

A Comparison of Protocols for Server-Side Binaural Audio Rendering on the Web

Hadeel M. S. Omar
Queen Mary University of London
United Kingdom
h.omar@se22.qmul.ac.uk

Johan Pauwels
Queen Mary University of London
United Kingdom
j.pauwels@qmul.ac.uk

Abstract—Binaural audio rendering currently relies on client-side processing through the Web Audio API, limiting HRTF personalisation and straining resource-constrained devices. In this paper, we present two real-time APIs for server-side binaural audio rendering using the 3D Tune-In Toolkit. These APIs differ only in their underlying communication protocol, respectively WebSocket and HTTP-SSE. This enables a systematic comparison using containerised deployment to AWS EC2 to achieve realistic scalability evaluation under distributed load conditions. Single-session benchmarking revealed that our HTTP-SSE implementation achieves 64% lower pipeline latency (17.4ms vs 48.1ms) despite WebSocket’s superior transport efficiency (3.3ms vs 11.2ms RTT). The root cause for the former was traced back to the abstraction overhead in the Flask-SocketIO library we built upon. The WebSocket-based implementation excels at position updates (2.0ms vs 10.7ms), critical for reactivity and audio-visual synchronisation. Load testing demonstrated viable performance up to 100 concurrent users for audio streaming, but position updates exhibit severe degradation (57ms to 5000ms p95 latency) at this number of users due to limited parallelism. Overall, the current state of the APIs already achieve sub-100ms latency for real-time applications, and its identified constraints will guide future implementation updates.

Index Terms—spatial audio, binaural rendering, HRTF, server-side rendering, WebSocket, HTTP-SSE, real-time systems.

I. INTRODUCTION

Spatial audio refers to sound that is reproduced with directional characteristics, creating the perception of sound sources positioned in three-dimensional space around the listener. A specific topic within spatial audio is *binaural audio*, where the reproduction is delivered through standard stereo headphones. This technology has become increasingly important for immersive applications including virtual reality, gaming, hearing aids and telepresence systems. Web applications integrating binaural audio can use the Web Audio API as a standardised client-side implementation [1], more specifically its PannerNode. This standard abstracts away the complexity of implementing binaural rendering, thereby greatly reducing the barrier to entry.

As convenient as the Web Audio API is, however, it does come with a couple of limitations. First, due to the nature of the standardisation process, the exact implementation of the underlying rendering techniques is left undefined. This leads to inconsistencies between browsers and platforms, which can be undesirable, for instance in the context of scientific research. Second, the W3C standard is limited in scope and does

not include certain advanced feature such as personalisation. Third, client-side rendering puts the computational burden on the user’s device, which can be challenging for mobile devices and hearing aids where battery life and thermal constraints limit sustained DSP workloads.

To address these limitations, we present a server-side binaural audio processing API. This architecture comes with its own trade-offs worth investigating. The server side rendering gives us full control of the implementation details and consistency across platforms. We build on an existing C++ library for binaural rendering that provides advanced rendering capabilities and customisation. We wrap this C++ core in a Python-based API that can be self-hosted.

The API streams audio to the client from the server and responds to control signals sent from the client to the server. Latency is therefore one of the most critical challenges compared to client-side implementations. Choosing the right communication protocol becomes a critical design decision that affects both performance and scalability, so we compare two. WebSocket, standardised in RFC 6455, provides full-duplex communication over persistent TCP connections [2]. WebSocket is generally assumed to be superior for real-time applications because its persistent connection model eliminates per-request handshake overhead and its minimal 2-6 byte framing is considerably lighter than HTTP’s verbose headers. However, this assumption derives primarily from studies examining transport-dominant workloads with minimal server processing - conditions that do not adequately reflect the computational intensity of HRTF-based spatial audio rendering. A feature-equivalent HTTP implementation using Server-Sent Events (SSE) allows for systematic protocol comparison, while containerised deployment to cloud infrastructure enables realistic scalability evaluation under distributed load conditions.

In summary, this project enables applications that require consistent binaural rendering quality, personalisation capabilities, or reduced client computational burden. As such, it complements existing ways to create immersive sonic experiences over the internet [3]. In addition to an open-source, real-time API for server-side binaural rendering, a study on the effects of its design choices and ideas for future improvements are presented. The software itself is provided for self-hosting ¹

¹<https://github.com/hadeelomar/spatialsocket>

with extensive documentation and tests.

The remainder of this text is organised as follows. We start by giving an overview of background material related to this work in section II, before going into the direct dependencies of our code in section III. The API specification is then discussed in section IV, along with some implementation details. The experimental setup used to test our APIs is described in section V, and its results reviewed in section VI. Finally, a conclusion is drawn in section VII, including ideas for further work.

II. BACKGROUND

A. Binaural Audio Fundamentals

Understanding how humans perceive spatial sound is fundamental to creating convincing artificial spatial experiences, since effective binaural rendering requires accurate reproduction of the acoustic cues that the auditory system employs for localisation. The human auditory system localises sound sources through psychoacoustic cues that can be categorised as binaural (requiring comparison between both ears) and monaural (available to a single ear) [4]. These cues work together, with the auditory cortex integrating information across frequency bands and time windows to form coherent spatial percepts. A visual illustration of binaural audio cues can be found in fig. 1

Interaural Time Difference (ITD) arises from the path length difference for sound travelling to each ear. Sound arriving from the side reaches the nearer ear before the farther ear, creating timing offsets ranging from 0 microseconds (sound directly ahead or behind) to approximately 700 microseconds (sound directly to one side). Blauert [4] identifies ITD as particularly effective for low-frequency sounds below 1.5 kHz, where wavelength is large relative to head dimensions and phase differences remain unambiguous. The auditory system demonstrates remarkable sensitivity to ITD, with some well-practiced listeners capable of detecting differences as small as 10-20 microseconds.

