The Journal of Computational Science and Engineering (TJCSE)
ISSN 2583-9055 (Media Online)
Vol 4, No 5, May 2026
PP 1–11

XenShare: A Peer-to-Peer File Sharing System Using WebRTC with Chunk-Based Transfer and Server-Side Performance Logging

Mr. K. Shankar[1], D.Sai Krishna[2], Nitin.K[3]*

[1,2] Dept. Of CSE, NSRIT, NSRIT, Visakhapatnam, AP, India.

[3] Dept. of MCA, NSRIT, Visakhapatnam, AP, India.

Corresponding Author *: [1] mkopanati@gmail.com, [3] nitinkosuru@icloud.com

Abstract

Most browser-based file sharing tools depend on a central server to receive, store and forward the file, which leads to privacy exposure, storage cost and a ceiling on file size. XenShare doesn't transmit file data to the server at all. After the connection is established, it uses the WebRTC RTCDataChannel API abstracted through PeerJS to transmit bytes directly between browsers. It's a static, single page application hosted on Vercel, with serverless functions, no persistent backend and no writing any file content to server-side storage.This is a stable peer identifier stored in localStorage in each browser . So identity is stable between sessions without an account . Files are transferred in 64 KB binary chunks with explicit backpressure control the sender monitors bufferedAmount against an 8 MB high-water mark and pauses until a bufferedamountlow event fires at a 1 MB low-water mark, addressing the buffer mismanagement shown to degrade data-channel throughput under default settings [11]. On larger payloads, say, 333 MB or greater, the system uses the File System Access API rather than buffering in memory, streaming chunks directly to a location selected by the receiver, avoiding the memory pressure that plagues many browser-based transfer tools.The JSON Web Token is short-lived (30 seconds) and is issued and verified across two serverless endpoints, proving valid before the application logic is served The 30 second expiry is to limit casual inspection, not to claim unbreakable obfuscation. All completed transfers are logged to MongoDB Atlas asynchronously, regardless of the transfer path, so a database failure never impacts the download. XenShare was able to maintain 11–12 MB/s between two PCs on a local Wi-Fi network. Disc streaming path 586.56 MB file transferred at 12.27 MB/s in 47.8 seconds. Mobile transfers were limited to 2-3 MB/s with mobile because of the mobile WiFi stack, not the application. The results indicate that a true server-free transfer path is possible on typical consumer hardware using only standard browser APIs.

Keywords: WebRTC, RTCDataChannel, Peer-to-Peer File Transfer, PeerJS, Backpressure Control, JSON Web Token, File System Access API, Serverless Architecture, MongoDB Atlas, DTLS.

The Journal of Computational Science and Engineering (TJCSE)
ISSN 2583-9055 (Media Online)
Vol 4, No 5, May 2026
PP 1–11

1. Introduction

Sharing a file with someone else on the internet almost always means giving it to a third party first. Email attachments are stored on a mail server, links to cloud-storage are routed through the provider's infrastructure and, even dedicated services like WeTransfer require the entire upload before the recipient can start downloading. That's handy, but it comes at a cost. The operator can see, keep, or be forced to disclose anything that passes through it. And the size of files a user can send is limited to whatever quota the service allows.

Modern browsers offer an alternative. The WebRTC RTCDataChannel API is intended for real-time applications like video chats, where two browsers can establish a direct, encrypted channel and exchange arbitrary binary data after the signalling is done [1]. Community tools like ShareDrop and FilePizza have demonstrated this to be possible for file transfer, but they are typically proof-of-concept: Big files occupy browser memory because chunks are stored entirely in RAM, mobile devices disconnect when the screen locks and there's no log of how a transfer actually went.

XenShare takes that proof-of-concept further, with a single developer working with standard browser APIs and free cloud services. It adds a 64 KB chunk-based transfer protocol with explicit backpressure control, an adaptive large-file strategy that switches from in-memory buffering to direct disc streaming via the File System Access API past an empirically determined size threshold, and two supporting layers that most hobbyist tools omit — JWT-gated delivery of the client script and a MongoDB Atlas logging pipeline — all without a single persistent server process.

The rest of this article surveys current browser-based and server-mediated file transfer, outlines the architecture and implementation of XenShare, shows results in PC-to-PC, PC-to-mobile and mobile-to-PC scenarios on real hardware, and ends with a candid discussion of current limitations and remaining work..

