https://www.researchgate.net/publication/267202351_BTStream_-_Um_ambiente_para_desenvolvimento_e_teste_de_aplica_oes_de_streaming_P2P. Acesso em: 23 mar. 2026.

ORACLE. **JDK Mission Control and Java Flight Recorder Documentation**. Redwood Shores: Oracle Corporation, 2026. Disponível em: https://docs.oracle.com/en/java/java-components/jdk-mission-control/. Acesso em: 10 abr. 2026.

PANDITA, Vishesh. **Benchmarking the Performance of Java *Virtual threads* in High-Throughput Workloads**. National College of Ireland, Dublin, 2024. Disponível em: https://norma.ncirl.ie/id/eprint/8134. Acesso em: 23 mar. 2026.

SANTOS, Anderson Mendes dos. **Um método para apoiar decisões de projeto em aplicações web com streaming de mídia visando desempenho e escalabilidade**. Universidade Federal de Alagoas (UFAL), Maceió, 2016. Disponível em: http://www.repositorio.ufal.br/handle/riufal/2043. Acesso em: 10 abr. 2026.

SPITTKA, Julian; VOS, Koen; VALIN, Jean-Marc. **RTP Payload Format for the Opus Speech and Audio *Codec* (RFC 7587)**. Fremont: Internet Engineering Task Force (IETF), jun. 2015. Disponível em: https://datatracker.ietf.org/doc/html/rfc7587. Acesso em: 10 abr. 2026.

TANENBAUM, Andrew S.; WETHERALL, David J. **Redes de Computadores.** 5. ed. São Paulo: Pearson Prentice Hall, 2011.

VALIN, Jean-Marc et al. **Definition of the Opus Audio *Codec* (RFC 6716)**. Fremont: Internet Engineering Task Force (IETF), set. 2012. Disponível em: https://datatracker.ietf.org/doc/html/rfc6716. Acesso em: 23 mar. 2026.

VIEIRA, Alex Borges. **Transmissão de Mídia Contínua Ao Vivo em P2P: Modelagem, Caracterização e Implementação**. Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, 2010. Disponível em: https://hdl.handle.net/1843/SLSS-85BNKG. Acesso em: 23 mar. 2026.

Este artigo é publicado em acesso aberto (Open Access) sob a licença Creative Commons Atribuição 4.0 Internacional (CC-BY), que permite uso, distribuição e reprodução irrestritos em qualquer meio, desde que o autor original e a fonte sejam creditados.