Interaural Level Difference (ILD) results from acoustic shadowing by the head, which acts as a barrier that attenuates sound reaching the farther ear. The degree of attenuation varies with frequency: higher frequencies experience greater attenuation because their shorter wavelengths are more readily blocked by the head. ILD becomes increasingly important above 1.5 kHz, providing complementary localisation information to ITD in frequency ranges where phase-based ITD becomes ambiguous [4]. This combination of ITD dominance at low frequencies and ILD dominance at high frequencies is known as the duplex theory of sound localisation.

Monaural spectral cues arise from direction-dependent filtering effects of the head, torso and pinnae (outer ears). The complex geometry of the pinna creates resonances and reflections that vary systematically with sound source elevation and front-back position, encoding directional information in the spectral shape of sounds reaching the ear canal. These spectral modifications are crucial for vertical localisation and resolving front-back ambiguities that binaural cues cannot distinguish,

since ITD and ILD are largely symmetrical for sources in frontal and rear hemispheres at the same horizontal angle [4]. The pinna's small size means these effects predominantly occur at high frequencies above 4 kHz.

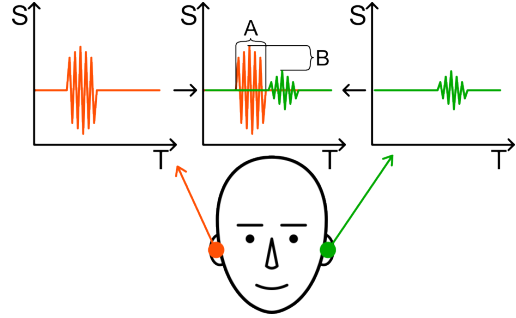


Fig. 1. Illustration of binaural spatial hearing cues. The waveforms show signals arriving at the left (orange) and right (green) ears from a sound source positioned to the listener's left. Label (A) indicates the interaural time difference - the delay between when the sound reaches each ear. Label (B) indicates the interaural level difference - the amplitude variation between the two ears. These cues enable the auditory system to determine the horizontal location of sound sources.

Head-Related Transfer Functions (HRTFs) mathematically characterise how sound transforms as it travels from a source position to the listener's ear canals, encoding all spatial cues necessary to reproduce directional hearing [5].

HRTFs are typically measured in anechoic chambers using miniature microphones at the ear canal entrance, with measurements taken for sound sources at many positions around the listener. This procedure creates an HRTF covering the full sphere of possible source directions [6].

Critically, HRTFs are highly individual due to anatomical variations in head size, ear shape and torso dimensions. While generic or averaged HRTFs provide convincing spatial audio for many listeners, personalised HRTFs measured for specific individuals typically provide superior localisation accuracy and reduced front-back confusion [7], [8]. The Spatially Oriented Format for Acoustics (SOFA), standardised as AES69-2022, provides consistent format for storing and sharing HRTF data between research groups and software implementations [9].

Binaural rendering creates spatial audio over headphones by convolving sound sources with appropriate HRTFs. For a source at a given position, binaural output signals for left and right ears are obtained through convolution with corresponding Head-Related Impulse Responses (HRIRs) - the time-domain representations of HRTFs [5].

Real-time binaural rendering for interactive applications requires dynamic HRTF processing, where HRTFs are continuously updated as sources move or listener head orientation changes. This presents computational challenges, as HRTF convolution is relatively expensive, particularly with high-quality HRTFs having long impulse responses. Distance-dependent effects follow the inverse square law, where sound pressure level decreases by 6 dB for each doubling of distance [5].

B. Communication Protocols for Real-Time Applications

HTTP, originally defined in RFC 2616 and subsequently updated through RFC 7230-7235 and RFC 9110, operates as a request-response protocol where clients initiate all communication and servers respond to each request independently [10]. HTTP/1.1 introduced persistent connections (Connection: keep-alive), allowing multiple requests over a single TCP connection without repeated TCP handshake overhead, which significantly improves efficiency for applications making many sequential requests.

HTTP's request-response model creates natural backpressure. Clients implicitly wait for responses before sending subsequent requests, preventing server overload from request flooding. This makes server implementation simpler since the communication pattern is inherently synchronous from the server's perspective. However, the model means servers cannot initiate communication; they can only respond to client requests, making server-push scenarios awkward to implement.

Server-Sent Events (SSE) provide a mechanism for server-to-client streaming over HTTP, enabling server push without WebSocket's full bidirectionality. SSE uses a long-lived HTTP response with chunked transfer encoding, where the server sends events as text lines following a simple format. SSE integrates well with HTTP infrastructure including proxies, load balancers and caching layers, though it only supports server-to-client communication. Client messages require separate HTTP requests.

HTTP/2 [11] and HTTP/3 [12] provide multiplexing and header compression that further improve efficiency, though these improvements primarily benefit scenarios with many concurrent requests and do not fundamentally alter the request-response communication model.

RFC 6455 defines WebSocket as establishing persistent bidirectional communication over TCP connections [2]. Connection establishment begins with an HTTP Upgrade handshake, after which both parties communicate using a lightweight binary framing protocol with 2-6 bytes overhead per message, substantially less than HTTP headers (typically 200-400 bytes even for minimal requests).

WebSocket's full-duplex nature enables server-initiated push without client polling and persistent connections eliminate per-request handshake overhead. Messages can be sent in either direction at any time without coordination, enabling truly asynchronous communication patterns. This flexibility is valuable for real-time applications where both client and server need to send unpredictable messages.