2. Literature Survey

File transfer used to be choosing between two bad things: a server that would hold your file for you, or a direct connection that was hard for regular people to set up. Centralised services Google Drive, Dropbox, WeTransfer, email attachments solved the usability problem decades ago, by making the server an unavoidable participant in every transfer and so every byte sent is a byte the operator can read, log, or lose, bounded by a storage quota someone is paying for.

This is changed with WebRTC, which provided a standardised way for browsers to negotiate a direct connection without either side installing anything. Its RTCDataChannel part is protocol

agnostical anything expressible as bytes can go over it as long as the connection exists [3] making plugin-free, server-free file transfer possible in principle. How well it works in practice depends on implementation details that the specification doesn't constrain, like how aggressively a sender pushes data before checking whether the other side has caught up.

Early efforts such as ShareDrop and FilePizza proved the concept, but also exposed its rough edges: both keep the whole file in browser memory on at least one side, limiting the practical file size to the available heap; neither tracks how a transfer went; and both rely on third-party signalling infrastructure they do not control. XenShare addresses these particular gaps -- memory limits, performance visibility, code protection -- without reinventing the WebRTC file transfer wheel.

3. Related Work

WebRTC Data Channels and Throughput Behaviour. The underlying stack for RTCDataChannel is SCTP over DTLS over UDP as described in the IETF documents, RFC 8825 for the architectural picture [1] and RFC 8829 for the session negotiation via RTCPeerConnection [2]. Neither dictates send-buffer management, and that difference matters [11] observed that modern browsers did not adjust the default window size of SCTP, resulting in poor throughput when round-trip time deviated from ideal LAN conditions. This is directly relevant to the design of XenShare. Instead of sending send() in a tight loop, the sender watches bufferedAmount and pauses when it is at a high-water mark, and resumes only when a bufferedamountlow event indicates the receiver has drained it.

Browser-Based Peer-to-Peer File Sharing Tools. ShareDrop and FilePizza remain the most visible community implementations and informed the comparative evaluation in this work (see Results). Mozilla's Firefox Send pursued encrypted server-side storage rather than peer-to-peer delivery and was discontinued in 2020 a reminder of how quickly tools in this space disappear, and a point in favour of depending on nothing beyond commodity browser APIs.

Relay-Mediated Alternatives. Not every recent system avoids the server entirely. [12] show a WebRTC-inspired transport protocol that sends encrypted chunks via a relay server to avoid the NAT-traversal difficulty. This is a useful contrast, showing that NAT topology often determines whether a true peer-to-peer path is possible. XenShare uses the stricter STUN-only NAT traversal with no relay fallback [9] easier to deploy, but connections across symmetric NATs may simply fail, a limitation discussed later rather than hidden behind a fallback relay.

Security and Delivery Protection. JSON Web Tokens usually authenticate a user's identity to an API [7], XenShare repurposes the mechanism more narrowly, to gate access to the client-side script itself via a 30-second-expiry token checked between two serverless functions a modest claim that raises the bar against casual inspection rather than asserting genuine obfuscation.

The Journal of Computational Science and Engineering (TJCSE)
ISSN 2583-9055 (Media Online)
Vol 4, No 5, May 2026
PP 1–11

4. System Architecture and Design

XenShare's architecture is organised into three layers, shown in Fig. 1. The Client Layer is the two browser instances sender and receiver running the same script and communicating directly over an RTCDataChannel once connected. The Infrastructure Layer handles connection setup: PeerJS's cloud signaling server exchanges SDP and ICE candidates, Google's public STUN servers assist NAT traversal, and Vercel hosts the static assets and the serverless functions that issue JWTs and accept transfer logs. The Data Layer is MongoDB Atlas, reached only by the logging function after a transfer completes it never sees the file itself, only metadata describing it.