However, WebSocket connections require maintaining persistent state on the server, with implications for horizontal scaling and failover. Stateful connections complicate load balancing (sticky sessions may be required) and make connection migration during server restarts more complex than stateless HTTP. WebSocket also receives less consistent support from intermediaries: some proxies, firewalls and CDNs may not correctly handle WebSocket connections, though support has improved substantially since the protocol's standardisation.

C. Prior Studies

Multiple tools to render binaural audio in the browser have been compared in [3]. These all focus on client-side rendering, and therefore come with considerably different trade-offs than the ones made in our setup. One of the discussed tools is based on the same C++ core library that underlies our rendering. Consequently, rendering quality is not a differentiator when comparing with our study, but the location of the rendering and its implications are. Similarly, research in networked music performance, such as [13], provides relevant data about latency and its effect on music performers. A setup that reduces latency when streaming multiple audio sources is discussed (based on JackTrip [14]), but binaural rendering is done on the client-side.

Several studies have compared WebSocket and HTTP performance, but critical analysis reveals limitations relevant to this project. Pimentel & Nickerson [15] found WebSocket reduced latency by 50% for stock ticker applications with minimal server processing (<1 ms for database lookup). However, their workload was transport-dominant: processing time approached zero, making transport efficiency the determining factor. This finding cannot generalise to processing-dominant workloads where server computation dominates.

Škvorc et al. [16] found WebSocket superior for bidirectional IoT data collection, but their benchmark involved simple echo servers with negligible processing: receive message, immediately return. The processing to transport ratio approached zero. Łasocha & Badurowicz [17] concluded WebSocket excels for frequent, small messages while REST remains competitive for infrequent, large payloads. Importantly, they noted framework implementation quality significantly affects results but did not quantify this effect.

III. API DEPENDENCIES

Our proposed API builds on proven, well-tested technologies and libraries, which are discussed in this section.

A. The 3D Tune-In Toolkit

The 3D Tune-In Toolkit (3DTI) is an open-source C++ library² providing professional-grade binaural audio rendering, developed through EU Horizon 2020 research focused on spatial audio technology for hearing aid applications [8]. 3DTI implements HRTF-based spatialisation with comprehensive SOFA format support, enabling use of both generic and personalised HRTFs. The toolkit can load any standards-compliant SOFA file, automatically handling coordinate system transformations and sample rate conversion. This flexibility is essential for research applications that need to compare different HRTF datasets and for personalisation scenarios where individual measurements are available.

Multiple simultaneous sound sources with independent positions can be rendered, enabling complex auditory scenes with many concurrent virtual objects. Each source maintains its own position state and audio buffer, with the renderer mixing

²https://github.com/3dtune-in/3dti_audiotoolkit

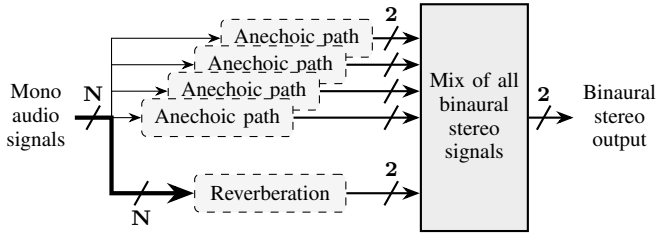


Fig. 2. High-level processing structure of the 3D Tune-In Toolkit. The architecture separates binaural spatialisation into two parallel processing paths: the anechoic path handles direct sound from sources with HRTF-based spatialisation, whilst the reverberation path processes environmental reflections. This dual-path approach enables realistic spatial audio reproduction combining direct sound localisation with room acoustic effects. (reproduced from [8])

spatialised outputs to produce the final binaural signal. The toolkit also models listener position and head orientation with support for dynamic head tracking, enabling responsive spatial audio that updates as the listener moves or turns.

The architecture employs a dual-path processing approach that separates direct sound from environmental reflections, as illustrated in fig. 2. The anechoic path processes direct sound using HRTF-based spatialisation for accurate localisation cues, while the reverberation path models environmental acoustics including early reflections and late reverberation. This separation allows independent control of spatialisation accuracy and environmental ambience [8].

The processing pipeline performs HRIR interpolation to calculate appropriate impulse responses based on source position relative to listener. The toolkit implements barycentric interpolation between the three nearest measured HRTF positions, providing smooth transitions during source movement. These interpolated HRIRs are convolved with source audio using overlap-save FFT-based algorithms, reducing computational requirements compared to time-domain convolution.

Importantly, ITD simulation is handled separately from magnitude-only HRTF convolution—a design choice that allows ITD customisation according to listener head size (important for personalisation) and reduces comb-filtering artifacts when interpolating between HRTF positions. The separated ITD is implemented as a fractional delay line applied after HRTF convolution, as shown in fig. 3 [8].

B. *py3dti*: Python Bindings for the 3DTI Toolkit

As a first contribution in this paper, we created Python bindings for 3DTI. These allow to use the performance of compiled C++ code in Python’s extensive ecosystem, not just for this project. The resulting open-source Python package³ is called *py3dti* and uses *pybind11* [18] to expose 3DTI’s core functionality through a Pythonic API, handling memory management and type conversion automatically.

The package provides a single `BinauralRenderer` class with methods for adding and positioning sources and listeners, and HRTF and room acoustics loading from SOFA files. All possible use-cases are covered:

- whether the rendering can happen offline or requires streaming block-by-block
- whether the positions of the audio sources and/or listener are static or moving around
- whether the audio material for the sources themselves is available offline or provided as streams

The online rendering capabilities make *py3dti* viable for server-side spatial audio processing in our proposed web API.

C. *Flask* and *Flask-SocketIO*

The actual web API is built using Flask, a lightweight WSGI web application framework. For the implementation of the WebSocket protocol, the Flask-SocketIO extension is used. This extends Flask with WebSocket support through the Socket.IO protocol, providing automatic reconnection, room-based broadcasting, namespace support and fallback to HTTP long polling when WebSocket is unavailable [19]. The event-driven programming model enables structured communication with different message types handled by separate handler functions, simplifying application organisation.

Socket.IO is not raw WebSocket: it adds a protocol layer on top of WebSocket providing features such as automatic reconnection, heartbeats and multiplexing. This abstraction introduces processing overhead that is not present in raw WebSocket implementations: incoming messages must be parsed for Socket.IO packet type, namespace, event name and JSON payload before reaching application handlers. The convenience of Socket.IO’s features comes at a performance cost that the literature has not quantified for processing-heavy workloads.

Flask-SocketIO supports multiple asynchronous modes including *threading* (default), *eventlet* and *gevent*. The *gevent* mode uses greenlet-based cooperative multitasking, enabling thousands of concurrent connections on a single worker through lightweight coroutines. This mode is the recommended one for production deployments handling many simultaneous connections.

IV. API SPECIFICATION AND IMPLEMENTATION

A. *System architecture*

The proposed system employs a multi-tier architecture separating concerns across three distinct layers. This separation serves two design goals: enabling the WebSocket and HTTP transports to share identical audio processing logic, ensuring comparison fairness, and isolating I/O-bound network handling from CPU-bound HRTF convolution so each can be independently optimised.

1) *Transport layer*: The transport layer will handle all protocol-specific communication. Both implementations will delegate to the same service layer, with only the transport mechanism differing between them.

WebSocket Transport Flask-SocketIO with *gevent* async mode will be used, providing event-driven messaging over persistent connections. Each connected client will be identified by Socket.IO’s `request.sid`, a unique per-connection identifier assigned automatically at handshake time. Socket.IO is chosen over raw WebSocket for its automatic reconnection,

³<https://github.com/jpauwels/py3dti>

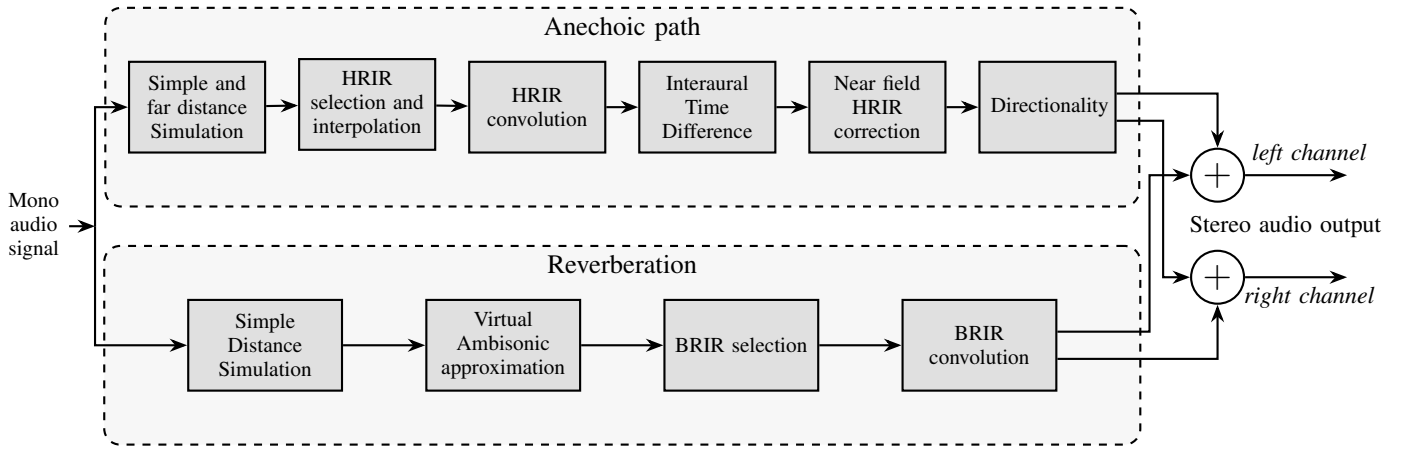


Fig. 3. Detailed binaural processing pipeline of the 3D Tune-In Toolkit. The diagram illustrates the complete signal flow, including HRIR (Head-Related Impulse Response) interpolation, convolution processing, ITD (Interaural Time Difference) simulation, ILD (Interaural Level Difference) processing, distance attenuation and near-field corrections [8]

heartbeat management and structured event naming, features valuable for maintaining sessions across transient network disruptions.

HTTP Transport Standard Flask routes will handle request-response control operations (session management, position updates, audio upload), with Server-Sent Events for streaming rendered audio. SSE is chosen because it allows the server to push audio chunks asynchronously over a single persistent HTTP connection without requiring the client to poll, while keeping HTTP semantics for all control operations.

In order to keep full compatibility for clients of our API that run in a browser, raw UDP was not considered as a transport layer.

2) *Service layer*: The service layer will implement business logic independently of transport protocol:

SessionManager Manages session lifecycle (creation, lookup, expiry detection and cleanup) using a thread-safe dictionary protected by a lock. A background thread will periodically purge idle sessions to prevent unbounded memory growth. The WebSocket implementation will key sessions by `request.sid` while the HTTP implementation will use UUID-based identifiers.

SourceManager Handles audio file operations: accepting uploads, writing to a per-session directory, decoding audio into fixed-size buffer arrays via *miniaudio* (with *SciPy* as fallback) and caching decoded buffers in memory to avoid re-decoding on every render tick.