Fig. 1: XenShare — System Architecture Diagram

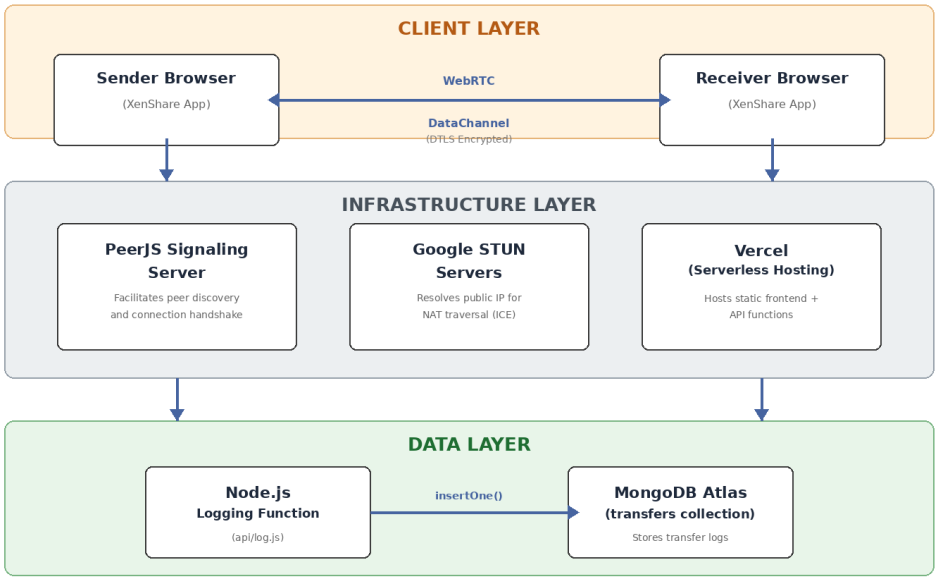

Fig. 1. System architecture of XenShare.

File data flows only within the Client Layer. The Infrastructure Layer handles only signaling, NAT traversal, script delivery, and log storage; once the RTCDataChannel opens, every chunk travels directly between sender and receiver, encrypted end-to-end by the DTLS layer WebRTC mandates beneath the data channel.

5. Implementation Details

The Journal of Computational Science and Engineering (TJCSE)
ISSN 2583-9055 (Media Online)
Vol 4, No 5, May 2026
PP 1–11

Peer Identification and Connection Establishment

On first load, a browser generates a persistent identifier of the form peer-XXXXXXX and stores it in localStorage under peershare:id, so that coming back to the app later will show the same identity rather than a new one each session a lightweight alternative to accounts that requires no registration or password.

Connection setup is carried out via standard WebRTC offer/answer exchange (Fig. 2) abstracted by PeerJS, so the application never builds SDP messages itself. The sender's browser asks PeerJS's signalling server to relay a request to the receiver's known peer ID. After both sides exchange session descriptions and ICE candidates via STUN, an RTCDataChannel opens between both. XenShare adds one step: after the opening of the channel the sender sends a start message detailing the incoming file and waits for an explicit ready acknowledgement before sending any data necessary because the large-file path below may require the receiver to first answer a save-location prompt, which has to happen before chunks start arriving