AudioProcessor Controls *py3dti* rendering, maintaining the *BinauralRenderer*, listener and source objects in the processing layer.

3) *Processing layer*: The processing layer will wrap *py3dti*, isolating the rest of the system from the library’s API conventions. Responsibilities will include loading HRTF datasets from SOFA files, managing the renderer and listener lifecycle, converting audio buffers to float32 NumPy arrays as required by *py3dti* and applying the coordinate transformation from client space to *py3dti*’s expected axes. This transformation

will take place as browser-based 3D environments, including Three.js used in the example browser client, follow a Y-up convention (Y axis pointing upward, Z axis toward the viewer), whereas *py3dti* follows the audio-industry Z-up convention used by the SOFA format for HRTF datasets. Without this remapping, source positions supplied by the browser would be interpreted incorrectly by the renderer, misplacing sound sources in the spatial scene.

B. API specification

1) *WebSocket API*: The WebSocket API will employ Socket.IO events with JSON payloads, which are listed in table I. Audio upload will be handled via HTTP POST rather than a Socket.IO event - binary file transfer is better suited to HTTP multipart semantics and avoids the overhead of framing large files through Socket.IO.

2) *HTTP-SSE API*: The HTTP API will employ RESTful endpoints with an `/api` prefix plus a long-lived SSE connection for server-pushed events. These are listed in table II. RESTful resource semantics (POST to create, DELETE to remove, PATCH to update) are used for control operations as they map naturally onto the session, source and listener resources the system manages.

3) *Shared endpoints*: The endpoints in table III are shared between both implementations and accessed via HTTP in both cases, as binary file transfer is better suited to HTTP multipart semantics than WebSocket or SSE framing. The returned `saved_filename` is subsequently passed to `stream_uploaded_file` (WebSocket) or the `/stream` endpoint (HTTP) to begin rendering.

C. Example browser client

To enable practical demonstration and testing of the spatial audio API, an interactive browser-based client was developed featuring comprehensive documentation, a live API playground and real-time 3D visualisation of spatial audio rendering. This interactive documentation website is hosted at <https://hadeelomar.github.io/spatialsocket>.

TABLE I
WEBSOCKET API

Event	Direction	Purpose
connect	Client → Server	Establishes Socket.IO connection; server assigns session
connected	Server → Client	Session confirmation with <code>session_id</code> , <code>sample_rate</code> , <code>buffer_size</code>
init_audio	Client → Server	Triggers HRTF loading from SOFA file
create_source	Client → Server	Creates a 3D audio source at a given position
update_position	Client → Server	Updates a source's 3D coordinates
update_listener	Client → Server	Updates listener position and forward/up orientation
stream_uploaded_file	Client → Server	Registers a pre-uploaded file in the render loop
stop_file_stream	Client → Server	Removes all sources from the render loop
file_audio_chunk	Server → Client	Streams rendered binaural audio (base64 float32 stereo)
status	Server → Client	Acknowledges operations with structured status codes
error	Server → Client	Reports processing errors

TABLE II
HTTP-SSE API ENDPOINTS

Endpoint	Method	Purpose
/api/session	POST	Creates new session, returns <code>session_id</code>
/api/session/<sid>	DELETE	Terminates session and cleans up resources
/api/<sid>/events	GET	SSE stream for all server-pushed events
/api/<sid>/init	POST	Initialises HRTF renderer
/api/<sid>/sources	POST	Creates a new audio source
/api/<sid>/sources/<src_id>	DELETE	Removes a source
/api/<sid>/sources/<src_id>/position	PATCH	Updates source position
/api/<sid>/sources/positions	PATCH	Batch position update
/api/<sid>/listener	PATCH	Updates listener position and orientation
/api/<sid>/sources/<src_id>/upload	POST	Uploads audio file for a source
/api/<sid>/sources/<src_id>/stream	POST	Starts streaming a source
/api/<sid>/stream/stop	POST	Stops all active streams

TABLE III
SHARED ENDPOINTS

Endpoint	Method	Purpose
/upload/<session_id>/<source_id>	POST	Upload an audio file for a specific source; returns <code>saved_filename</code>
/upload/<session_id>/hrtf	POST	Upload a custom SOFA-format HRTF dataset for the session

1) *Documentation website structure*: The client was implemented as a single-page application serving multiple purposes:

API documentation section Comprehensive reference documentation covering all WebSocket events and HTTP endpoints, including request/response formats, parameter specifications and usage examples. The documentation was structured to guide developers through the API systematically, from initial connection establishment through audio streaming workflows.

Live API playground An interactive testing interface allowing users to connect to the API, create sessions, upload audio files, update source positions in real-time and receive spatialised binaural output, all without writing code. This playground provides immediate feedback for each operation, displaying session IDs, acknowledgement messages and streaming status.

Three.js 3D visualiser A real-time visualisation component rendering the spatial audio scene in three dimensions, showing listener position (represented as a head model with orientation

indicators), audio source positions (represented as colored spheres) and dynamic position updates reflected immediately in the 3D scene.

The client provides real-time feedback for all operations, with status indicators showing connection state, active sessions, streaming progress and any errors encountered. This enabled thorough manual testing of the API and served as a practical demonstration tool for evaluating the spatial audio rendering quality and system responsiveness.

In addition to this example client, unit tests and integration tests covering the codebase are provided.

V. EXPERIMENTAL SETUP

A. Containerisation and production deployment

The application was containerised using Docker to enable realistic load testing and production deployment. A docker-compose file defines application services for each transport protocol with asymmetric resource allocations reflecting their workloads. The WebSocket service receives 4 CPUs and 12

GB RAM with a renderer pool of 100 pre-loaded processors, enough to serve 100 concurrent users without any per-session HRTF load delay. The HTTP service receives 2 CPUs and 4 GB RAM, reflecting its stateless-per-request design.

Local testing carries a fundamental limitation: when the client, server and load generator all run on the same machine, they compete for CPU resources, conflating server performance with resource contention. Therefore the API was also deployed on Amazon AWS for testing purposes.

B. Protocol equivalence validation

Both transport protocols share the same `AudioProcessor` and `SourceManager` implementations. Equivalence is established by feeding identical source data into the WebSocket and HTTP render helpers, then comparing the WebSocket stereo frame with the HTTP payload after decoding. This validates that both transport paths preserve the same rendered audio and that the HTTP-specific base64 framing doesn't alter sample values.

This test exercises the HTTP render helper used in production and compares its payload with the byte stream produced by the WebSocket framing path for the same rendered frame. By decoding the HTTP response and matching it against the WebSocket-encoded output, the test verifies both shared DSP behaviour and protocol-specific serialisation.

C. Browser client validation

The browser client was validated through manual testing covering the complete workflow:

Connection testing Verified WebSocket connection establishment, session ID retrieval and connection state indicators functioning correctly.

Audio upload and playback Tested audio file upload through the playground interface, confirmed binaural output playback through Web Audio API and validated that spatialised audio correctly reflected source positions.

Position update testing Verified real-time position updates through both manual controls and programmatic API calls, confirmed visual feedback in Three.js visualiser synchronised with server state and tested smooth animation of position transitions.

Documentation accuracy Cross-referenced documentation examples with actual API behaviour, confirmed all documented endpoints and events functioned as described and validated parameter specifications and response formats.

D. Distributed load testing with Locust

Locust is an open-source distributed load testing framework that simulates concurrent users by executing configurable user behaviour scripts and reporting latency percentiles and failure rates across defined concurrency levels [20].

To address local testing limitations (resource contention when client, server and load generator share CPU), distributed load testing was implemented using Locust against the containerised server on AWS EC2.

Test configuration

- Server: AWS EC2 c5a.2xlarge (8 vCPU AMD EPYC 7R32, 16GB RAM) in eu-west-2
- Load generator: Separate machine running Locust
- Network-isolated setup simulating real traffic patterns

E. Core audio parameters

Sample rate: 48,000 Hz All tests are conducted with an audio sample rate of 48 kHz by default. SOFA-format HRTF databases vary in their native sample rates. Common values include 44.1 kHz, 48 kHz and 96 kHz, with some datasets providing measurements at multiple rates and the chosen audio rate must be compatible with the loaded HRTF. An HRTF measured from KEMAR GRAS EarSim dummy head with small ears is used for the test, as made available through the SONICOM HRTF dataset [21].

Streaming buffer size: 1,024 samples At 48 kHz, 1,024 samples yields a render tick of approximately 21.3 ms. This balances latency against overhead: smaller buffers reduce latency but increase thread pool dispatch frequency, while larger buffers reduce overhead but push latency toward the 100 ms budget. At 21.3 ms per tick, there is roughly 4 ticks of headroom before exceeding 100 ms, providing margin for network jitter and processing variation.

Maximum sources per session: 16 Sixteen simultaneous spatialised sources is sufficient for complex auditory scenes while bounding per-session compute cost. Each source requires a full HRTF convolution pass; at 16 sources, render time per tick remains tractable on a single core.

VI. PERFORMANCE EVALUATION

A. Single-session protocol comparison

Single-session benchmarks comparing WebSocket vs HTTP are conducted locally for controlled protocol comparison. The results are shown in table IV.

Both protocols show identical session memory at 88.8MB, confirming that the footprint is entirely attributable to *py3dti*'s in-memory HRTF representation rather than to protocol-specific state. The memory delta was measured using *psutil* RSS before and after initialisation. The actual footprint depends on the angular resolution of the loaded SOFA dataset, and the SONICOM KEMAR dataset used here has fine enough resolution to produce this result. A coarser dataset or a lower-resolution HRTF mesh would reduce the per-session footprint proportionally.

The counterintuitive finding is that HTTP achieved 64% lower pipeline latency despite WebSocket's superior transport efficiency. Component-level profiling revealed Flask-SocketIO's abstraction layer as the cause: Socket.IO adds protocol overhead (packet type parsing, namespace decoding, event extraction, JSON parsing) before messages reach application handlers. Flask's simpler HTTP routing has minimal parsing overhead. The overhead is not inherent to WebSocket but specific to the Socket.IO abstraction.

Despite overall pipeline disadvantage, WebSocket demonstrated 81% advantage for position update responsiveness (2.0ms vs 10.7ms), critical for spatial audio where source

TABLE IV
COMPARING TRANSPORT PROTOCOLS

Metric	WebSocket	HTTP	Winner	Notes
Transport RTT (ms)	3.3	11.2	WS (70% faster)	Raw protocol efficiency
Pipeline Latency (ms)	48.1	17.4	HTTP (64% faster)	End-to-end processing
Position Update RTT (ms)	2.0	10.7	WS (81% faster)	Critical for spatial coherence
Chunk Jitter σ (ms)	4.8	0.9	HTTP (81% lower)	Timing consistency
Connect Time (ms)	53.6	9.8	HTTP (82% faster)	Session establishment
HRTF Init Time (s)	13.355	13.086	$\approx$ Tie	Dominated by SOFA loading
Session Memory (MB)	88.8	88.8	Tie	py3dti dominates footprint

movement must be reflected immediately. For lightweight operations, WebSocket’s persistent connection and minimal framing overhead dominate.