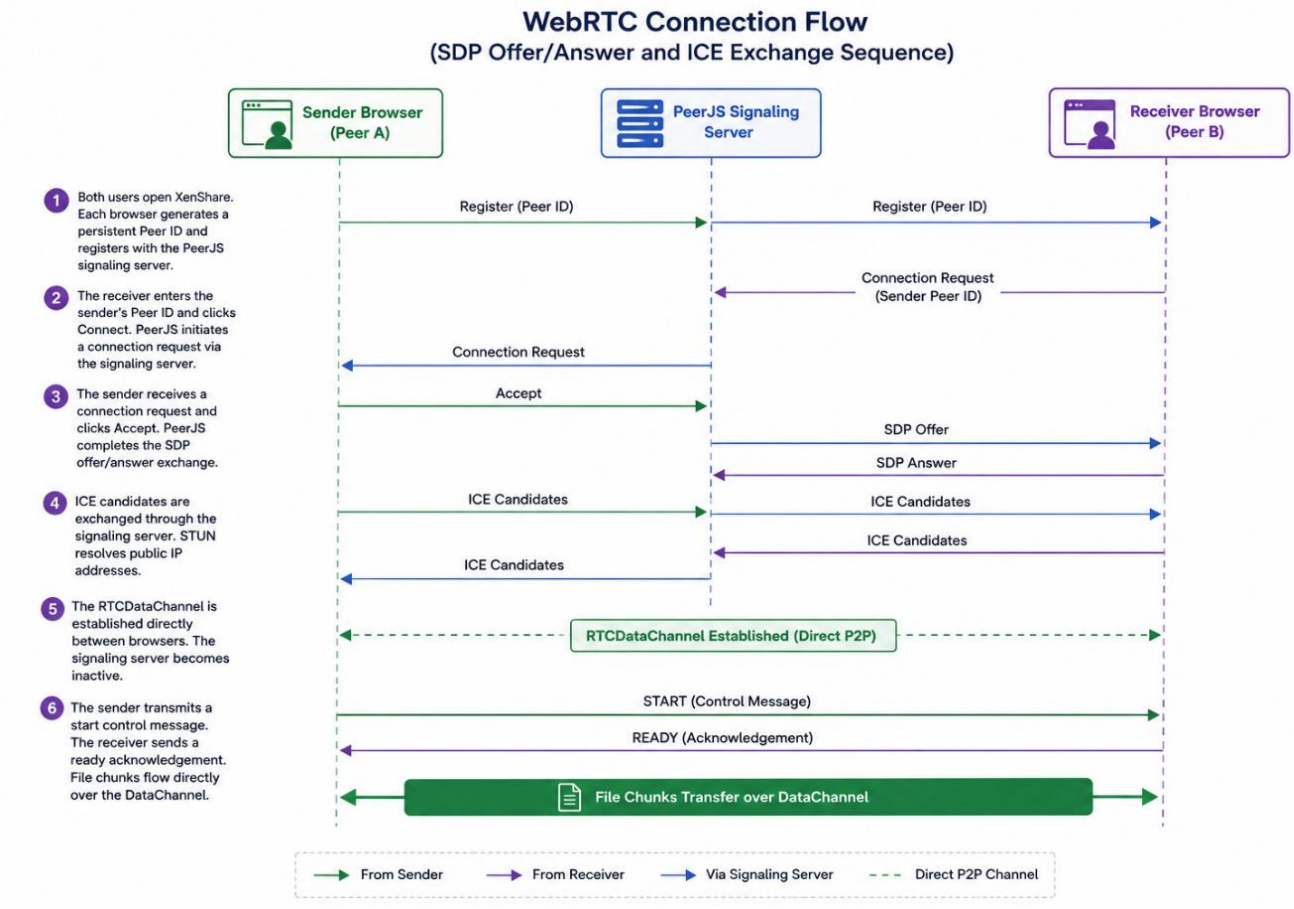

Fig. 2. Sequence of the connection handshake prior to transfer.

Chunk-Based Transfer with Backpressure Control

Once ready arrives, the sender's pumpSend() function slices the file into 64 KB ArrayBuffer chunks via file.slice().arrayBuffer() and writes each to the channel as binary data through conn.send(ab). An earlier implementation intercepted the channel's onmessage handler directly outside PeerJS's own data path, which silently conflicted with PeerJS's internal wiring and dropped incoming data entirely. The corrected design routes all incoming data exclusively through PeerJS's conn.on('data'), reserving direct RTCDataChannel access for one purpose only: reading bufferedAmount and configuring bufferedAmountLowThreshold for backpressure.

Backpressure keeps a fast sender from overwhelming the channel: pumpSend() checks bufferedAmount before each chunk and, once it reaches an 8 MB high-water mark, pauses until a bufferedamountlow event fires at the 1 MB low-water mark. Zero buffer-overflow events were observed across all transfers measured for this article, a result the Discussion returns to in light of prior findings on default SCTP window behaviour (Nurminen & Eskola, 2015).

Adaptive Handling of Large Files

Small and large files are received differently. Below 333 MB, incoming chunks accumulate in a JavaScript array and are concatenated into a single Blob for download once complete simple, but bounded by available heap. At or above that threshold, the receiver instead prompts the user via window.showSaveFilePicker() to choose a disk location before transfer begins, and each chunk is written directly to disk through a FileSystemWritableFileStream rather than held in memory.

Both the 333 MB cutoff and the disk-streaming approach emerged from an earlier failed attempt using the Origin Private File System (OPFS), which produced intermittent zero-byte downloads from a race between the write stream and transfer completion. Replacing OPFS with the File System Access API, plus the ready handshake above, resolved the issue; the largest file moved through this path during testing was 586.56 MB, delivered without any memory-related failure.