B. Multi-session scalability analysis

Scalability testing was conducted on AWS EC2 using Locust. Tests ran for 5 minutes at each load level with 30-second ramp-up. The results can be found in table V.

TABLE V
SHARED ENDPOINTS

Concurrent users	Connect p95 (ms)	HRTF Init p95 (ms)	First Chunk p95 (ms)	Position Update p95 (ms)	Position Failures
10	110	25	35	57	0%
25	300	92	120	1300	0%
50	310	100	180	3000	0%
100	540	190	300	5000	11%
				(timeout)	

Connection scalability Zero connection failures up to 100 users. The renderer pool (100 pre-loaded processors) ensures connections receive a ready renderer immediately, eliminating the 8.7-second per-session HRTF load delay that would otherwise serialise all initialisations.

Audio streaming scalability Graceful degradation from 35 ms (10 users) to 300 ms (100 users) p95 time-to-first-chunk with zero streaming failures. Audio renders are bounded to one per 21 ms buffer interval per session, so even with serialisation of each render call, a 50-worker `_render_pool` can service 100 concurrent sessions without catastrophic queuing.

Position update bottleneck Severe degradation from 57 ms to 5000 ms p95 latency with 11% failures at 100 users. Investigation revealed ineffective usage of multiple cores as the fundamental constraint.

C. Analysis of the main bottleneck: single core serialisation of concurrent operations

The Global Interpreter Lock (GIL) is a mutex within the Python interpreter that prevents more than one thread from executing Python bytecode at a time within a single process; because *py3dti* operations hold it throughout their execution, concurrent threads cannot execute these operations in parallel regardless of how many CPU cores are available.

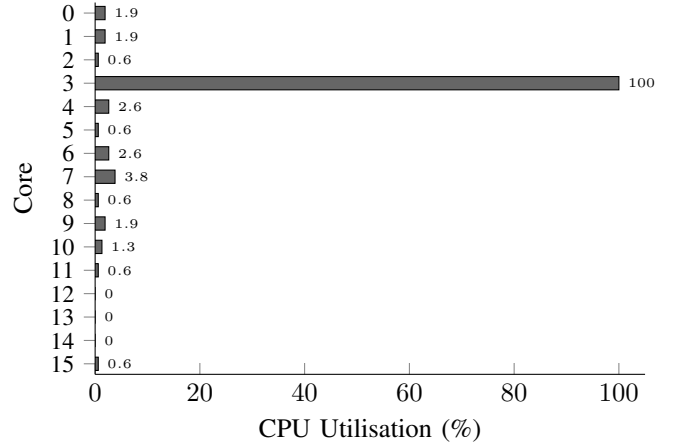


Fig. 4. Per-core CPU utilisation for the *gunicorn* worker (htop snapshot)

CPU utilisation evidence Per-core CPU monitoring in *htop* on a Linux-based OS during peak position update load revealed one core pinned at 100% with all remaining cores near-idle - the diagnostic signature of GIL serialisation. On the 16-core development machine used for local profiling, this corresponds to approximately 6% total system CPU (1/16 cores active). The equivalent measurement during the EC2 load test produced 13% total utilisation on the 8-core instance (1/8 cores), consistent with the same single-threaded execution pattern at different core counts. In both cases, the bottleneck is not CPU capacity, the remaining cores have headroom, but GIL contention forcing all concurrent position updates onto a single thread. If updates were truly parallel, utilisation would distribute across cores and approach 100% system-wide before latency degraded.

The solution is to replace thread-based concurrency with process-based concurrency using Python’s multiprocessing module. Unlike threads, which share a single GIL within one process, multiprocessing spawns separate Python interpreter processes (each with its own independent GIL) allowing the programmer to fully leverage multiple processors on the machine [22]. Critically, this doesn’t require any changes to *py3dti* itself: the library continues to function exactly it does within each worker process. Our session-isolated design makes it a natural fit: each session already maintains fully independent

AudioProcessor state with no cross-session shared mutable data, so assigning sessions to dedicated worker processes doesn't need any restructuring of the service layer. Control messages (position updates, source creation) would be passed between the transport layer and workers via `multiprocessing.Queue`; rendered audio buffers would be returned via `multiprocessing.shared_memory` to avoid copy overhead. Position update throughput would then scale with the number of worker processes rather than being capped at a single core. Within the current single-process architecture, a near-term partial mitigation is server-side rate-limiting of *py3dti* dispatch, coalescing client updates to a fixed tick rate (e.g. 30 Hz) which reduces serialisation pressure without architectural changes.

VII. CONCLUSION AND FUTURE WORK

In this paper, we present an open-source Python API for server-side binaural rendering. We compare approaches based on two different transport protocols. The API based on WebSocket achieves 48.1 ms pipeline latency with 3.3 ms transport RTT. HTTP-SSE achieved lower pipeline latency (17.4 ms) due to Socket.IO abstraction overhead, a counter-intuitive result that contradicts the standard assumption that WebSocket is faster for real-time applications. The finding demonstrates that framework overhead can dominate end-to-end latency when server processing time is substantial. For position updates, however, WebSocket's 2 ms RTT versus HTTP's 10.7 ms shows that the persistent connection model does provide meaningful advantage for lightweight, frequent operations where transport time is the dominant factor.

HTTP demonstrated 81% lower chunk jitter ($\sigma=0.9$ ms vs 4.8 ms), making it more suitable for applications where temporal consistency of the audio stream is the primary concern. WebSocket's higher jitter likely reflects variability in the event loop's scheduling under Socket.IO parsing overhead. For position updates, WebSocket's 2 ms responsiveness is critical for spatial coherence; at the 50 ms NFR2 threshold, WebSocket comfortably passes while HTTP at 10.7 ms still passes but with less headroom at scale.

In practice, of course, the latency introduced by our API pipeline is only one part of the total latency in a realistic system. Further latency will be added because of the block-based processing of audio recording and playback, and sensor polling and transmission of a head tracker, for instance.

Both protocols exhibit identical memory footprint, dominated by *py3dti*'s in-memory HRTF representation. Implementation complexity is similar since the shared service layer means only the transport layer differs. Scalability is constrained by Python's GIL rather than protocol choice: both transports share the same bottleneck.

While the currently available API already enables real use-cases, profiling its performance has shown a clear path towards further improvements. First, process-based concurrency via multiprocessing will be evaluated. Spawning separate interpreter processes, each with their own GIL, would allow concurrent sessions to execute in true parallel across CPU

cores without requiring any changes to *py3dti* or the existing session-isolated architecture. Second, a more efficient usage of the WebSocket protocol could be explored to verify that this would reduce the overhead introduced by Socket.IO. The *websockets* Python package⁴ could be a way to achieve this.

REFERENCES

- [1] W3C, "Web audio API 1.1," Tech. Rep., 2024. [Online]. Available: <https://www.w3.org/TR/webaudio-1.1/>
- [2] I. Fette and A. Melnikov, "The websocket protocol," RFC 6455, Tech. Rep., 2011. [Online]. Available: <https://doi.org/10.17487/rfc6455>
- [3] M. Tomasetti, A. Boem, and L. Turchet, "How to Spatial Audio with the WebXR API: a comparison of the tools and techniques for creating immersive sonic experiences on the browser," in *2023 Immersive and 3D Audio: from Architecture to Automotive (I3DA)*, Sep. 2023, pp. 1–9.
- [4] J. Blauert, *Spatial hearing: the psychophysics of human sound localization*. Cambridge: The MIT Press, 1997.
- [5] F. Rumsey, *Spatial audio*. Oxford: Focal Press, 2001.
- [6] G. Kailas, A. Behera, and N. Tiwari, "Tracing the evolution of headphone-based spatial audio: from principles to applications," *International Journal of Interactive Design and Manufacturing*, 2025.
- [7] E. Wenzel, M. Arruda, D. Kistler, and F. Wightman, "Localization using nonindividualized head-related transfer functions," *Journal of the Acoustical Society of America*, vol. 94, no. 1, pp. 111–123, 1993.
- [8] M. Cuevas-Rodríguez, L. Picinali, D. González-Toledo, C. Garre, E. de la Rubia-Cuevas, L. Molina-Tanco, and A. Reyes-Lecuona, "3D Tune-In Toolkit: An open-source library for real-time binaural spatialisation," *PLOS ONE*, vol. 14, no. 3, p. e0211899, 2019.
- [9] P. Majdak, F. Zotter, F. Brinkmann, J. De Muyne, M. Mihocic, and M. Noisternig, "Spatially oriented format for acoustics 2.1: Introduction and recent advances," *Journal of the Audio Engineering Society*, vol. 70, no. 7/8, pp. 565–584, 2022.
- [10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol - HTTP/1.1," RFC 2616, Tech. Rep., 1999. [Online]. Available: <https://doi.org/10.17487/rfc2616>
- [11] M. Thomson and C. Benfield, "HTTP/2," RFC 9113, Tech. Rep., 2022. [Online]. Available: <https://doi.org/10.17487/rfc9113>
- [12] M. Bishop, "HTTP/3," RFC 9114, Tech. Rep., 2022. [Online]. Available: <https://doi.org/10.17487/rfc9114>
- [13] P. Cairns, H. Daffern, and G. Kearney, "Immersive Network Music Performance: Design and Practical Deployment of a system for Immersive Vocal Performance," *Journal of the Audio Engineering Society*, Oct. 2020.
- [14] J.-P. Cáceres and C. Chafe, "JackTrip: Under the Hood of an Engine for Network Audio," *Journal of New Music Research*, vol. 39, no. 3, pp. 183–187, Sep. 2010.
- [15] V. Pimentel and B. Nickerson, "Communicating and displaying real-time data with WebSocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.
- [16] D. Škvorc, M. Horvat, and S. Srbljic, "Performance evaluation of WebSocket protocol for implementation of full-duplex web streams," in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, May 2014, pp. 1003–1008.
- [17] W. Łasocha and M. Badurowicz, "Comparison of websocket and HTTP protocol performance," *Journal of Computer Sciences Institute*, vol. 19, pp. 67–74, 2021. [Online]. Available: <https://doi.org/10.35784/jcsi.2452>
- [18] W. Jakob, J. Rhineland, and D. Moldovan, "pybind11 – seamless operability between C++11 and Python," 2017, <https://github.com/pybind/pybind11>.
- [19] M. Grinberg, "Flask-SocketIO documentation." [Online]. Available: <https://flask-socketio.readthedocs.io/en/latest/>
- [20] Locust Contributors, "Locust: An open source load testing tool," 2024. [Online]. Available: <https://docs.locust.io/en/stable/what-is-locust.html>
- [21] I. Engel, R. Daigintis, T. Vicente, A. Hogg, J. Pauwels, A. Tournier, and L. Picinali, "The SONICOM HRTF dataset," *Journal of the Audio Engineering Society*, vol. 71, no. 5, pp. 241–253, 2023.
- [22] Python Software Foundation, "multiprocessing – process-based parallelism," 2024. [Online]. Available: <https://docs.python.org/3/library/multiprocessing.html>

⁴<https://websockets.readthedocs.io/en/stable/>