JWT-Protected Script Delivery

XenShare’s application logic is in one script that is never served from a guessable path. On load the HTML shell asks a serverless function for a token, signed with HS64 and a 30 second expiry. That token is passed as a bearer credential to a second function, which verifies it and only then returns the script with a Cache-Control: no-store header. This is a deliberately narrow protection, it pushes the effort beyond “open developer tools” without claiming to defeat a determined adversary.

The Journal of Computational Science and Engineering (TJCSE)
ISSN 2583-9055 (Media Online)
Vol 4, No 5, May 2026
PP 1–11

Transfer Logging to MongoDB Atlas

After a transfer finishes through either the RAM or disk-streaming path the client calls sendLog(), posting file name, size, duration, throughput, peer identifiers, and status to a serverless endpoint that inserts the document into a MongoDB Atlas collection. Logging failures are caught and discarded inside the endpoint itself, so nothing about the user-visible transfer depends on whether the database write succeeded.

Mobile Resilience

Two mechanisms target mobile devices specifically: the Screen Wake Lock API keeps the display active for a transfer's duration, and a ping/pong message exchanged every five seconds lets the client detect a dropped connection after the page's visibility changes for instance, the user switching apps and returning and attempt to re-establish it.

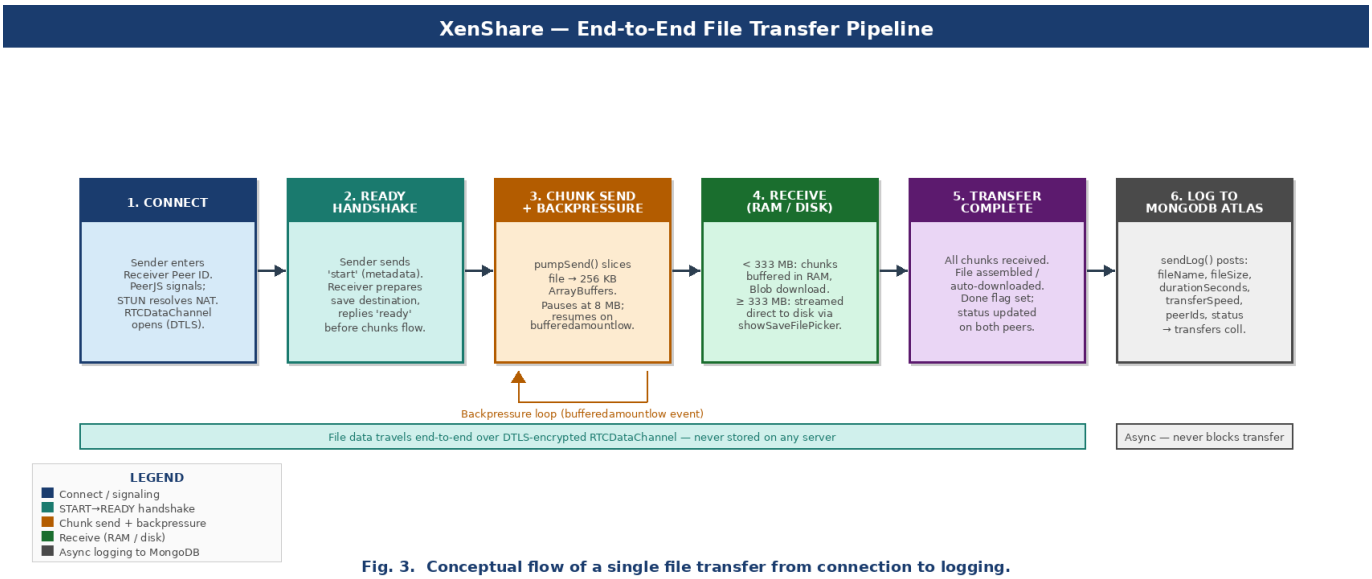

Fig. 3. Conceptual flow of a single file transfer from connection to logging.

Fig. 3. Conceptual flow of a single file transfer from connection to logging.

Example Use Cases

XenShare covers a few situations existing tools handle awkwardly: moving a large file a video export, a VM image, a folder of photos between two devices on the same network without uploading it anywhere first; a one-off transfer to someone outside the local network where the alternative would be a cloud upload bounded by a free-tier quota; and simply not wanting the file

The Journal of Computational Science and Engineering (TJCSE)
ISSN 2583-9055 (Media Online)
Vol 4, No 5, May 2026
PP 1–11

to exist anywhere except the two participating devices, a guarantee several existing services cannot make regardless of their stated privacy policy since the file necessarily passes through their infrastructure during upload.

6. Experimental Evaluation

Setup

Testing used ordinary consumer hardware rather than a dedicated lab network: a Windows PC running Chrome and Edge, an Android phone running Chrome, and a shared home WiFi network for all transfers. The figures below describe one network, not a distribution of networks, and should be read as indicative of typical behaviour rather than a controlled laboratory measurement. Functional correctness connection setup, chunk transfer, backpressure, JWT issuance/verification, and MongoDB log insertion was validated across 33 individual test cases, all of which passed.

Performance Metrics

Table 1 summarises throughput and latency collected during testing, alongside two reliability indicators that speak to whether the backpressure and chunking design held up under real use.

Metric	Observed Value	Observation
PC-to-PC transfer speed (same WiFi)	11 – 12 MB/s	Approaches WiFi bandwidth ceiling
PC-to-Mobile transfer speed	2 – 3 MB/s	Limited by mobile WiFi stack
Mobile-to-PC transfer speed	2 – 3 MB/s	Consistent with mobile hardware
586.56 MB file transfer time	47.8 s	12.27 MB/s average speed
JWT token request latency	< 100 ms	Serverless cold start included
DataChannel buffer-overflow events	0	Backpressure control effective
Transfer log insertion latency	< 500 ms	MongoDB Atlas free tier
Largest file tested	586 MB	Disk-streaming path, no RAM issues
File integrity (size match)	100%	All transfers verified by size

Table 1. Performance metrics observed during testing.

The Journal of Computational Science and Engineering (TJCSE)
ISSN 2583-9055 (Media Online)
Vol 4, No 5, May 2026
PP 1–11

7. Results

Table 2 places XenShare alongside three existing tools chosen for their position on the server-involvement spectrum: ShareDrop and FilePizza are WebRTC-based peer-to-peer tools, while WeTransfer represents the conventional server-upload model XenShare was designed to avoid.

Parameter	XenShare	ShareDrop	FilePizza	WeTransfer
File data through server	Never	Never	Never	Always
End-to-end encryption	DTLS (mandatory)	DTLS	DTLS	TLS (server sees file)
Large-file RAM handling	Disk streaming (333 MB+)	RAM only	Service Worker	Server upload
Source code protection	JWT two-step auth	None	None	N/A
Transfer logging	MongoDB Atlas	None	None	Server-side logs
Mobile Wake Lock	Yes	No	No	N/A
Auto-reconnect	Yes	No	No	N/A
File size limit	Unlimited (disk path)	RAM-limited	Unlimited (SW)	Plan-dependent

Table 2. Comparison of XenShare against existing file-sharing tools.

A 586.56 MB file took 47.8 seconds to traverse the disk-streaming path with the receiver's memory usage essentially flat, since chunks were written to disc as they arrived, instead of being accumulated in an array. Backpressure performed well under this load, as it did in every other test, with zero buffer-overflow events across the entire set. The most significant difference in Table 2 is not raw speed, which is bounded mostly by network conditions common to all of these tools, but the file-data-through-server row, where XenShare and the other WebRTC-based tools share a "never" that WeTransfer cannot match by design.

8. Discussion

The results show that XenShare meets its design goals: a transfer path that does not store data on the server, throughput near the WiFi ceiling, and robustness to the buffer-overflow failure mode exhibited by naive WebRTC implementations. A few insights came out in development and testing. Backpressure as the deciding factor. Throughput remained stable across all runs, in agreement with Nurminen and Eskola (2015) who found that WebRTC data channels do not manage their own buffering well by default. Instead of calling send() in a loop and hoping the channel keeps up, explicit HIGH_WATER (8 MB) and LOW_WATER (1 MB) thresholds were used.The 333 MB threshold as a pragmatic compromise. Moving from in-memory buffering to disc streaming trades simplicity for memory safety; 333 MB was settled on empirically, after the OPFS-based approach proved unreliable, as a point comfortably below where consumer hardware starts to struggle.Security and logging as decoupled layers. If the token cannot be transferred (via the JWT mechanism or the logging pipeline), the app still loads (a visible, fail-safe failure). If the logging fails, it is silently absorbed and never reaches the user.

9. Limitations and Future work

Several limitations need to be stated clearly. XenShare uses PeerJS's cloud signalling and STUN only for NAT traversal no TURN relay so connections over symmetric NATs or restrictive corporate networks may fail outright. Resumable transfer is not supported by the system: a connection drop during a transfer restarts from zero, a costly operation for large disk-streamed files. It only supports one-to-one transfers, multi-peer broadcasting is not implemented. The File System Access API that powers large-file streaming is not available in Firefox and mobile Safari, leaving those browsers stuck with the in-memory path, no matter how big the file is. Finally the performance figures are based on a single local WiFi network and a single mobile data path tested by the developer, not a large scale or geographically distributed test bed, and should be taken as indicative rather than a guarantee across all network conditions.There are several extensions planned but not yet implemented: a self-hosted PeerJS signalling server (Railway or Render) to remove dependency on shared cloud infrastructure; a TURN server to handle symmetric NATs and restrictive networks where STUN alone fails; resumable transfer via chunk-index tracking and persisted partial progress, likely through IndexedDB; and a real-time dashboard over the existing MongoDB logs to make collected performance data actually visible. A Service Worker-based streaming fallback was an attempt to add large-file support to Firefox and mobile Safari, but was reverted due to zero-byte downloads and race conditions of its own; revisiting it is still future work, not a solved problem. One-to-one, fixed-chunk design would be extended with multi-peer broadcast and adaptive chunk sizing based on measured round-trip time and bandwidth.

Conclusion

In this article we introduced XenShare, a WebRTC RTCDataChannel-based browser-to-browser peer-to-peer file transfer system with explicit backpressure control, adaptive large-file transfer with the File System Access API, JWT-protected script delivery, and server-less transfer logging with MongoDB Atlas. Real hardware testing of PC-to-PC, PC-to-mobile and mobile-to-PC throughput topped out near the local WiFi ceiling, logged zero buffer-overflow events and successfully delivered a 586.56 MB file without memory exhaustion.
In a larger sense, the system shows that a truly server-free path of transfer can be realised today using only standard browser APIs and free-tier cloud services for the limited supporting functions of signalling, script delivery, and logging. The issues discussed no TURN fallback, no resumable transfer, an incomplete browser-compatibility surface are not fundamental barriers, but aspects of the implementation that have been deliberately left unfinished, providing a clear road-map for the future work described.

12. References

[1] Alvestrand, H. (2021). Overview: Real-Time Protocols for Browser-Based Applications. IETF RFC 8825.

[2] Uberti, J., Jennings, C., & Rescorla, E. (2021). JavaScript Session Establishment Protocol (JSEP). IETF RFC 8829.

[3] W3C. (2023). WebRTC 1.0: Real-Time Communication Between Browsers. W3C Recommendation. https://www.w3.org/TR/webrtc/

[4] PeerJS Team. (2024). PeerJS Documentation: Peer-to-Peer Simplified. https://peerjs.com/docs/

[5] W3C. (2023). File System Access API. W3C Editor's Draft. https://wicg.github.io/file-system-access/

[6] W3C. (2023). Screen Wake Lock API. W3C Candidate Recommendation. https://www.w3.org/TR/screen-wake-lock/

[7] Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT). IETF RFC 7519.

[8] Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446.

[9] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., & Matthews, P. (2020). Session Traversal Utilities for NAT (STUN). IETF RFC 8489.

[10] Mahy, R., Matthews, P., & Rosenberg, J. (2010). Traversal Using Relays around NAT (TURN). IETF RFC 5766.

[11] Nurminen, J. K., & Eskola, R. (2015). Performance evaluation of WebRTC data channels. In Proceedings of the 20th IEEE Symposium on Computers and Communication (ISCC) (pp. 676–680). IEEE.

[12] Rahalkar, C., & Virgaonkar, A. (2024). Designing a secure device-to-device file transfer mechanism. arXiv preprint arXiv:2411.13827.

[13] MongoDB, Inc. (2024). MongoDB Atlas Documentation. https://www.mongodb.com/docs/atlas/

[14] Vercel, Inc. (2024). Vercel Documentation: Serverless Functions. https://vercel.com/docs/functions

[15] WHATWG. (2024). HTML Living Standard: Web Storage. https://html.spec.whatwg.org/multipage/webstorage.html